Undergraduate Topics in Computer Science

John Hunt

Advanced Guide to Python 3 Programming

Second Edition

UTiCS

python™

Springer

Undergraduate Topics in Computer Science

'Undergraduate Topics in Computer Science' (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems, many of which include fully worked solutions.

The UTiCS concept relies on high-quality, concise books in softback format, and generally a maximum of 275–300 pages. For undergraduate textbooks that are likely to be longer, more expository, Springer continues to offer the highly regarded Texts in Computer Science series, to which we refer potential authors.

John Hunt

Advanced Guide to Python 3 Programming

Second Edition

Springer

John Hunt
Midmarsh Technology Ltd.
Chippenham, Wiltshire, UK

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-031-40335-4 ISBN 978-3-031-40336-1 (eBook)
https://doi.org/10.1007/978-3-031-40336-1

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

For Denise, my wife, my soulmate, my best friend.

Preface to the Second Edition

This second edition represents a significant expansion of the material in the first edition, as well as an update of that book from Python 3.7 to 3.12.

This book includes whole new sections on advanced language features, Reactive Programming in Python and data analysts. New chapters on working with Tkinter, on event handling with Tkinter and a simple drawing application using Tkinter have been added. A new chapter on performance monitoring and profiling has also been added. A chapter on pip and conda is included at the end of the book.

In all there are 18 completely new chapters that take you far further on your Python journey. Enjoy the book and I hope you find it useful.

Chippenham, UK John Hunt

Preface to the First Edition

Some of the key aspects of this book are

1. It assumes knowledge of Python 3 and of concepts such as functions, classes, protocols, abstract base classes, decorators, iterables and collection types (such as list and tuple).
2. However, the book assumes very little knowledge or experience of the topics presented.
3. The book is divided into eleven topic areas: advanced language features, Computer Graphics, games, testing, file input/output, database access, logging, concurrency and parallelism, Reactive Programming, network programming and data analytics.
4. Each topic in the book has an introductory chapter followed by chapters that delve into that topic.
5. The book includes exercises at the end of most chapters.
6. All code examples (and exercise solutions) are provided on line in a GitHub repository.

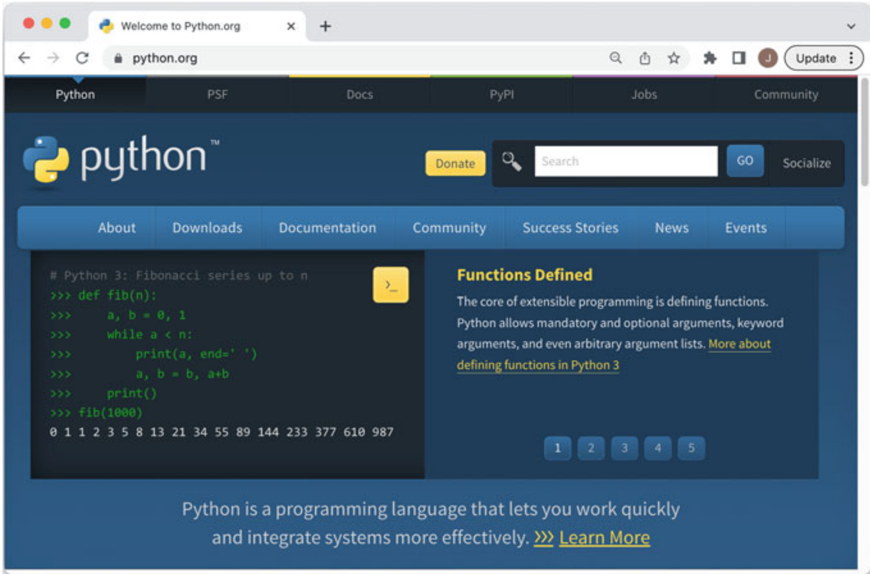
What You Need

You can of course just read this book; however following the examples in this book will ensure that you get as much as possible out of the content. For this you will need a computer.

Python is a cross-platform programming language, and as such you can use Python on a Windows PC, a Linux box, an Apple Mac, etc. So you are not tied to a particular type of operating system; you can use whatever you have available.

However you will need to install some software on that computer. At a minimum you will need Python. The focus of this book is Python 3 so that is the version that is assumed for all examples and exercises. As Python is available for a wide range of platforms from Windows, to Mac OS and Linux, you will need to ensure that you download the version for your operating system.

Python can be downloaded from the main Python website which can be found at http://www.python.org/.



You will also need some form of editor to write your programs. There are numerous generic programming editors available for different operating systems with VIM on Linux, Notepad++ on Windows and Sublime Text on windows and Macs being popular choices.

However, using an Integrated Development Environment (IDE) editor such as PyCharm, Visual Studio Code or Spyder can make writing and running your programs much easier.

However, this book does not assume any particular editor, IDE or environment (other than Python 3 itself).

Conventions

Throughout this book you will find a number of conventions used for text styles. These text styles distinguish between different kinds of information. Code words, variable and Python values, used within the main body of the text, are shown using a Courier font. A block of Python code is set out as shown here:

```
def draw_koch(size, depth):
    if depth > 0:
      for angle in ANGLES:
        draw_koch(size / 3, depth - 1)
        turtle.left(angle)
  else:
      turtle.forward(size)
  # Draw three sides of snowflake
  for_in range(3):
    draw_koch(SIZE_OF_SNOWFLAKE, depth)
    turtle.right(120)
```

Note that keywords and points of interest are shown in bold font.

Any command line or user input is shown in standard font as shown below, for example:

```
Hello, world
Enter your name: John
Hello John
```

Example Code and Sample Solutions

The examples used in this book (along with sample solutions for the exercises at the end of most chapters) are available in a GitHub repository. GitHub provides a web interface to Git, as well as a server environment hosting Git.

Git is a version control system typically used to manage source code files (such as those used to create systems in programming languages such as Python but also Java, C#, C++ and Scala). Systems such as Git are very useful for collaborative development as they allow multiple people to work on an implementation and to merge their work together. They also provide a useful historical view of the code (which also allows developers to roll back changes if modifications prove to be unsuitable).

The GitHub repository for this book can be found at:

- https://github.com/johnehunt/advancedpython3_2nd

If you already have Git installed on your computer, then you can clone (obtain a copy of) the repository locally using:

```
git clone   https://github.com/johnehunt/advancedpyth
on3_2nd.git
```

If you do not have Git, then you can obtain a zip file of the examples using

```
https://github.com/johnehunt/advancedpython3_2nd/arc
hive/refs/heads/main.zip
```

You can of course install Git yourself if you wish. To do this, see https://git-scm.com/downloads. Versions of the Git client for Mac OS, Windows and Linux/Unix are available here.

However, many IDEs such as PyCharm come with Git support and so offer another approach to obtaining a Git repository.

For more information on Git see `http://git-scm.com/doc`. This Git guide provides a very good primer and is highly recommended.

Acknowledgement I would like to thank Phoebe Hunt for creating the pixel images used for the Starship Meteors game in Chap. 22.

Contents

Chapter 1
Introduction

1.1 Introduction

I have heard many people over the years say that Python is an easy language to learn and that Python is also a simple language.

To some extent both of these statements are true; but only to some extent.

While the core of the Python language is easy to lean and relatively simple (in part thanks to its consistency), the sheer richness of the language constructs and flexibility available can be overwhelming. In addition the Python environment, its eco system, the range of libraries available, the often competing options available, etc., can make moving to the next-level daunting.

Once you have learned the core elements of the language such as how classes and inheritance work, how functions work, what are protocols and Abstract Base Classes, etc. where do you go next?

The aim of this book is to delve into those next steps. The book is organised into eleven different topics:

1. **Advanced Language Features**. The first section in the book covers topics that are often missed out from introductory Python books such as slots, weak references __init__() versus __new__() and metaclasses.
2. **Computer Graphics**. The book covers Computer Graphics and Computer Generated Art in Python as well as graphical user interfaces and graphing/charting via Matplotlib.
3. **Games Programming**. This topic is covered using the `pygame` library.
4. **Testing and Mocking**. Testing is an important aspect of any software development; this book introduces testing in general and the PyTest module in detail. It also considers mocking within testing including what and when to mock.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_1

5. **File Input/Output**. The book covers text file reading and writing as well as reading and writing CSV and Excel files. Although not strictly related to file input, regulator expressions are included in this section as they can be used to process textual data held in files.
6. **Database Access**. The book introduces databases and relational database in particular. It then presents the Python DB-API database access standard and one implementation of this standard, the PyMySQL module used to access a MySQL database.
7. **Logging**. An often missed topic is that of logging. The book therefore introduces logging the need for logging, what to log and what not to log as well as the Python logging module.
8. **Concurrency and Parallelism**. The book provides extensive coverage of concurrency topics including threads, processes and inter-thread or process synchronisation. It also presents futures and AsyncIO.
9. **Reactive Programming**. This section of the book introduces Reactive Programming using the PyRx Reactive Programming library.
10. **Network Programming**. The book introduces socket and web service communications in Python. It looks at both the Flask and the Django web service libraries.
11. **Data Analytics**. A very hot topic for any potential Python programmer is data analytics (and the related use of machine learning). The book concludes by introducing these topics and there Pandas and scikit-learn (or SK-learn as it is sometimes known) libraries.

Each section is introduced by a chapter providing the background and key concepts of that topic. Subsequent chapters then cover various aspects of the topic.

For example, the second topic covered is on Computer Graphics. This section has an introductory chapter on Computer Graphics in general. It then introduces the Turtle Graphics Python library which can be used to generate a graphical display.

The following chapter considers the subject of Computer Generated Art and uses the Turtle Graphics library to illustrate these ideas. Thus several examples are presented that might be considered art. The chapter concludes by presenting the well-known Koch Snowflake and the Mandelbrot Fractal set.

This is followed by a chapter presenting the Matplotlib library used for generating 2D and 3D charts and graphs (such as a line chart, bar chart or scatter graph).

The section concludes with a chapter on graphical user interfaces (or GUIs) using the `wxpython` library. This chapter explores what we mean by a GUI and some of the alternatives available in Python for creating a GUI.

Other topics follow a similar pattern.

Each programming or library-oriented chapter also includes numerous sample programs that can be downloaded from the GitHub repository and executed. These chapters also include one or more end of chapter exercises (with sample solutions also in the GitHub repository).

The topics within the book can be read mostly independently of each other. This allows the reader to dip into subject areas as and when required. For example, the File Input/Output section and the Database Access section can read independently of each other (although in this case assessing both technologies may be useful in selecting an appropriate approach to adopt for the long-term persistent storage of data in a particular system).

Within each section there are usually dependencies; for example, it is necessary to understand `pygame` library from the 'Building Games with pygame' introductory chapter, before exploring the worked case study presented by the chapter on the StarshipMeteors game. Similarly it is necessary to have read the threading and multiprocessing chapters before reading the inter-thread/process synchronisation chapter.

1.2 Useful Python Resources

There are a wide range of resources on the web for Python; we will highlight a few here that you should bookmark. We will not keep referring to these to avoid repetition but you can refer back to this section whenever you need to:

- https://en.wikipedia.org/wiki/Python_Software_Foundation Python Software Foundation.
- https://docs.python.org/3/ The main Python 3 documentation site. It contains tutorials, library references, set up and installation guides as well as Python how-tos.
- https://docs.python.org/3/library/index.html A list of all the built-in features for the Python language—this is where you can find online documentation for the various class and functions that we will be using throughout this book.
- https://pymotw.com/3/ the Python 3 Module of the week site. This site contains many, many Python modules with short examples and explanations of what the modules do. A Python module is a library of features that build on and expand the core Python language. For example, if you are interested in building games using Python then pygame is a module specifically designed to make this easier.
- https://www.fullstackpython.com/email.html is a monthly newsletter that focusses on a single Python topic each month, such as a new library or module.
- http://www.pythonweekly.com/ is a free weekly summary of the latest Python articles, projects, videos and upcoming events.

Each section of the book will provide additional online references relevant to the topic being discussed.

Part I
Advanced Language Features

Chapter 2
Python Type Hints

2.1 Introduction

Python is a dynamically typed language right—well yes it is however there is a feature known as *Type Hints* that allows typing information to be provided when functions and methods are defined. These Type Hints are extremely useful and can help a developer understand what types are expected by a function or a method and indeed what types are likely to be returned.

Recent versions of Python, including 3.10 and 3.11, have increased the support for Type Hints so that they are now quite usable. These can be used by analysis tools and IDEs to help developers create more stable and reliable applications.

2.2 Pythons Type System

Many people consider Python to be an untyped programming language. However, that is not quite true. The type system in Python can be referred to as representing a dynamically typed programming language. That is a variable holds a value and the type of that value is known and understood by the language. At runtime Python checks that what you are trying to do is valid given the types involved. For example, you can use the `type()` function to find out what type of thing a variable holds at any point in time:

```
a_variable = 42
print(f'a_variable type = {type(a_variable)} = {a_variable}')
a_variable = 1.345
print(f'a_variable type = {type(a_variable)} = {a_variable}')
a_variable = "Hello"
print(f'a_variable type = {type(a_variable)} = {a_variable}')
a_variable = True
print(f'a_variable type = {type(a_variable)} = {a_variable}')
```


J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_2

This produces as output:

```
a_variable type = <class 'int'> = 42
a_variable type = <class 'float'> = 1.345
a_variable type = <class 'str'> = Hello
a_variable type = <class 'bool'> = True
```

Of course, as the above shows, a variable in Python can hold different types of things at different types, hence the term dynamically typed.

As Python knows what types variables hold it can check at runtime that your programs are valid/correct given the types involved for example in a particular operation. Thus, it is valid to add two integers together and indeed two strings together (as this is string concatenation) but attempting to add an integer to a string will result in a `TypeError`:

```
print(1 + 1)
print(1.2 + 3.4)
print("Hello" + "world")
print("Hello" + 1)
```

This code produced the following output including the `TypeError`:

```
2
4.6
Helloworld
Traceback (most recent call last):
  File "/Users/jeh/temp/pythonProjects/course/main.py", line 15,
in <module>
    print("Hello" + 1)
TypeError: can only concatenate str (not "int") to str
```

2.3 The Challenge for Python Developers

The challenge for Python developers comes when they need to understand what types are required by, or work with, some API. As a very simple example, consider the following function:

```
def add(x, y):
    return x + y
```

What types can be used with this function?

In essence any type can be used for the parameter `x` that supports the *plus* operator (+) with the type in `y`. From the above we know that integers and strings can be used, but we can also use floating point numbers, for example:

```
print(add(1, 2))
print(add(1.2, 3.4))
```

```
print(add(1, 3.4))
print(add(5.5, 1))
print(add("Hi", "There"))
```

All of the above are valid parameters, and the output produced from the above code is:

```
3
4.6
4.4
6.5
HiThere
```

Even custom types can be used if they implement the special `__add__(self, other)` operator method, for example:

```
class Quantity:
    def __init__(self, amount):
        self.amount = amount

    def __add__(self, other):
        return Quantity(self.amount + other.amount)

    def __str__(self):
        return f"Quantity({self.amount})"

q1 = Quantity(5)
q2 = Quantity(4)
print(add(q1, q2))
```

The `__add__()` method allows the custom type (class) being defined to be used with the add operator ('+'). Thus this program generates the following output:

```
Quantity(9)
```

However, what was the intent of the designer of this `add()` function? What did they expect you to add together? The only option in *traditional* Python code is for the developer to provide some form of documentation, for example in the form of a docstring:

```
def add(x, y):
    """adds two integers together and
       returns the resulting integer."""
    return x + y
```

2.4 Static Typing

Languages such as Java, C# and C are statically typed languages. That is when a variable, object attribute, parameter or return type is defined then the type of that element is specified statically a compile time.

Thus, an `add()` method on a Java class `Calculator` might be written as follows:

```
package com.jjh;

public class Calculator {

   public int add(int x, int y) {
      return x + y;
   }

}
```

This makes it clear to a Java programmer and to the Java compiler that the `add()` method will only handle integers and will return as a result an integer type. Thus, there is no possibility that a developer might try to add a number to a Boolean value, etc. In fact the compiler will not even allow it!

The Java `Calculator` class can be used as shown below, note that this code will not even compile if the developer tries to add two strings together. In this case we are adding two integers together, so all is fine:

```
package com.jjh;

public class App {
   public static void main( String[] args ) {
      System.out.println( "Starting" );
      Calculator calc = new Calculator();
      System.out.println(calc.add(4, 5));
      System.out.println("Done");
   }
}
```

As the above program uses valid integer types with the `add()` method, the output from the compiled and executed program is:

```
Starting
9
Done
```

2.5 Python Type Hints

Python's Type Hints are more like a half-way house between traditional Python's lack of typing information at all and the very strict string static typing approach of languages such as Java.

A Type Hint is additional type information that can be used with a function definition to indicate what types parameters should be and what type is returned. This is illustrated below:

```
def add(x: int, y: int) -> int:
```

```
    return x + y
```

In this case it makes it clear that both `x` and `y` should be of type `int` (integer types) and the returned result will be an `int`. However, adding Type Hints as shown above has no effect on the runtime execution of the program; they are only *hints* and are not enforced by Python per se. For example, it is still possible to pass a string into the `add()` function as far as Python is concerned.

However, static analysis tools (such as MyPy) can be applied to the code to check for such misuse. Some editors, such as the widely used PyCharm, already have such tools integrated into their code checking behaviour.

If you want to use a tool such as `mypy` instead, or in addition to that available in your IDE, then you can install it using

```
pip install mypy
```

Or if you want to use conda/Anaconda by using

```
conda install mypy
```

You can now analyse your code by applying MyPy to a Python file, for example:

```
% mypy main.py
main.py:3: error: Incompatible types in assignment (expression has type "float", variable has type "int")
main.py:5: error: Incompatible types in assignment (expression has type "str", variable has type "int")
main.py:24: error: Argument 1 to "add" has incompatible type "str"; expected "int"
main.py:24: error: Argument 2 to "add" has incompatible type "str"; expected "int"
main.py:44: error: Argument 1 to "add" has incompatible type "Quantity"; expected "int"
main.py:44: error: Argument 2 to "add" has incompatible type "Quantity"; expected "int"
Found 6 errors in 1 file (checked 1 source file)
```

2.6 Type Hint Layout

The Python Style Guide defined by Python Enhancement Proposal 8 (PEP 8) provides some guidance for using Type Hints, for example:

- Use normal rules for colons, that is, no space before and one space after a colon: text: str.
- Use spaces around the = sign when combining an argument annotation with a default value: align: bool = True.
- Use spaces around the -> arrow: def headline(…) -> str.

2.7 Type Hints for Multiple Types

Of course our `add()` function could work with floating point numbers as well as it works with integers. It would therefore be useful to be able to state this in terms of the Type Hints. Prior to Python 3.10 this could be done using a Union type, for example `Union[int, float]` which while it worked was a little unwieldy. Since Python 3.10 we can use the style syntax bar '|' for example `int | float` as shown below:

```
def add(x: int | float, y: int | float) -> int:
    return x + y
```

2.8 The Self Type

Python 3.11 introduced the Self type which is defined in PEP 673. This can be used to indicate that a method returns a reference to itself, for example:

```
from typing import Self

class Shape:

    def __init__(self):
        self.scale = 0.0

    def set_scale(self, scale: float) -> Self:
        self.scale = scale
        return self
```

2.9 The Benefits of Type Hints

There are a range of benefits to using Type Hints in Python, for example:

- **They help catch some errors within programs**. Obviously, the biggest benefit is that Type Hints can help developers catch certain types of problems in their code (assuming that some form of type checker is used).
- **They provide documentation**. Type Hints can also act as a level of document that editors such as IDEs can pick up and display to other developers.
- **They can be work with IDEs**. They can help with code generation and IDE auto-complete functionality.
- **They can make developers stop and think**. They can help ensure that developers think about their code and what types should be supported.
- **They can improve understanding of libraries**. Although Type Hints may offer little advantage in a single use script, or throw away program, they can be of significant benefit when a library is being created. Such libraries will be used

by a range of different developers, and some may be released into the wild, for example via PyPI, the Python Package Index. The use of Type Hints can greatly enhance others understanding of the APIs provided by these libraries.

2.10 Summary

If you are just starting out with Python, or you are writing scripts that will only be used once, then Type Hints may not be particularly useful. However, if you are creating libraries or developing larger more complex applications with teams of developers, then they can be very useful indeed.

2.11 Online Resources

- https://docs.python.org/3.10/ Python 3.10 documentation.
- https://docs.python.org/3.11/ Python 3.11 documentation.
- https://en.wikipedia.org/wiki/Type_system Wikipedia Type System page.
- https://en.wikipedia.org/wiki/Dynamic_programming_language Dynamically typed languages.
- https://docs.python.org/3/library/exceptions.html Exception handling in Python.
- https://www.pythontutorial.net/python-basics/python-type-hints/ Tutorial on Python Type Hints.
- https://pypi.org/project/mypy/ MyPy static type hint analysis tool.
- https://mypy.readthedocs.io/en/stable/ MyPy documentation.
- https://www.jetbrains.com/pycharm/ PyCharm IDE tool.
- https://peps.python.org/pep-0008/ Python (PEP 8) Style Guide including guidance of how to layout Type Hints.
- https://peps.python.org/pep-0673/ Information on the Self type.
- https://pypi.org/ The Python package Index PyPi.

Chapter 3
Class Slots

3.1 Introduction

Python classes are very flexible, they allow data and behaviour to be defined when the class is created, but also dynamically at any point in the lifetime of the class and its instances. This technique is known as Monkey Patching and can be extremely useful. However, in other situations, allowing the data or behaviour of a class to change dynamically after the class has been defined, might be very confusing and make the system harder to maintain. The issue is that a class's attributes can be added at any time, and there is no formal specification of the attributes—that is until we look at slots. Slots allow us to specify what attributes a class will have and to ensure that those attributes and only those attributes are used with the class and its instances. This chapter introduces Python class slots.

3.2 The Problem

In many object-oriented languages it is necessary to define the attributes or fields that a class will have explicitly within the class definition. For example, in Java we might write the following:

```
public class Person {
   private int age = 0;
   private String name = "";

   public Person (name String, int age) {
       this.name = name;
       this.age = age;
   }
   public int getAge () {
    return age;
```


J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_3

```
    }
    public void setAge (int newAge) {
     age = newAge;
    }
    public String getName () {
     return name;
    }
    public void birthday () {
     int oldAge, newAge;
     oldAge = getAge();
     System.out.println("Happy birthday " + getName());
     System.out.print("You were " + oldAge);
     System.out.print(" but now you are ");
     this.age = this.age + 1;
     System.out.println(age);
    }
}
```

In the above class the two lines:

```
private int age = 0;
private String name = "";
```

declare that the Java class `Person` will have two attributes (also known as instance variables in Java) called `name` and `age`. It cannot have any dynamically added additional attributes, and it is not possible to create on the fly within a method.

The equivalent class definition in Python might look like:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def birthday(self):
        print(f'Happy birthday {self.name}',
              f', you were {self.age}',
              end = '')
        self.age = self.age + 1
        print(f' but now you are {self.age}')
```

This does essentially the same thing (although in a more concise format). Using this class we can create instances of the class and print out the details associated with the object, for example:

```
p1 = Person('Phoebe', 25)
print(f'p1: {p1.name} {p1.age}')
```

which produces:

```
p1: Phoebe 25
```

However there is nothing to stop us adding a new attribute address to the class, for example:

```
p1.address = '10 High Street'
print(f'p1.address: {p1.address}')
```

When we run this we get:

```
p1.address: 10 High Street
```

This is not possible in Java as address was not defined within the scope of the class and thus objects of class Person in Java can never have any additional attributes such as address.

In Python not only is this legal, it is also sometimes quite useful.

However, how do you know that there is an attribute address on the object in p1? Only be reading through the code using p1, you cannot see it by looking at the Python class definition for Person.

Perhaps even more confusingly if we write:

```
p2 = Person('Gryff', 24)
print(f'p2: {p2.name} {p2.age}')
print(f'p2.address: {p2.address}')
```

That is we create a new instance of the class Person and try to access the attributes name, age and address (which apparently all work for the instance in p1), and we will raise a runtime AttributeError, for example:

```
p2: Gryff 24
Traceback (most recent call last):
  File "/person.py", line 19, in <module>
    print(f'p2.address: {p2.address}')
                         ^^^^^^^^^^
AttributeError: 'Person' object has no attribute 'address'
```

This indicates that the Person object in p2 does not possess an attribute address yet p1 did! Of course this is because we added the attribute only to the object in p1 not to the class in general.

This can be very confusing and make maintaining code much more difficult!

3.3 Slots to the Rescue

There is a special class attribute called __slots__ which can be used to provide a sequence of strings that define or specify the attributes that the class will hold. It is a class attribute as it is part of the class not part of an instance of object of the class. However, it defines the attributes any instance of the class can use.

Thus if an attribute is not included in the slots sequence then it cannot be defined within the class. This means that to find out what attributes a class defines for its objects all you have to do is look at the slots attribute and they will be listed there.

It also means that it is not possible to dynamically *monkey patch* a class with additional attributes at runtime.

As an example, see the modified definition for the class Person below:

```
class Person:
    __slots__ = ['name', 'age']

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f'Person({self.name} is {self.age})'
```

This version of the class `Person` lists the attributes `name` and `age` in the `__slots__` class attribute. Note that the names of the attributes are defined as strings—so don't forget the quotes around each attribute.

Then within the class an initialiser sets up the values for the attributes `self.name`, `self.age`, etc.

We can now create an instance of this class and for example print out the age, the name and use the `__repr__()` method to convert the object to a string for printing purposes:

```
p1 = Person('Phoebe', 25)
print(p1)
print(f'p1: {p1.name} {p1.age}')
```

The output from this is:

```
Person(Phoebe is 25)
p1: Phoebe 25
```

So far so good, but what has this given us over the original version?

If we now try to dynamically add an attribute such as `address` to this version of `Person`, for example:

```
p1.address = '10 High Street'
```

We will now generate a runtime `AttributeError` indicating that the object does not have an attribute `address`, for example:

```
Traceback (most recent call last):
  File "main.py", line 16, in <module>
    p1.address = '10 High Street'
    ^^^^^^^^^^
AttributeError: 'Person' object has no attribute 'address'
```

We have now fixed the attributes defined within the class to be `name` and `age` and only ever name and age.

This is actually true even if we tried to define an additional attribute within the *initialiser* method, for example, in the following version of the Person class has added the self.address attribute within the __init__() method:

```
class Person:
    __slots__ = ['name', 'age']

    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.address = None

    def __repr__(self):
        return f'Person({self.name} is {self.age})
```

When we try and use this class to create a new instance of the class Person we again get an AttributeError raised:

```
Traceback (most recent call last):
  File "main.py", line 13, in <module>
    p1 = Person('Phoebe', 25)
         ^^^^^^^^^^^^^^^^^^^^
  File "main.py", line 7, in __init__
    self.address = None
    ^^^^^^^^^^^^
AttributeError: 'Person' object has no attribute 'address'
```

Thus we are guaranteed that the class Person and all instances of the class Person, all work with just the name and age specified in the __slots__ class attribute.

3.4 Performance Benefits

There are in fact additional benefits to be had from using the __slots__ class attribute. These benefits relate to performance. This is because attributes defined using slots are more efficient in terms of memory space and speed of access and a bit safer than the default Python method of data access.

By default, when Python creates a new instance of a class, it creates a __dict__ attribute for the class. The __dict__ attribute is a dictionary whose keys are the variable names and whose values are the variable values. This allows for dynamic variable creation but can also lead to uncaught errors.

The fact that under the hood a simple dict is used for attribute storage and lookup as a few implications:

- Dictionaries are memory expensive objects. While this may not be a problem for a small class or for a class with only a few instances it can become far more significant with millions of objects as they will use a lot of memory.

- Dictionaries are based on a hash map. In the worst-case scenarios the time complexity of the get and set options on a hash map is of O(n) that is of Order(n). This means that when the time taken to access a value is measured as the number of elements in the dictionary increases so does the access time and that it increases linearly.

When using slots the attributes are created directly as properties of the class and by pass the default dict-based implementation. This is both more efficient in terms of access times and in terms of memory usage.

To illustrate this we can compare the size of the object created for the first version and the second version of the Person class.

The size of the object created without using slots is:

```
print(f'sys.getsizeof(p1) noslots - {sys.getsizeof(p1)}')
```

which generates:

```
sys.getsizeof(p1) noslots - 56
```

And if we run the version using slots:

```
print(f'sys.getsizeof(p1) slots - {sys.getsizeof(p1)}')
```

This produces:

```
sys.getsizeof(p1) slots - 48
```

which is 8 bytes less. This may not sound like much but multiple this by a million times and it starts becoming significant!

3.5 Why Not Use Slots?

So if slots are so good why doesn't everyone use them for all classes? In short although many people consider them a very useful features there as many who consider them poorly understood, difficult to get right and restrictive.

The main issue comes down to flexibility—do you want or need to be able to dynamically update the attributes in your class? If the answer is yes then slots are not for you, if you don't then slots may well be advantageous. Although the counter-argument is that you are making this design decision which will impact anyone who uses your class at any time in future!

Additionally by default slots cannot be used with weak references. Thus if you want your class to use weak references you can't use `__slots__`.

Although it should be noted that if you want to use slots with dynamic attributes or indeed with weak references then you can do so by adding '`__dict__`' or '`__weakref__`' as the last element in the `__slots__` declaration.

The final issue might be that a library you are using might rely on the presence of a `__dict__` for attributes for it to function correctly. For example, the `functools.cached_property()` is an example of a function that requires an instance `__dict__` to function correctly.

3.6 Online Resource

- https://wiki.python.org/moin/UsingSlots Tutorial on using slots.

Chapter 4
Weak References

4.1 Introduction

In this chapter we will look at *weak references*. Most of the time an object has one or more strong references associated with it. As long as at least one strong reference is associated with an object it cannot be automatically garbage collected. However, in some situations it may be useful to allow an object to be garbage collected (removed from memory) while a program is still using it, particularly if the available memory is becoming limited. To allow this to happen we can use weak references; weak references do not stop an object being removed from memory and can be very useful when used with a data cache, etc.

4.2 How Garbage Collection Works: Reference Counting

Before we go any further it is worth considering how the traditional Python garbage collector determines if an object can be removed or not from memory.

To do this the Python runtime keeps a record of the number of references to an object. If there is at least one (strong) reference to an object, then that object cannot be garbage collected. However, if an object has no (strong) references to it then the garbage collector can safely reclaim the memory used by the object.

In fact all Python objects include a reference count, which counts how many things are referencing it. If an object is referenced by another object, then its counter is incremented, if a variable references an object then the objects reference counter will again be incremented. If a variable or another object dereferences the object then the objects' reference counter is decremented, etc.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_4

By a reference we mean a variable or an attribute in another object is currently referencing, pointing at, the object.

As an example, we will define a simple class `Data` that will hold some sort of value:

```
class Data:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return f'Data({self.value})'
```

We can create a simple program to create an instance of this class and store it into a variable called `data`. Using the `id` of the object we can then find out how many references there are to the data object:

```
import ctypes

print('Create data item')
data = Data(1)

print('Obtain the id of list object')
data_id = id(data)

print('Find the reference count of data')
ref_count = ctypes.c_long.from_address(data_id).value
print(f"Reference count for data is: {ref_count}")
```

The above code uses the `ctypes.c_long.from_address` value attribute to find the number of (strong) references.

When we run this code we will see:

```
Create data item
Obtain the id of list object
Find the reference count of data
Reference count for data is: 1
```

That is, there is one reference to the `Data` object. This reference is currently held in the variable `data`. If we now make another two variables reference the object held in `data` we will increment the reference count:

```
print('Add some more references')
other_data1 = data
other_data2 = data

print('Find the reference count of data now')
ref_count = ctypes.c_long.from_address(data_id).value
print(f"Reference count for data is now: {ref_count}")
```

The output from this is:

```
Add some more references
Find the reference count of data now
Reference count for data is now: 3
```

As you can see there are now *3* things referencing the `Data` object.

And finally if we reset the `data` variable to `None` and check the number of references we will see that there are 2:

```
print('Reset data to None')
data = None
print('Find the reference count of data now')
ref_count = ctypes.c_long.from_address(data_id).value
print(f"Reference count for data is now: {ref_count}")
```

The output from this is:

```
Reset data to None
Find the reference count of data now
Reference count for data is now: 2
```

Notice that setting the `data` variable to `None` has no effect on the `other_data1` and `other_data2` variables, all it does is reduce the number of references to the `Data` instance from 3 down to 2.

4.3 Weak References

Unlike strong or normal references, a **weak reference** is a reference that does not protect the object from being garbage collected.

A weak reference is not included in the main count for an objects' reference. Thus if an object has no (strong) references to it but only *weak references,* then if the garbage collector needs to free up memory it can remove a weak reference object from memory. The result is that the weak reference no long references an object and any attempt to do so will return `None`.

That is to say that if the object is available in memory, calling the weak reference returns it, otherwise if the object has been garbage collected, then `None` is returned.

In terms of terminology, we say that the object that is the target of a reference is called the *referent*. Therefore, we can say that a *weak reference* does not prevent the *referent* from being *garbage collected*.

4.4 When to Use Weak References

Why do we want weak references as a thing in the first place?

There are two main applications of weak references:

- Implement caches for large objects (weak dictionaries).
- Handling circular references.

To create weak references Python has provided us with a module named `weakref`. We will explore the `weakref` module in the next section.

Note that some of the built-in types do not support weak references such as `tuple` and `int`.

4.5 The Weakref Module

The `weakref` module is a built-in module, and it is therefore provided as part of the reference implementation of Python. However, it is not made available to your code by default, and thus you must *import* weakref to use the module.

The following classes and methods are provided by the `weakref` module:

- **class weakref.ref(object[, callback])**—This returns a weak reference to the *object.*
- **weakref.proxy(*object*[, *callback*])—**This returns a proxy to *object* which uses a weak reference.
- **weakref.getweakrefcount(*object*)**—Return the number of weak references and proxies which refer to *object*.
- **weakref.getweakrefs(*object*)**—Return a list of all weak reference and proxy objects which refer to *object.*

4.6 Creating Weak References

The `ref()` function, in the `weakref` module, can be used to create a weak reference to an object. In turn the `weakref.getweakrefcount()` function can be used to obtain the number of weak references associated with an object. Both of these functions are illustrated below:

```
import ctypes
import weakref

print('Create data item')
data = Data(1)

print('Obtain the id of list object')
data_id = id(data)

print('Find the reference count of data')
ref_count = ctypes.c_long.from_address(data_id).value
print(f"Reference count for data is: {ref_count}")

print('Initial weak ref count')
weak_ref_count = weakref.getweakrefcount(data)
print(f"Number of weak references: {weak_ref_count}")

print('Add a weakref reference')
weakref_data = weakref.ref(data)

print('Find the reference count of data now')
```

```
ref_count = ctypes.c_long.from_address(data_id).value
print(f"Reference count for data is now: {ref_count}")

print('Find the weak ref count')
weak_ref_count = weakref.getweakrefcount(data)
print(f"Number of weak references: {weak_ref_count}")
```

In the above code the `Data` object has a strong reference to it via the `data` variable but a *weak reference* held in the `weakref_data` variable. The code uses the `types.c_long.from_address` value attribute to find the number of (strong) references and the `weakref.getweakrefcount()` to find the number of weak references.

The output from this code is:

```
Create data item
Obtain the id of list object
Find the reference count of data
Reference count for data is: 1
Initial weak ref count
Number of weak references: 0
Add a weakref reference
Find the reference count of data now
Reference count for data is now: 1
Find the weak ref count
Number of weak references: 1
```

This shows that the `Data` object has one (strong) reference and one weak reference to it.

If we now run the following code (after running the above) we will see that the `data` object no longer has a strong reference to it, and thus it can be (and in this case may have been) garbage collected:

```
print('Set data to None')
data = None
print('Find the reference count of data now')
ref_count = ctypes.c_long.from_address(data_id).value
print(f"Reference count for data is now: {ref_count}")

print('Find the weak ref count')
weak_ref_count = weakref.getweakrefcount(data)
print(f"Number of weak references: {weak_ref_count}")
```

The output is:

```
Set data to None
Find the reference count of data now
Reference count for data is now: 0
Find the weak ref count
Number of weak references: 0
```

which shows that the `Data` object has no strong references and now also does not have a weak reference either.

If we subsequently try and print out the weakref we will see that the associated object is dead:

```
print(weakref_data)
```

Generates:

```
<weakref at 0x10c84d2b0; dead>
```

4.7 Retrieving Objects from Weak References

Once we have created a weak reference, if you print it out, you see the output indicates that you have a *weakref* to an object of a specific type. To actually access the object being referenced, you need to apply the execution operator to *invoke* the weak reference (using the call operator `()`) and return the actual object being referenced.

For example the following code creates a weak reference and the accesses that weak reference directly before invoking it:

```
print('Create data item')
data = Data(1)
print('Add a weakref reference')
weakref_data = weakref.ref(data)
print(weakref_data)
print(weakref_data())
```

The output from this code is:

```
Create data item
Add a weakref reference
<weakref at 0x1020a91c0; to 'Data'at 0x1020b3c10>
Data(1)
```

As you can see we only access the `Data` object via the `weakref_data()` call, and the previous print accesses the weak reference wrapper.

4.8 The WeakValueDicitonary

The most common situation in which you might want to use weak references is with a cache. In such a situation you might want the cache to release cached values when the applications memory becomes used up. Fortunately, the `weakref` module provides a class called `WeakValueDictionary` which is a dictionary that associates keys with weak references. It is thus a very good basis for a simple weak reference style cache.

The `WeakValueDictionary` can be used in exactly the same was as a normal dictionary via the index accessor (`[]`) and via methods such as `get()`. For example:

```
import weakref

print('Create data item')
data = Data(1)

# creates a Weak Value Dictionary
weak_dict = weakref.WeakValueDictionary()

# inserting value into the dictionary
weak_dict['info'] = data

# getting the weak ref count
print(f'Weak reference count is: ',
      f'{weakref.getweakrefcount(weak_dict)}')
print(f'weak_dict: {weak_dict}')

print(f"weak_dict['info']: {weak_dict['info']}")
```

In this code a `WeakValueDictionary` is created and the data object is stored into the dictionary using the key '`info`'. Various checks are then made to see the number of weak reference counts it has and what the weak_dict contains. We then access the dictionary to retrieve the value associated with the key 'info'.

The output from this program is:

```
Create data item
Weak reference count is: 1
weak_dict: <WeakValueDictionary at 0x10f2c5b90>
weak_dict['info']: Data(1)
```

If the memory used by the program becomes limited, then the garbage collector could reclaim the memory associated with the `Data` object. We can simulate this by setting the original data variable to `None` and then attempt to access the WeakValueDictionary for the object:

```
data = None
if 'info'in weak_dict.keys():
    print(f"weak_dict['info']: {weak_dict['info']}")
```

However, we have to protect the access to the key behind and `if` statement. This is because if we access the key directly we would generate a `KeyError` as the entry would have been removed from the dictionary. For example:

```
Traceback (most recent call last):
  File "main3.py", line 22, in <module>
    print(f"weak_dict['info']: {weak_dict['info']}")
                              ~~~~~~~~~^^^^^^^^
  File "weakref.py", line 136, in __getitem__
    o = self.data[key]()
        ~~~~~~~~~^^^^^
KeyError: 'info'
```

4.9 WeakKeyDictionary

The `WeakKeyDictionary` is an alternative to the `WeakValueDictionary` in that in this version it is the keys that are weak references. The Python documentation says that the `WeakKeyDictionary` can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

4.10 Proxy Objects

A proxy object is another type of weak reference. The difference between a proxy object and a weak reference is that the proxy object tries to act as a proxy for the original object as much as it can.

An example of using a proxy object is shown below:

```
print('Create data item')
data = Data(1)

print('Obtain the id of list object')
data_id = id(data)

print('Find the reference count of data')
ref_count = ctypes.c_long.from_address(data_id).value
print(f"Reference count for data is: {ref_count}")

print('Add a weakref reference')
weakref_data = weakref.ref(data)

print('Find the weak ref count')
weak_ref_count = weakref.getweakrefcount(data)
print(f"Number of weak references: {weak_ref_count}")

print('Create a proxy of original object')
proxy_object = weakref.proxy(data)
print(f"This is a proxy object: {proxy_object}”)

print('Find the weak ref count')
weak_ref_count = weakref.getweakrefcount(data)
print(f"Number of weak references: {weak_ref_count}")
```

Notice that we have created both a proxy object and a weak reference to the `Data` object. When we run this we will see that both the weak reference and the proxy object are considered weak in terms of the `weakrefcount()` function. The output is:

```
Create data item
Obtain the id of the object
Find the reference count of data
Reference count for data is: 1
```

```
Add a weakref reference
Find the weak ref count
Number of weak references: 1
Create a proxy of original object
This is a proxy object: Data(1)
Find the weak ref count
Number of weak references: 2
```

The advantage in using the Proxy Object is that it tries to look like the original object as much as possible, thus we don't need to call the proxy object to return the `Data` object being referenced. Instead we can treat the proxy object as if it was the actual data object, for example:

```
print(proxy_object)
print(proxy_object.value)
```

This code prints the proxy object directly and accesses the value attribute directly—although in actual fact it goes via the proxy to the wrapped Data object. The output from the code is:

```
Data(1)
1
```

4.11 Online Resources

- https://docs.python.org/3/library/weakref.html Python weak reference documentation.
- https://www.educative.io/answers/what-is-weak-reference-in-python Short tutorial on weak references.

Chapter 5
Data Classes

5.1 Introduction

Python 3.7 introduced a special type of class called a *data class*. Data classes can be used to represent data-oriented concepts. That is, concepts that represent data but tend not to have much related functionality. Such a class might contain several properties but other than member functions for equality or string conversions they do not contain any behaviour. They are often used in larger applications as they are a useful way to group associated attributes together using a named concept and as such have far more semantic meaning than say a simple dictionary. This chapter presents such data classes.

5.2 A Traditional Class

We will start off by defining a typical Python class. This class represents a (very simplified) Equity Trade. That is, one person or organisation wishes to sell some equity (e.g. shares) to another person or organisation. Those involved in the trade are referred to as counter parties. The equity or shares in an organisation are represented by a symbol, and there will be some number of shares involved in the sale.

A simple definition of such as class in Python might look like:

```
class Trade:
    """Class for representing Equity Trades"""

    def __init__(self,
                 counter_party1,
                 counter_party2,
                 symbol,
                 amount=0):
        self.counter_party1 = counter_party1
```


J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_5

```
        self.counter_party2 = counter_party2
        self.symbol = symbol
        self.amount = amount

    def __repr__(self):
        return f'Trade(counter_party1={self.counter_party1}, ' \
               f'counter_party_2={self.counter_party2}, ' \
               f'symbol={self.symbol}, ' \
               f'amount={self.amount})'

    def __eq__(self, other):
        if not isinstance(other, Trade):
            return False
        return self.counter_party1 == other.counter_party1 and \
            self.counter_party2 == other.counter_party2 and \
            self.symbol == other.symbol and \
            self.amount == other.amount
```

We can use this class as shown below:

```
trade1 = Trade('John', 'Denise', 'IBM', 100)
print(trade1)
trade2 = Trade('John', 'Gryff', 'MSFT', 50)
print(trade2)
print(trade1 == trade2)
trade3 = Trade('John', 'Denise', 'IBM', 100)
print(trade1 == trade3)
```

The output from this code is:

```
Trade(counter_party1=John, counter_party_2=Denise, symbol=IBM,
amount=100)
Trade(counter_party1=John, counter_party_2=Adam, symbol=MSFT,
amount=50)
False
True
```

If we look at the class definition for the `Trade` type we can see that:

- It defines a new class `Trade`.
- A `Trade` contains two counter parties, a `symbol` and an `amount` attribute. That we have defined an initialiser to set up these attributes and that the `amount` has a default value of zero.
- The `Trade` class defines a `__repr__` method that can be used to represent this type as a string (note we are not defining a `__str__` method here as if it is not present `__repr__` will be used instead).
- The `Trade` class also defines an equals methods (`__eq__`) to allow two instances of the `Trade` class to be compared by value rather than by reference. Thus above `trade1` and `trade3` are equivalent by value although they are different instances of the `Trade` class.

This pattern is actually not that uncommon. Many classes in an application will start off like this. Some will have significant additional behaviour defined for them but many others will not.

So is there a problem here? Well from one aspect no; as this is just a Python class and it does what it sets out to do, it associated a group of attributes together within a named type (class); that is it represents a *Trade*.

However, there is a lot of boiler plate code here. If we were to define a `Book` class or a `Person` class or a `Temperature` class then all those classes might look very similar to the above and that a lot of very similar repetitive code to write. Indeed if you look at the initialise (`__init__`) then that itself is a repetitive structure with a lot of repeated patterns to take in some parameters and record them as attributes.

As you might suspect, as this is such a common pattern, there is a feature within Python that allows you to define such data-oriented classes in a much more concise and simpler way; these are *data classes*.

5.3 Defining Data Classes

Data classes can be used to represent data-oriented concepts. That is concepts that represent data, but tend not to have much related functionality (although there is no reason that they cannot have additional functionality).

Such a class might contain several attributes but other than methods for equality or string conversions they do not contain any (much) behaviour.

To define a data class, a class is annotated with the `@dataclass` decorator followed by a set of attributes (known as *fields* in a data class) with Type Hints associated with them. The definitions are thus:

```
from dataclasses import dataclass

@dataclass
def <classname>:
    """Docstring"""
    <fieldname>: type annotation
    <fieldname>: type annotation
    …
```

When a data class is defined it automatically receives:

- An initialiser, that is `__init__(self, …)` method.
- A `__repr__(self)` method that uses the information held in the attributes.
- A `__eq__(self, other)` method which compares values held in the attributes.

This is what we manually wrote in the previous section.

The Trade class can therefore be rewritten as a data class as shown below:

```
from dataclasses import dataclass

@dataclass
class Trade:
    """Class for representing Equity Trades"""
    counter_party1: str
    counter_party2: str
    symbol: str
    amount: int = 0
```

As you can see this is far shorter and much more concise and by default includes type information. Thus if you use a tool such as `mypy` it can ensure that you are using the correct types with the fields in the data class.

Note that the `amount` field has a *default* value provided this is used to make this field an *optional* field. A Trade can be created with or with the amount being specified. However any fields defined after the amount field must also have a default value.

This data class is used in exactly the same way as the original `Trade` class, and thus we can run the same application code with it, that is:

```
trade1 = Trade('John', 'Denise', 'IBM', 100)
print(trade1)
trade2 = Trade('John', 'Gryff', 'MSFT', 50)
print(trade2)
print(trade1 == trade2)
trade3 = Trade('John', 'Denise', 'IBM', 100)
print(trade1 == trade3)
```

The output from this code is exactly the same as that generated in the previous section:

```
Trade(counter_party1='John', counter_party2='Denise', symbol=
'IBM', amount=100)
Trade(counter_party1='John', counter_party2='Adam', symbol=
'MSFT', amount=50)
False
True
```

The difference is that we have written far less code, and a developer looking at our class will immediately see that it is a data class and that it holds a set of fields and will have a `__repr__` and an `__eq__` method defined. This means that will not have to scroll through a lot of code to determine that for themselves.

5.4 Defining Additional Behaviour

As mentioned at the start of this chelater, data classes often only hold data (excluding methods such as `__repr__` and `__eq__`). However, there is nothing stopping a data class having additional behaviour defined for them. For example, the following version of `Trade` has a method defined called `calculate_value` (of the trade). This takes a parameter indicating the price of each share and multiples it by the `amount` of shares being sold:

```
from dataclasses import dataclass

@dataclass
class Trade:
    """Class for representing Equity Trades"""
    counter_party1: str
    counter_party2: str
    symbol: str
    amount: int = 0

    # dataclasses can still have methods defined for them
    def calculate_value(self, price):
        return self.amount * price
```

We can invoke the `calculate_value()` method in exactly the same way as any method on any class:

```
trade1 = Trade('John', 'Denise', 'IBM', 100)
print(trade1)

print(f'The value of this trade at 1.55 per share =
{trade1.calculate_value(1.55)}')
```

The output from this code is:

```
Trade(counter_party1='John', counter_party2='Denise', symbol=
'IBM', amount=100)
The value of this trade at 1.55 per share = 155.0
```

In this case, this method makes perfect sense as we might want to hold the trade back until the value of the trade meets our objectives, and this might depend on the current stock market share price.

5.5 The Dataclass Decorator

The data class decorator itself has a set of parameters that can be used to configure how Python actually creates the data class implementation. There are numerous parameters as shown below:

```
@dataclass(init=True, repr=True, eq=True, order=False,
           unsafe_hash=False, frozen=False,
           match_args=True, kw_only=False, slots=False,
           weakref_slot=False)
```

The meaning of these parameters is outlined below:

- `init`: If `True` (the default), a `__init__()` method will be generated. If the class already defines `__init__()`, this parameter is ignored.
- `repr`: If `True` (the default), a `__repr__()` method will be generated. The generated repr string will have the class name and the name and repr of each field, in the order they are defined in the class. Fields that are marked as being excluded from the repr are not included. If the class already defines `__repr__()`, this parameter is ignored.
- `eq`: If `True` (the default), an `__eq__()` method will be generated. This method compares the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type. If the class already defines `__eq__()`, this parameter is ignored.
- `order`: If `True` (the default is `False`), `__lt__()`, `__le__()`, `__gt__()` and `__ge__()` methods will be generated. If the class already defines any of `__lt__()`, `__le__()`, `__gt__()` or `__ge__()`, then `TypeError` is raised.
- `unsafe_hash`: If `False` (the default), a `__hash__()` method is generated according to how `eq` and `frozen` are set.
- `frozen`: If `True` (the default is `False`), assigning to fields will generate an exception. This emulates read-only immutable instances.
- `match_args`: If True (the default is `True`), the `__match_args__` tuple will be created from the list of parameters to the generated `__init__()` method (even if `__init__()` is not generated, see above). If false, or if `__match_args__` is already defined in the class, then `__match_args__` will not be generated.
- `kw_only`: If `True` (the default value is `False`), then all fields will be marked as keyword-only. If a field is marked as keyword-only, then the only effect is that the `__init__()` parameter generated from a keyword-only field must be specified with a keyword when `__init__()` is called.
- `slots`: If `True` (the default is `False`), `__slots__` attribute will be generated. If `__slots__` is already defined in the class, then `TypeError` is raised.
- `weakref_slot`: If `True` (the default is `False`), add a slot named "`__weakref__`", which is required to make an instance weakref-able. It is an error to specify `weakref_slot=True` without also specifying `slots=True`.

Some of the above were added in Python 3.10 and 3.11 so if you are using an older version of Python not all of the decorator parameters may be available.

5.6 Custom Factory for Fields

Providing a default value for a field using a literal is straightforward, all you need to do is to provide that value, for example:

```
amount: int = 0
```

However, if you want to use a function to generate a value for a field, there is a subtle but important consideration to consider.

If you were to write:

```
market: StockExchange = make_stock_exchange()
```

This will indeed initialise the market field to a new StockExchange. However, the important point to note is when the `make_stock_exchange()` function would run. It would run when the class is first loaded, thus a new stock exchange instance would be created and used to initialise the definition for the whole class. This definition would then be shared amongst all instances of the class. That is, all instances of the trade would share the same `StockMarket` instance. This is fine if that is what you want! However, if you wanted different trades to be handled by different Stock Markets then it would not work.

If you need each instance to have their own StockExchange instance then the answer is to use a default *factory configuration* for your field. This allows a function to be referenced rather than invoked or executed, when the definition of the class is parsed by the Python runtime. This function reference will actually only be executed when a new instance of the class is generated. As such each new instance will run the function and create a new instance of the `StockMarket` class.

This can be done using the field function and specifying a `default_factory` which takes a reference to a named (or indeed a `lambda`) function. For example, updating the `Trade` class to have a `StockMarket` field using the `make_stock_exchange()` function can be defined as shown below:

```
from dataclasses import dataclass, field

class StockExchange:
     def __init__(self, name):
         self.name = name

     def __repr__(self):
         return f'StockExchange({self.name}'

def make_stock_exchange():
    return StockExchange('London Stock Exchange')

@dataclass
class Trade:
    """Class for representing Equity Trades"""
    counter_party1: str
    counter_party2: str
    symbol: str
    amount: int = 0
    market: StockExchange =
                field(default_factory=make_stock_exchange)

trade1 = Trade('John', 'Denise', 'IBM', 100)
print(trade1)
```

In this example each `Trade` instance will have their own instance of the StockMarket class held in the market field. Note that the `StockMarket` class could also have been a data class but is defined here as a plain old Python class to avoid any confusion.

Running this code generates the following output:

```
Trade(counter_party1='John', counter_party2='Denise', symbol=
'IBM', amount=100, market=StockExchange(London Stock Exchange)
```

5.7 Immutable Dataclasses

It is possible to indicate that a data class should be *immutable* using the frozen parameter to the `@dataclass` decorator. This ensures that all the fields within the data class are treated as read-only, and any attempt at modifying a value will generate a runtime error.

For example, the following data class `Book` is marked as *frozen* and is thus immutable:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Book:
    title: str
    author: str = 'Anonymous'

book1 = Book('Python for ever!', 'Gryff Smith')
print(book1)
```

Note that from a creation perspective, nothing has changed. The data class instance is created in exactly the same way as a normal class or indeed data class. Fields can also be accessed in the normal manner. If we run this code we see:

```
Book(title='Python for ever!', author='Gryff Smith')
```

However, if an attempt is made to modify either of the fields in the `Book` instance then a runtime error will be generated. For example:

```
book1.author = 'Adam Davies'
```

If we try and run the above line we will generate a `dataclasses.FrozenInstanceError` indicating that the author field cannot be assigned to:

```
Traceback (most recent call last):
   File "/Users/Shared/workspaces/pycharm/advancedpython3_2nd/
chapter2_dataclasses/main5.py", line 14, in <module>
     book1.author = 'Adam Davies'
     ^^^^^^^^^^^^
  File "<string>", line 4, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'author'
```

5.8 Data Classes and Inheritance

Data classes can extend any *other* class including other data classes. As usual this means that they will inherit all the fields and methods defined in the parent class. They can also override any fields or methods, etc. A simple example is given below:

```
from dataclasses import dataclass

@dataclass
class Trade:
    """Class for representing Equity Trades"""
    counter_party1: str
    counter_party2: str

@dataclass
class EquityTrade(Trade):
    symbol: str
    amount: int = 0

trade1 = EquityTrade('John', 'Denise', 'IBM', 100)
print(trade1)
trade2 = EquityTrade('John', 'Gryff', 'MSFT', 50)
print(trade2)
print(trade1 == trade2)
trade3 = EquityTrade('John', 'Denise', 'IBM', 100)
print(trade1 == trade3)
```

In this case the `EquityTrade` data class extends the `Trade` data class.

This means that the `EquityTrade` has four fields `counter_party1`, `counter_party2`, `symbol` and `amount`. The EquityTrade class also has an `__init__` methods that takes four parameters (with `amount` being an optional parameter with a default value of zero), a `__repr__` method that prints an `EquityTrade` with all four fields and an `__eq__` method that compares two instance of EquityTrade base don't he value of all four fields.

If we run the above code we see:

```
EquityTrade(counter_party1='John', counter_party2='Denise',
symbol= 'IBM', amount=100)
EquityTrade(counter_party1='John', counter_party2='Adam',
symbol= 'MSFT', amount=50)
False
```

```
True
```

However, there is one subtlety associated with inheritance that should be noted. When the *hidden* methods are created for you by Python the `__init__` initialiser is created by concatenating the definitions for the fields together to create the parameter list. This means that in the above example, the resulting initialised parameters look like:

```
def __init__(self,
   counter_party1,
   counter_party2,
   symbol,
   amount = 0)
```

This may not look like an issue but it becomes an issue if the parent class defines a default value for any of the fields. All subsequent fields must also have a default value; otherwise when the initialiser is generated the parameter list would be invalid (as all parameters to the right of a parameter with a default value must also have a default value).

This means the following is illegal:

```
from dataclasses import dataclass

@dataclass
class Location:
    name: str
    longitude: float = 0.0
    latitude: float = 0.0

@dataclass
class City(Location):
    country: str # Does NOT work

city = City('Dublin', country='Ireland')
print(city)
```

The above is invalid because the resulting initialiser for the City class would be:

```
def __init__(self, name, longitude = 0.0, latitude=0.0, country)
```

This is illegal because the `country` parameter comes after the `longitude` and `latitude` parameters, and these have a *default* value. It is caused by the simple listing of the fields in the order they are defined within the class inheritance hierarchy.

Thus if the parent class has a default value for a field, all fields in the subclass must have default values as well.

If you tried to run the above code you would get a runtime error:

```
Traceback (most recent call last):
  File "main7.py", line 9, in <module>
    @dataclass
     ^^^^^^^^^
  File "dataclasses.py", line 1223, in dataclass
```

```
    return wrap(cls)
           ^^^^^^^^^
  File "dataclasses.py", line 1213, in wrap
    return _process_class(cls, init, repr, eq, order, unsafe_hash,

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "dataclasses.py", line 1027, in _process_class
    _init_fn(all_init_fields,
  File "dataclasses.py", line 545, in _init_fn
    raise TypeError(f'non-default argument {f.name!r} '
TypeError: non-default argument 'country' follows default argument
```

The valid definition of the `City` class should look something like:

```python
from dataclasses import dataclass

@dataclass
class Location:
    name: str
    longitude: float = 0.0
    latitude: float = 0.0

@dataclass
class City(Location):
    country: str = None # Does work

city = City('Dublin', country='Ireland')
print(city)
```

Now when we run this code we see:

```
City(name='Dublin', longitude=0.0, latitude=0.0, country=
'Ireland')
```

5.9 Post Initialisation

One thing that may be a problem is what happens if you want to perform some initialisation behaviour of your own. By default a data class generates its own `__init__` initialisation method.

You can of course override this yourself and define your own `__init__` method as the processor won't generate an `__init__` method if one is manually defined by the programmer. However, this means that you have to include a lot of boiler plate code for each of the parameters, which was one of the benefits of having a data class in the first place!

Another option is to define a `__post_init__(self)` method. This method is run after the `__init__` method and can be used to perform any initialisation that you want to execute after all the fields have been set up—such as logging the fact, reading a preferences file, accessing a database and setting up further non-field attributes.

As a simple example, the following `Trade` class has had a `__post_init__` method added:

```
from dataclasses import dataclass

@dataclass
class Trade:
    """Class for representing Equity Trades"""
    counter_party1: str
    counter_party2: str
    symbol: str
    amount: int = 0

    def __post_init__(self):
        print('In __post_init__() method')

print('Starting')
trade1 = Trade('John', 'Denise', 'IBM', 100)
print(trade1)
print('Done')
```

When this class is run the output is:

```
Starting
In __post_init__() method
Trade(counter_party1='John', counter_party2='Denise', symbol=
'IBM', amount=100)
Done
```

As you can see from this the `__post_init__()` method is run after the instance is created but before the following code can access the trade instance.

5.10 Initialisation Variables

A final option is to mark a field as being something that the `__post_init__()` method should receive. That is a field can effectively be made into a parameter passed to the `__post_init__()` and not be added to the class `__repr__()` and `__eq__()` methods. This is done using the `InitVar` type. This type takes a parameter to indicate the actual type to be held by the attribute but passes the field along to the `__post_init__()` method to handle.

For example:

```
from dataclasses import dataclass, InitVar

@dataclass
class Trade:
    """Class for representing Equity Trades"""
    counter_party1: str
    counter_party2: str
    symbol: str
    amount: int = 0
    status: InitVar[str] = 'Live'

    def __post_init__(self, status):
        print('In __post_init__() method')
        print(f'status = {status}')

print('Starting')
trade1 = Trade('John', 'Denise', 'IBM', 100)
print(trade1)
print('Done')
```

When we run this code we can see that the output from the `__repr__()` method (when it is connected to a string for printing purposes) does not include the status:

```
Starting
In __post_init__() method
status = Live
Trade(counter_party1='John', counter_party2='Denise', symbol=
'IBM', amount=100)
Done
```

5.11 Positional Attributes

Traditional classes have no concept of a natural or default ordering to the attributes they hold. However, for data classes, there is a default ordering to the attributes, this ordering is the order in which they are defined in the data class. Thus for the data class `Book` presented below the ordering is `title` followed by `author`:

```
@dataclass
class Book:
    title: str
    author: str = 'Anonymous'
```

How important is this? This is mainly of interest for *Structural Pattern* matching where the values can be extracted from the data class instance. This is discussed in the next chapter.

5.12 Exercise

The aim of this exercise is to create a data class to represent a Customer for a *fintech* system.

In this simple example, the Customer data class will have a name, an address and an email. All three of these properties will be read-only fields and will hold strings.

You should be able to create a Customer using:

```
customer1 = Customer('John',
                     '10 High Street',
                     'john@gmail.com')
```

Next create an Account data class and subclasses for CurrentAccount and DepositAccount. The Account class should have an account number, a customer and an opening balance.

The `CurrentAccout` class should additionally have an `overdraft_limit` property. This can be set when an instance of a class is created and altered during the lifetime of the object.

The `CurrentAccount` `withdraw()` member function should verify that the balance never goes below the overdraft limit. If it does then the `withdraw()` member function should not reduce the balance instead it should raise a ValueError.

The `DepositAccount` should have an `interest _rate` associated with it which is included when the account is converted to a string.

You will now need to create a customer for each account instance, for example:

```
customer1 = Customer('John',
                     '10 High Street',
                     'john@gmail.com')

acc1 = CurrentAccount("123", customer1, 10.0, -100.00)
print(acc1)
acc1.withdraw(1)
print(acc1)

customer2 = Customer('Denise',
                     '11 Main Street',
                     'denise@gmail.com')
acc2 = DepositAccount("345", customer2, 23.55, 0.5)
print(acc2)

try:
    acc1.withdraw(200)
except ValueError as err:
    print(err)
```

An example of the sort of output this code might generate is given below:

```
CurrentAccount(number='123', customer=Customer(name='John',
address='10 High Street', email='john@gmail.com'), balance=10.0,
overdraft_limit=-100.0)
```

```
CurrentAccount(number='123', customer=Customer(name='John',
address='10 High Street', email='john@gmail.com'), balance=9.0,
overdraft_limit=-100.0)
DepositAccount(number='345', customer=Customer(name='Denise',
address='11 Main Street', email='denise@gmail.com'),
balance=23.55, interest_rate=0.5)
-191.0 Exceeds Overdraft Limit of -100.0
```

Note that the overdraft is presented as a negative number in this example.

Chapter 6
Structural Pattern Matching

6.1 Introduction

Structural pattern matching was introduced into Python in version 3.10. Although the basic form of structural pattern matching looks very much like pattern matching in other languages, there are some specific features that relate to data classes. In this chapter we will look at how data classes can be used with structural pattern matching.

6.2 The Match Statement

As a simple example of using the match statement in Python consider the following code:

```
def get_status_message(status):
    match status:
        case 400:
            return 'Bad request'
        case 404:
            return 'Not found'
        case 418:
            return "I'm a teapot"
        case _:
            return 'Something is wrong'

print(get_status_message(404))
print(get_status_message(401))
```


J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_6

The output from this example is:

```
Not found
Something is wrong
```

This illustrates the basic structure and behaviour of the `match` statement. The value to be matched against is compared with literal values specified in a series of case conditions. If the value in `status` is the same as one of the `case` statements then the associated body of the case statement is executed. Thus if the value of status is `404` then the string ‘Not found’ will be returned. The `case _:` option is a *default* or *wild card* case statement which will run if none of the others match. It is optional but is a common feature. Note only one car condition will run and that will be the first one found that can run.

6.3 Matching Classes with Positional Attributes

Structural pattern matching can also be used with data classes. Such classes provide the concept of positional attributes. These attributes can be extracted from a data class instance using the structural pattern matcher and the values used within a case condition block. For example, given the following simple data class `Click`:

```
from dataclasses import dataclass

@dataclass
class Click:
    x: int
    y: int
```

We can write a function that takes a `Click` as a parameter and then can extract the values for `x` and `y` from the `Click` instance and make them available for use within the body of the `case` condition. This is illustrated below:

```
def handle_click(point):
    match point:
        case Click(x, y):
            print(f'Click x={x}, y={y}')
        case _:
            print('Not a click')
```

In this example, if the value held in `point` is a `Click` instance, then it will match against the first case condition and the positional values `x` and `y` will be extracted from the instance and stored into the local variables x and y which can then be used within the body of the `case` condition—that is in the `print` statement.

If the value held in `point` is *not* a Click instance then the wildcard will trigger and a message that the value is not a click will be printed.

To illustrate this we will use a simple program which will call the handle_click() function twice, once with a Click instance and once with a string:

```
cursor = Click(10, 5)
handle_click(cursor)
handle_click('john')
```

The output from this code snippet is:

```
Click x=10, y=5
Not a click
```

6.4 Matching Against Standard Classes

In most user-defined classes there is no *natural* or default ordering for the attributes held. However, this issue can be overcome by defining a __match_args__ class property which contains a tuple with the names of the attributes and their order.

For example, a plain class Person does not have a natural ordering to its properties; however it does have a name and an age attributes. Thus by adding the __match_args__ class property to the definition, it can now be used with a case condition extraction.

Thus we can define the class Person as:

```
class Person:
    __match_args__ = ("name", "age")

    def __init__(self, pos, age):
        self.name = pos
        self.age = age
```

We can now define a function that takes an instance of the class Person and uses the structural matcher to extract the values for name and age and print them out. If the value provided when the function is called is not a Person then an appropriate message will be printed.

For example:

```
def print_person(person):
    match person:
        case Person(name, age):
            print(f'Person name={name}, age={age}')
        case _:
            print('Not a person')
```

To illustrate the use of the Person class and this function we will use the following three lines that create a Person and call the print_person() function, once with the Person instance and once with an integer 42:

```
p = Person('Adam', 21)
print_person(p)
print_person(42)
```

The output generated by this code is:

```
Person name=Adam, age=21
Not a person
```

6.5 Online Resource

- https://peps.python.org/pep-0636/ Structural Pattern Matching Tutorial.

Chapter 7
Working with pprint

7.1 Introduction

It is common to need to print containers/collections in Python. This is often done to the standard output stream which in most cases is the terminal window from which Python is being run. The problem with this is that lists of lists or dictionaries of key-value pairs can become long and difficult to read if merely printed across the screen. The built-in `pprint` module is designed to help with this. The `pprint` module provides functions that can be used to print out collections with options to control how the data contained within the collection should be displayed. It is a very useful module which is often overlooked when learning Python.

7.2 The pprint Data Printer Module

The `pprint` module is a *built-in* in module; it is provided as part of the Python reference implementation but is not made available by default, and thus it must be imported.

The module supports pretty-printing lists, tuples and dictionaries recursively. Thus it can pretty-print a dictionary containing keys and values where the values are themselves dicts, lists, tuples, etc.

It is a very simple, but useful, module especially when you need to debug applications with nested (and potentially large) data collections.

It provides three functions:

- `pprint()` This function will pretty-print a Python object to an output stream (the default is the standard output).
- `pformat()` This function formats a Python object into a pretty-printed representation.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_7

- `saferepr()` This function will create a 'standard' `__repr__()`-like value, but protect against recursive data structures.

The most commonly used of these functions is the `pprint()` function.

7.3 Basic pprint Usage

To illustrate why we might want to use the `pprint` module, consider the following list of list:

```
data = [[1, 2, 3, 4, 5, 6, 7, 8, 9],
        [1, 2, 3, 4, 5, 6, 7, 8, 9],
        [1, 2, 3, 4, 5, 6, 7, 8, 9]]
print(data)
```

This is not a particularly large list, but it contains three inner lists, each of which is made up of 10 values. In many cases the data structures you might want to create may be much bigger and more complex. However, this simple example will illustrate the idea behind the `pprint` module.

If we run this code we will generate output laid out as shown below:

```
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4, 5, 6, 7, 8, 9], [1, 2, 3, 4,
5, 6, 7, 8, 9]]
```

This is not too difficult to read but it is laid out across the screen (or page) as one long line. On the page this becomes distributed across two lines (or more).

However, there is an inherent structure here, as illustrated by the way that the source code was laid out. Wouldn't it be nice if the print function understood this and laid the data out in a similar way?

This is exactly what the `pprint()` function from the `pprint` module does. To use this function we will need to import the built-in module `pprint`. For example using:

```
import pprint
```

If we now rewrite the earlier code to use the `pprint()` function we can see how the output changes:

```
import pprint

data = [[1, 2, 3, 4, 5, 6, 7, 8, 9],
        [1, 2, 3, 4, 5, 6, 7, 8, 9],
        [1, 2, 3, 4, 5, 6, 7, 8, 9]]
pprint.pprint(data)
```

The output generated by this revised code is now:

```
[[1, 2, 3, 4, 5, 6, 7, 8, 9],
 [1, 2, 3, 4, 5, 6, 7, 8, 9],
 [1, 2, 3, 4, 5, 6, 7, 8, 9]]
```

As you can see the *list of lists* has been printed in a way that makes it much easier to read as the inner lists are each printed on a separate line down the screen (or page in this case). The `pprint()` function determines where to break the lines based on a default screen width of 80 characters.

To illustrate the idea further we can see how the `pprint()` function works with a dictionary (`dict`) in Python.

The following code creates a structure containing a list of dictionaries, and each dictionary has three entries called ‘Name’, ‘Grades’ and ‘Course’. The ‘Grades’ key relates to a value that is itself a list of integer values.

The structure we are using is illustrated below along with the plain old print function:

```
grades = [{'Name': 'John',
           'Grades': [55, 34, 76],
           'Course': 'Csi'},
          {'Name': 'Adam',
           'Grades': [71, 55, 64],
           'Course': 'MedPharm'},
          {'Name': 'Natalia',
           'Grades': [85, 91, 78],
           'Course': 'BioSci'},
          {'Name': 'Denise',
           'Grades': [68, 71, 82],
           'Course': 'Chem'}]
print(grades)
```

Notice the way that we have laid out the code so that it is easy to read for the programmer. However, the output from this is

```
[{'Name': 'John', 'Grades': [55, 34, 76], 'Course': 'Csi'},
{'Name': 'Adam', 'Grades': [71, 55, 64], 'Course': 'MedPharm'},
{'Name': 'Natalia', 'Grades': [85, 91, 78], 'Course': 'BioSci'},
{'Name': 'Denise', 'Grades': [68, 71, 82], 'Course': 'Chem'}]
```

Again it has laid the output out in a single line which has been broken up across the screen.

Depending on the size of the terminal or the command prompt being used this may or may not be that readable.

If we now use the `pprint()` function on the same structure:

```
pprint.pprint(grades)
```

The output is now:

```
[{'Course': 'Csi', 'Grades': [55, 34, 76], 'Name': 'John'},
 {'Course': 'MedPharm', 'Grades': [71, 55, 64], 'Name': 'Adam'},
 {'Course': 'BioSci', 'Grades': [85, 91, 78], 'Name': 'Natalia'},
 {'Course': 'Chem', 'Grades': [68, 71, 82], 'Name': 'Denise'}]
```

Notice that again the `pprint()` function tries to lay the data structures out in a meaningful way based not the contents of the lists and dictionary and the default line length (which is 80 characters).

7.4 Changing the Width

The `pprint()` function has numerous parameters available, all of which have default values provided and all of which can be used to control how `pprint()` generates its output. One of these parameters is the `width` parameter. This has a default value of `80` and has been used above to determine where and how to break each line in the output. We will use the named parameter passing style with the `width` parameter as there are several parameters available, and we need to indicate which the width value relates to. The syntax is thus

```
pprint.pprint(<data>, width=<an int>)
```

We can rerun the above grades example using different integer width values. For example, if we override the default value of 80 with 60 as shown below:

```
pprint.pprint(grades, width=40)
```

The output generated from this is changed to:

```
[{'Course': 'Csi',
  'Grades': [55, 34, 76],
  'Name': 'John'},
 {'Course': 'MedPharm',
  'Grades': [71, 55, 64],
  'Name': 'Adam'},
 {'Course': 'BioSci',
  'Grades': [85, 91, 78],
  'Name': 'Natalia'},
 {'Course': 'Chem',
  'Grades': [68, 71, 82],
  'Name': 'Denise'}]
```

As you can see from this the inner dictionary is now laid out down the page as the maximum width of 60 characters is no longer enough to display the contents.

To illustrate this further we can change the width down to just 20 characters:

```
pprint.pprint(grades, width=20)
```

Now the output is further compressed across the page:

```
[{'Course': 'Csi',
  'Grades': [55,
             34,
             76],
  'Name': 'John'},
 {'Course': 'MedPharm',
  'Grades': [71,
             55,
             64],
  'Name': 'Adam'},
 {'Course': 'BioSci',
  'Grades': [85,
             91,
             78],
  'Name': 'Natalia'},
 {'Course': 'Chem',
  'Grades': [68,
             71,
             82],
  'Name': 'Denise'}]
```

Now the inner list is also displayed down the screen.

7.5 Changing the Depth

By default the `pprint()` function prints out all nested collections down to whatever depth exists. That is, if a list contains a dictionary that contains a list it will print all 3 collections (as illustrated for the grades examples above). However, this can be overridden. There is another parameter to the `pprint()` function called `depth`. By default this parameter is set to the value `None` which indicates that all nested sequences should be printed. However, you can set this to an integer to indicate the depth of nesting to be printed.

In the grades example we could say that we have a depth of nesting of 3, as there is a list containing a dictionary containing a list. We could thus set to depth to 2 to see what happens:

```
pprint.pprint(grades, depth=2)
```

When this code is run the output generated is:

```
[{'Course': 'Csi', 'Grades': [...], 'Name': 'John'},
 {'Course': 'MedPharm', 'Grades': [...], 'Name': 'Adam'},
 {'Course': 'BioSci', 'Grades': [...], 'Name': 'Natalia'},
 {'Course': 'Chem', 'Grades': [...], 'Name': 'Denise'}]
```

As you can see the inner most list (the value element associated with the key 'Grades') is shown as a list with three dots inside it. This illustrates that there is a data present here but we are not displaying the contents of this list. This is useful if you don't need to know what the values are just that there is a list here, and the output takes up less space on the screen, for example during debugging.

It is of course possible to mix the `width` and the `depth` parameters in the same call to control how the data is laid out across the screen, for example:

```
pprint.pprint(grades, depth=2, width=40)
```

This will print out the grades to a depth of 2 but within only 40 characters horizontally across the screen, for example:

```
[{'Course': 'Csi',
  'Grades': [...],
  'Name': 'John'},
 {'Course': 'MedPharm',
  'Grades': [...],
  'Name': 'Adam'},
 {'Course': 'BioSci',
  'Grades': [...],
  'Name': 'Natalia'},
 {'Course': 'Chem',
  'Grades': [...],
  'Name': 'Denise'}]
```

7.6 Managing the Indentation Level

A third parameter for the `pprint()` function is the `indent` parameter. This parameter has a default value of 1. The indent value determines by how much a nested value is indented in the output relative to the containing collection when printed on a different line.

For example to change the indentation level from 1 to 6 with the grades example, we could write:

```
pprint.pprint(grades, indent=6)
```

The output from this is now:

```
[     {'Course': 'Csi', 'Grades': [55, 34, 76], 'Name': 'John'},
      {'Course': 'MedPharm', 'Grades': [71, 55, 64], 'Name': 'Adam'},
      {'Course': 'BioSci', 'Grades': [85, 91, 78], 'Name': 'Natalia'},
      {'Course': 'Chem', 'Grades': [68, 71, 82], 'Name': 'Denise'}]
```

As you can see there is a big gap between the outer most square brackets and the inner dictionary.

It is of course possible to use the `indent` parameter with both the `width` and/or the `depth` parameters. For example:

```
pprint.pprint(grades, depth=2, width=40, indent=6)
```

This line will generate output that is displayed only to depth 2 and is laid out across 40 characters on the screen with each subsequent line indented by 6 spaces:

```
[   {   'Course': 'Csi',
        'Grades': [...],
        'Name': 'John'},
    {   'Course': 'MedPharm',
        'Grades': [...],
        'Name': 'Adam'},
    {   'Course': 'BioSci',
        'Grades': [...],
        'Name': 'Natalia'},
    {   'Course': 'Chem',
        'Grades': [...],
        'Name': 'Denise'}]
```

7.7 Reducing Line Breaks Using Compact

One problem with using the `width` parameter is that it looks at the contents of a collection and tries to determine whether it can print the whole contents out within the specified `width`. If not then it defaults to printing each value on a separate line. This was fine for the grades example as it only held three values, but what about a list containing say just a collection of integers. For example, the following code creates a list of 25 integers:

```
marks_range = list(range(0, 25))
print(marks_range)
```

The use of the basic `print()` function prints out the integers across the screen: The output from this is:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24]
```

which is probably ok in most situations. However, if we do decide to use the `pprint()` function and specify a width of `20` (which is 5 less than the number of values in the list) something odd happens:

```
pprint.pprint(marks_range, width=20)
```

This produces an output where each integer is printed down the screen:

```
[0,
 1,
 2,
 3,
 4,
..
```

Due to space we are not including all 25 values but hopefully you get the idea. Note that the two dots here are added by us to indicate that there is more data that would be displayed if you ran this program yourself.

However, Python 3.4 added another parameter to the `pprint()` function, `compact` (which is set to `False`) by default. If compact is set to `True`, then elements that fit into the `width` are printed on a single line. It is therefore considered best practice to use `compact=True` for lists with many elements.

For example:

```
pprint.pprint(marks_range, width=20, compact=True)
```

generates the following output:

```
[0, 1, 2, 3, 4, 5,
 6, 7, 8, 9, 10, 11,
 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21,
 22, 23, 24]
```

which is probably more desirable than the default behaviour.

7.8 The pformat Function

The `pprint.pformat()` function is used to convert a container such as a list, dict or tuple into a string. Unlike the `str()` conversion function, the `pformat()` function uses the same layout options and behaviour as the `pprint.pprint()` function. Thus a string can be generated that is based on the same `width`, `depth` and `indent` options as `pprint()` can be generated.

To illustrate this, let us again use the grade structure and convert this into a string using the basic `str()` function:

```
grades = [{'Name': 'John',
           'Grades': [55, 34, 76],
           'Course': 'Csi'},
          {'Name': 'Adam',
           'Grades': [71, 55, 64],
           'Course': 'MedPharm'},
          {'Name': 'Natalia',
           'Grades': [85, 91, 78],
```

```
            'Course': 'BioSci'},
           {'Name': 'Denise',
            'Grades': [68, 71, 82],
            'Course': 'Chem'}]
data_str = str(grades)
print(data_str)
```

The output from this is:

```
[{'Name':  'John',  'Grades':  [55,  34,  76],  'Course':  'Csi'},
{'Name': 'Adam', 'Grades': [71, 55, 64], 'Course': 'MedPharm'},
{'Name': 'Natalia', 'Grades': [85, 91, 78], 'Course': 'BioSci'},
{'Name': 'Denise', 'Grades': [68, 71, 82], 'Course': 'Chem'}]
```

which is a single string representing the contents of the grades structure.

However, using the `pprint.pformat()` function we can generate a string which is formatted as it would if we had used the `pprint()` function, for example:

```
data_str = pprint.pformat(grades)
print(data_str)
```

The output from this is:

```
[{'Course': 'Csi', 'Grades': [55, 34, 76], 'Name': 'John'},
 {'Course': 'MedPharm', 'Grades': [71, 55, 64], 'Name': 'Adam'},
 {'Course': 'BioSci', 'Grades': [85, 91, 78], 'Name': 'Natalia'},
 {'Course': 'Chem', 'Grades': [68, 71, 82], 'Name': 'Denise'}]
```

In addition we can use the same options as are available on `pprint()` to control the `depth`, `width` and indentation, for example:

```
import pprint
data_str = pprint.pformat(grades, depth=2, width=40, indent=2)
print(data_str)
```

which produces the following output:

```
[ { 'Course': 'Csi',
    'Grades': [...],
    'Name': 'John'},
  { 'Course': 'MedPharm',
    'Grades': [...],
    'Name': 'Adam'},
  { 'Course': 'BioSci',
    'Grades': [...],
    'Name': 'Natalia'},
  { 'Course': 'Chem',
    'Grades': [...],
    'Name': 'Denise'}]
```

Note that this is a single string representing the nested grades data structures.

7.9 The saferepr() Function

In some situations a data structure might be recursive; that is a structure may at some point refer to itself resulting in an endless loop. An example of this sort of structure is illustrated below where `list1` contains three strings and a reference to `list3`. However, `list3` is made up of `list1` and `list2`:

```
list1 = ['a', 'b', 'c']
list2 = ['d', 'e', 'f']
list3 = [list1, list2]
list1.append(list3)
print(list1)
```

Python handles this normally by just using the three dots to indicate something—however it is not necessarily obvious that this indicates a recursive structure. For example, the output of the above is:

```
['a', 'b', 'c', [[...], ['d', 'e', 'f']]]
```

However, the `saferepr()` function returns a string which will include a '<Recursion on list with >' element when data becomes recursive, for example:

```
print(pprint.saferepr(list1))
```

This generates:

```
['a', 'b', 'c', [<Recursion on list with id=4349488128>, ['d', 'e',
'f']]]
```

To help determine if you want to use `saferepr()` there is a `isrecusrive()` function in the `pprint` module that will return `True` if a structure is recursive. For example:

```
print(f'pprint.isrecursive(grades):
{pprint.isrecursive(grades)}')
print(f'pprint.isrecursive(list1): {pprint.isrecursive(list1)}')
```

The output from this is:

```
pprint.isrecursive(grades): False
pprint.isrecursive(list1): True
```

7.10 Using the PrettyPrinter Class

If you are going to use a particular option on the `pprint()` function all the time you can create your own custom `PrettyPrint` instance with the parameter set to your default. This means that you only need to specify the new settings once when you are instantiating the `PrettyPrint` object and can use these settings throughout your program.

```
import pprint

ppr = pprint.PrettyPrinter(depth=2, indent=2)
ppr.pprint(grades)
```

The output from this is:

```
[ {'Course': 'Csi', 'Grades': [...], 'Name': 'John'},
  {'Course': 'MedPharm', 'Grades': [...], 'Name': 'Adam'},
  {'Course': 'BioSci', 'Grades': [...], 'Name': 'Natalia'},
  {'Course': 'Chem', 'Grades': [...], 'Name': 'Denise'}]
```

Now each time you use the per object you will have the depth set to 2 and the indent also set 2 but the width is the default 80, etc.

7.11 Online Resource

- https://docs.python.org/3/library/pprint.html Documentation on the `pprint` module.

Chapter 8
Shallow v Deep Copy

8.1 Introduction

When a container or collection type is copied there is an issue of what should be copied about the elements it contains. By default if a list of lists is copied, then the copy contains the addresses of the sublists; thus the inner lists are shared. This is efficient as it means that complex or deep structures aren't duplicated but can cause problems if programmers do not realise that a sublist is shared between data structures and start to modify the sublists. In this chapter we will look at the `copy` module that provides for both shallow and deep copy options.

8.2 Copying a List of Lists

There are several ways in which a list can be copied. The two most common approaches are to use the `copy()` method on a list or to the use *copy slice* syntax.

The `copy()` method is invoked using the dot notation and generates a *shallow copy* of the data held in a container; for example when using a list of lists we might write:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
List3 = [list1, list2]

# Copy using the copy method
list4 = list3.copy()
for item in list4:
   print(f'item: {item}')
```

In this example we have a `list3` which contains references to two sublists `list1` and `list2`. These sublists contain integers. We then create a copy of `list3`

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_8

and store that in `list4`. We then loop through `list4` printing out each item in turn. The output from this is:

```
item: [1, 2, 3]
item: [4, 5, 6]
```

We thus end up with two lists (`list3` and `list4`) which contain sublists with integers in them.

We could also have used the slice syntax to create a copy of `list3` as shown below:

```
# Copy using slice syntax
list5 = list3[:]
for item in list5:
    print(f'item: {item}')
```

Again the output from this is:

```
item: [1, 2, 3]
item: [4, 5, 6]
```

8.3 The Problem with Copying

So what is the problem with copy? The issue comes in the way that Python copies the contents of the top-level list `list3`. By default it does a *shallow copy*. That is, it copies the references to the sublists held in `list3` into `list4` and `list5`. This is efficient in that we don't create lots of additional lists but it is dangerous in that all three top-level lists, `list3`, `list4` and `list5` reference the same instances of the lists originally held in `list1` and `list2`. Thus if a sublist is modified via `list5`, it is also modified for `list3` and `list4`.

This is illustrated by the ids of the sublists held in each of the top-level lists:

```
list1_id = id(list1)
print(f'list1_id: {list1_id}')

list2_id = id(list2)
print(f'list2_id: {list2_id}')

print('=' * 25)
for sublist in list3:
    print(f'sublist id: {id(sublist)}')
print('-' * 25)

for sublist in list4:
    print(f'sublist id: {id(sublist)}')
print('-' * 25)

list5 = list3[:]
for sublist in list5:
```

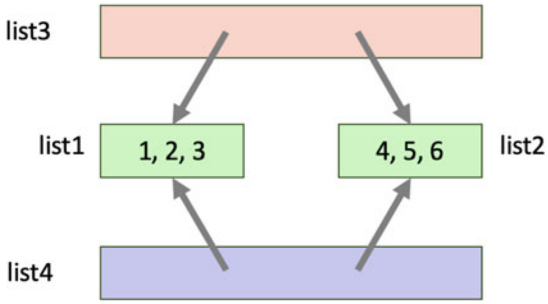
```
    print(f'sublist id: {id(sublist)}')
print('=' * 25)
```

When we run this code the output is:

```
list1_id: 4521910080
list2_id: 4522960512
=========================
sublist id: 4341751808
sublist id: 4342801984
-------------------------
sublist id: 4341751808
sublist id: 4342801984
-------------------------
sublist id: 4341751808
sublist id: 4342801984
=========================
```

This indicates that the ids of the sublists are the same across all three top-level lists. As every object in Python has a unique id they must be the same instances of the inner lists.

Pictorially what happens when we copy `list3` to `list4` is shown below:



As we can see from the diagram both `list3` and `list4` are referencing the same instances of `list1` and `list2`.

The danger occurs when we append a value to a sublist via, for example, `list5`. This is shown below:

```
list5[0].append(100)
print(list5)
print(list4)
print(list3)
```

The output from this is:

```
[[1, 2, 3, 100], [4, 5, 6]]
[[1, 2, 3, 100], [4, 5, 6]]
[[1, 2, 3, 100], [4, 5, 6]]
```

Here you can see that the integer `100` has apparently been added to `list5`, `list4` and `list3`. Actually it was added to the shared sublist originally represented by `list1`.

8.4 The Copy Module to the Rescue

The copy module provides two functions, the `copy()` function and the `deepcopy()` function. These are described below:

- `copy.copy(x)` returns a *shallow* copy of x.
- `copy.deepcopy(x)` returns a *deep* copy of x.

The difference between shallow and deep copying is only relevant for container or collection like objects such as lists, tuples, dictionaries or class instances. In essence the difference is that:

- A *shallow* copy constructs a new top-level object (such as a list) and then (to the extent possible) inserts references (that is the addresses) of the contained objects into the copy.
- A *deep* copy constructs a new top-level object (such as a list) and then, recursively, makes copies of the contained objects which are added to the new top-level container. Thus copies are made down to any depth within the structure being copied.

At this point you might be wondering why all copies are not performed as deep copies as they are safer. There are several issues to consider with deep copies, including:

- *Recursive objects.* These are objects which at some point refer back to another compound object within the same data structure. This may happen directly or indirectly because of some deep network within the structure. These can cause recursive loops which mean that a deep copy can fail.
- *Intentionally shared data.* Because deep copy copies everything it may copy too much, such as data which is intended to be shared between copies.
- The *amount of data being copied may be prohibitively expensive* both in terms of the memory being used but also in terms of the time taken to make a copy.

For these reasons most programming languages default to a shallow copy mechanism and often provide a deep copy option—which is exactly what Python does.

To try to alleviate some of the issues associated with the first two points above, the `deepcopy()` function in Python has a couple of enhancement strategies; these are:

- Keeping a *memo dictionary of objects* already copied during the current copying pass. The term memo is short of memoization and is a form of caching, and it is typically used where the cache concept is specific to a particular task or function.
- Letting *user-defined classes override* the copying operation or the set of components copied.

This second point is worthy of some additional explanation. Classes in Python can define special methods that typically start with a __ in their name. Copying is no different to other operations such as addition or subtract in this way. In order for a class to define its own copy implementation, it can define special methods `__copy__()` and `__deepcopy__()`. The former is called to implement the shallow copy operation; no additional arguments are passed. The latter is called to implement the deep copy operation; it is passed one argument, the memo dictionary. If the `__deepcopy__()` implementation needs to make a deep copy of a component, it should call the `copy.deepcopy()` function with the component as first argument and the memo dictionary as second argument.

8.5 Using the deepcopy() Function

We can now update our earlier examples to use `copy.deepcopy()` rather than just the `copy()` method or the `[:]` copy slice syntax. We now have to import the `copy` module and then call the `copy.deepcopy()` function passing in the list to be copied (in this case `list3`). The new list is then stored in `list4`.

In the following code we use the `id()` function to again check the unique id for each sublist within lists and list4:

```
import copy

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [list1, list2]

list4 = copy.deepcopy(list3)

for sublist in list3:
    print(f'sublist id: {id(sublist)}')
print('-' * 25)

for sublist in list4:
    print(f'sublist id: {id(sublist)}')
print('-' * 25)
```

The output from this code is given below:

```
sublist id: 4539736128
sublist id: 4539735488
--------------------------
sublist id: 4539737536
sublist id: 4539474560
--------------------------
```

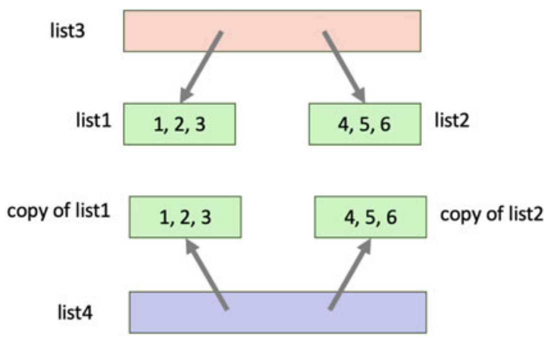
This time the ids for the sublists in `list3` are different to the ids of the sublists in `list4`. They no longer hold the same references, but they instead hold copies of the sublists. This means that if we now add the integer 100 to `list4` it will have no effect on the contents of `list3`, for example:

```
list4[0].append(100)
print(list4)
print(list3)
```

which produces:

```
[[1, 2, 3, 100], [4, 5, 6]]
[[1, 2, 3], [4, 5, 6]]
```

As we can see list3 has not been modified.
Thus the effect of using `deepcopy()` pictorially on list3 and list 4 is:



8.6 Online Resource

- https://docs.python.org/3/library/copy.html Shallow and deep copy operations.

Chapter 9
The __init__ Versus __new__ and __call

9.1 Introduction

Python classes have many special methods that are of the form __<name of method>__. These methods are sometimes referred to as magic methods or double underscore methods (also dunder methods which is short for double underscore methods). Many of these support operators such as __eq__ (for the equality operator==) or __lt__ (for the less than operator<) or for specification functionality such as __len__ for determining the length of an object or __str__ and __repr__ for converting objects into string formats. However, two special methods that often get confused are __new__ and __init__. These two special methods are associated with instance creation but have different roles and are run at different times in the object creation process. For many developers they only ever use __init__ and may not even realise that there is a __new__ method available. This chapter looks at these two methods, their roles and how they are defined. The chapter concludes by briefly introducing the __call__ method.

9.2 The __new__ and __init__ Methods

In Python, both the __new__ and __init__ methods are special methods used in class definitions. They serve different purposes and are invoked at different stages of an object's lifecycle.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_9

The `__new__` method is responsible for creating and returning the instance of the object, while the `__init__()` method initialises the object's attributes and performs any necessary setup. When you create an object using the class constructor, both methods are automatically invoked in the following sequence:

```
<Object Instantiated> -> __new__() -> __init()
```

The `__new__()` method is called first, which creates the object instance and returns it. It is a static method that takes the class as its first argument, followed by any additional arguments passed to the class constructor. The `__new__` method is responsible for creating the object and can return an instance of the class or an instance of a different class.

After the object has been created by the `__new__() method`, the `__init__()` method is called with the newly created object instance as its first argument, followed by any additional arguments passed to the class constructor. The __init__ method is expected to initialise the state of the object, that is to set up the attributes held by the object and their initial values. It can also invoke any additional set up or initialisation behaviour as required.

9.3 The __new__ Method

The `__new__()` method is the object creation method. Its purpose is to create the instance or object of the class. The default behaviour of the `__new__` method is defined in the class object from which all class in Python inherit. As such all classes can create instances of a class in the same way. However, subclasses can override the `__new__` method when and if required.

The `__new__` method is a *static* method, and it is therefore part of the class itself and not part of a particular object or instance. However, it is a special case which means that you do not need to mark it explicitly as a static method; it automatically is a static method.

The `__new__` method is typically used in situations where you need more control over the object creation process, such as when dealing with immutable objects or implementing singletons.

Typical implementations of the `__new__` method create a new instance of the class by invoking the superclass's `__new__()` method using `super().__new__(cls[,...])` with appropriate arguments and then modifying the newly created instance as necessary before returning it.

The definition of the `__new__` method is:

```
def class (class, *args, **kwargs)
```

The parameters are:

- `class` this is the class of the new object that you want to create.

- The `*args` and `**kwargs` parameters must match the parameters of the `__init__()` of the class. However, the `__new__()` method is not expected to use them.

  Here's a basic example of how the `__new__` method can be used:

```
class MyClass:
    def __new__(cls, *args, **kwargs):
        print('Entering __new__')
        # Custom object creation logic
        instance = super().__new__(cls)
        print('New instance created')
        # Additional initialisation of the instance if needed
        return instance

print('Starting')
obj = MyClass()
print('Done')
```

  The output generated from running this code is:

```
Starting
Entering __new__
New instance created
Done
```

As you can see the output is generated when the instance of `MyClass` is created.

An important point to note is that it is quite rare to need to implement your own `__new__` method.

The default `__new__` method in the object class is usually perfectly fine for most situations. If you do not define your own `__new__` method then the default is inherited from the parent class (e.g. `object`). This default simply creates and returns a new instance of the class.

9.4 When to Use the __new__ Method

The `__new__` method is usually used in specific situations where there is a need for fine-grained control over object creation. Here are a few scenarios where the `__new__` method might be used:

- **Implementing Immutable Objects**: If we want to create an object that is immutable (i.e. its state cannot be changed after creation), we can override the __new__ method to ensure that no modifications can be made to the object's attributes.
- **Singleton Pattern**: The `__new__` method can be used to implement the singleton design pattern, where only one instance of a class can exist. By controlling the creation process in `__new__`, you can ensure that subsequent requests for the object return the same instance. That is, the class

can record the single instance to be used the first time a new instance is created. Following this, each subsequent request for a new instance will return the original instance.

- **Customising Object Creation**: If you need to perform additional operations during object creation, such as validating arguments or initialising internal state, you can override the `__new__` method to incorporate your custom logic before the object is initialised by `__init__`.
- **Subclassing an Immutable Base Class**: When subclassing an immutable class, like `tuple` or `str`, you need to override the `__new__` method instead of `__init__` because the base class does not allow modification after creation.

9.5 Using __new__ to Create a Singleton Object

The following code creates a simple implementation of a singleton class in Python. This class overrides the __new__ method such that the first instance of the class is rec order in the class side attribute instance. After this further requests to return a new instance just return the previously created instance:

```
class Singleton:

  instance = None

  def __new__(cls, *args, **kwargs):
    print('Entering __new__')
    if Singleton.instance is None:
       print('Creating instance')
       # create the single instance
       Singleton.instance = super().__new__(cls)

    print('Returning instance')
    return Singleton.instance

print('Starting')
s1 = Singleton()
print('-' * 25)
s2 = Singleton()
print('-' * 25)
s3 = Singleton()
print('-' * 25)

print(id(s1))
print(id(s2))
print(id(s3))
print('Done')
```

If we run this code the output from this is:

```
Starting
Entering __new__
Creating instance
```

```
Returning instance
-------------------------
Entering __new__
Returning instance
-------------------------
Entering __new__
Returning instance
-------------------------
4321848656
4321848656
4321848656
Done
```

From this we can see that only once does the code print out ‘Creating instance’ and that the ids for each of the objects being referenced by `s1`, `s2` and `s3` are the same. Thus the same instances is being used each time a request is made to make a new object of type Singleton.

9.6 The __init__ Method

The `__init__` method is called after the object has been created by the `__new__` method. It initialises the object’s attributes and performs any necessary setup. The `__init__` method takes the newly created object instance as its first argument, followed by any additional arguments passed to the class constructor. It doesn’t return anything and is primarily used for initialisation purposes.

The following code provides a simple example of creating a class `Person` in which the `name` and `age` attributes are initialised by values passed into the `__init__` method:

```
class Person:

    def __init__(self, name, age):
        print('In __init__')
        self.name = name
        self.age = age

    def __repr__(self):
        return f'Person({self.name}), {self.age})'

print('Starting')
p1 = Person('John', 21)
print(p1)
p2 = Person('Denise', 18)
print(p2)
print('Done')
```

The result of running this code is:

```
Starting
In __init__
```

```
Person(John), 21)
In __init__
Person(Denise), 18)
Done
```

As can be seen from this the `__init__` method runs when each of the instances of the class `Person` is created.

9.7 Can __new__ and __init__ Be Used Together?

The short answer here is yes they can—they serve different purposes as has been indicated above.

Thus an individual class may well have need for both a `__new__` and an `__init__` methods. For example, the `__new__` method may be used to limit the number of instances of a class that are created whereas the `__init__` method may be used to initialise the state of any instances that are actually created.

The following class has both a `__new__` and an `__init__` methods.

```python
class MyClass:
    def __new__(cls, *args, **kwargs):
        print('Entering __new__')
        # Custom object creation logic
        instance = super().__new__(cls)
        print('New instance created')
        # Additional initialisation of the instance if needed
        return instance

    def __init__(self):
        print('In __init__')

print('Starting')
obj = MyClass()
print('Done')
```

When we run this code the output generated will be:

```
Starting
Entering __new__
New instance created
In __init__
Done
```

As this shows the __new__ method runs before the __init__ method.

9.8 The __call__ Method

A final special method that we should consider is the `__call__` method. This special method allows an object to be called as if it were a function. When an object defines the `__call__` method, it can be invoked using parentheses `()` as if it were a function call.

A simple example is given below:

```
class CallableClass:
    def __call__(self, *args, **kwargs):
      print("The object was called!")

print('Start')
obj = CallableClass()
obj()
print('Done')
```

The output generated for this code is:

```
Start
The object was called!
Done
```

We can therefore make any class into a callable or executable thing. The inverse of this is that not all objects in Python can be called. Only objects that define the `__call__()` method can be invoked as functions.

Another point to note is that when an object is instantiated in Python using the standard notation, it is this method that makes the class creation behaviour an executable thing; that is the `__call__()` method is invoked when creating a new instance of a class. Thus the following invokes the call method to create a new Person instance:

```
P1 = Person('John')
```

9.9 Summary

To summarise, the `__new__` method is responsible for creating the instance of the object, while the `__init__` method initialises the object's attributes and performs setup operations. The `__new__` method is rarely used in everyday Python programming, except in cases where you need more control over object creation. On the other hand, the `__init__` method is commonly used for typical initialisation tasks.

Chapter 10
Python Metaclasses and Meta Programming

10.1 Introduction

This chapter looks at metaprogramming and metaclasses in Python. Metaprogramming relates to the idea that a Python program can generate or modify code dynamically at runtime. This is a very powerful (although potentially dangerous) feature in Python which many statically compiled languages such as C++ do not provide. This chapter introduces metaprogramming before discussing three ways in which metaprogramming can be achieved in Python; using decorators, metaclasses and dynamic code execution.

10.2 Metaprogramming

Metaprogramming is exactly that, it is software that can generate program code (or indeed modify existing code). Hence it is 'meta' programming—that is programs that generate programs. In Python such metaprogramming happens dynamically at runtime. There are several features in the language that can support metaprogramming including decorators, metaclasses and dynamic code execution using exec() and eval():

- **Decorators**: Decorators are functions that modify the behaviour of other functions. They are denoted by the "@" symbol and can be applied to functions, classes, or methods. Decorators allow you to add functionality to existing code without modifying it directly.
- **Metaclasses**: Metaclasses are classes that define the behaviour of other classes. By defining a metaclass, you can customise the creation and behaviour of class objects. Metaclasses are often used to implement frameworks and libraries.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_10

- **Dynamic code execution**: Python provides the `exec` and `eval` functions, which allow you to execute dynamically generated code. These functions take a string containing Python code and execute it at runtime. This enables dynamic code generation and execution based on specific conditions or user input.

Thus the key aspect to metaprogramming is that the developer must create code that will generate or manipulate other code. This may sound non-sensical, but it does allow developers to create very flexible and powerful code abstractions that would not be possible using traditional coding techniques.

A word of caution is appropriate at this point. Metaprogramming is very powerful; however, it can also have significant drawbacks. The most notable issue with metaprogramming is that it can make code much harder to understand, debug and maintain as there is a level of indirection and additional complexity involved. Metaprogramming can be very useful for generic, reusable frameworks but should only be used when they are appropriate and not as a general technique.

It is therefore useful to consider the situations in which metaprogramming can be useful. Some of these include:

- **Code Generation**: Metaprogramming allows a developer to generate code dynamically based on specific conditions, configurations or input. This can be helpful when an application or library needs to automate repetitive code generation tasks or customise code behaviour at runtime.
- **Frameworks and Libraries**: Metaprogramming is often used in frameworks and libraries to provide flexible and extensible APIs. By using metaprogramming techniques, frameworks can automatically handle common tasks, enforce conventions, and provide abstractions that simplify development.
- **Dynamic Configuration**: Metaprogramming allows the developer to load and modify configuration files or data structures at runtime. This can be helpful when it is necessary to dynamically adjust the behaviour of an application based on different environments, user preferences, or external data.
- **Domain-Specific Languages** (DSLs): Metaprogramming can be used to create domain-specific languages that provide a higher level of abstraction tailored to specific problem domains. DSLs enable solutions to be expressed in a more concise and intuitive manner, improving productivity and code readability. For example a DSL might be created to represent submitting jobs to some task processing system, or to describe a set of domain concepts such as trades in a financial trading system, etc.
- **Aspect-Oriented Programming** (AOP): Metaprogramming can be applied to implement aspect-oriented programming techniques. AOP allows you to modularise cross-cutting concerns, such as logging, error handling or performance monitoring, by separating them from the core logic of your application. AOP is widely used in several common frameworks although it is often hidden from the end user, for example turning logging on within a framework may appear to be a simple 'on' function but behind the scenes metaprogramming may be used to enable cross-framework logging.

The overall guiding principle regarding whether to use a metaprogramming technique or not should be 'does its use improve the flexibility, comprehensibility or readability or maintainability of your code without significantly comprising that code?'. Thus an appropriate balance between these aspects should be borne in mind.

10.3 Decorators as a Form of Metaprogramming

Decorators in Python are a language feature that can be used to implement a metaprogramming approach.

Decorators are higher-order functions that take a function as input and can return a modified or wrapped version of that function as their result. They can thus replace, modify or wrap a function there by dynamically modifying the behaviour of a function.

Some common uses for metaprogramming style decorators include:

- **Function Wrapping**: This is the classic use of decorators. A decorator can wrap a function with additional behaviour. For example, you can create a decorator to log function calls, measure execution time, or handle exceptions.
- **Access Control**: Decorators can be used to control access to the associated wrapped function or method. This access can be used to check for security constraints such as is the current user valid and do they have access rights to the function or method.
- **Caching and Memoization**: Decorators can be used to create a form of caching often referred to as *memoization.* This approach allows a function to cache the results generated for particular parameter values. The cache holds information about the parameters and the results previously generated. This means that when the function is invoked a look up is performed to see if the function has already cached the results. This is particularly useful for computations that are expensive to perform as the computation is only required to be performed once. However, it also relies on the function or method not using mutable external values. For memoization to work a function or method must rely sole on the parameter values passed into it to generate the result or only reference immutable external values. This is referred to as Referential Transparency (aka RT).
- **Input valuation and Sanitisation**: Decorators can also be used to validate and sanitise input values. By defining reusable decorators that can do such input valuation/sanitisation you can provide common integrity support across functions and methods.
- **Frameworks and Libraries**: Decorators are widely used within many frameworks and library APIs to link functions, classes and methods into those frameworks, etc. Flask discussed later in this book is an example of such a framework.

One specific benefit of using decorators to perform the above metaprogramming use cases is that it enables a separation of concerns as well as code reuse. For example, the logic associated with a Flask RESTFul service is encapsulated to a large extent within a set of decorators, while the behaviour required when a service is invoked is encapsulated within a function.

Here is an example that demonstrates how decorators can be used for metaprogramming in Python:

```
def uppercase_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        if isinstance(result, str):
            return result.upper()
        return result
return wrapper

@uppercase_decorator
def greeter(name):
    return f"Hello, {name}!"

print(greeter('Hello Denise'))
```

The output generated by this code is:

```
HELLO, DENISE!
```

In this example, we define a decorator called `uppercase_decorator`. This decorator takes a function `func` as an argument and returns a new function `wrapper`. The `wrapper` function wraps the original function `func` and modifies its behaviour. That is when wrapper is called it calls the original function and then adds some behaviour which may modify the result of the function.

In this case, the `uppercase_decorator` converts the return value of the decorated function into uppercase if it is a string. The wrapper function receives the arguments passed to the decorated function, calls the original function `func` and stores the result in the result variable.

Then, it checks if the result is a string using the `isinstance` function. If it is, the wrapper function converts the result to uppercase using the `upper()` method. Finally, the modified result is returned.

The `@uppercase_decorator` syntax is used to apply the decorator to the `greeter()` function. Now, whenever the `greeter()` function is called, the `uppercase_decorator` is automatically invoked, modifying the output by converting it to uppercase if it's a string.

When we call `greeter('Denise')`, the output is "HELLO, DENISE!", demonstrating how the decorator modifies the behaviour of the original function.

Of course the `@uppercase_decorator` could be applied to any function or method and would convert all strings to upper case; it is not tied to the `greeter()` function.

10.4 Metaclasses for Metaprogramming

Metaclasses in Python provide a way to define the behaviour of classes themselves. A metaclass is the class of a class, meaning it is responsible for creating and defining the behaviour of class objects. Metaclasses allow you to modify class creation and control how classes behave at runtime. Metaclasses are used with Abstract Base Classes to provide the basic behaviour of an ABC.

Metaclasses are classes for classes, and there are several points worth noting about metaclasses including:

- **Metaclasses are Templates for Classes**: By template here we mean that a metaclass acts as a pattern or blueprint for creating a class. It defines the structure, behaviour and attributes of the classes that will be created using the metaclass.
- **Class Creation and Initialisation is Handled by Metaclasses**: If a class has a metaclass defined for it, then when an instance of a class is created, the metaclass will be used to create that instance (object). Python will therefore invoke the metaclasses `__new__()` and `__init__()` methods. Thus it's the metaclass that decides how a new object will be created and its state set up.
- **Metaclasses can modify the behaviour of a class**: A metaclass can intercept and modify class attributes, methods and their behaviour. Metaclasses can therefore add, modify or remove attributes, override methods, modify the effect of a method or add new methods to a class.
- **Metaclasses can add class-level behaviour**: Class-level methods are methods that apply to the class itself rather than instances of the class. This includes defining custom class methods, class properties or class-level methods.
- **Metaclasses can control inheritance**: Metaclasses can control how classes inherit from parent/super classes. They can modify the inheritance order, enforce specific constraints on inheritance or dynamically generate parent classes based on specific conditions.

Although metaclasses provide a very powerful way to create abstractions, libraries, frameworks and the like, care should be taken with them. As the saying goes, with great power comes great responsibility! Best practice suggests that metaclasses should be used with caution and only where their use has clear benefits in terms of readability, reusability, maintenance and further extensions.

10.4.1 Singleton Metaclass

As an alternative to the singleton pattern implementation presented in the last chapter, here is a metaclass version. This version allows the metaclass `SingletonMetaclass` to be used with any class that we wish to convert into a singleton pattern. This provides significant benefits in terms of reusability, that is we only need to define this behaviour once and we can use it with any class.

Here's an example that demonstrates a `SingletonMetaclass` used for metaprogramming in Python:

```
class SingletonMetaclass(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        print('In SingletonMetaclass.__call__')
        if cls not in cls._instances:
            print(f'Creating new instance of {cls}')
            cls._instances[cls] = super().__call__(*args, **kwargs)
        print('Returning instance')
        return cls._instances[cls]

class Session(metaclass=SingletonMetaclass):
    def __init__(self):
        print('In Session initialiser')

print('Starting')
s1 = Session()
s2 = Session()

print(f'id(s1): {id(s1)}')
print(f'id(s2): {id(s2)}')
# checks to see if they are the same instance
print(f's1 is s2: {s1 is s2}')
print('Done')
```

The output from this sample code is:

```
Starting
In SingletonMetaclass.__call__
Creating new instance of <class '__main__.Session'>
In Session initialiser
Returning instance
In SingletonMetaclass.__call__
Returning instance
id(s1): 4305011728
id(s2): 4305011728
s1 is s2: True
Done
```

If we examine the output we can see that only one instance of the class `__main__.Session` is created. Subsequent requests to create a new instance will return the stored instance. This is illustrated by the ids of `s1` and `s2` being the same and by the `is` operator returning `True`.

To understand how this works let us look at the implementation. The class `SingletonMetaclass` is a metaclass that inherits from the class metaclass (this is what makes it a metaclass). It overrides the default behaviour of the `__call__()` method which is used when creating a new instance of a class.

The `SingletonMetaclass` maintains a class side dictionary _instances that holds the instances created for any classes that the `SingletonMetaclass` is applied to. Note that we are using the _ notation here to indicate that it is a protected attribute. When a new instance of a class is to be created the `__call__()` method checks to see if an instance of that class already exists in the dictionary _instances. If it does, it returns the existing instance. Otherwise, it creates a new instance using the `super().__call__` method and stores it in `_instances` before returning it.

Once we have defined the `SingletonMetaclass` we define a class `Session` which has its metaclass set to be `SingletonMetaclass`. This means that the `SingletonMetaclass` metaclass will be used to create and customise the behaviour of instances of the Session class.

When we create instances of `Session` (`s1` and `s2`), the `SingletonMetaclass` metaclass ensures that only one instance of Session is created. Subsequent calls to create new instances will return the existing instance.

When we print out the id of the objects held in `s1` and `s2` we can see that they are the same indicating that `s1` and `s2` both hold the same instances. This is also confirmed by the is operator which checks for referential equality rather than value based equality.

10.5 Exec and Eval for Metaprogramming

Both `exec()` and `eval()` functions in Python allow a developer to dynamically compile and execute code. That is Python can on the fly create new code which can be compiled and executed at runtime. This allows them to be used as a way to implement metaprogramming style behaviours.

10.5.1 The exec() Function

The `exec()` function allows a developer to execute dynamically generated code as a statement block. It takes a string containing Python code as an argument and executes it within the *current scope*.

As an example of the `exec()` function, consider the following code:

```
MAX = 4

code = '''
for i in range(MAX):
  print(i)
'''

exec(code)
```

When this code is run the output is:

```
0
1
```

```
2
3
```

The above example has a global value MAX set to 4. A string is then created containing valid, well formed, Python code. The code within the string actually references the value in MAX but of course at this point it's just a string. We then call exec() passing in the string within code. The exec() function now executes the contents of the string as Python within the current execution context. This means that it runs the code as if it had been in the Python file as normal code and it can therefore access the MAX value. The result of running the code is a series of integers printed to the standard output.

The code that is executed is created on the fly and could have been loaded from a database, a text file or constructed based on other information available dynamically at runtime.

10.5.2 The eval() Function

The eval function evaluates a string containing a Python *expression* and returns the result. It allows you to dynamically *compute* values based on code provided as a string.

The key here is that eval executes expressions that are expected to return a value.

An example of using eval to evaluate an expression is given below:

```
expression = '((2 + 3) - 4) * 5'

result = eval(expression)

print(result)
```

The output from this code is:

```
5
```

In this example the expression '((2 + 3) − 4) * 5' is held in a string stored in the variable expression. This is evaluated using eval() and the result is printed out. In this case the result is 5.

The eval() function can be used to dynamically generate a string containing an expression which is executed at runtime.

10.5.3 eval Versus exec()

It is worth noting a difference between eval() and exec() as they can both appear to do the same thing at first glance. The main difference between eval() and exec() in Python lies in their functionality and the type of code they handle:

- `eval()` is a built-in Python function that evaluates a single *expression* and returns the result. The result returned by `eval()` is the result of evaluating the expression. It is typically used for evaluating mathematical or logical expressions, or for dynamically computing values based on user input or configuration files.
- `exec()` is a built-in Python function that executes a block of code (statements) in the current context. It takes a string containing one or more lines of Python code as input and executes but does not return any value. It is commonly used for executing dynamically generated code, code generation tasks, or running code obtained from external source.

Both exec and eval should be used with caution since they execute arbitrary code and can introduce security risks if used with untrusted input. It's essential to validate and sanitise any input used with exec or eval to prevent potential security vulnerabilities.

Part II
Computer Graphics and GUIs

Chapter 11
Introduction to Computer Graphics

11.1 Introduction

Computer Graphics are everywhere; they are on your TV, in cinema adverts, the core of many films, on your tablet or mobile phone and certainly on your PC or Mac as well as on the dashboard of your car, on your smart watch and in children's electronic toys.

However what do we mean by the term Computer Graphics? The term goes back to a time when many (most) computers were purely textual in terms of their input and output, and very few computers could generate graphical displays let alone handle input via such a display. However, in terms of this book we take the term Computer Graphics to include the creation of Graphical User Interfaces (or GUIs), graphs and charts such as bar charts or line plots of data, graphics in computer games (such as Space Invaders or Flight Simulator) as well as the generation of 2D and 3D scenes or images. We also use the term to include Computer Generated Art.

The availability of Computer Graphics is very important for the huge acceptance of computer systems by non-computer scientists over the last 40 years. It is in part thanks to the accessibility of computer systems via computer graphic interfaces that almost everybody now uses some form of computer system (whether that is a PC, a tablet, a mobile phone or a smart TV).

A Graphical User Interface (GUI) can capture the essence of an idea or a situation, often avoiding the need for a long passage of text or textual commands. It is also because a picture can paint a thousand words; as long as it is the right picture.

In many situations where the relationships between large amounts of information must be conveyed, it is much easier for the user to assimilate this graphically than textually. Similarly, it is often easier to convey some meaning by manipulating some system entities on screen, than by combinations of text commands.

For example, a well-chosen graph can make clear information that is hard to determine from a table of the same data. In turn, an adventure style game can become engaging and immersive with computer graphics which is in marked contrast to the

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_11

textual versions of the 1980s. This highlights the advantages of a visual presentation compared to a purely textual one.

11.2 Background

Every interactive software system has a human–computer interface, whether it be a single text line system or an advanced graphic display. It is the vehicle used by developers for obtaining information from their user(s), and in turn, every user has to face some form of computer interface in order to perform any desired computer operation.

Historically computer systems did not have a Graphical User Interface and rarely generated a graphical view. These systems from the 60s, 70s and 80s typically focussed on numerical or data processing tasks. They were accessed via green or grey screens on a text oriented terminal. There was little or no opportunity for graphical output.

However, during this period various researchers at laboratories such as Stanford, MIT, Bell Telephone Labs and Xerox were looking at the possibilities that graphic systems might offer to computers. Indeed even as far back as the 1963 Ivan Sutherland showed that Interactive Computer Graphics were feasible with his Ph.D. thesis on the Sketchpad system.

11.3 The Graphical Computer Era

Graphical computer displays and interactive graphical interfaces became a common means of human–computer interaction during the 1980s. Such interfaces can save a user from the need to learn complex commands. They are less likely to intimidate computer naives and can provide a large amount of information quickly in a form which can be easily assimilated by the user.

The widespread use of high-quality graphical interfaces (such as those provided by the Apple Macintosh and the early Windows interface) led many computer users to expect such interfaces to any software they use. Indeed these systems paved the way for the type of interface that is now omnipresent on PCs, Macs, Linux boxes, tablets and smart phones. This graphical user interface is based on the WIMP paradigm (Windows, Icons, Menus and Pointers) which is now the prevalent type of graphical user interface in use today.

The main advantage of any window-based system, and particularly of a WIMP environment, is that it requires only a small amount of user training. There is no need to learn complex commands, as most operations are available either as icons, operations on icons, user actions (such as swiping) or from menu options, and are easy to use. (An icon is a small graphic object that is usually symbolic of an operation or of a larger entity such as an application program or a file). In general, WIMP-based systems are simple to learn, intuitive to use, easy to retain and straightforward to work with.

These WIMP systems are exemplified by the Apple Macintosh interface (see Goldberg and Robson as well as Tesler), which was influenced by the pioneering work done at the Palo Alto Research Center on the Xerox Star Machine. It was, however, the Macintosh which brought such interfaces to the mass market, and first gained acceptance for them as tools for business, home and industry. This interface transformed the way in which humans expected to interact with their computers, becoming a de facto standard, which forced other manufacturers to provide similar interfaces on their own machines, for example Microsoft Windows for the PC.

This type of interface can be augmented by providing direct manipulation graphics. These are graphics which can be *grabbed* and manipulated by the user, using a mouse, to perform some operation or action. Icons are a simple version of this, the "opening" of an icon causes either the associated application to execute or the associated window to be displayed.

11.4 Interactive and Non Interactive Graphics

Computer graphics can be broadly subdivided into two categories:

- Non-Interactive Computer Graphics
- Interactive Computer Graphics

In Non-Interactive Computer Graphics (aka Passive Computer Graphics) an image is generated by a computer typically on a computer screen; this image can be viewed by the user (however they cannot interact with the image). Examples of non-interactive graphics presented later in this book include Computer Generated Art in which an image is generated using Pythons Turtle graphics library. Such an image can reviewed by the user but not modified. Another example might be a basic bar chart generated using Matplotlib which presents some set of data.

Interactive Computer Graphics involve the user interacting with the image displayed in the screen in some way. This might be to modify the data being displayed or to change the way in which the image is being rendered, etc. It is typified by interactive Graphical User Interfaces (GUIs) in which a user interacts with menus, buttons, input field, sliders, scrollbars, etc. However, other visual displays can also be interactive. For example, a slider could be used with a Matplotlib chart. This display could present the number of sales made on a particular date; as the slider is moved so the data changes and the chart is modified to show different data sets.

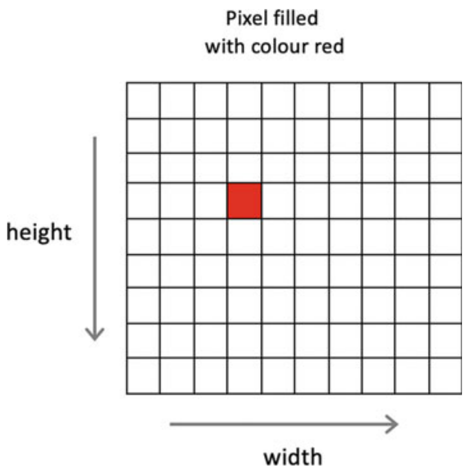
Another example is represented by all computer games which are inherently interactive and most, if not all, update their visual display in response to some user inputs. For example in the classic flight simulator game, as the user moves the joystick or mouse, the simulated plane moves accordingly and the display presented to the user updates.

11.5 Pixels

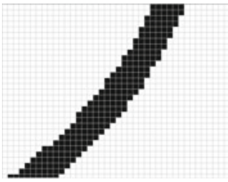
A key concept for all computer graphics systems is the pixel. Pixel was originally a word formed from combining and shortening the words picture (or pix) and element. A pixel is a cell on the computer screen. Each cell represents a dot on the screen. The size of this dot or cell and the number of cells available will vary depending upon the type, size and resolution of the screen. For example, it was common for early Windows PCs to have a 640 by 480 resolution display (using a VGA graphics card). This relates to the number of pixels in terms of the width and height. This meant that there were 640 pixels across the screen with 480 rows of pixels down the screen. By contrast todays 4 K TV displays have 4096 by 2160 pixels.

The size and number of pixels available affect the quality of the image as presented to a user. With lower-resolution displays (with fewer individual pixels) the image may appear blocky or poorly defined; whereas with a higher resolution it may appear sharp and clear.

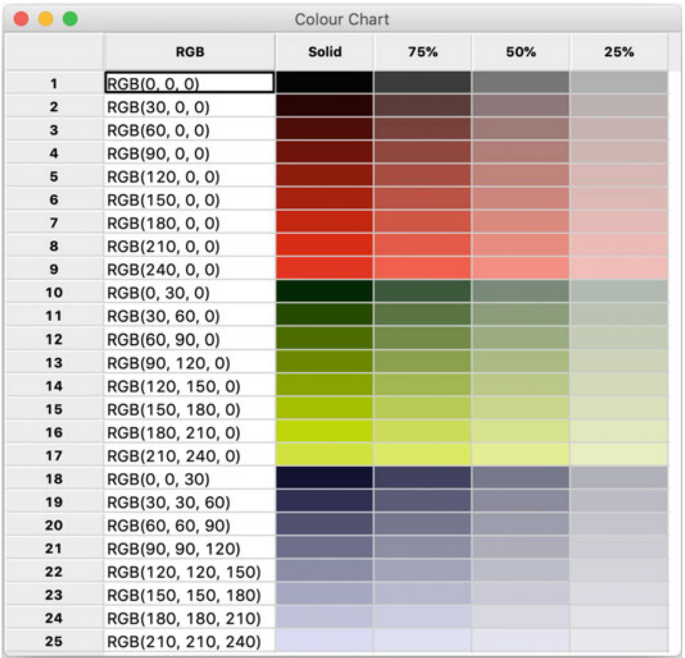
Each pixel can be referenced by its location in the display grid. By filling a pixels on the screen with different colours various images/displays can be created. For example, in the following picture a single pixel has been filled at position 4 by 4:



A sequence of pixels can form a line, a circle or any number of different shapes. However, since the grid of pixels is based open individual points, a diagonal line or a circle may need to utilise multiple pixels which when zoomed may have jagged edges. For example, the following picture shows part of a circle on which we have zoomed in:



Each pixel can have a colour and a transparency associated with it. The range of colours available depends on the display system being used. For example, monochrome displays only allow black and white, whereas a grey scale displays only allows various shades of grey to be displayed. On modern systems it is usually possible to represent a wide range of colours using the tradition RGB colour codes (where R represents Red, G represents Green and B represents Blue). In this encoding solid Red is represented by a code such as [255, 0, 0] where as solid Green is represented by [0, 255, 0] and solid Blue by [0, 0, 255]. Based on these various shades can be represented by combination of these codes such as Orange which might be represented by [255, 150, 50]. This is illustrated below for a set of RGB colours using different red, green and blue values:



In addition it is possible to apply a transparency to a pixel. This is used to indicate how solid the fill colour should be. The above grid illustrates the effect of applying a 75%, 50% and 25% transparency to colours displayed using the Python wxPython GUI library. In this library the transparency is referred to as the alpha opaque value. It can have values in the range 0 to 255 where 0 is completely transparent and 255 is completely solid.

11.6 Bit Map Versus Vector Graphics

There are two ways of generating an image/display across the pixels on the screen. One approach is known as bit mapped (or raster) graphics, and the other is known as vector graphics. In the bit mapped approach each pixel is mapped to the values to be displayed to create the image. In the vector graphics approach geometric shapes are described (such as lines and points), and these are then *rendered* onto a display. Raster graphics are simpler, but vector graphics provide much more flexibility and scalability.

11.7 Buffering

One issue for interactive graphical displays is the ability to change the display as smoothly and cleanly as possible. If a display is jerky or seems to jump from one image to another, then users will find it uncomfortable. It is therefore common to drawn the next display on some in memory structure; often referred to as a buffer. This buffer can then be rendered on the display once the whole image has been created. For example, Turtle graphics allows the user to define how many changes should be made to the display before it is rendered (or drawn) on to the screen. This can significantly speed up the performance if a graphic application.

In some cases systems will use two buffers; often referred to as double buffering. In this approach one buffer is being rendered or drawn onto the screen while the other buffer is being updated. This can significantly improve the overall performance of the system as modern computers can perform calculations and generate data much faster than it can typically be drawn onto a screen.

11.8 Python and Computer Graphics

In the remainder of this section of the book we will look at generating computer graphics using the Python Turtle graphics library. We will also discuss using this library to create Computer Generated Art. Following this we will explore the Matplotlib library used to generate charts and data plots such as bar charts, scatter graphs, line plots and heat maps. We will then explore the use of Python libraries to create GUIs using menus, fields, tables, etc.

11.9 References

The following are referenced in this chapter:

- I.E. Sutherland, Sketchpad: a man–machine graphical communication system (courtesy Computer Laboratory, University of Cambridge UCAM-CL-TR-574 September 2003), January 1963.
- D.C. Smith, C. Irby, R. Kimball, B. Verplank, E. Harslem, Designing the Star user interface. BYTE **7**(4), 242–282 (1982).

11.10 Online Resources

The following provide further reading material:

- https://en.wikipedia.org/wiki/Sketchpad Ivan Sutherlands Sketchpad from 1963.
- http://images.designworldonline.com.s3.amazonaws.com/CADhistory/Sketchpad_A_Man-Machine_Graphical_Communication_System_Jan63.pdf Ivan Sutherlands Ph.D. 1963.
- https://en.wikipedia.org/wiki/Xerox_Star The Xerox Star computer and GUI.

Chapter 12
Python Turtle Graphics

12.1 Introduction

Python is very well supported in terms of graphics libraries. One of the most widely used graphics libraries is the Turtle graphics library. This is partly because it is straight forward to use and partly because it is provided by default with the Python environment (and thus you do not need to install any additional libraries to use it). This chapter introduces the Python Turtle Graphics library.

The chapter concludes by briefly considering a number of other graphic libraries including PyOpenGL. The PyOpenGL library can be used to create sophisticated 3D scenes.

12.2 The Turtle Graphics Library

12.2.1 The Turtle Module

This provides a library of features that allow what are known as vector graphics to be created. Vector graphics refers to the lines (or vectors) that can be drawn on the screen. The drawing area is often referred to as a drawing plane or drawing board and has the idea of x, y coordinates.

The Turtle graphics library is intended just as a basic drawing tool; other libraries can be used for drawing two- and three-dimensional graphs (such as Matplotlib) but those tend to focus on specific types of graphical displays.

The idea behind the Turtle module (and its name) derives from the Logo programming language from the 60s and 70s that was designed to introduce programming to children. It had an on screen turtle that could be controlled by commands such as forward (which would move the turtle forward), right (which would turn the turtle

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_12

buy a certain number of degrees), left (which turns the turtle left by a certain number of degrees), etc. This idea has continued into the current Python Turtle graphics library where commands such as turtle.forward(10) moves the turtle (or cursor as it is now) forward 10 pixels, etc. By combining together these apparently simple commands, it is possible to create intricate and quite complex shapes.

12.2.2 Basic Turtle Graphics

Although the `turtle` module is built into Python 3 it is necessary to *import* the module before you use it:

```
import turtle
```

There are in fact two ways of working with the `turtle` module; one is to use the classes available with the library, and the other is to use a simpler set of functions that hide the classes and objects. In this chapter we will focus on the set of functions you can use to create drawings with the turtle library.

The first thing we will do is to set up the window we will use for our drawings; the `TurtleScreen` class is the parent of all screen implementations used for whatever operating system you are running on.

If we are using the turtle module functions then the screen object is initialised as appropriate for our operating system. This means that we can just focus on the following functions to configure the layout/display such as this screen can have a title, a size, a starting location, etc.

The key functions are:

- `setup(width, height, startx, starty)` Sets the size and position of the main window/screen. The parameters are:
  - `width`—if an integer, a size in pixels, if a float, a fraction of the screen; default is 50% of screen
  - `height`—if an integer, the height in pixels, if a float, a fraction of the screen; default is 75% of screen
  - `startx`—if positive, starting position in pixels from the left edge of the screen, if negative from the right edge, if None, centre window horizontally
  - `starty`—if positive, starting position in pixels from the top edge of the screen, if negative from the bottom edge, if None, centre window vertically
- `title(titlestring)` sets the title of the screen/window.
- `exitonclick()` shuts down the turtle graphics screen/window when the use clicks on the screen.
- `bye()` shuts down the turtle graphics screen/window.
- `done()` starts the main event loop; this must be the last statement in a turtle graphics program.

- `speed(speed)` the drawing speed to use, the default is 3. The higher the value the faster the drawing takes place, values in the range 0–10 are accepted.
- `turtle.tracer(n=None)` can be used to batch updates to the Turtle graphics screen. It is very useful when a drawing become large and complex. By setting the number (*n*) to a large number (say 600) then 600 elements will be drawn in memory before the actual screen is updated in one go; this can significantly speed up the generation of for example, a fractal picture. When called without arguments, returns the currently stored value of `n`.
- `turtle.update()` performs an update of the turtle screen; this should be called at the end of a program when `tracer()` has been used as it will ensure that all elements have been drawn even if the tracer threshold has not yet been reached.
- `pencolor(color)` used to set the colour used to draw lines on the screen; the colour can be specified in numerous ways including using named colours set as 'red', 'blue', 'green' or using the RGB colour codes or by specifying the colour using hexadecimal numbers. For more information on the named colours and RGB colour codes to use see https://www.tcl.tk/man/tcl/TkCmd/colors.htm. Note all colour methods use American spellings for example this method is `pencolor` (not pencolour).
- `fillcolor(color)` used to set the colour to use to fill in closed areas within drawn lines. Again note the spelling of colour!

The following code snippet illustrates some of these functions:

```
import turtle

# set a title for your canvas window
turtle.title('My Turtle Animation')

# set up the screen size (in pixels)
# set the starting point of the turtle (0, 0)
turtle.setup(width=200, height=200, startx=0, starty=0)

# sets the pen color to red
turtle.pencolor('red')

# …

# Add this so that the window will close when clicked on
turtle.exitonclick()
```

We can now look at how to actually draw a shape onto the screen.

The cursor on the screen has several properties; these include the current drawing colour of the *pen* that the cursor moves, but also its current position (in the *x*, *y* coordinates of the screen) and the direction it is currently facing. We have already seen that we can control one of these properties using the `pencolor()` method; other methods are used to control the cursor (or turtle) and are presented below.

The direction in which the *cursor* is pointing can be altered using several functions including:

- `right(angle)` Turn cursor right by angle units.
- `left(angle)` Turn the cursor left by angle units
- `setheading(to_angle)` Set the orientation of the cursor to `to_angle`. Where 0 is east, 90 is north, 180 is west and 270 is south.

We can move the cursor (and if the pen is down we will draw a line) using:

- `forward(distance)` move the cursor forward by the specified distance in the direction that the cursor is currently pointing. If the pen is down draw a line.
- `backward(distance)` move the cursor backward by distance in the opposite direction that that in which the cursor is pointing.

And we can also explicitly position the cursor:

- `goto(x, y)` move the cursor to the *x*, *y* location on the screen specified; if the pen is down draw a line. You can also use steps and set position to do the same thing.
- `setx(x)` sets the cursor's *x* coordinate, leaves the *y* coordinate unchanged.
- `sety(y)` sets the cursor's *y* coordinate, leaves the *x* coordinate unchanged.

It is also possible to move the cursor without drawing by modifying whether the pen is up or down:

- `penup()` move the pen up—moving the cursor will no longer draw a line.
- `pendown()` move the pen down—moving the cursor will now draw a line in the current pen colour.

The size of the pen can also be controlled:

- `pensize(width)` set the line thickness to width. The method `width()` is an alias for this method.

It is also possible to draw a circle or a dot:

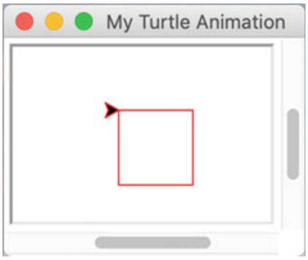
- `circle(radius, extent, steps)` draws a circle using the given radius. The extent determines how much of the circle is drawn; if the extent is not given then the whole circle is drawn. Steps indicate the number of steps to be used to drawn the circle (it can be used to draw regular polygons).
- `dot(size, color)` draws a filled circle with the diameter of size using the specified colour.

We can now use some of the above method to draw a shape on our screen. For this first example, we will keep it very simple, we will draw a simple square:

```
# Draw a square
turtle.forward(50)
turtle.right(90)
turtle.forward(50)
turtle.right(90)
```

```
turtle.forward(50)
turtle.right(90)
turtle.forward(50)
turtle.right(90)
```

The above moves the cursor forward 50 pixels then turns 90 degrees before repeating these steps three times. The end result is that a square of 50 × 50 pixels is drawn on the screen:



Note that the cursor is displayed during drawing (this can be turned off with `turtle.hideturtle()` as the cursor was originally referred to as the turtle).

12.2.3 Drawing Shapes

Of course you do not need to just use fixed values for the shapes you draw, you can use variables or calculate positions based on expressions, etc.

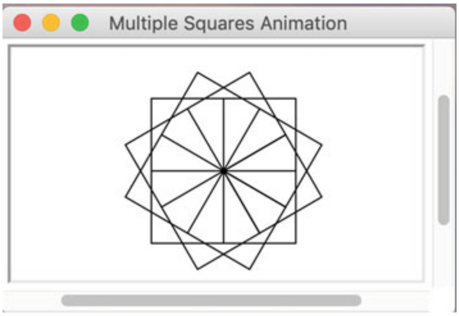
For example, the following program creates a sequences of squares rotated around a central location to create an engaging image:

```
import turtle

def setup():
    """ Provide the config for the screen """
    turtle.title('Multiple Squares Animation')
    turtle.setup(100, 100, 0, 0)
    turtle.hideturtle()

def draw_square(size):
    """ Draw a square in the current direction """
    turtle.forward(size)
    turtle.right(90)
    turtle.forward(size)
    turtle.right(90)
    turtle.forward(size)
    turtle.right(90)
    turtle.forward(size)

setup()
```

```
for _ in range(0, 12):
    draw_square(50)
    # Rotate the starting direction
    turtle.right(120)

# Add this so that the window will close when clicked on
turtle.exitonclick()
```

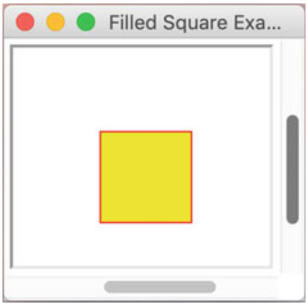
In this program two functions have been defined, one to setup the screen or window with a title and a size and to turn off the cursor display. The second function takes a size parameter and uses that to draw a square. The main part of the program then sets up the window and uses a for loop to draw 12 squares of 50 pixels each by continuously rotating 120 degrees between each square. Note that as we do not need to reference the loop variable we are using the '_' format which is considered an anonymous loop variable.

The image generated by this program is shown below:



12.2.4 Filling Shapes

It is also possible to fill in the area within a drawn shape. For example, if we wanted to fill in one of the squares we have drawn as shown below:



To do this we can use the `begin_fill()` and `end_fill()` functions:

- `begin_fill()` indicates that shapes should be filled with the current fill colour, this function should be called just before drawing the shape to be filled.
- `end_fill()` called after the shape to be filled has been finished. This will cause the shape drawn since the last call to `begin_fill()` to be filled using the current fill colour.
- `filling()` Return the current fill state (`True` if filling, `False` if not).

The following program uses this (and the earlier `draw_square()` function) to draw the above filled square:

```
turtle.title('Filled Square Example')
turtle.setup(100, 100, 0, 0)
turtle.hideturtle()

turtle.pencolor('red')
turtle.fillcolor('yellow')
turtle.begin_fill()

draw_square(60)

turtle.end_fill()
turtle.done()
```

12.3 Other Graphics Libraries

Of course Turtle graphics is not the only graphics option available for Python; however, other graphics libraries do not come pre-packed with Python and must be downloaded using a tool such as Anaconda or PyCharm.

- **PyQtGraph**: The PyQtGraph library is pure Python library oriented towards mathematics, scientific and engineering graphic applications as well as GUI applications. For more information see http://www.pyqtgraph.org.

- **Pillow**: Pillow is a Python Imaging Library (based on PIL the Python Imaging Library) that provides image processing capabilities for use in Python. For more information on Pillow see https://pillow.readthedocs.io/en/stable.
- **Pyglet**: Pyglet is another windowing and multimedia library for Python. See https://bitbucket.org/pyglet/pyglet/wiki/Home.

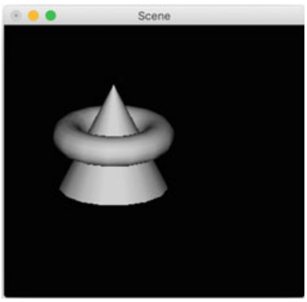
12.4 3D Graphics

Although it is certainly possible for a developer to create convincing 3D images using Turtle graphics, it is not the primary aim of the library. This means that there is no direct support for creating 3D images other than the basic cursor moving facilities and the programmers skill.

However, there are 3D graphics libraries available for Python. One such library is Pand3D (https://www.panda3d.org) while another is VPython (https://vpython.org) while a third is pi3d (https://pypi.org/project/pi3d). However we will briefly look at the PyOpenGL library as this builds on the very widely used OpenGL library.

12.4.1 PyOpenGL

PyOpenGL is an open-source project that provides a set of bindings (or wrappings around) the OpenGL library. OpenGL is the Open Graphics Library which is a cross-language, cross-platform API for rendering 2D and 3D vector graphics. OpenGL is used in a wide range of applications from games, to virtual reality, through data and information visualisation systems to computer-aided design (CAD) systems. PyOpenGL provides a set of Python functions that call out from Python to the underlying openGL libraries. This makes it very easy to create 3D vector-based images in Python using the industry standard OpenGL library. A very simple example of an image created using PyOpenGL is given below:



12.5 Online Resources

The following provides further reading material:

- https://docs.python.org/3/library/turtle.html Turtle graphics documentation.
- http://pythonturtle.org/ The Python Turtle programming environment—this intended for teaching the basic concepts behind programming using the Turtle graphics library.
- http://pyopengl.sourceforge.net The PyOpenGL home page.
- https://www.opengl.org The OpenGL home page.

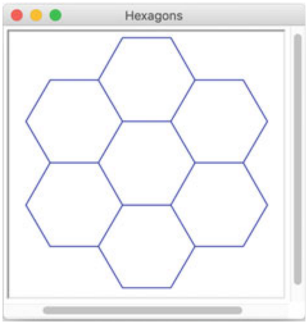
12.6 Exercises

The aim of this exercise is to create a graphic display using Python Turtle graphics.

You should create a simple program to draw an octagon on the Turtle graphics screen.

Modify your program so that there is a hexagon drawing function. This function should take three parameters, the x and y coordinates to start drawing the octagon and the size of each side of the octagon.

Modify your program to draw the hexagon in multiple locations to create the following picture:

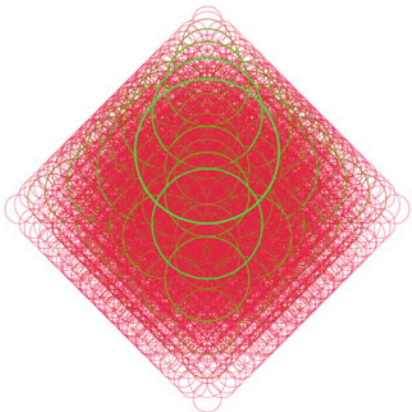


Chapter 13
Computer Generated Art

13.1 Creating Computer Art

Computer Art is defined as any art that uses a computer. However, in the context of this book we mean it to be art that is generated by a computer or more specifically a computer program. The following example illustrates how in a very few lines of Python code, using the Turtle graphics library, you can create images that might be considered to be Computer Art.

The following image is generated by a recursive function that draws a circle at a given x, y location of a specified size. This function recursively calls itself by modifying the parameters so that smaller and smaller circles are drawn at different locations until the size of the circles goes below 20 pixels.



J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_13

The program used to generate this picture is given below for reference:

```
import turtle

WIDTH = 640
HEIGHT = 360

def setup_window():
   # Set up the window
   turtle.title('Circles in My Mind')
   turtle.setup(WIDTH, HEIGHT, 0, 0)
   # Indicates RGB numbers will be in the range 0 to 255
   turtle.colormode(255)
   turtle.hideturtle()
   # Batch drawing to the screen for faster rendering
   turtle.tracer(2000)

   # Speed up drawing process
   turtle.speed(10)
   turtle.penup()

def draw_circle(x, y, radius, red=50, green=255, blue=10, width=7):
   """ Draw a circle at a specific x, y location.
   Then draw four smaller circles recursively """
   colour = (red, green, blue)

   # Recursively drawn smaller circles
   if radius > 50:
      # Calculate colours and line width for smaller circles
      if red < 216:
         red = red + 33
         green = green - 42
         blue = blue + 10
         width -= 1
      else:
         red = 0
         green = 255
      # Calculate the radius for the smaller circles
      new_radius = int(radius / 1.3)
      # Drawn four circles
       draw_circle(int(x + new_radius), y, new_radius, red, green,
blue, width)
      draw_circle(x - new_radius, y, new_radius, red, green, blue,
width)
       draw_circle(x, int(y + new_radius), new_radius, red, green,
blue, width)
       draw_circle(x, int(y - new_radius), new_radius, red, green,
blue, width)
```

```
    # Draw the original circle
    turtle.goto(x, y)
    turtle.color(colour)
    turtle.width(width)
    turtle.pendown()
    turtle.circle(radius)
    turtle.penup()

# Run the program
print('Starting')
setup_window()
draw_circle(25, -100, 200)

# Ensure that all the drawing is rendered
turtle.update()
print('Done')
turtle.done()
```

There are a few points to note about this program. It uses recursion to draw the circles with smaller and smaller circles being drawn until the radius of the circles falls below a certain threshold (there termination point).

It also uses the `turtle.tracer()` function to speed up drawing the picture as 2000 changes will be buffered before the screen is updated.

Finally, the colours used for the circles are changed at each level of recession; a very simple approach is used to that the Red, Green and Blue codes are changed resulting in different colour circles. Also a line width is used to reduce the size of the circle outline to add more interest to the image.

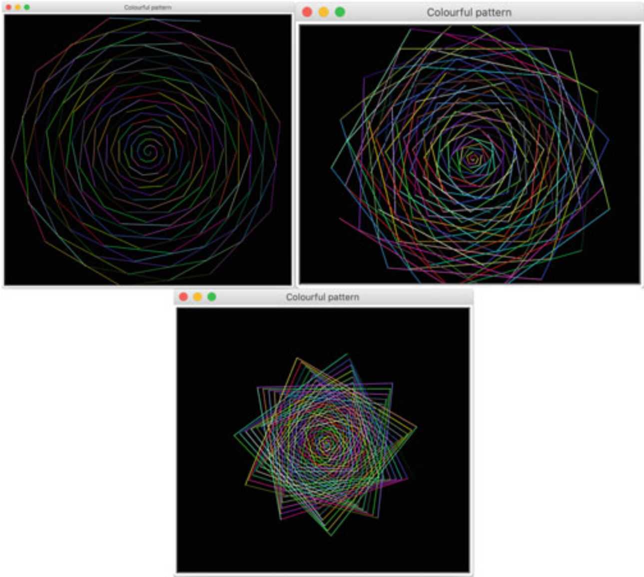
13.2 A Computer Art Generator

As an another example of how you can use Turtle graphics to create Computer Art, the following program randomly generates RGB colours to use for the lines being drawn which gives the pictures more interest. It also allows the user to input an angle to use when changing the direction in which the line is draw. As the drawing happens within a loop even this simple change to the angle used to draw the lines can generate very different pictures.

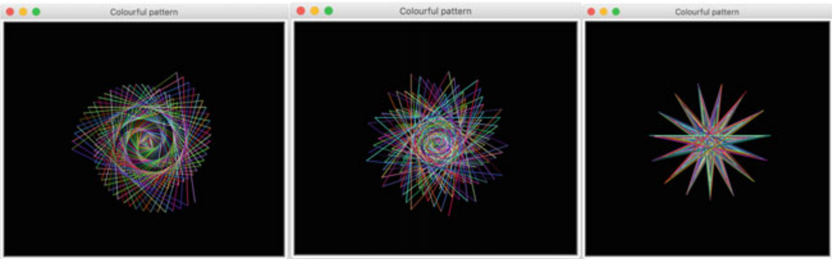
```
# Lets play with some colours
import turtle
from random import randint

def get_input_angle():
    """ Obtain input from user and convert to an int """
    message = 'Please provide an angle:'
    value_as_string = input(message)
    while not value_as_string.isnumeric():
```

```
        print('The input must be an integer!')
        value_as_string = input(message)
    return int(value_as_string)

def generate_random_colour():
    """ Generates an R,G,B values randomly in range
    0 to 255 """
    r = randint(0, 255)
    g = randint(0, 255)
    b = randint(0, 255)
    return r, g, b

print('Set up Screen')
turtle.title('Colourful pattern')
turtle.setup(640, 600)
turtle.hideturtle()
turtle.bgcolor('black')# Set the background colour of the
screen
turtle.colormode(255)# Indicates RGB numbers will be in the
range 0 to 255
turtle.speed(10)
angle = get_input_angle()

print('Start the drawing')
for i in range(0, 200):
    turtle.color(generate_random_colour())
    turtle.forward(i)
    turtle.right(angle)

print('Done')
turtle.done()
```

Some sample images generated from this program are given below. The left most picture is generated by inputting an angle of 38, the picture on the right uses an angle of 68 and the bottom picture an angle of 98.



The following pictures below use an angles of 118, 138 and 168 respectively.



What is interesting about these images is how different each is; even though they use exactly the same program. This illustrates how algorithmic or Computer Generated Art can be as subtle and flexible as any other art form. It also illustrates that even with such a process it is still up to the human to determine which image (if any) is the most aesthetically pleasing.

13.3 Fractals in Python

Within the arena of Computer Art fractals are a very well-known art form. Fractals are recurring patterns that are calculated either using an iterative approach (such as for loop) or a recursive approach (when a function calls itself but with modified parameters). One of the really interesting features of fractals is that they exhibit the same pattern (or nearly the same pattern) at successive levels of granularity. That is, if you magnified a fractal image you would find that the same pattern is being repeated at successively smaller and smaller magnifications. This is known as expanding symmetry or unfolding symmetry; if this replication is exactly the same at every scale, then it is called affine self-similar.

Fractals have their roots in the world of mathematics starting in the seventeenth century, with the term fractal being coined in the twentieth century by mathematical Benoit Mandelbrot in 1975. One often cited description that Mandelbrot published to describe geometric fractals is:

> a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole.

For more information see Mandelbrot, Benoît B. (1983). The fractal geometry of nature. Macmillan. ISBN 978-0-7167-1186-5).

Since the latter part of the twentieth century fractals have been a commonly used way of creating Computer Art.

One example of a fractal often used in Computer Art is the Koch snowflake, while another is the Mandelbrot set. Both of these are used as examples to illustrate how Python and the Turtle graphics library can be used to create fractal-based art.

13.4 The Koch Snowflake

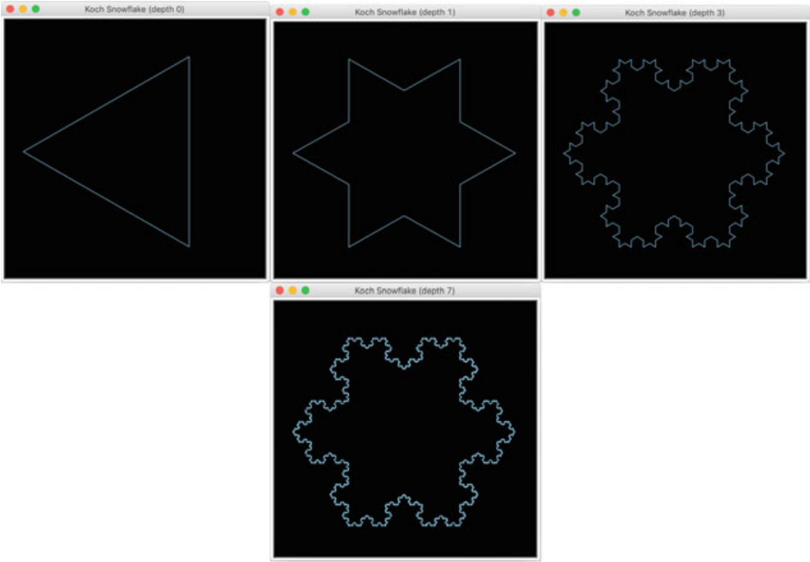
The Koch snowflake is a fractal that begins with equilateral triangle and then replaces the middle third of every line segment with a pair of line segments that form an equilateral bump. This replacement can be performed to any depth generating finer and finer grained (smaller and smaller) triangles until the overall shape resembles a snow flake.

The following program can be used to generate a Koch snowflake with different levels of recursion. The larger the number of levels of recursion the more times each line segment is dissected.

```
import turtle

# Set up Constants
ANGLES = [60, -120, 60, 0]
SIZE_OF_SNOWFLAKE = 300

def get_input_depth():
    """ Obtain input from user and convert to an int """
```

```
    message = 'Please provide the depth (0 or a positive integer):'
    value_as_string = input(message)
    while not value_as_string.isnumeric():
        print('The input must be an integer!')
        value_as_string = input(message)
    return int(value_as_string)

def setup_screen(title, background='white', screen_size_x=640,
screen_size_y=320, tracer_size=800):
    print('Set up Screen')
    turtle.title(title)
    turtle.setup(screen_size_x, screen_size_y)
    turtle.hideturtle()
    turtle.penup()
    turtle.backward(240)
    # Batch drawing to the screen for faster rendering
    turtle.tracer(tracer_size)
    turtle.bgcolor(background) # Set the background colour of the
screen

def draw_koch(size, depth):
    if depth > 0:
        for angle in ANGLES:
            draw_koch(size / 3, depth - 1)
            turtle.left(angle)
    else:
        turtle.forward(size)

depth = get_input_depth()

setup_screen('Koch Snowflake (depth ' + str(depth) + ')',
             background='black',
             screen_size_x=420, screen_size_y=420)
# Set foreground colours
turtle.color('sky blue')

# Ensure snowflake is centred
turtle.penup()
turtle.setposition(-180,0)
turtle.left(30)
turtle.pendown()

# Draw three sides of snowflake
for _ in range(3):
    draw_koch(SIZE_OF_SNOWFLAKE, depth)
    turtle.right(120)

# Ensure that all the drawing is rendered
turtle.update()
print('Done')
turtle.done()
```

Several different runs of the program are shown below with the depth set at 0, 1, 3 and 7.



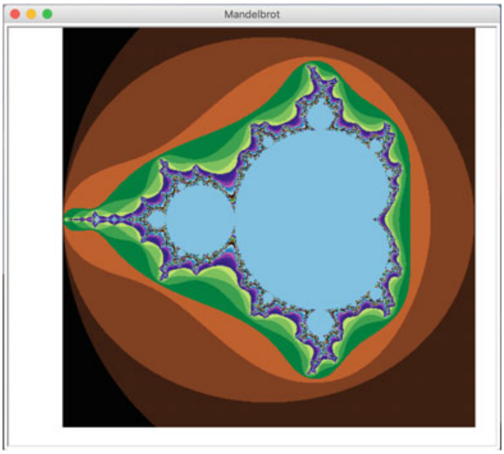
Running the simple `draw_koch()` function with different depths makes it easy to see the way in which each side of a triangle can be dissected into a further triangle like shape. This can be repeated to multiple depths giving a more detailed structured in which the same shape is repeated again and again.

13.5 Mandelbrot Set

Probably one of the most famous fractal images is based on the Mandelbrot set. The Mandelbrot set is the set of complex numbers c for which the function `z * z + c` does not diverge when iterated from $z = 0$ for which the sequence of functions (func(0), func(func(0)), etc.) remains bounded by an absolute value. The definition of the Mandelbrot set and its name is down to the French mathematician Adrien Douady, who named it as a tribute to the mathematician Benoit Mandelbrot.

Mandelbrot set images may be created by sampling the complex numbers and testing, for each sample point *c*, whether the sequence func(0), func(func(0)), etc. ranges to infinity (in practice this means that a test is made to see if it leaves some predetermined bounded neighbourhood of 0 after a predetermined number of iterations). Treating the real and imaginary parts of *c* as image coordinates on the complex plane, pixels may then be coloured according to how soon the sequence crosses an arbitrarily chosen threshold, with a special colour (usually black) used for the values of *c* for which the sequence has not crossed the threshold after the predetermined number of iterations (this is necessary to clearly distinguish the Mandelbrot set image from the image of its complement).

The following image was generated for the Mandelbrot set using Python and Turtle graphics.



The program used to generate this image is given below:

```
import turtle

# Set up constants
SCREEN_OFFSET_X = 250
SCREEN_OFFSET_Y = 240

# max iterations allowed
MAX_ITERATIONS = 255

# image size
IMAGE_SIZE_X = 512
IMAGE_SIZE_Y = 512

# Drawing area
MIN_X = -2.0
MAX_X = 1.0
MIN_Y = -1.5
```

```
MAX_Y = 1.5

def setup_screen(title, background='white', screen_size_x=640,
screen_size_y=320, tracer_size=200):
    print('Set up Screen')
    turtle.title(title)
    turtle.setup(screen_size_x, screen_size_y)
    turtle.hideturtle()
    turtle.penup()
    turtle.backward(240)
    turtle.tracer(tracer_size)
    turtle.bgcolor(background) # Set the background colour of the
screen

setup_screen('Mandelbrot', screen_size_x=IMAGE_SIZE_X, screen_
size_y=IMAGE_SIZE_Y, tracer_size=20000)
turtle.colormode(255)# Indicates RGB numbers will be in the
range 0 to 255

# Generate Mandelbrot
for y in range(IMAGE_SIZE_Y):
    zy = y * (MAX_Y - MIN_Y) / (IMAGE_SIZE_Y - 1) + MIN_Y
    for x in range(IMAGE_SIZE_X):
        zx = x * (MAX_X - MIN_X) / (IMAGE_SIZE_Y - 1) + MIN_X
        z = zx + zy * 1j
        c = z
        for i in range(MAX_ITERATIONS):
            if abs(z) > 2.0:
                break
            z = z * z + c
        turtle.color((i % 4 * 64, i % 8 * 32, i % 16 * 16))
        turtle.setposition(x - SCREEN_OFFSET_X, y - SCREEN_OFFSET_Y)
        turtle.pendown()
        turtle.dot(1)
        turtle.penup()

# Ensure that all the drawing is rendered
turtle.update()

print('Done')
turtle.done()
```

13.6 Online Resources

The following provide further reading material:

- https://en.wikipedia.org/wiki/Fractal For the Wikipedia page on Fractals.
- https://en.wikipedia.org/wiki/Koch_snowflake the Wikipedia page on the Koch snowflake.
- https://en.wikipedia.org/wiki/Mandelbrot_set Wikipedia page on the Mandelbrot set.

13.7 Exercises

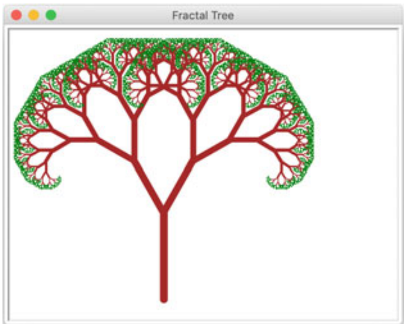
The aim of this exercise is to create a fractal tree.

A fractal tree is a tree in which the overall structure is replicated at finer and finer levels through the tree until a set of leaf elements are reached.

To draw the fractal tree you will need to:

- Draw the trunk.
- At the end of the trunk, split the trunk in two with the left trunk and the right trunk being 30° left/right of the original trunk. For aesthetic purposes the trunk may become thinner each time it is split. The trunk may be drawn in a particular colour such as brown.
- Continue this until a maximum number of splits have occurred (or the trunk size reduces to a particular minimum). You have now reached the leaves (you may draw the leaves in a different colour, e.g. green).

An example of a fractal tree is given below:

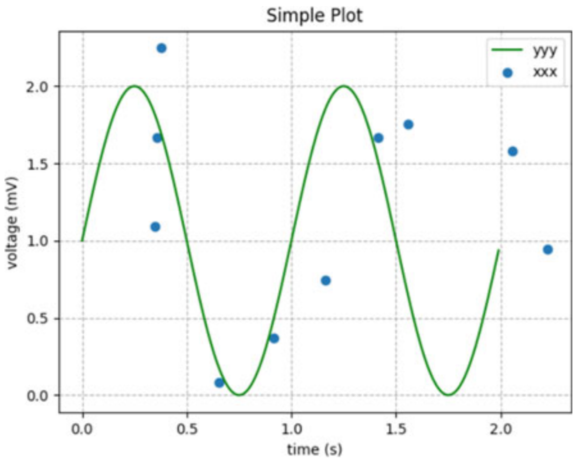


Chapter 14
Introduction to Matplotlib

14.1 Introduction

Matplotlib is a Python graphing and plotting library that can generate a variety of different types of graph or chart in a variety of different formats. It can be used to generate line charts, scatter graphs, heat maps, bar charts, pie charts and 3D plots. It can even support animations and interactive displays.

An example of a graph generated using Matplotlib is given below. This shows a line chart used to plot a simple sign wave:



J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_14

Matplotlib is a very flexible and powerful graphing library. It can support a variety of different Python graphics platforms and operating system windowing environments. It can also generate output graphics in a variety of different formats including PNG, JPEG, SVG and PDF.

Matplotlib can be used on its own or in conjunction with other libraries to provide a wide variety of facilities. One library that is often used in conjunction with Matplotlib is NumPy which is a library often used in Data Science applications that provides a variety of functions and data structures (such as n-dimensional arrays) that can be very useful when processing data for display within a chart.

However, Matplotlib does not come pre-built into the Python environment; it is an optional module which must be added to your environment of IDE.

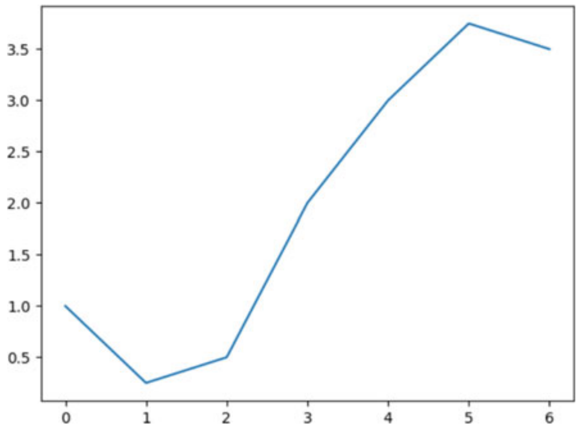
In this chapter we will introduce the Matplotlib library, its architecture, the components that comprise a chart and the `pyplot` API. The `pyplot` API is the simplest and most common way in which a programmer interacts with Matplotlib. We will then explore a variety of different types of chart and how they can be created using Matplotlib, from simple line charts, through scatter charts, to bar charts and pie charts. We will finish by looking at a simple 3D chart.

14.2 Matplotlib

Matplotlib is a graph plotting library for Python. For simple graphs Matplotlib is very easy to use, for example to create a simple line graph for a set of x and y coordinates you can use the `matplotlib.pyplot.plot` function:

```
import matplotlib.pyplot as pyplot

# Plot a sequence of values
pyplot.plot([1, 0.25, 0.5, 2, 3, 3.75, 3.5])

# Display the chart in a window
pyplot.show()
```

This very simple program generates the following graph:

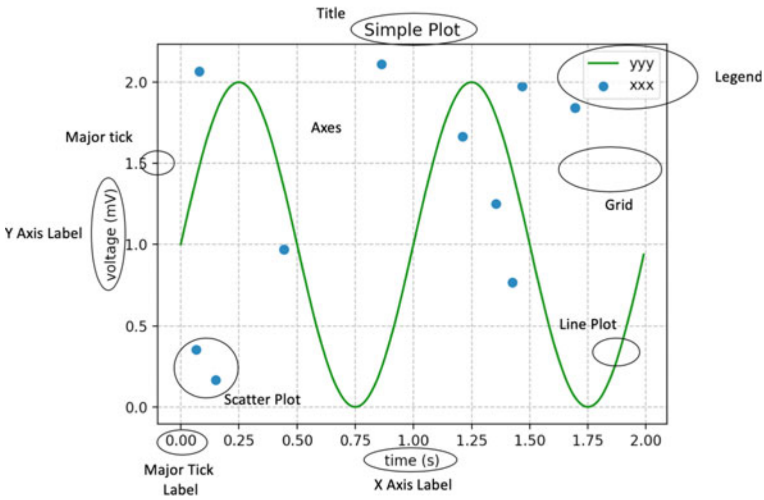


In this example, the `plot()` function takes a sequence of values which will be treated as the *y* axis values; the *x* axis values are implied by the position of the value within the list. Thus as the list has six elements in it the *x* axis has the range 0–6. In turn as the maximum value contained in the list is 3.75, then the *y* value ranges from 0 to 4.

14.3 Plot Components

Although they may seem simple, there are numerous elements that comprise a Matplotlib graph or plot. These elements can all be manipulated and modified independently. It is therefore useful to be familiar with the Matplotlib terminology associated with these elements, such as ticks, legends, labels, etc.

The elements that make up a plot are illustrated below:



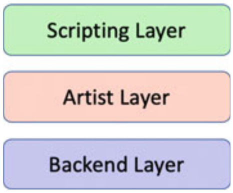
The diagram illustrates the following elements:

- **Axes** An Axes is defined by the `matplotlib.axes.Axes` class. It is used to maintain most of the elements of a figure, namely the *X* and *Y* Axis, the ticks, the line plots, any text and any polygon shapes.
- **Title** This is the title of the whole figure.
- **Ticks** (major and minor) The ticks are represented by the class `matplotlib.axis.Tick`. A tick is the mark on the Axis indicating a new value. There can be major ticks which are larger and may be labelled. There are also minor ticks which can be smaller (and may also be labelled).
- **Tick Labels** (major and minor) This is a label on a tick.
- **Axis** The `maplotlib.axis.Axis` class defines an Axis object (such as an *X* or *Y* axis) within a parent `Axes` instance. It can have formatters used to format the labels used for the major and minor ticks. It is also possible to set the locations of the major and minor ticks.
- **Axis Labels** (*X*, *Y* and in some cases *Z*) These are labels used to describe the Axis.
- **Plot** types such as line and scatter plots. Various types of plots and graphs are supported by Matplotlib including line plots, scatter graphs, bar charts and pie charts.
- **Grid** This is an optional grid displayed behind a plot, graph or chart. The grid can be displayed with a variety of different line styles (such as solid or dashed lines), colours and line widths.

14.4 Matplotlib Architecture

The Matplotlib library has a layered architecture that hides much of the complexity associated with different windowing systems and graphic outputs. This architecture has three main layers, the Scripting Layer, the Artist Layer and the Backend Layer. Each layer has specific responsibilities and components. For example, the Backend is responsible for reading and interacting with the graph or plot being generated. In turn the Artist Layer is responsible for creating the graph objects that will be rendered by the Backend Layer. Finally the Scripting Layer is used by the developer to create the graphs.

This architecture is illustrated below:



14.4.1 Backend Layer

The Matplotlib backend *layer* handles the generation of output to different target formats. Matplotlib itself can be used in many different ways to generate many different outputs.

Matplotlib can be used interactively, it can be embedded in an application (or graphical user interface), and it may be used as part of a batch application with plots being stored as PNG, SVG, PDF or other images.

To support all of these use cases, Matplotlib can target different outputs, and each of these capabilities is called a *backend*; the "frontend" is the developer facing code. The Backend Layer maintains all the different backends, and the programmer can either use the default backend or select a different backend as required.

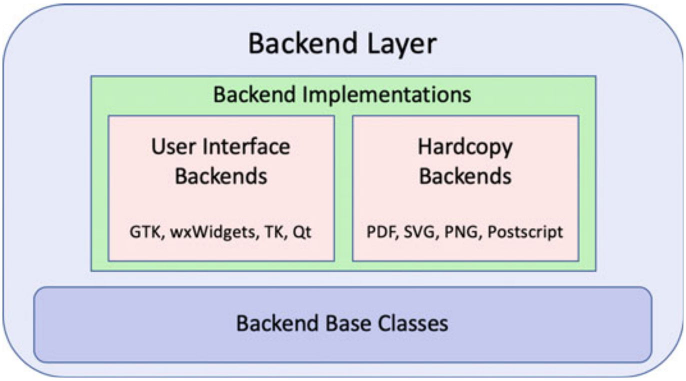
The backend to be used can be set via the `matplotlib.use()` function. For example, to set the backend to render Postscript use: matplotlib.use('PS') this is illustrated below:

```
import matplotlib

if 'matplotlib.backends'not in sys.modules:
 matplotlib.use('PS')

import matplotlib.pyplot as pyplot
```

It should be noted that if you use the `matplotlib.use()` function, this must be done *before* importing `matplotlib.pyplot`. Calling `matplotlib.use()` after `matplotlib.pyplot` has been *imported* will have no effect. Note that the argument passed to the `matplotlib.use()` function is case sensitive.

The default renderer is the '`Agg`' which uses the Anti-Grain Geometry C++ library to make a raster (pixel) image of the figure. This produces high-quality raster graphics-based images of the data plots.

The '`Agg`' backend was chosen as the default backend as it works on a broad selection of Linux machines as its supporting requirements are quite small; other backends may run on one particular machine, but may not work on another machine. This occurs if a particular machine does not have all the dependencies loaded that the specified Matplotlib backend relies on.



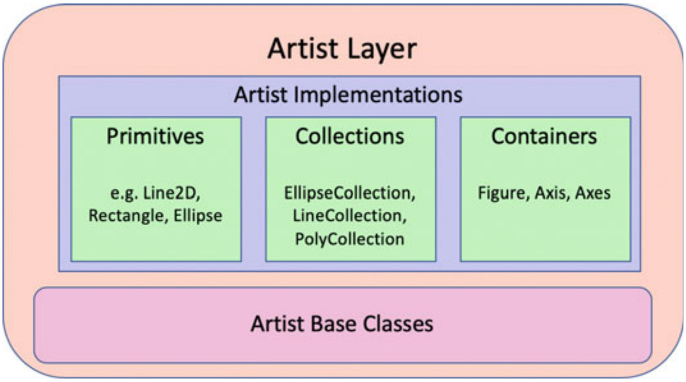
The Backend Layer can be divided into two categories:

- User interface backends (interactive) that support various Python windowing systems such as `wxWidgets`, Qt, TK, etc.
- Hardcopy Backends (non-interactive) that support raster and vector graphic outputs.

The user interface and hardcopy backends are built upon common abstractions referred to as the Backend base classes.

14.4.2 The Artist Layer

The Artist Layer provides the majority of the functionality that you might consider to be what Matplotlib actually does; that is the generation of the plots and graphs that are rendered/displayed to the user (or output in a particular format).



The artist layer is concerned with things such as the lines, shapes, axis, axes, text, etc. that comprise a plot.

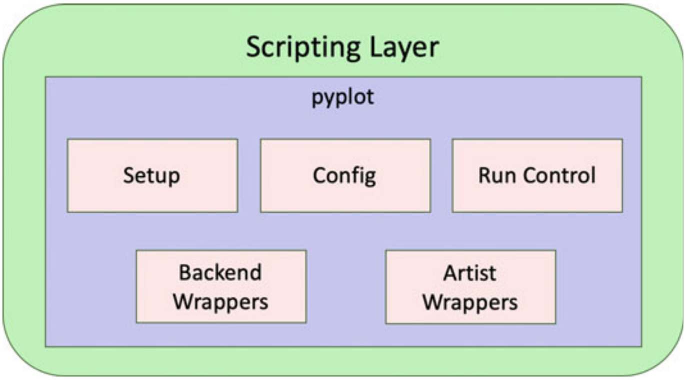
The classes used by the Artist Layer can be classified into one of the following three groups; primitives, containers and collections:

- Primitives are classes used to represent graphical objects that will be drawn on to a figures canvas.
- Containers are objects that hold primitives. For example, typically a figure would be instantiated and used to create one or more Axes, etc.
- Collections are used to efficiently handle large numbers of similar types of objects.

Although it is useful to be aware of these classes, in many cases you will not need to work with them directly as the `pyplot` API hides much of the detail. However, it is possible to work at the level of figures, axes, ticks, etc. if required.

14.4.3 The Scripting Layer

The scripting layer is the developer facing interface that simplifies the task of working with the other layers.



Note that from the programmers' point of view, the Scripting Layer is represented by the `pyplot` module. Under the covers `pyplot` uses module-level objects to track the state of the data, handle drawing the graphs, etc.

When imported `pyplot` selects either the *default* backend for the system or the one that has been configured; for example via the `matplotlib.use()` function.

It then calls a `setup()` function that:

- Creates a figure manager factory function, which when called will create a new figure manager appropriate for the selected backend,
- Prepares the drawing function that should be used with the selected backend,
- Identifies the callable function that integrates with the backend `mainloop` function,
- Provides the *module* for the selected backend.

The `pyplot` interface simplifies interactions with the internal wrappers by providing methods such as `plot()`, `pie()`, `bar()`, `title()`, `savefig()`, `draw()` and `figure()`.

Most of the examples presented later in this chapter will use the functions provided by the `pyplot` module to create the required charts, thereby hiding the lower level details.

14.5 Online Resources

See the online documentation for:

- https://matplotlib.org The Matplotlib library. This incorporates numerous examples with complete listings, documentation, galleries and a detailed user guide and FAQ.
- https://pythonprogramming.net/matplotlib-python-3-basics-tutorial Python Matplotlib crash course.

Chapter 15
Graphing with Matplotlib Pyplot

15.1 Introduction

In this chapter we will explore the Matplotlib pyplot API. This is the most common way in which developers generate different types of graphs or plots using Matplotlib.

15.2 The pyplot API

The purpose of the `pyplot` module and the API it presents is to simplify the generation and manipulation of Matplotlib plots and charts. As a whole the Matplotlib library tried to make simple things easy and complex things possible. The primary way in which it achieves the first of these aims is through the `pyplot` API as this API has high-level functions such as bar(), plot(), scatter() and pie() that make it easy to create bar charts, line plots, scatter graphs and pie charts.

One point to note about the functions provided by the `pyplot` API is that they can often take very many parameters; however most of these parameters will have default values that in many situations will give you a reasonable default behaviour/default visual representations. You can therefore ignore most of the parameters available until such time as you actually need to do something different; at which point you should refer to the Matplotlib documentation as this has extensive material as well as numerous examples.

It is of course necessary to import the `pyplot` module; as it is a module within the Matplotlib (e.g. `matplotlib.pyplot`) library. It is often given an alias within a program to make it easier to reference. Common alias for this module are `pyplot` or `plt`.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_15

A typical import for the `pyplot` module is given below:

```
import matplotlib.pyplot as pyplot
```

The pyplot API can be used to

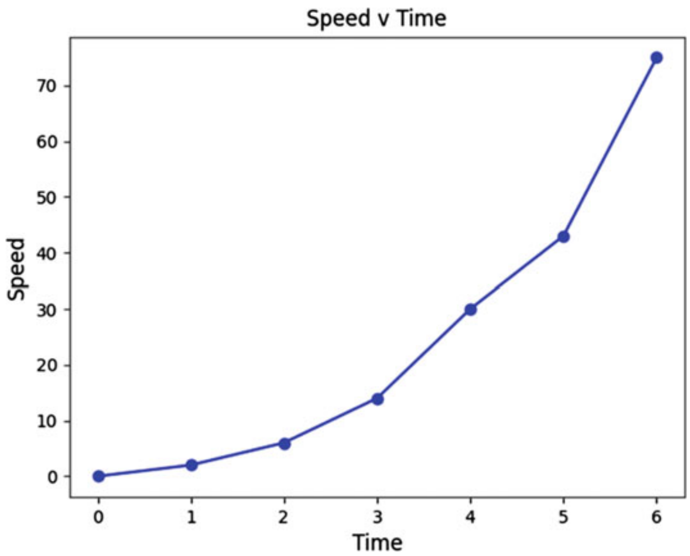
- construct the plot,
- configure labels and axis,
- manage colour and line styles,
- handles events/allows plots to be interactive,
- display (show) the plot.

We will see examples of using the pyplot API in the following sections.

15.3 Line Graphs

A line graph or line plot is a graph with the points on the graph (often referred to as markers) connected by lines to show how something changes in value as some set of values (typically the x axis) changes. For example, over a series to time intervals (also known as a time series). Such line charts are typically drawn in chronological order; such charts are known as run charts.

The following chart is an example of such a run chart; it charts time across the bottom (x axis) against speed (represented by the y axis).



The program used to generate this chart is given below:

```
import matplotlib.pyplot as pyplot

# Set up the data
x = [0, 1, 2, 3, 4, 5, 6]
y = [0, 2, 6, 14, 30, 43, 75]

# Set the axes headings
pyplot.ylabel('Speed', fontsize=12)
pyplot.xlabel('Time', fontsize=12)

# Set the title
pyplot.title("Speed v Time")

# Plot and display the graph
# Using blue circles for markers ('bo')
# and a solid line ('-')
pyplot.plot(x, y, 'bo-')
pyplot.show()
```

The first thing that this program does is to import the `matplotlib.pyplot` module and give it an alias of pyplot (as this is a shorter name it makes the code easier to read).

Two lists of values are then created for the *x* and *y* coordinates of each marker or plot point.

The graph itself is then configured with labels being provided for the *x* and *y* axis (using the pyplot functions `xlabel()` and `ylabel()`). The title of the graph is then set (again using a pyplot function).

After this the *x* and *y* values are then plotted as a line chart on the graph. This is done using the `pyplot.plot()` function. This function can take a wide range of parameters, the only compulsory parameters being the data used to define the plot points. In the above example a third parameter is provided; this is a string 'bo-'. This is a *coded format string* in that each element of the string is meaningful to the `pyplot.plot()` function. The elements of the string are:

- *b*—this indicates the colour to use when drawing the line; in this case the letter '*b*' indicates the colour blue (in the same way '*r*' would indicate red and '*g*' would indicate green).
- *o*—this indicates that each marker (each point being plotted) should be represented by a circle. The lines between the markers then create the line plot.
- '-'—This indicates the line style to use. A single dash ('-') indicates a solid line, where as a double dash ('--') indicates a dashed line.

Finally the program then uses the `show()` function to render the figure on the screen; alternatively `savefig()` could have been used to save the figure to a file.

15.3.1 Coded Format Strings

There are numerous options that can be provided via the format string; the following tables summarises some of these:

The following colour abbreviations are supported by the format string:

Character	Colour
‘*b*’	Blue
‘*g*’	Green
‘*r*’	Red
‘*c*’	Cyan
‘*m*’	Magenta
‘*y*’	Yellow
‘*k*’	Black
‘*w*’	White

Different ways of representing the markers (points on the graph) connected by the lines are also supported including:

Character	Description
‘.’	Point marker
‘,’	Pixel marker
‘*o*’	Circle marker
‘*v*’	Triangle_down marker
‘^’	Triangle_up marker
‘<’	Triangle_left marker
‘>’	Triangle_right marker
‘*s*’	Square marker
‘*p*’	Pentagon marker
‘*’	Star marker
‘*h*’	Hexagon1 marker
‘ + ’	Plus marker
‘*x*’	*x* marker
‘*D*’	Diamond marker

Finally, the format string supports different line styles:

Character	Description
'-'	Solid line style
'--'	Dashed line style
'-.'	Dash-dot line style
':'	Dotted line style

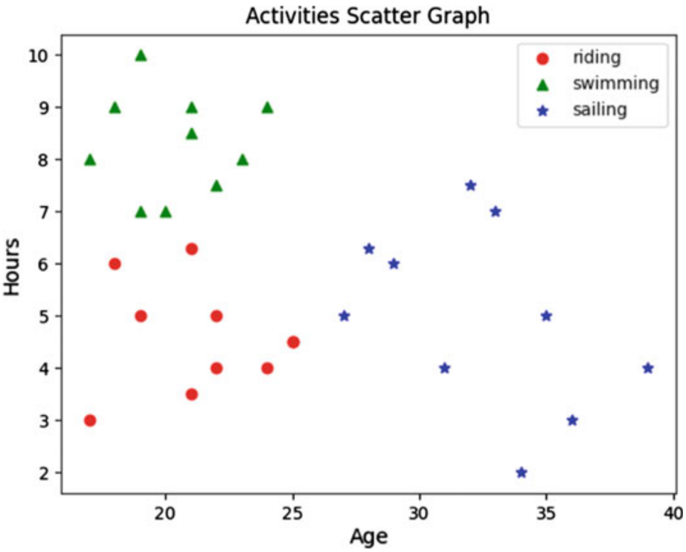
Some examples of formatting strings:

- '*r*' red line with default markers and line style.
- '*g-*' green solid line.
- '--' dashed line with the default colour and default markers.
- 'yo:' yellow dotted line with circle markers.

15.4 Scatter Graph

A scatter graph or scatter plot is type of plot where individual values are indicated using Cartesian (or *x* and *y*) coordinates to display values. Each value is indicated via a mark (such as a circle or triangle) on the graph. They can be used to represent values obtained for two different variables; one plotted on the *x* axis and the other plotted on the *y* axis.

An example of a scatter chart with three sets of scatter values is given below:



In this graph each dot represents the amount of time people of different ages spend on three different activities.

The program that was used to generate the above graph is shown below:

```
import matplotlib.pyplot as pyplot

# Create data
riding = ((17, 18, 21, 22, 19, 21, 25, 22, 25, 24),
      (3, 6, 3.5, 4, 5, 6.3, 4.5, 5, 4.5, 4))
swimming = ((17, 18, 20, 19, 22, 21, 23, 19, 21, 24),
       (8, 9, 7, 10, 7.5, 9, 8, 7, 8.5, 9))
sailing = ((31, 28, 29, 36, 27, 32, 34, 35, 33, 39),
       (4, 6.3, 6, 3, 5, 7.5, 2, 5, 7, 4))

# Plot the data
pyplot.scatter(x=riding[0], y=riding[1], c='red', marker='o',
label='riding')
pyplot.scatter(x=swimming[0], y=swimming[1], c='green',
marker='^', label='swimming')
pyplot.scatter(x=sailing[0], y=sailing[1], c='blue', marker='*',
label='sailing')

# Configure graph
pyplot.xlabel('Age')
pyplot.ylabel('Hours')
pyplot.title('Activities Scatter Graph')
pyplot.legend()

# Display the chart
pyplot.show()
```

In the above example the `plot.scatter()` function is used to generate the scatter graph for the data defined by the riding, swimming and sailing tuples.

The colours of the markers have been specified using the named parameter *c*. This parameter can take a string representing the name of a colour or a two-dimensional array with a single row in which each value in the row represents an RGB colour code. The `marker` Indicates the marker style such as '*o*' for a circle, a '^' for a triangle and '*' for a star shape. The `label` is used in the chart legend for the marker.

Other options available on the `pyplot.scatter()` function include:

- `alpha`: indicates the alpha blending value, between 0 (transparent) and 1 (opaque).
- `linewidths`: which is used to indicate the line width of the marker edges.
- `edgecolors`: indicates the colour to use for the marker edges if different from the fill colour used for the marker (indicates by the parameter '*c*').

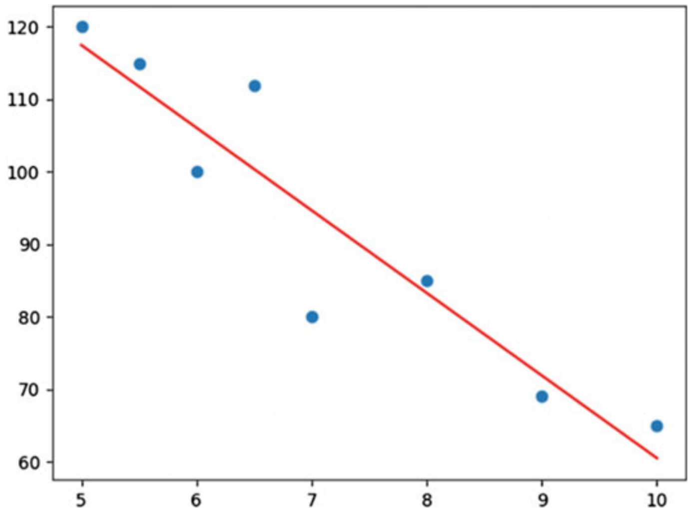
15.4.1 When to Use Scatter Graphs

A useful question to consider is when should a scatter plot be used? In general scatter plots are used when it is necessary to show the relationship between two variables.

Scatter plots are sometimes called correlation plots because they show how two variables are correlated.

In many cases a trend can be discerned around the points plotted on a scatter chart (although there may be outlying values). To help visualise the trend it can be useful to draw a trend line along with the scatter graph. The trend line helps to make the relationship of the scatter plots to the general trend clearer.

The following chart represents a set of values as a scatter graph and draws the trend line of this scatter graph. As can be seen some values are closer to the *trendline* than others.



The trend line has been created in this case using the `numpy` function `polyfit()`.

The `polyfit()` function performs a least squares polynomial fit for the data it is given. A `poly1d` class is then created based on the array returned by `polyfit()`. This class is a one-dimensional polynomial class. It is a convenience class, used to encapsulate "natural" operations on polynomials. The poly1d object is then used to generate a set of values for use with the set of *x* values for the function `pyplot.plot()`.

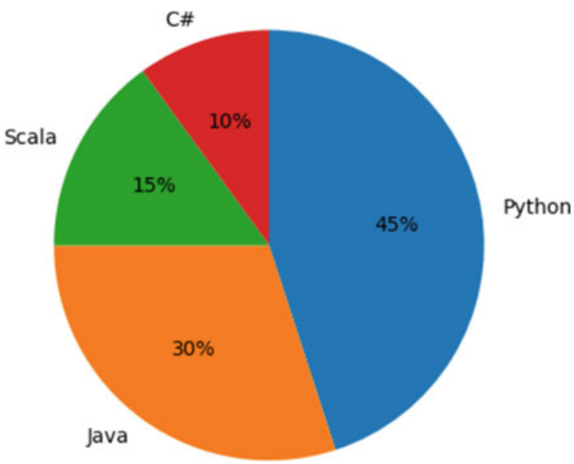
```
import numpy as np
import matplotlib.pyplot as pyplot

x = (5, 5.5, 6, 6.5, 7, 8, 9, 10)
y = (120, 115, 100, 112, 80, 85, 69, 65)

# Generate the scatter plot
pyplot.scatter(x, y)

# Generate the trend line
z = np.polyfit(x, y, 1)
p = np.poly1d(z)
pyplot.plot(x, p(x), 'r')

# Display the figure
pyplot.show()
```

15.5 Pie Charts

A pie chart is a type of graph in which a circle is divided into sectors (or wedges) that each represent a proportion of the whole. A wedge of the circle represents a category's contribution to the overall total. As such the graph resembles a pie that has been cut into different sized slices.

Typically, the different sectors of the pie chart are presented in different colours and are arranged clockwise around the chart in order of magnitude. However, if there is a slice that does not contain a unique category of data but summarises several, for example "other types" or "other answers", then even if it is not the smallest category, it is usual to display it last in order that it does not detract from the named categories of interest.

The following chart illustrates a pie chart used to represent programming language usage within a particular organisation.



The pie chart is created using the `pyplot.pie()` function.

```
import matplotlib.pyplot as pyplot

labels = ('Python', 'Java', 'Scala', 'C#')
sizes = [45, 30, 15, 10]

pyplot.pie(sizes,
       labels=labels,
       autopct='%1.f%%',
       counterclock=False,
       startangle=90)

pyplot.show()
```

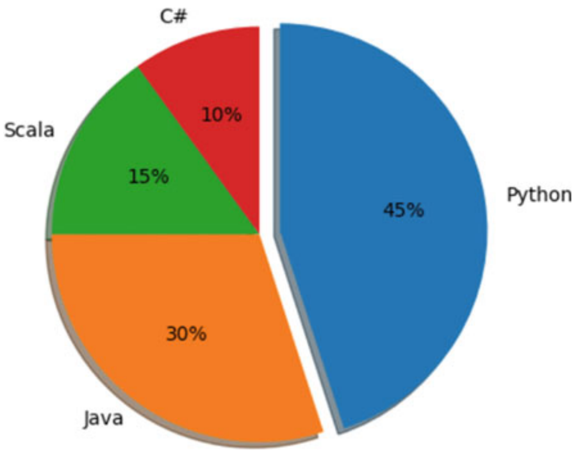
The `pyplot.pie()` function takes several parameters, most of which are optional. The only required parameter is the first one that provides the values to be used for the wedge or segment sizes. The following optional parameters are used in the above example:

- The `labels` parameter is an optional parameter that can take a sequence of strings that are used to provide labels for each wedge.
- The `autopct` parameter takes a string (or function) to be used to format the numeric values used with each wedge.
- The `counterclockwise` parameter. By default wedges are plotted counter clockwise in pyplot and so to ensure that the layout is more like the traditional clockwise approach the `counterclock` parameter is set to `False`.
- The `startangle` parameter. The starting angle has also been moved 90° using the `startangle` parameter so that the first segment starts at the top of the chart.

15.5.1 Expanding Segments

It can be useful to emphasise a particular segment of the pie chart by *exploding* it; that is separating it out from the rest of the pie chart. This can be done using the `explode` parameter of the `pie()` function that takes a sequence of values indicating how much a segment should be explored by.

The visual impact of the pie chart can also be enhanced in this case by adding a shadow to the segments using the named `shadow` Boolean parameter. The effect of these is shown below:



The program that generated this modified chart is given below for reference:

```
import matplotlib.pyplot as pyplot

labels = ('Python', 'Java', 'Scala', 'C#')
sizes = [45, 30, 15, 10]

# only "explode" the 1st slice (i.e. 'Python') explode = (0.1, 0,
0, 0)

pyplot.pie(sizes,
        explode=explode,
        labels=labels,
        autopct='%1.f%%',
        shadow=True,
        counterclock=False,
        startangle=90)

pyplot.show()
```

15.5.2 When to Use Pie Charts

It is useful to consider what data can be/should be presented using a pie chart. In general pie charts are useful for displaying data that can be classified into nominal or ordinal categories. Nominal data is categorised according to descriptive or qualitative information such as program languages, type of car, country of birth. Ordinal data is similar but the categories can also be ranked, for example in a survey people may be asked to say whether they classed something as very poor, poor, fair, good, very good.

Pie charts can also be used to show percentage or proportional data and usually the percentage represented by each category is provided next to the corresponding slice of pie.

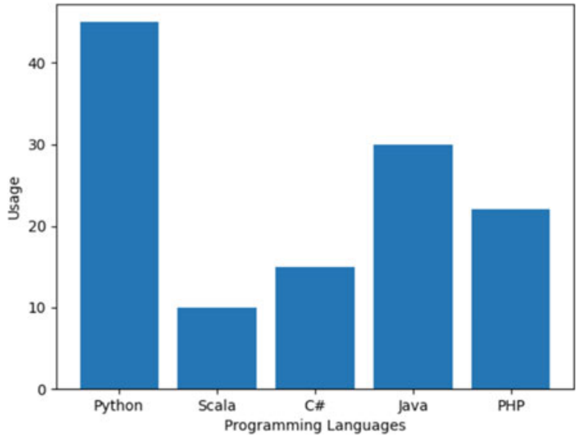
Pie charts are also typically limited to presenting data for six or less categories. When there are more categories it is difficult for the eye to distinguish between the relative sizes of the different sectors and so the chart becomes difficult to interpret.

15.6 Bar Charts

A bar chart is a type of chart or graph that is used to present different discrete categories of data. The data is usually presented vertically although in some cases horizontal bar charts may be used. Each category is represented by a bar whose height (or length) represent the data for that category.

Because it is easy to interpret bar charts, and how each category relates to another, they are one of the most commonly used types of chart. There are also several different common variations such as grouped bar charts and stacked bar charts.

The following is an example of a typical bar chart. Five categories of programming languages are presented along the *x* axis while the *y* axis indicates percentage usage. Each bar then represents the usage percentage associated with each programming language.



The program used to generate the above figure is given below:

```
import matplotlib.pyplot as pyplot

# Set up the data
labels = ('Python', 'Scala', 'C#', 'Java', 'PHP')
index = (1, 2, 3, 4, 5) # provides locations on x axis
```

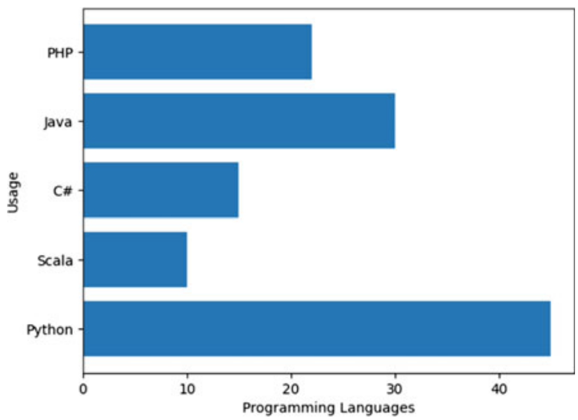
```
sizes = [45, 10, 15, 30, 22]

# Set up the bar chart
pyplot.bar(index, sizes, tick_label=labels)

# Configure the layout
pyplot.ylabel('Usage')
pyplot.xlabel('Programming Languages')

# Display the chart
pyplot.show()
```

The chart is constructed such that the lengths of the different bars are proportional to the size of the category they represent. The *x* axis represents the different categories and so has no scale. In order to emphasise the fact that the categories are discrete, a gap is left between the bars on the *x* axis. The *y* axis does have a scale, and this indicates the units of measurement.

15.6.1 Horizontal Bar Charts

Bar charts are normally drawn so that the bars are vertical which means that the taller the bar, the larger the category. However, it is also possible to draw bar charts so that the bars are horizontal which means that the longer the bar, the larger the category. This is a particularly effective way of presenting a large number of different categories when there is insufficient space to fit all the columns required for a vertical bar chart across the page.

In Matplotlib the `pyplot.barh()` function can be used to generate a horizontal bar chart:

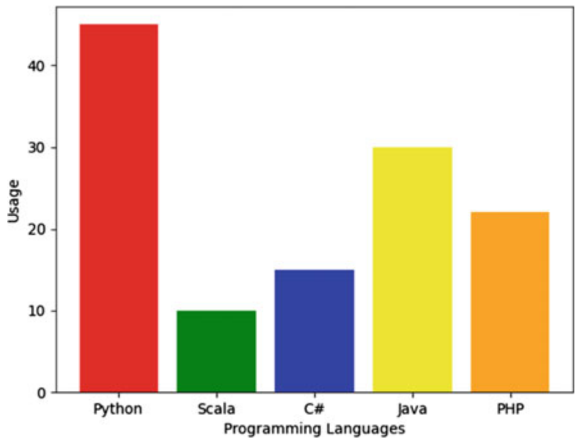


In this case the only line of code to change from the previous example is:

```
pyplot.barh(x_values, sizes, tick_label=labels)
```

15.6.2 Coloured Bars

It is also common to colour different bars in the chart in different colours or using different shades. This can help to distinguish one bar from another. An example is given below:



The colour to be used for each category can be provided via the `color` parameter to the `bar()` (and `barh()`) function. This is a sequence of the colours to apply. For example, the above coloured bar chart can be generated using:

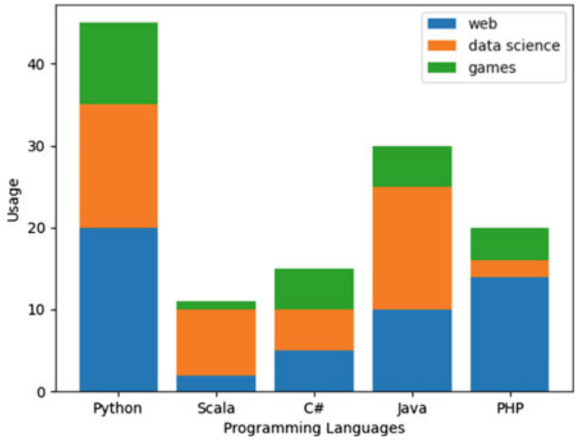
```
pyplot.bar(x_values, sizes, tick_label=labels, color=('red',
'green', 'blue', 'yellow', 'orange'))
```

15.6.3 Stacked Bar Charts

Bar charts can also be stacked. This can be a way of showing total values (and what contributes to those total values) across several categories. That is, it is a way of viewing overall totals, for several different categories based on how different elements contribute to those totals.

Different colours are used for the different subgroups that contribute to the overall bar. In such cases, a legend or key is usually provided to indicate what subgroup each of the shadings/colours represent. The legend can be placed in the plot area or may be located below the chart.

For example, in the following chart the total usage of a particular programming language is composed of its use in games and web development as well as data science analytics.



From this figure we can see how much each use of a programming language contributes to the overall usage of that language. The program that generated this chart is given below:

```
import matplotlib.pyplot as pyplot

# Set up the data
labels = ('Python', 'Scala', 'C#', 'Java', 'PHP')
index = (1, 2, 3, 4, 5)
web_usage = [20, 2, 5, 10, 14]
data_science_usage = [15, 8, 5, 15, 2]
games_usage = [10, 1, 5, 5, 4]

# Set up the bar chart
pyplot.bar(index, web_usage, tick_label=labels, label='web')
pyplot.bar(index,     data_science_usage,     tick_label=labels,
label='data science', bottom=web_usage)

web_and_games_usage = [web_usage[i] + data_science_usage[i] for i
in range(0, len(web_usage))]
pyplot.bar(index, games_usage, tick_label=labels, label='games',
bottom=web_and_games_usage)

# Configure the layout
pyplot.ylabel('Usage')
pyplot.xlabel('Programming Languages')
pyplot.legend()
```

```
# Display the chart
pyplot.show()
```

One thing to note from this example is that after the first set of values are added using the `pyplot.bar()` function, it is necessary to specify the bottom locations for the next set of bars using the `bottom` parameter. We can do this just using the values already used for `web_usage` for the second bar chart; however for the third bar chart we must add the values used for `web_usage` and `data_science_usage` together (in this case using a for list comprehension).

15.6.4 Grouped Bar Charts

Finally, grouped bar charts are a way of showing information about different subgroups of the main categories. In such cases, a legend or key is usually provided to indicate what subgroup each of the shadings/colours represent. The legend can be placed in the plot area or may be located below the chart.

For a particular category separate bar charts are drawn for each of the subgroups. For example, in the following chart the results obtained for two sets of teams across a series of lab exercises are displayed. Thus each team has a bar for lab1, lab2, lab3, etc. A space is left between each category to make it easier to compare the sub categories.

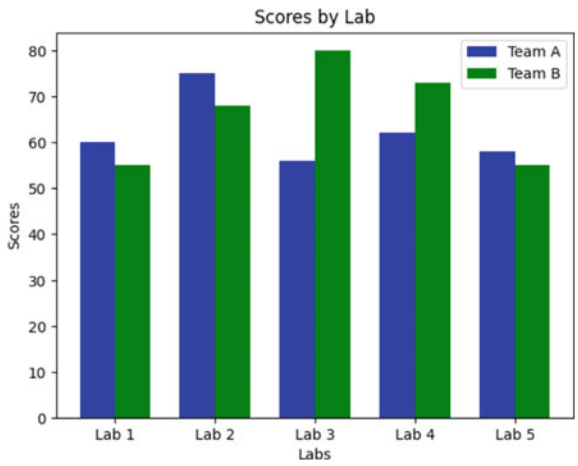
The following program generates the grouped bar chart for the lab exercises example:

```
import matplotlib.pyplot as pyplot

BAR_WIDTH = 0.35

# set up grouped bar charts
teama_results = (60, 75, 56, 62, 58)
teamb_results = (55, 68, 80, 73, 55)
# Set up the index for each bar
index_teama = (1, 2, 3, 4, 5)
index_teamb = [i + BAR_WIDTH for i in index_teama]

# Determine the mid point for the ticks
ticks = [i + BAR_WIDTH / 2 for i in index_teama]
tick_labels = ('Lab 1', 'Lab 2', 'Lab 3', 'Lab 4', 'Lab 5')

# Plot the bar charts
pyplot.bar(index_teama, teama_results, BAR_WIDTH, color='b',
label='Team A')
pyplot.bar(index_teamb, teamb_results, BAR_WIDTH, color='g',
label='Team B')

# Set up the graph
pyplot.xlabel('Labs')
pyplot.ylabel('Scores')
pyplot.title('Scores by Lab')
```

```
pyplot.xticks(ticks, tick_labels)
pyplot.legend()

# Display the graph
pyplot.show()
```

Notice in the above program that it has been necessary to calculate the index for the second team as we want the bars presented next to each other. Thus the index for the teams includes the width of the bar for each index point, thus the first bar is at index position `1.35`, the second at index position `2.35,` etc. Finally the tick positions must therefore be between the two bars and thus is calculated by taking into account the bar widths.

This program generates the following grouped bar chart:



15.7 Figures and Subplots

A Matplotlib figure is the object that contains all the graphical elements displayed on a plot. That is the axes, the legend, the title as well as the line plot or bar chart itself. It thus represents the overall window or page and is the top level graphical component.

In many cases the figure is implicit as the developer interacts with the `pyplot` API; however the figure can be accessed directly if required.

The `matplotlib.pyplot.subplots()` function is a useful function that creates common layouts of subplots, including the enclosing figure object. This function can take optional parameters to indicate the number and organisation of the subplots, for example:

```
fig, axs = plt.subplots(2, 2)
```

Indicates 4 subplots display in a 2 by 2 grid.

This function returns a `matplotlib.figure.Figure` object and a two-dimensional array of axis objects that can be used with the subplots. It is then possible to interact directly with the appropriate axis object. For example it is possible to add subplots and titles for these subplots to the axes.

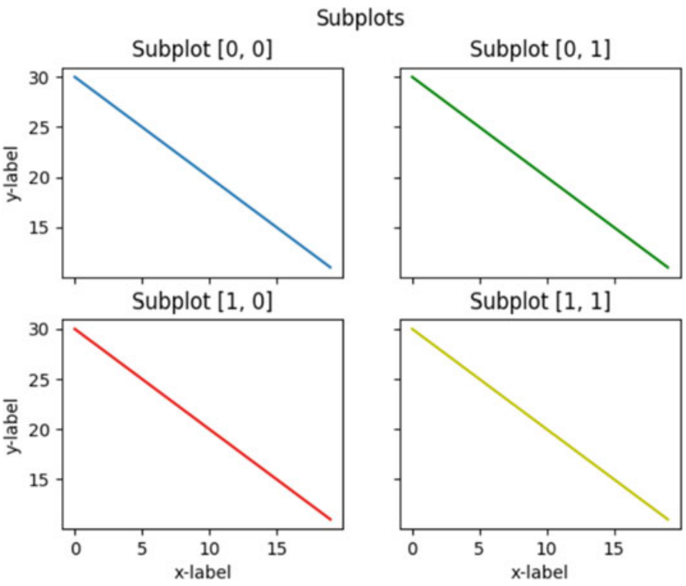
Working directly with the axes is necessary if you want to add multiple subplots to a figure. This can be useful if what is required is to be able to compare different views of the same data side by side. Each subplot has its own axes which can coexist within the figure.

One or more subplots can be added to a figure using the appropriate axis. As the axis is a two-dimensional array (an `ndarray`) it is possible to access each axis individually and add a plot to that axis, for example:

```
axs[0, 0].plot(t, s)
axs[0, 0].set_title('Subplot [0, 0]')
```

This adds a plot to axis in position 0, 0 within the `ndarray` and then sets the title of this plot.

For example, the following figure illustrates four subplots presented within a single figure. Each subplot is added via the `Axes.plot()` method.



This figure is generated by the following program:

```
import matplotlib.pyplot as plt
```

```
# Generate some data to display
t = range(0, 20)
s = range(30, 10, -1)

# Set up the grid of subplots to be 2 by 2
fig, axs = plt.subplots(2, 2)
fig.suptitle('Subplots')

# Add first subplot
print('Adding first subplot to position [0, 0]')
axs[0, 0].plot(t, s)
axs[0, 0].set_title('Subplot [0, 0]')

# Add second subplot
print('Adding second subplot to position [0, 1]')
axs[0, 1].plot(t, s, 'g-')
axs[0, 1].set_title('Subplot [0, 1]')

# Add third subplot
print('Adding third subplot to position [1, 0]')
axs[1, 0].plot(t, s, 'r-')
axs[1, 0].set_title('Subplot [1, 0]')

# Add fourth subplot
print('Adding fourth subplot to position [1, 1]')
axs[1, 1].plot(t, s, 'y-')
axs[1, 1].set_title('Subplot [1, 1]')

# Set up X and y axis labels
for ax in axs.flat:
  ax.set(xlabel='x-label', ylabel='y-label')

# Hide x labels and tick labels for top plots and y ticks for
right plots.
for ax in axs.flat:
  ax.label_outer()

# Display the chart
plt.show()
```

The console output from this program is given below:

```
Adding first subplot to position [0, 0]
Adding second subplot to position [0, 1]
Adding third subplot to position [1, 0]
Adding fourth subplot to position [1, 1]
```

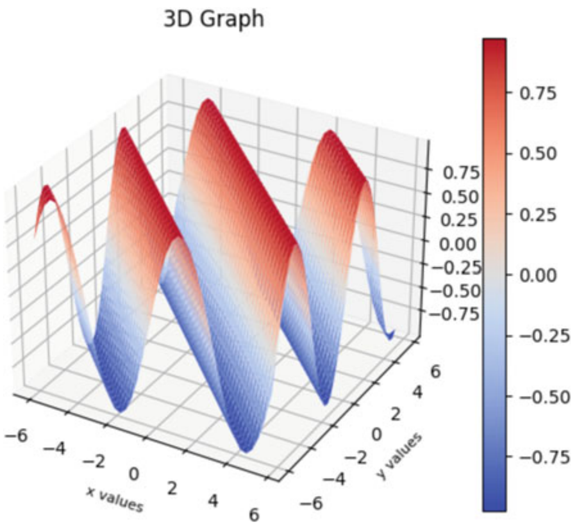
15.8 3D Graphs

A three-dimensional graph is used to plot the relationships between three sets of values (instead of the two used in the examples presented so far in this chapter). In a three-dimensional graph as well as the *x* and *y* axis there is also a *z* axis.

The following program creates a simple 3D graph using two sets of values generates using the bumpy range function. These are then converted into a coordinate matrices using the bumpy `meshgrid()` function. The z axis values are created using the bumpy sin() function. The 3D graph surface is plotted using the plot_ surface function of the futures axes object. This takes the x, y and z coordinates. The function is also given a colour map to use when rendering the surface (in this case the Matplotlib cool to warm colour map is used).

```
import matplotlib.pyplot as pyplot
# Import matplotlib colour map
from matplotlib import cm as colourmap
# Provide access to numpy functions
import numpy as np

# Make the data to be displayed
x_values = np.arange(-6, 6, 0.3)
y_values = np.arange(-6, 6, 0.3)

# Generate coordinate matrices from coordinate vectors
x_values, y_values = np.meshgrid(x_values, y_values)

# Generate Z values as sin of x plus y values
z_values = np.sin(x_values + y_values)

# Obtain the figure object / get the axes object for the 3D
graph
figure, axes = pyplot.subplots(subplot_kw={"projection": "3d"})

# Plot the surface.
surf = axes.plot_surface(x_values,
                         y_values,
                         z_values,
                         cmap=colourmap.coolwarm)

# Add a color bar which maps values to colors.
figure.colorbar(surf)

# Add labels to the graph
pyplot.title("3D Graph")
axes.set_ylabel('y values', fontsize=8)
axes.set_xlabel('x values', fontsize=8)
axes.set_zlabel('z values', fontsize=8)

# Display the graph
pyplot.show()
```

This program generates the following 3D graph:



One point to note about three-dimensional graphs is that they are not universally accepted as being a good way to present the data. One of the maxims of data visualisation is keep it simple/keep it clean. Many consider that a three-dimensional chart does not do this and that it can be difficult to see what is really being shown or that it can be hard to interpret the data appropriately. For example, in the above chart what are the values associated with any of the peaks? This is difficult to determine as it is hard to see where the peaks are relative to the X, Y and Z axis. Many consider such 3D charts to be eye candy; pretty to look at but not proving much information. As such the use of a 3 D chart should be minimised and only used when actually necessary.

15.9 Exercises

The following table provides information on cities in the UK and their populations (note that London has been omitted as its population is so much larger than that of any other city).

City	Population
Bristol	617,280
Cardiff	447,287
Bath	94,782
Liverpool	864,122
Glasgow	591,620

(continued)

(continued)

City	Population
Edinburgh	464,990
Leeds	455,123
Reading	318,014
Swansea	300,352
Manchester	395,515

Using this data create:

1. A scatter plot for the city to population data.
2. A bar chart for the city to population data.

Chapter 16
Graphical User Interfaces

16.1 Introduction

A Graphical User Interface can capture the essence of an idea or a situation, often avoiding the need for a long passage of text. Such interfaces can save a user from the need to learn complex commands. They are less likely to intimidate computer users and can provide a large amount of information quickly in a form which can be easily assimilated by the user.

The widespread use of high-quality graphical interfaces has led many computer users to expect such interfaces to any software they use. Most programming languages either incorporate a graphical user interface (GUI) library or have third-party libraries available.

Python is of course a cross-platform programming language, and this brings in additional complexities as the underlying operating system may provide different windowing facilities depending upon whether the program is running on Unix, Linux, macOS or Windows operating systems.

In this chapter we will first introduce what we mean by a GUI and by WIMP-based UIs in particular. We will then consider the range of libraries available for Python before selecting one to use. This chapter will then describe how to create rich client graphical displays (desktop application) using one of these GUI libraries. Thus in this chapter we consider how windows, buttons, text fields and labels are created, added to windows, positioned and organised.

16.2 GUIs and WIMPS

Graphical user interfaces (GUIs) and Windows, Icons, Mice and Pop-up Menus (WIMP) style interfaces have been available within computer systems for many decades, but they are still one of the most significant developments to have occurred.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_16

These interfaces were originally developed out of a desire to address many of the perceived weaknesses of purely textual interfaces.

The textual interface to an operating system was typified by a peremptory prompt. In Unix/Linux systems for example, the prompt is often merely a single character such as `%, > or $`, which can be intimidating. This is true even for experienced computer users if they are not familiar with the Unix/Linux family of operating systems.

For example, a user wishing to copy a file from one directory to another might have to type something like:

```
> cp file.pdf ~otheruser/projdir/srcdir/newfile.pdf
```

This long sequence needs to be entered with no mistakes in order to be accepted. Any syntax error in this command will cause the system to generate an error message which might or might not be enlightening. Even where systems attempt to be more "user friendly" through features like command histories, much typing of arrow keys and filenames is typically needed.

The main issue on both input and output is one of bandwidth. For example, in situations where the relationships between large amounts of information must be described, it is much easier to assimilate this if output is displayed graphically than if it is displayed as a tables of figures. On input, combinations of mouse actions can be given a meaning that could otherwise only be conveyed by several lines of text.

WIMP stands for Windows (or Window Managers), Icons, Mice and Pop-up menus. WIMP interfaces allow the user to overcome at least some of the weaknesses of their textual counterparts—it is possible to provide a pictorial image of the operating system which can be based on a concept the user can relate to, menus can be used instead of textual commands and information in general can be displayed graphically.

The fundamental concepts presented via a WIMP interface were originally developed at XEROX's Palo Alto Research Center and used on the Xerox Star machine, but gained much wider acceptance through first the Apple Macintosh and then IBM PC implementations of WIMP interfaces.

Most WIMP style environments use a desktop analogy (although this is less true of mobile devices such as phones and tablets):

- the whole screen represents a working surface (a desktop),
- graphic windows that can overlap represent sheets of paper on that desktop,
- graphic objects are used for specific concepts, for example filing cabinets for disks or a waste bin for file disposal (these could be regarded as desk accessories),
- various application programs are displayed on the screen, these stand for tools that you might use on your desktop.

In order to interact with this display, the WIMP user is provided with a mouse (or a light pen or a touch sensitive screen), which can be used to select icons and menus or to manipulate windows.

The software basis of any WIMP style environment is the *window manager*. It controls the multiple, possibly overlapping windows and icons displayed on the screen. It also handles the transfer of information about events which occur in those windows to the appropriate application and generates the various menus and prompts used.

A window is an area of the graphic screen in which a page or piece of a page of information may be displayed; it may display text, graphics or a combination of both. These windows may be overlapping, and associated with the same process, or they may be associated with separate processes. Windows can generally be created, opened, closed, moved and resized.

An icon is a small graphic object that is usually symbolic of an operation or of a larger entity such as an application program or a file. The *opening* of an icon causes either the associated application to execute or the associated window to be displayed.

At the heart of the users ability to interact with such WIMP-based programs is the *event loop*. This loop *listens* for events such as the user clicking a button or selecting a menu item or entering a text field. When such an *event* occurs it triggers the associated behaviour (such as running a function linked with a button).

16.3 Windowing Frameworks for Python

Python is a cross-platform programming language. As such Python programs can be written on one platform (such as a Linux box) and then run on that platform or another operating system platforms (such as Windows or macOS). This can however generate issues for libraries that need to be available across multiple operating system platforms. The area of GUIs is particularly an issue as a library written to exploit features available in the Windows system may not be available (or may look different) on macOS or Linux systems.

Each operating system that Python runs on may have one or more windowing systems written for it and these systems may or may not be available on other operating systems. This makes the job of providing a GUI library for Python that much more difficult.

Developers of Python GUIs have taken one of two approaches to handle this:

- One approach is to write a wrapper that abstracts the underlying GUI facilities so that the developer works at a level above a specific windowing system's facilities. The Python library then maps (as best it can) the facilities to the underlying system that is currently being used.
- The other approach is to provide a closer wrapping to a particular set of facilities on the underlying GUI system and to only target systems that support those facilities.

Some of the libraries available for Python are listed below and have been categorised into platform-independent libraries and platform-specific libraries:

16.3.1 Platform-Independent GUI Libraries

- **Tkinter**. This is a widely used standard Python GUI library. It is built on top of the Tcl/Tk widget set that has been around for very many years for many different operating systems. Tcl stands for Tool Command Language while Tk is the graphical user interface toolkit for Tcl.
- **wxPython**. wxWidgets is a free, highly portable GUI library. It is written in C++, and it can provide a native look and feel on operating systems such as Windows, macOS and Linux. wxPython is a set of Python bindings for wxWidgets. This is the library that we will be using in this chapter.
- **PyQT** or **PySide** both of these libraries wrap the Qt toolkit facilities. Qt is a cross-platform software development system for the implementation of cross-platform GUIs and applications.

16.3.2 Platform-Specific GUI Libraries

1. **PyObjc** is a macOS specific library that provides an Objective-C bridge to the Apple Mac Cocoa GUI libraries.
2. **PythonWin** provides a set of wrappings around the Microsoft Windows Foundation classes and can be used to create Windows-based GUIs.

16.4 Online Resources

There are numerous online references that support the development of GUIs and of Python GUIs in particular, including:

1. https://www.wxpython.org wxPython home page.
2. https://www.tcl.tk for Information on Tcl/Tk.
3. https://www.qt.io For information on the Qt cross-platform software and UI development library.
4. https://wiki.python.org/moin/PyQt For information about PyQt.
5. https://pypi.org/project/PySide/ which provides project information for PySide.
6. https://en.wikipedia.org/wiki/Cocoa_(API) for the Wikipedia page on the MacOS Cocoa library.
7. https://pythonhosted.org/pyobjc/ for information on the Python to Objective-C bridge.
8. https://docs.microsoft.com/en-us/cpp/mfc/mfc-desktop-applications?view=vs-2019 Provides an introduction to the Microsoft Foundation classes.
9. https://www.cgl.ucsf.edu/Outreach/pc204/pythonwin.html for information on PythonWin.

Chapter 17
Tkinter GUI Library

17.1 Introduction

The Tkinter library is a cross-platform GUI library (or toolkit) for Python. It allows programmers to develop highly graphical user interfaces for their programs using common concepts such as menu bars, menus, buttons, fields, panels and frames. This chapter introduces Tkinter. It allows programmers to develop highly graphical user interfaces for their programs using common concepts such as menu bars, menus, buttons, fields, panels and frames. This chapter introduces Tkinter.

17.2 Tkinter

Tkinter is the de-facto standard for creating graphical user interfaces (GUIs) in Python. Although there are other libraries available Tkinter is provided as part of the Python environment and provides all the features required to create desktop applications.

Some of the key features of Tkinter include:

- **Cross-platform**: Tkinter is available on most operating systems, including Windows, macOS, and Linux, making it a platform-independent choice for GUI development.
- **Widget library**: Tkinter provides a set of pre-built GUI elements called widgets, such as buttons, labels, fields, checkboxes, menus and more.
- **Event-driven programming**: Tkinter follows an event-driven programming paradigm common to most GUI libraries in most languages. It allows developers to define event handlers for various user actions, such as button clicks or key presses, which allows for interactive applications. This is the topic of the next chapter.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_17

- **Layout managers**: Tkinter offers different layout managers, such as `pack()`, `grid()`, and `place()`, to control the positioning and organisation of widgets within a window or frame.

17.3 Windows as Objects

In Tkinter, window frames, buttons and text labels as well as their contents are instances of appropriate classes (such as Frame, Button or Label). Thus when you create a window.

- You create an object that knows how to display itself on the computer screen. You must tell it what to display and then tell it to start listening for user input (if appropriate).
- You should bear the following points in mind during your reading of this chapter; they will help you understand what you are required to do:
- You create a window by instantiating a `Tk` object. This is the applications' top most window in a Tk application which typically represents the main window of an application. It is also possible to have other top-level windows represented by the `TopLevel` class.
- You define what the window displays by creating widgets that have an appropriate parent
- A widget can be played out within a window using one or three strategies such as pack().
- You can send messages to the window to change its state, perform an operation and display a graphic object.
- The window, or components within the window, can send messages to other objects in response to user (or program) actions.
- Everything displayed by a window is an instance of a class and is potentially subject to all of the above.
- `tk.Tk` class handles the main event loop of the GUI application.

17.4 Key Concepts

In Tkinter all the elements of a GUI are contained within a Frame held within a top-level windows such as a `Tk` class root window or a `TopLevel` window. These frames contain graphical components known as widgets. These concepts and are others are outlined below:

- `Tk`: The `Tk` class represents the main window or the root window of a Tkinter application. It serves as the container for other widgets and is responsible for managing the application's event loop.

- `Toplevel`: The `Toplevel` class represents additional top-level windows in a Tkinter application. These windows are separate from the root window and can be used for dialog boxes, pop-up windows, or secondary windows within the application.
- `Frame`: The `Frame` class is a container widget that provides a rectangular region to hold other widgets. It is used to group and organise widgets within the main window or other windows.
- Widgets, which are graphical components within a Frame, such as a button, a field, a label, etc.
- `Canvas`: The `Canvas` class provides a drawing area where you can draw graphics, lines, shapes and images.
- Dialogs which are like Frames but provide fewer border controls.

Using these components it is possible to construct complex user interfaces.

17.4.1 The Tk Class

The `Tk` class is the core class used to create a TK window or frame. The class methods and attributes that can be used to customise the main window, handle user events, and manage the application's behaviour. Some commonly used methods and attributes of the `Tk` class include:

- `title()`: Sets the title of the main window.
- `geometry()`: Can be used to set the size and position of the main window.
- `config()`: Configures various properties of the main window.
- `mainloop()`: Starts the Tkinter event loop.
- `destroy()`: Closes the main window and terminates the application.
- `bind()`: Binds an event to a event handler/callback function.

Additionally, the `Tk` class inherits methods and attributes from the Widget class, which is the base class for all Tkinter widgets. This allows the developer to further customise the main window using methods and attributes available to all widgets, such as setting the background colour, adding images or applying styles. By using the `Tk` class and its associated methods and attributes, the developer can create the main window of their Tkinter application and define its behaviour and appearance.

17.4.2 TK Widgets

There are a set of classes within TKinter referred to as widget classes. Each class represents a different type of graphical element such as buttons, labels, menus and list boxes. The widget classes all inherit (or mix in) from the `tk,Misc` class which means that all widgets provide a common set of behaviours and attributes as well as those specific to their type.

When working with a widget it is necessary to create an instance of the corresponding class and link it to its parent widget (which could be a window or a frame or some other container style widget).

17.4.3 The TopLevel Class

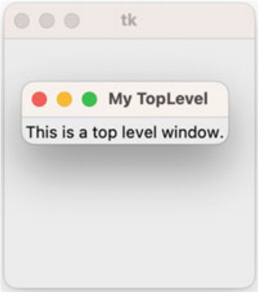
Tkinter also has a class called `tk.TopLevel`. It represents a top-level window or dialog box. It is a container widget that functions as a separate independent window, typically used for creating pop-up windows, dialog boxes or additional top-level windows in your application.

The `tk.TopLevel` class is derived from the `tk.Tk` class, however, unlike the root window, a `TopLevel` window is a separate window with its own title bar and borders.

To create a `TopLevel` window, you need to instantiate the class and specify the parent widget as the first argument, typically the root window or another `TopLevel` window. For example:

```
import tkinter as tk

root = tk.Tk()

# Create a top-level window
top_level = tk.Toplevel(root)
top_level.title("My TopLevel")

# Add widgets to the top-level window
label = tk.Label(top_level, text="This is a top level window.")
label.pack()

root.mainloop()
```

When this program is run it generates two windows as shown below:



In this code, we create a TopLevel instance named `top_level` with `root` as the parent widget. The title of the top-level window is then set to "My TopLevel" using the `title()` method. A `Label` widget is then added to the top-level window to display some text.

`TopLevel` windows are useful for creating additional windows or dialog boxes in a Tkinter application, allowing the developer to display information, prompt for input, or perform specific tasks independently of the main root window.

17.4.4 The Frame Class

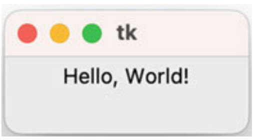
The TK `Frame` class is a widget that represents a rectangular region used to group and organise other widgets within a window. It thus acts as a container or a panel within a window which is very useful when creating hierarchical layouts. This is because each `Frame` can have their own layout management and thus by combining frames together complex UIs can be created.

The `tk.Frame()` constructor can be used to create a new instance of a Frame with respect to parent widget such as the root window. For example:

```
import tkinter as tk

root = tk.Tk()

# Create a frame
frame = tk.Frame(root)
frame.pack()

# Add widgets to the frame
label = tk.Label(frame, text="Hello, World!")
label.pack()

root.mainloop()
```

In this program an initial root window is created using `tk.Tk()`. This root is then used as the parent of the `tk.Frame()` that is subsequently instantiated. The frame is then packed with the root windows. Following this a label is created and that is added to the Frame. The main loop of the top-level window is then started.

The result of running this program is that a simple window is displayed as shown below:



Note that the frame does not by and of its self have a default visual presence within the top-level window.

17.4.5 Dialogs

You can use the `tkinter.simpledialog` or the `tkinter.messagebox` modules to create common dialog types, such as input dialogs, file dialogs or message boxes. These libraries provide pre-defined functions that make it easy to use dialogs in your applications. The `simpledialog` module provides:

- `SimpleDialog`—A simple modal dialog box.
- `Dialog`—a base class for dialogs.
- `askinteger()`—a function to display a dialog to get an integer from the user.
- `askfloat()`—a function to display a dialog to get a float from the user
- `askstring()`—a function to display a dialog to get a string from the user

For example, the following code illustrates creating a simple *Enter Your Name* style dialog:

```
import tkinter as tk
import tkinter.simpledialog as simpledialog

root = tk.Tk()

# Function to display an input dialog
def display_dialog():
   answer = simpledialog.askstring("Name Entry",
                         "Please enter your name:")
   if answer:
      print("Your name is:", answer)
   else:
      print("No input provided.")

# Create a button to show the dialog
button = tk.Button(root, text="Open Dialog", command=display_
dialog)
button.pack()

root.mainloop()
```

This program imports `tkinter.simpledialog` and use the `askstring()` function to display an input dialog box. The function takes two arguments: the *title* of the dialog box and the *prompt* message. It returns the user's input as a string or `None` if no input is provided. If the user entered their name it is printed out, if the user did not enter their name then a message telling the user that no input was provided is printed to the console.

When this program is run it generates a window containing a single button 'Open Dialog'. When you click on this button a 'dialog' is opened that will ask the user to enter their name (as a string), for example:

When the user enters their name it is then printed out to the console:

```
Your name is: John
```

17.4.6 The Canvas Class

The `Canvas` class in Tkinter provides a 2D drawing area that can be used to create and manipulate graphical elements such as lines, shapes, images and text.

To use the `Canvas` class, it is first necessary to create an instance of the `Canvas` and specify its parent container, usually the main window or a TopLevel window. For example:

```
import tkinter as tk

window = tk.Tk()
canvas = tk.Canvas(window, width=400, height=300)
canvas.pack()
```

In this example, we have created an instance of the `Canvas` class, passing the root `window` as the parent and specifying the desired `width` and `height` of the canvas. We then use the `pack()` method to layout and display the canvas within the main window.

Once the canvas is created, it can be used to draw and manipulate graphical elements suing methods such as:

- `create_line()`: Draws a straight line between two coordinates on the receiving canvas.
- `create_rectangle()`: Draws a rectangle specified by its top-left and bottom-right coordinates.
- `create_oval()`: Draws an oval specified by its bounding box coordinates.
- `create_polygon()`: Draws a polygon specified by a list of coordinates.
- `create_image()`: Displays an image on the canvas.
- `create_text()`: Places text on the canvas.
- As an example off using the canvas to display a graphical element, the following program draws a rectangle and a line as specific positions within the canvas:

```
# Draw a rectangle
canvas.create_rectangle(50, 50, 200, 150, fill='red')

# Draw a line
canvas.create_line(100, 100, 300, 200, fill='blue', width=3)

root.mainloop()
```

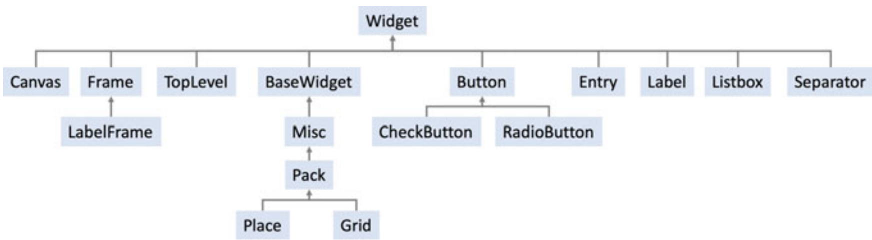
In this example, we use the `create_rectangle()` method to draw a red rectangle and the `create_line()` method to draw a blue line on the canvas. The coordinates passed to these methods specify the position and dimensions of the shapes.

The `Canvas` class provides additional methods for controlling the appearance and behaviour of the canvas and its elements. For example, you can change the colour, outline or fill of a shape, apply transformations such as scaling or rotation, handle events like mouse clicks or movements on the canvas and more.

Using the `Canvas` class it is possible to create complex graphics, interactive visualisations or custom drawing tools within your Tkinter application.

17.5 The Class Inheritance Hierarchy

The following diagram illustrates part of the inheritance tree for the graphical Tk components (or widgets).



At the top of the hierarchy is the `Widget` class, which serves as the base class for all Tkinter widgets. It provides the basic functionality and attributes shared by all widgets.

The `Widget` class is then subclassed by other intermediate classes such as `BaseWidget`, `Misc`, `TopLevel`, `Frame`, `Button` and `Canvas`, which further define common behaviour and attributes for groups of related widgets.

Below these intermediate classes, you find specific widget classes like `CheckButton`, `RadioButton` or `LabelFrame` which is a subclass of `Frame`.

Additionally, there are widget classes related to menus such as `Menu`.

Understanding the inheritance hierarchy can be useful when working with Tkinter, as it helps in identifying common methods, attributes and behaviours shared by groups of widgets and allows you to leverage the specific features and customisation options provided by each widget class.

17.5.1 Layout Management

All Tkinter widgets have access to specific geometry management methods, which have the purpose of organising widgets throughout the parent widget area. Tkinter exposes the following geometry manager classes: pack, grid and place.

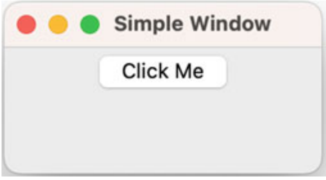
- `pack()` This geometry manager organises widgets in blocks before placing them in the parent widget.
- `grid()` This geometry manager organises widgets in a table-like structure in the parent widget.
- `place()` This geometry manager organises widgets by placing them in a specific position in the parent widget.

17.6 A Simple Example

To illustrate the basic usage of Tkinter, consider the following code that creates a window containing a simple button.

```
import tkinter as tk

def button_click():
   """ function to be run when button is clicked"""
   print("Button clicked!")

# Set up the window and the button within the window
window = tk.Tk()

window.geometry("200x80")

# Set the title of the tkinter window
window.title('Simple Window')

# Add a button to the window
button = tk.Button(window,
             text="Click Me",
             command=button_click)
button.pack()

# Start the main GUI processing loop
window.mainloop()
```

When this program is run the following window is displayed:



If the user clicks on the 'Click Me' button a message 'button clicked!' will be printed to the standard out, for example if we click the button 3 times we will see the following in the output console:

```
Button clicked!
Button clicked!
Button clicked!
```

The above code imports the `tkinter` module and creates a window using `Tk()`. It then sets the size of the window using `geometry()` and a string describing the dimensions of the window. Following this the title of the window is set using the `title()` method. The program then creates a button using `Button()` and associates it with a function `button_click` to handle the button click event. The `pack()` method is used to manage the layout of the button within the window. In this case there is only the button and so the layout is very simple. Finally, the `mainloop()` method is called to start the Tkinter GUI event loop that will listen for user events and trigger their related behaviours.

17.7 Tkinter Installation

Tkinter is typically included with the standard Python installation on macOS, so you don't need to install it separately.

17.7.1 Mac Installation

You may need to ensure that the Tcl/Tk libraries, which Tkinter relies on, are installed on your Mac. Most Unix/Linux operating system distributions, as well as Mac OS X, include Tcl/Tk. If not already installed, you can use your system's package manager to install the appropriate packages.

Here's how you can check and install Tkinter-related dependencies:

1. Open Terminal on your Mac.
2. Check if Tcl/Tk is installed by running the following command:

```
which wish
```

If Tcl/Tk is installed, the command will display the path to the Tcl/Tk interpreter.

3. If Tcl/Tk is not installed, you can install it using Homebrew, a popular package manager for macOS. Install Homebrew by executing the following command in Terminal:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/HEAD/install.sh)"
```

4. Once Homebrew is installed, run the following command to install Tcl/Tk:

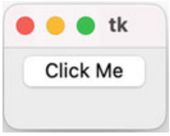
```
brew install tcl-tk
```

This command will install the Tcl/Tk libraries on your system.

After ensuring that Tcl/Tk is installed, you can start using Tkinter in your Python programs without any additional installation steps. To check that this is working try from the command line:

```
python -m tkinter
```

This should open a window with a sample tkinter application to verify everything is ok:



If you are still having a problem in your environment then it may be necessary to install the tkinter package on your Mac, this can be done again using homebrew via:

```
brew install python-tk
```

Now retest your system to ensure that you can run the tkinter sample application.

17.7.2 Windows Installation

On Windows, Tkinter is typically included with the standard Python installation, so you don't need to install it separately.

17.8 GUI Builders for Tkinter

Programming a GUI by hand can provide very powerful and flexible UIs; however, it is also time consuming and can be error prone. To alleviate this there are so called GUI builders available for Tkinter that allow you to essentially ‘draw’ the UI and then the GUI builder can generate the underlying code for you.

In general, these types of tools provide a visual interface for creating and arranging widgets, setting properties, and generating the corresponding Tkinter code.

A few of the more popular GUI builders/editors for Tkinter are:

- **Pygubu**: Pygubu is a simple GUI builder that allows the developer to visually design their GUI by dragging and dropping widgets onto a canvas, setting properties, and assigning event handlers. Pygubu generates the corresponding Python code, which can then be used within an application.
- **PAGE**: Python Automatic GUI Generator (PAGE) is another GUI builder for Tkinter. It provides a visual environment for designing GUIs and generates the corresponding Python code. PAGE offers a variety of widgets and properties, and it supports events and event binding.
- **Qt Designer with PyQt**: While not specific to Tkinter, you can use Qt Designer, a visual GUI builder for PyQt (a Python binding for the popular Qt framework), to design your GUI. PyQt provides a bridge between Tkinter and Qt, allowing you to use the Qt Designer tool to create the GUI and then convert the generated ‘.ui’ file to Python code that works with Tkinter.

GUI builders can save time and make it easier to create complex layouts and arrangements of widgets. However, bear in mind that learning the basics of Tkinter and writing code manually can provide you with more flexibility and control over your GUI application.

17.9 Online Resources

- https://docs.python.org/3/library/tkinter.html Main tkinter documentation.
- https://realpython.com/python-gui-tkinter/ Tutorial on Tkinter.
- https://tkdocs.com/tutorial/widgets.html Tutorial on the basic widgets.
- https://www.geeksforgeeks.org/what-are-widgets-in-tkinter/ Another tutorial on widgets in Tkinter.
- https://github.com/alejandroautalan/pygubu Pygubu reference.
- https://sourceforge.net/projects/page/ SourceForge site for PAGE.
- https://pythonbasics.org/qt-designer-python/ Qt Designer tutorial.

17.10 Exercises

Create a GUI application that has button and label. When the user clicks on the button, a dialog should be displayed asking the user to enter their name. The field should then be populated with the value the user provided.

For example, the window might initial look like:

When the user clicks on the button they should be prompted to enter their name:

If they enter their name the value should then update the display, for example if they enter 'John', then the display should now update to:

Note that to create the label you can use:

```
label = tk.Label(window, text="<<...>>")
label.pack()
```

And to update the label once it has been created (for example within your button click function) you can use the config method:

```
label.config(text=answer)
```

Chapter 18
Events in Tkinter User Interfaces

18.1 Introduction

This chapter introduces event handling within a GUI application and Tkinter's event handling mechanism in particular. It outlines what an event is, what event handlers are, and how events are bound to event handlers. It consider what types of events there are in Tkinter. It then takes the reader through how a simple application is built to handle several different events.

18.2 Event Handling

Events are an integral part of any GUI; they represent user interactions with the interface such as clicking on a button, entering text into a field and selecting a menu option.

The main *event loo*p of GUI listens for an event; when one occurs it processes that event (which usually results in a function or method being called) and then waits for the next event to happen. This loop is initiated in Tkinter via the call to the `mainloop()` method on the `tk.Tk` object.

This raises the question 'what is an Event?' An event object is a piece of information representing some interaction that occurred typically with the GUI (although an event can be generated by anything). An event is processed by an event handler. This is a method or function that is called when the event occurs. The event is passed to the handler as a parameter. An event binder is used to bind an event to an event handler.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_18

18.3 What is Event Handling?

In Tkinter, an event refers to a specific action that can happen within a graphical user interface (GUI). It can be triggered by user interactions, such as clicking a button or pressing a key, or by system events, such as window resizing or mouse movements.

Events are the primary means by which Tkinter communicates user actions or system changes to the application, allowing that application to respond and handle those actions appropriately.

When an event occurs, Tkinter generates an event object that contains information about the event, such as the event type, widget involved, mouse coordinates or key details.

For example, some of the common Tkinter events are:

- Button Click Event ("<Button-1>"): Occurs when the left mouse button is clicked.
- Key Press Event ("<Key>"): Occurs when a key on the keyboard is pressed.
- Mouse Motion Event ("<Motion>"): Occurs when the mouse moves within a widget or window.
- Window Close Event ("<Destroy>"): Occurs when a window is closed.

Events are bound to specific widgets using the 'bind()' method (the binder). By specifying the event type and the associated event handler function, a developer can define how the application should respond when that event occurs.

For example, you a Button Click Event can be bound to button widget for the left mouse button such that then it is clicked a function will be run. Similarly, you can bind the Key Press Event ("<Key>") to a widget or the root window and handle different key presses by calling specific functions.

When the event is triggered, Tkinter invokes the event handler function associated with that event, passing an event object as an argument. The event object provides access to event-specific information, allowing you to determine the type of event, the widget involved, or extract relevant details.

By understanding events and their associated event types, you can capture and respond to user interactions and system changes in a Tkinter application, making it interactive and responsive to user actions.

18.4 What Are Event Handlers?

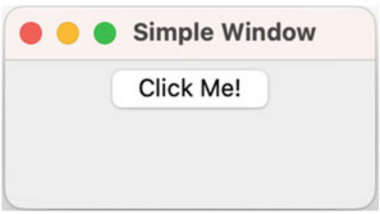
Event handlers, also known as event callback functions or event listeners, are functions that are associated with specific events in a graphical user interface (GUI) application. They are responsible for handling and responding to events when they occur.

As a programmer this means that you will associate a function to be invoked with a widget using the `bind()` method. When the associated event is triggered, the function will be called. Tkinter will pass into the function an object representing the event as an argument. This parameter represents the event object and provides information about the event that occurred, such as the event type, the widget involved or additional details like mouse coordinates or key information.

Here’s an example of an event handler function for a button click event in Tkinter. The button is displayed winning the main Tk window:

```
import tkinter as tk
from datetime import datetime

def button_click(event):
   print('Button clicked!')
   print(event)
   print(f'Event type: {event.type}')
   print(f'Widget: {event.widget}')
   print(f'Time: {datetime.fromtimestamp(event.time / 1e3)}')

# Create main window
window = tk.Tk()
# Set the size of the window
window.geometry('200x80')
# Set the title of the window
window.title('Simple Window')

# Create button
button = tk.Button(window,
                   text="Click Me!",
                    name='my button')
button.pack()

# Bind the button_click function to the <Button-1> event
button.bind("<Button-1>", button_click)

window.mainloop()
```

When this program is run it displays a window with a single button in it as shown below:



When the user clicks on the button the following output is printed into the console (although the date will be different):

```
Button clicked!
<ButtonPress event num=1 x=63 y=16>
Event type: 4
Widget: .my button
Time: 2023-06-15 10:48:51.181000
```

The program defines a function `button_click(event)` which will be called when the left mouse button is used to click the button in the window. The function prints out the event (using the default print for events). This indicates that it is a button press event at location *x* 52 and *y* 12. The function then accesses several properties of the event such as the type of the event, the widget that generated the event and the timestamp of the event (which is then formatted into a date time object).

Note that in this example the button has been given an explicit name so that we can see this returned when we access the `event.widget` property.

The function is *bound* to the button using the `bind()` method and the event binder (which links the widget to the function for a specific event) is the "`<Button-1>`" event binder. When the button is clicked, Tkinter calls the `button_clicked` function and passes the event object as the argument.

18.5 Event Binders

Event binders are used to associate an event handler (for example the `button_clicked()` function above) with a *specific* event for a *particular* widget.

Event binders allow a widget to have multiple event handlers bound that will handle different event types. Thus a widget might have event handlers for when they are in focus or lose focus as well as for key presses, etc. Each appropriate function would be bound to the widget by a specific event binder.

The main method used to bind event handlers to widget for specific event bindings is the `bind()` method. The `bind()` method takes two or more arguments, the key two are:

- **The event binding** (the first parameter). This is a string representing the event type (or a sequence of event types) used for the binding. The value can be a specific event such as <Button1> or a sequence of events separated by a space such as "<Button-1><Button-3>" for the left and right mouse button clicks.
- **The event handler function** (the second argument). This is the function that will be invoked when the event is raised on the widget. Note it should be the name of the function rather than a direct invocation of that function.

As well as the `bind()` method, Tkinter providers an `unbind()` method. This allows a event handler to be removed from a widget. It also provides an `unbind_all()` method which removes all event handlers from a widget.

18.6 Virtual Events

One point to note is that built-in operating system generated events in Tkinter are all defined using a single set of brackets such as '`<KeyPress>`' or '`<Button1>`'. Most of the built-in events are directly tied to actual physical events and follow this format.

However, you may come across other events that have a double bracket name. These are called virtual events. They do not necessarily represent a physical event and often (although not always) are specific to a particular type of widget. For example the '`<<ListboxSelect>>`' is only used by the listbox.

Virtual events can be triggered by a combination of other events using the `event_add()` widget method, though they can also be generated by calling event_generate.

For example:

```
widget.event_add("<<abc>>")
```

18.7 Event Definitions

It is useful to summarise the definitions around events as the terminology used can be confusing and is very similar:

- **Event** represents information from the underlying GUI framework that describes something that has happened and any associated data. The specific data available will differ depending on what has occurred. For example, if a window has been moved then the associated data will relate to the window's new location.
- **Event Loop** the main processing loop of the GUI that waits for an event to occur. When an event occurs the associated event handler is called.
- **Event Handlers** these are methods (or functions) that are called when an event occurs.
- **Event Binders** associate a type of event with an event handler.

The relationship between the event, the event handler via the event binder is illustrated below:

18.8 What Types of Event Are There?

Tkinter provides a wide range of event types that you can capture and handle in your graphical user interface (GUI) application. These events cover various user interactions and system changes. Here are some common event types in Tkinter:

- **Mouse Events**. There are a range of mouse events that can be handled including those associated with:
- Button Click: <Button-1>, <Button-2>, <Button-3> (left, middle, and right mouse button clicks)
- Button Release: <ButtonRelease-1>, <ButtonRelease-2>, <ButtonRelease-3>
- Mouse Motion: <Motion>
- Mouse Scroll: <MouseWheel>

Key Events. These are associated with keys on the keyboard. They include:

- Key Press: <KeyPress>
- Key Release: <KeyRelease>
- Individual Key Events: <KeyPress-a>, <KeyRelease-Enter>, etc.

Focus Events. These events relate to a widget gaining or loosing focus in the UI:

- Focus In: <FocusIn>
- Focus Out: <FocusOut>

Window Events. Window events relate to events at the window level such as resizing and closing windows:

- Window Resize: <Configure >
- Window Close: <Destroy >

Widget-Specific Events. These are events associated with a specific type of widget such as an entry field, listbox or combo box:

- Entry Field Edit: <Key > events on an Entry widget
- Listbox Selection: <ListboxSelect >

Timer Events. These are events that are triggered after a period of time, for example every widget has an after method that will generate an event at a specific time interval from the time it is called. The method takes at least two arguments: the amount of time (in milliseconds) to wait before generating the event, and the callback function to call after the time has elapsed.

- Timer Trigger: <Timer >

These are just a few examples of event types available in Tkinter. Each event type has its own specific format and can be bound to widgets using the 'bind()' method.

18.9 Binding an Event to an Event Handler

An event is bound to an event handler using the `bind()` method of an event generating object (such as a button, field, menu item) via a named event binder such as “`<Button-1>`”.

For example:

```
import tkinter as tk

def button_click(event):
    print("Button clicked!")

root = tk.Tk()

button = tk.Button(root, text="Click Me!")
button.pack()
button.bind("<Button-1>", button_click)

root.mainloop()
```

18.10 Implementing Event Handling

There are four steps involved in implementing event handling for a widget or window, these are:

1. **Identify the event of interest**. Many widgets will generate different events in different situations; it may therefore be necessary to determine which event you are interested in.
2. **Find the correct Event Binder name**, e.g. <Button-1>, <MouseWheel >, <KeyPress> etc. Again you may find that the widget you are interested in supports numerous different event binders which may be used in different situations (even for the same event).
3. **Implement an event handler** that will be called when the event occurs. The event handler will be supplied with the event object.
4. **Bind the Event to the Event Handler** via the Binder Name using the `bind()` method of the event generating widget.

To illustrate this we will use a simple example.

We will write a very simple event handling application. This application will have a `Tk` window containing a Button, a Label and an Entry.

We will define a set of function that can be called for different events. These functions will react to the button click event (as we have seen above). However we will also define a function to be used with a key press and two functions to be used with focus (being gained and lost).

We will associate the `button_click` function with the <`Button1` >event of the `Button` widget. For example:

```
button = tk.Button(window, text='Click Me!')
button.pack()
button.bind('<Button-1>', button_click)
```

However, we will associate the other functions all with the `Entry` widget. Each function will be bound to a different event so that they will be called in different situations. We thus need to determine the correct bindings for the event handler functions.

A widget can support a wide range of events, and we can therefore choose which event bindings are relevant. In this case we will select:

- the '`<Key>`' binding for the `key_pressed` function,
- the '`<FocusIn>`' binding for the `focus_gained` function,
- and the '`<FocusOut>`' binding for the `focus_lost` function.

For example:

```
entry = tk.Entry(window, bd =5)
entry.pack(side = tk.RIGHT)
entry.bind('<FocusIn>', focus_gained)
entry.bind('<FocusOut>', focus_lost)
entry.bind('<Key>', key_press)
```

This illustrates how a single widget can invoke different event handler functions depending upon the type of event generated.

In this case we are also indicating to the `pack()` method that we would like it to orient the widget to the right of the container/parent widget (in this case the window).

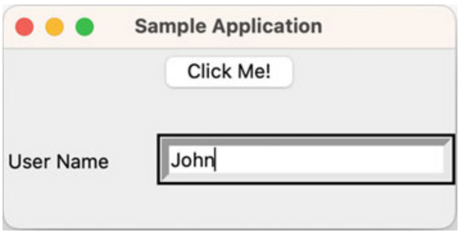
The end result is the program shown below:

```
import tkinter as tk

def button_click(event):
    print(f'Button clicked: {event}')

def key_press(event):
    print("Key pressed:", event.char)

def focus_lost(event):
    print(f'Widget {event.widget} lost focus')

def focus_gained(event):
    print(f'Widget {event.widget} gained focus')

# Create main window
window = tk.Tk()
# Set the size of the window
window.geometry('300x120')
# Set the title of the window
window.title('Sample Application')

button = tk.Button(window, text='Click Me!')
```

```
button.pack()
button.bind('<Button-1>', button_click)

label = tk.Label(window, text='User Name')
label.pack( side = tk.LEFT)

entry = tk.Entry(window, bd =5)
entry.pack(side = tk.RIGHT)
entry.bind('<FocusIn>', focus_gained)
entry.bind('<FocusOut>', focus_lost)
entry.bind('<Key>', key_press)

window.mainloop()
```

When this program is run the window is displayed with the button at the top of the display and the User Name Label and the Entry input field below.

If the user then clicks the left mouse button on the button then a message will be printed out. If they then click on the entry field the `focus_gained` function will be invoked, if they type anything into the field then each key press will be echoed to the output console. Thus if we end up with:



Then the output in the console will be:

```
Button clicked: <ButtonPress event num=1 x=47 y=14>
Widget .!entry gained focus
Key pressed:
Key pressed: J
Key pressed: o
Key pressed: h
Key pressed: n
Widget .!entry lost focus
```

18.11 An Interactive GUI Application

As an example of a slightly larger GUI application, that brings together many of the ideas presented in this chapter and the previous one, is given below.

In this application we have a text input field (a `tk.Entry`) that allows a user to enter their name. When they click on the Enter button (`tk.Button`) a dialog is displayed allowing them to enter they name. The value entered is then used to update the contents of the Entry field.

Note that for the `Entry` field we have used a `StringVar` as the text variable as this simplifies updating the on screen field. Using the text variable we can *set* the `Entry` value using the `set()` method. If we interacted with the `Entry` directly then we would have to use delete and insert methods to remove the existing content and replace it with a new value, for example:

```
e.delete(0,tk.END)
e.insert(0,text_to_add)
```

This is only necessary as we want to overwrite any existing values within the `Entry` field.

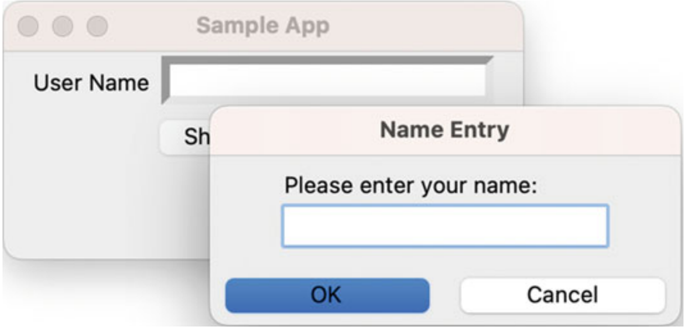
The code used to implement this GUI application is given below:

```
import tkinter as tk
import tkinter.simpledialog as simpledialog

def key_press(event):
   print("Key pressed:", event.char)

def focus_lost(event):
   print(f'Widget {event.widget} lost focus')

def focus_gained(event):
   print(f'Widget {event.widget} gained focus')

# Create main window
window = tk.Tk()
# Set the size of the window
window.geometry('300x120')
# Set the title of the window
```

```
window.title('Sample App')

# Create a frame
frame = tk.Frame(window)
frame.pack()

label = tk.Label(frame, text='User Name')
label.pack( side = tk.LEFT)

# Set up a 'ext variable' to use with the entry field
# Makes setting it programmatically easier (don't need to
delete and insert)
entry_text = tk.StringVar()
entry = tk.Entry(frame, textvariable=entry_text, bd =5)
entry.pack(side = tk.RIGHT)
entry.bind('<FocusIn>', focus_gained)
entry.bind('<FocusOut>', focus_lost)
entry.bind('<Key>', key_press)

def button_click(event):
  answer = simpledialog.askstring("Name Entry", "Please enter your
name:")
   if answer:
      print("Your name is:", answer)
      entry_text.set(answer)
   else:
      print("No input provided.")

button = tk.Button(window, text='Show Message')
button.pack()
button.bind('<Button-1>', button_click)

window.mainloop()
```

When the application is run then a window containing the various widgets will be displayed to the user:

If the user clicks on the 'Show Message' button then the `tkinter.simpledialog` will display prompt to the user asking for their name:



If the user then enters their name, such as John:

And then clicks on 'OK', the Entry field will be populated with their name:

Of course if the user clicks on the `Entry` field and types into it, then the event handlers for the entry field will be triggered, such as the key press and focus gained and lost event handlers. For example, if they type hunt into the entry field after John, then the output in the console would be:

```
Widget .!frame.!entry gained focus
Key pressed:
Key pressed: H
Key pressed: u
Key pressed: n
Key pressed: t
Widget .!frame.!entry lost focus
```

18.12 Online Resources

There are numerous online references that support the development of GUIs and of Python GUIs in particular, including:

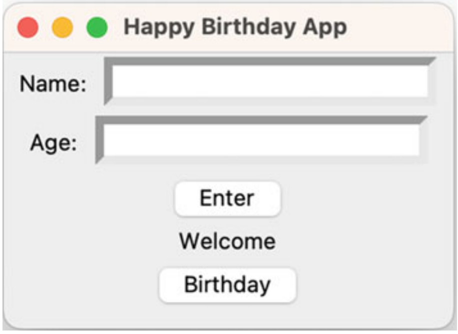
- https://docs.python.org/3/library/tkinter.html Main tkinter documentation.
- https://pythonprogramming.net/tkinter-tutorial-python-3-event-handling a tutorial on Tkinter event handling.

18.13 Exercises

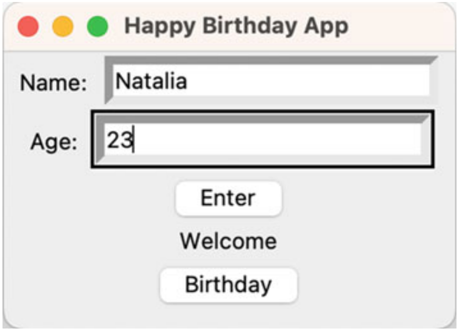
The application should allow a user to enter their name and age. You will need to check that the value entered into the age field is a numeric value (for example using `isnumeric()`). If the value is not a number then an error message dialog should be displayed.

A button should be provided labelled ‘Birthday’; when clicked it should increment the age by one and display a Happy Birthday message. The age should be updated within the GUI.

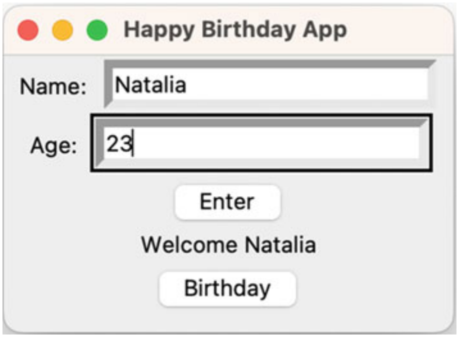
An example of the user interface you created in the last chapter is given below:



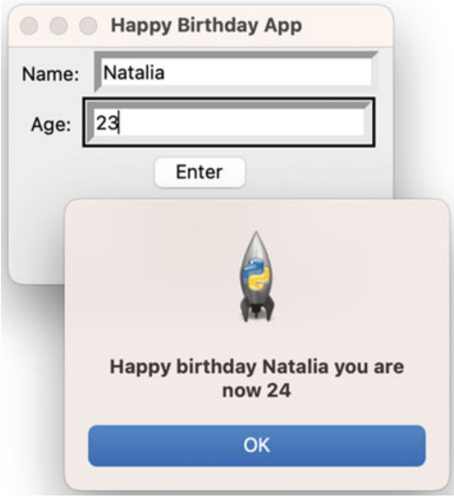
As an example, the user might enter their name and age as shown below:



If they press enter then the welcome message should update with their name:



When the user clickffigs on the 'birthday' button then the Happy Birthday message dialog is displayed:



Note look at from tkinter import messagebox as there are several useful simple to use dialogs available in this module such as:

```
messagebox.showerror('Error', msg)
```

And

```
messagebox.showinfo("Birthday", msg)
```

Chapter 19
PyDraw Tkinter Example Application

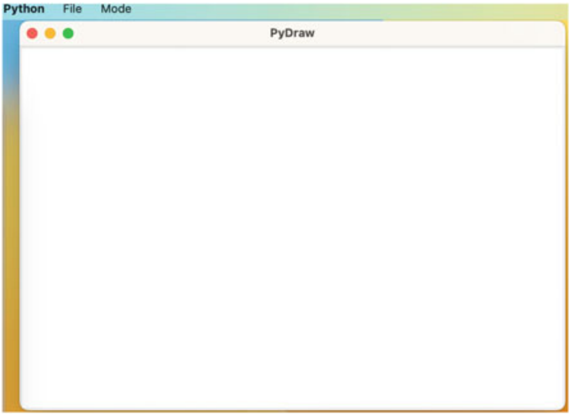
19.1 Introduction

This chapter builds on the GUI library presented in the last two chapters to illustrate how a larger application can be built. It presents a case study of a drawing tool akin to a tool such as Visio.

19.2 The PyDraw Application

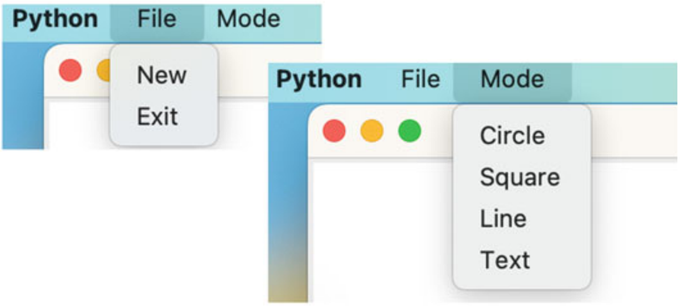
The PyDraw application allows a user to draw diagrams using squares, circles, lines and text. At present there is no select, resize, reposition or delete option available (although these could be added if required). PyDraw is implemented using the TkInter set of components. Below we can see the PyDraw application running on a Mac:

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_19



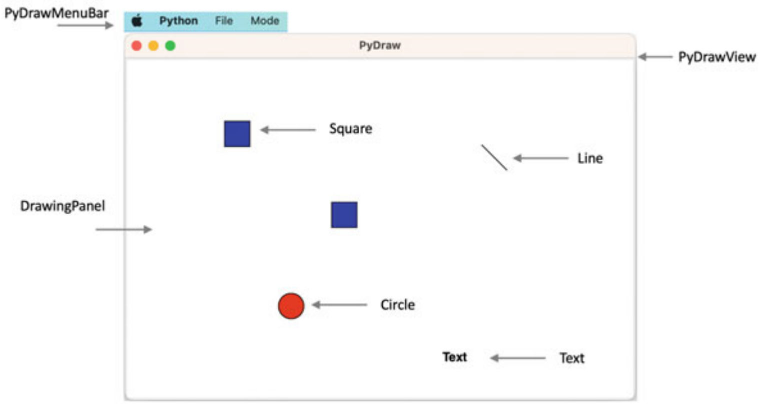
When a user starts the PyDraw application, they see the interface shown above. Depending on the operating system it has a menu bar across the top (on a Mac this menu bar is at the Top of the Mac display) and a drawing area below that.

There are two menus defined for the application, File and Mode:

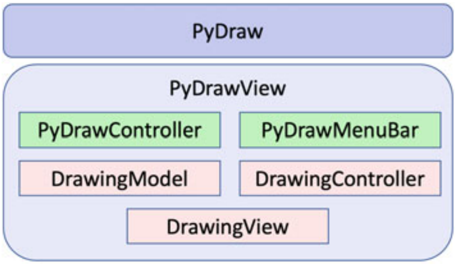


19.3 The Structure of the Application

The user interface created for the PyDraw application is made up of a number of elements (see below): the `PyDrawMenuBar`, the `PyDrawToolbar` containing a sequence of buttons across the top of the window, the drawing panel and the window frame (implemented by the `PyDrawFrame` class).

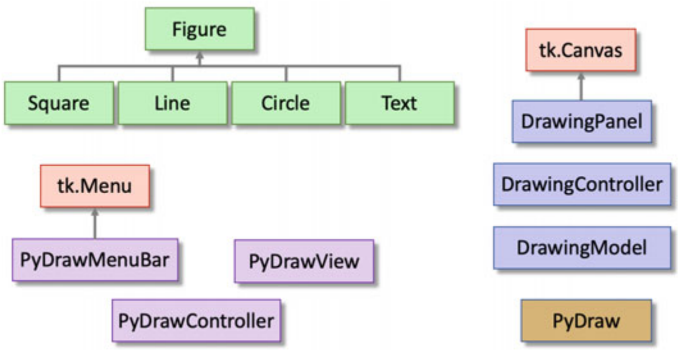


The following diagram shows the same information as that presented above, but as a containment hierarchy, this means that the diagram illustrates how one object is contained within another. The lower-level objects are contained within the higher-level objects.



It is important to visualise this as the majority of GUI interfaces are built up in this way, using containers and layout managers.

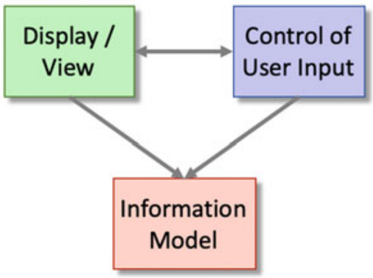
The inheritance structure between the classes used in the PyDraw application is illustrated below. This class hierarchy is typical of an application which incorporates user interface features with graphical elements.



19.3.1 Model, View and Controller Architecture

The application adopts the well-established Model-View-Controller (or MVC) design pattern for separating out the responsibilities between the view element (e.g. the Tk or Frame), the control element (for handling user input) and the model element (which holds the data to be displayed).

This separation of concerns is not a new idea and allows the construction of GUI applications that mirror the Model-View-Controller architecture. The intention of the MVC architecture is the separation of the user display, from the control of user input, from the underlying information model as illustrated below.



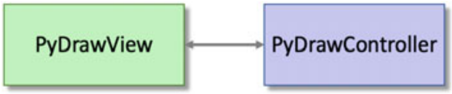
There are a number of reasons why this separation is useful:

- reusability of application and/or user interface components,
- ability to develop the application and user interface separately,
- ability to inherit from different parts of the class hierarchy.
- ability to define control style classes which provide common features separately from how these features may be displayed.

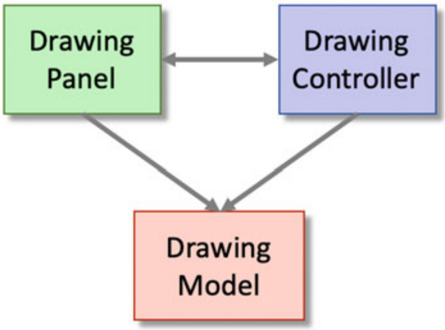
This means that different interfaces can be used with the same application, without the application knowing about it. It also means that any part of the system can be changed without affecting the operation of the other. For example, the way that the graphical interface (the look) displays the information could be changed without modifying the actual application or how input is handled (the feel). Indeed the application need not know what type of interface is currently connected to it at all.

19.3.2 PyDraw MVC Architecture

The MVC structure of the PyDraw application has a top-level controller class `PyDrawController` and a top-level view class the `PyDrawFrame` (there is no model as the top level MVC triad does not hold any explicit data itself). This is shown below:



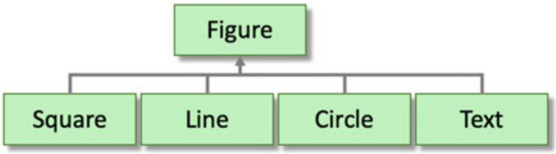
At the next level down there is another MVC structure; this time for the drawing element of the application. There is a `DrawingController`, with a `DrawingModel` and a `DrawingPanel` (the view) as illustrated below:



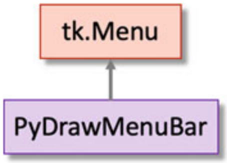
The `DrawingModel`, `DrawingPanel` and `DrawingController` classes exhibit the classic MVC structure. The view and the controller classes (`DrawingPanel` and `DrawingController`) know about each other and the drawing model, whereas the `DrawingModel` knows nothing about the view or the controller. The view is notified of changes in the drawing through the paint event.

19.3.3 Additional Classes

There are also four types of drawing object (of Figure): `Circle`, `Line`, `Square` and `Text` figures. The only difference between these classes is what is drawn on the graphic *device context* within the `draw()` method. The `Figure` class, from which they all inherit, defines the common attributes used by all objects within a Drawing (e.g. representing an *x* and *y* location and size).



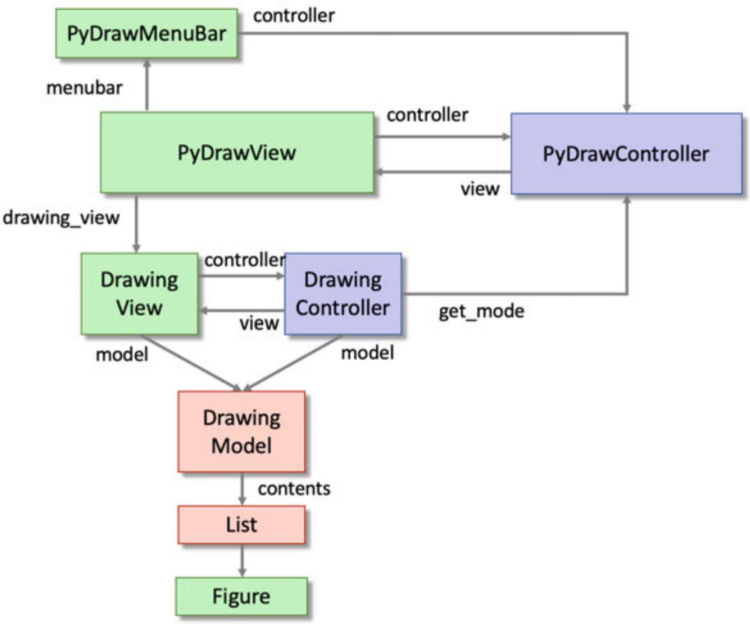
The PyDrawFrame class also uses a `PyDrawMenuBar` class which extends the `tk.Menu` class with menu items for use within the PyDraw application.



The final class is the `PyDraw` class that initiates the execution of the whole application.

19.3.4 Object Relationships

However, the inheritance hierarchy is only part of the story for any object-oriented application. The following figure illustrates how the objects relate to one another within the working application.



The `PyDrawView` is responsible for setting up the controller and the `DrawingView`.

The `PyDrawController` is responsible for handling menu and tool bar user interactions.

This separates graphical elements from the behaviour triggered by the user.

The `DrawingView` is responsible for displaying any figures held by the `DrawingModel`. The `DrawingController` manages all user interactions with the `DrawingView` including adding figures and clearing all figures from the model. The `DrawingModel` holds a list of figures to be displayed.

19.4 The Interactions Between Objects

We have now examined the physical structure of the application but not how the objects within that application interact.

In many situations this can be extracted from the source code of the application (with varying degrees of difficulty). However, in the case of an application such as PyDraw, which is made up of a number of different interacting components, it is useful to describe the system interactions explicitly.

The diagrams illustrating the interactions between the objects use the following conventions:

- a solid arrow indicates a message send,

- a square box indicates a class,
- a name in brackets indicates the type of instance,
- numbers indicate the sequence of message sends.

These diagrams are based on the sequence and collaboration diagrams found in the Unified Modelling Language (UML) notation.

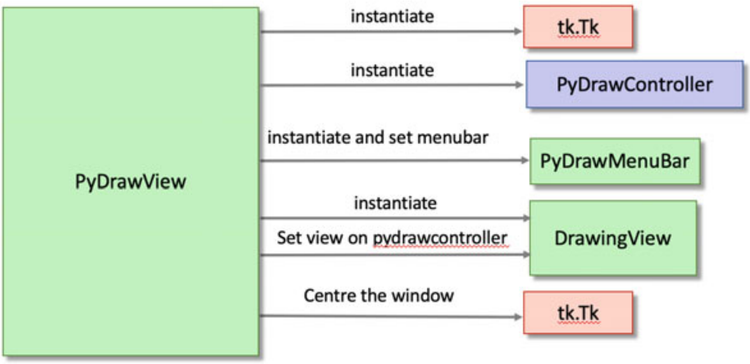
19.4.1 The PyDrawApp

When the `PyDraw` application is instantiated the `PyDraw` object in created, and this happens if this is the main entry point to the code. The PyDraw class creates the main PyDrawView display and causes it to be displayed to the user via the `mainloop()` method of the view:

```
class PyDraw:
    def __init__(self):
        self.view = PyDrawView()
        self.view.mainloop()

if __name__ == '__main__':
    PyDraw()
```

19.5 The PyDrawView Constructor

The `PyDrawView` initialiser method sets up the main display of the UI application and also initialises the controllers and drawing elements. This is shown below using a collaboration diagram:

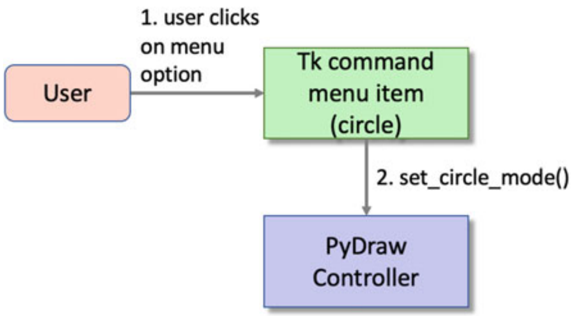


The PyDrawView constructor sets up the environment for the application. It creates the top level PyDrawController. It creates the DrawingView and initialises the display layout. It initialises the menu bar. It binds the DrawingView to the overall PyDrawController. It also uses the Tk object to centre the windowing using:

```
self.root.eval('tk::PlaceWindow . center')
```

19.5.1 Changing the Application Mode

One interesting thing to note is what happens when the user selects an option from the Drawing menu. This allows the *mode* to be changed to a square, circle, line or text. The interactions involved are shown below for the situation where a user selects the 'Circle' menu item on the Drawing menu (using a collaboration diagram):

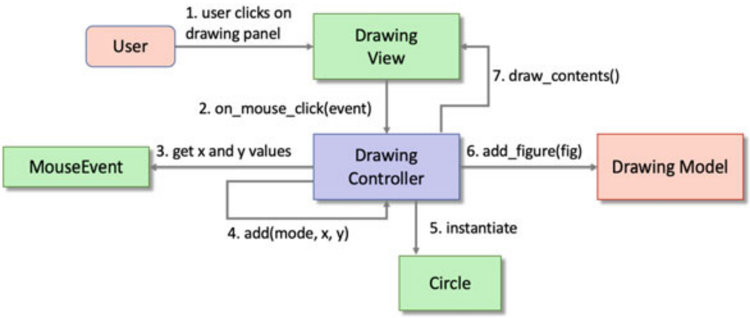


When the user selects one of the menu items the `command menu item calls a referenced PyDrawController method (such as the set_circle_mode()` method). Thus each command menu item is linked to the appropriate method defined on the overall controller (such as `set_circle_mode()` or `set_line_mode()`). These methods set the `mode` attribute of the controller to an appropriate value.

19.5.2 Adding a Graphic Object

A user adds a graphic object to the drawing displayed by the `DrawingView` by pressing the mouse button.

When the user clicks on the drawing panel, the `DrawingController` responds as shown below:



The above illustrates what happens when the user presses and releases a mouse button over the drawing panel, to create a new figure.

When the user presses the left mouse button, a mouse clicked message is sent to the DrawingController, which decides what action to perform in response (see above). In PyDraw, it obtains the cursor *x* and *y* values at which the event was generated.

The controller then calls its own `add()` method passing in the current mode and the current mouse location. The controller obtains the current *mode* (from the `PyDrawController` using the method callback provided when the `DrawingController` is instantiated) and adds the appropriate type of figure to the `DrawingModel`.

The `add()` method then adds a new figure to the drawing model based on the specified mode. It finally asks the view to redraw its contents.

19.6 The Classes

This section presents the classes in the PyDraw application. As these classes build on concepts already presented in the last few chapters, they shall be presented in their entirety with comments highlighting specific points of their implementations. Note that the code imports the `tk` module from the tkinter library, e.g.

```
import tkinter as tk
from abc import abstractmethod
```

19.6.1 The PyDrawConstants Class

The purpose of this class is to provide a set of constants that can be referenced in the remainder of the application. It is used to provide constants used to represent the current mode (to indicate whether a line, square, circle or test should be added to the display) default window sizes and background colour etc.

```
class PyDrawConstants:
    WIDTH = 600
    HEIGHT = 400

    BACKGROUND_COLOUR = 'white'

    CIRCLE_MODE = 'Circle'
    SQUARE_MODE = 'Square'
    LINE_MODE = 'Line'
    TEXT_MODE = 'Text'

    SIZE = 30
```

19.6.2 The PyDrawView Class

The `PyDrawView` class provides the main window for the application. Note that due to the separation of concerns introduced via the MVC architecture, the *view* class is only concerned with the layout of the components:

```
class PyDrawView:
    """ Main Frame responsible for the
        layout of the UI."""

    def __init__(self):
        self.root = tk.Tk()

        # Set the title of the window
        self.root.title('PyDraw')

        # Set up the controller
        self.controller = PyDrawController(self.root)

        # Set up menus
        self.menubar = PyDrawMenuBar(self.root, self.controller)
        self.root.config(menu=self.menubar)

        # Setup drawing view
                    self.drawing_view = DrawingView(self.root,
self.controller.get_mode)
        self.controller.view = self.drawing_view

        self.root.eval('tk::PlaceWindow . center')

    def mainloop(self):
       """Delegate method that passes responsibility onto the root"""
        self.root.mainloop()
```

19.6.3 The PyDrawMenuBar Class

The `PyDrawMenuBar` class is a subclass of the `tk.Menu` class which defines the contents of the menu bar for the PyDraw application. It does this by creating two `tk.Menu` objects and adding them to the menu bar. Each `tk.Menu` implements a dropdown menu from the menu bar. To add individual menu items the `add_command() function` is used. These menu items are appended to the menu. The menus are themselves appended to the menu bar using the add_cascade() function. Each menu item invokes a method on the controller associated with the menu bar—thus separating out the concerns of defining the structure of the menu bar and defining the functionality of each item in the menu bar.

```
class PyDrawMenuBar(tk.Menu):
    def __init__(self, parent, controller):
        super().__init__(parent)
        self.controller = controller
        self.create_file_menu()
        self.create_mode_menu()

    def create_mode_menu(self):
        mode_menu = tk.Menu(self, tearoff=0)
        mode_menu.add_command(label=PyDrawConstants.CIRCLE_MODE,
                              command=self.controller.set_circle_mode)
        mode_menu.add_command(label=PyDrawConstants.SQUARE_MODE,
                              command=self.controller.set_square_mode)
        mode_menu.add_command(label=PyDrawConstants.LINE_MODE,
                              command=self.controller.set_line_mode)
        mode_menu.add_command(label=PyDrawConstants.TEXT_MODE,
                              command=self.controller.set_text_mode)
        self.add_cascade(label='Mode', menu=mode_menu)

    def create_file_menu(self):
        file_menu = tk.Menu(self, tearoff=0)
        file_menu.add_command(label='New',
                              command=self.controller.new_canvas)
        file_menu.add_command(label='Exit',
                              command=self.controller.exit_app)
        self.add_cascade(label='File', menu=file_menu)
```

19.6.4 The PyDrawController Class

This class provides the control element of the top-level view. It maintains the current `mode`. It also provides a method which can be used to obtain the current mode. The final two methods support cleaning out the display and quitting the application.

```
class PyDrawController:

    def __init__(self, root):
        self.root = root
        self.view = None
        # Set the initial mode
        self.mode = PyDrawConstants.SQUARE_MODE

    def set_circle_mode(self):
        self.mode = PyDrawConstants.CIRCLE_MODE

    def set_line_mode(self):
        self.mode = PyDrawConstants.LINE_MODE

    def set_square_mode(self):
        self.mode = PyDrawConstants.SQUARE_MODE

    def set_text_mode(self):
        self.mode = PyDrawConstants.TEXT_MODE

    def clear_drawing(self):
        self.view.drawing_controller.clear()

    def get_mode(self):
        return self.mode

    def new_canvas(self):
        self.view.delete('all')

    def exit_app(self):
        self.root.quit()
```

19.6.5 The DrawingModel Class

The `DrawingModel` class has a `contents` attribute that is used to hold all the figures in the drawing. It also provides some convenience methods to reset the `contents` and to add a figure to the `contents`.

```
class DrawingModel:

    def __init__(self):
        self.contents = []

    def clear_figures(self):
        self.contents = []

    def add_figure(self, figure):
        self.contents.append(figure)
```

The `DrawingModel` is a relatively simple model which merely records a set of graphical figures in a List. These can be any type of object and can be displayed in any way as long as they implement the `draw()` method. It is the objects themselves which determine what they look like when drawn.

19.6.6 The DrawingView Class

The `DrawingView` class is a subclass of the `tk.Canvas` class. It provides the view for the drawing data model. This uses the classical MVC architecture and has a `model` (DrawingModel), a `view` (the DrawingPanel) and a `controller` (the DrawingController).

The `DrawingPanel` instantiates its own `DrawingModel` to hold the figures to be drawn and `DrawingController` to handle mouse events.

It also registers for button events to be handled by the on_mouse_click method of the controller.

```
class DrawingView(tk.Canvas):
  def __init__(self, parent, get_mode,
          width=PyDrawConstants.WIDTH,
          height=PyDrawConstants.HEIGHT,
          bg=PyDrawConstants.BACKGROUND_COLOUR):
    super().__init__(parent, width=width, height=height, bg=bg)
    self.model = DrawingModel()
     self.controller = DrawingController(self, self.model, get_
mode)
    self.pack()
    self.bind('<Button-1>', self.controller.on_mouse_click)

  def draw_contents(self):
    for figure in self.model.contents:
      figure.draw()
```

19.6.7 The DrawingController Class

The `DrawingController` class provides the control class for the top level MVC architecture used with the `DrawingModel` (model) and `DrawingPanel` (view) classes. In particular it handles the mouse events in the `DrawingView` via the `on_mouse_click()` method.

It also defines an add method that is used to add a figure to the DrawingModel (the actual figure depends on the current mode of the `PyDrawController`) and requests that the view refresh itself. A final method, the `clear()` method, removes all figures from the drawing model and refreshes the display.

```
class DrawingController:

  def __init__(self, view, model, get_mode):
    self.view = view
    self.model = model
    self.get_mode = get_mode

  def on_mouse_click(self, mouse_event):
    x = mouse_event.x
```

```
        y = mouse_event.y
        self.add(self.get_mode(), x, y)

    def add(self, mode, x, y, size=PyDrawConstants.SIZE):
        if mode == PyDrawConstants.SQUARE_MODE:
            fig = Square(self.view, x, y, size)
        elif mode == PyDrawConstants.CIRCLE_MODE:
            fig = Circle(self.view, x, y, size)
        elif mode == PyDrawConstants.TEXT_MODE:
            fig = Text(self.view, x, y)
        else:
            fig = Line(self.view, x, y, size)
        self.model.add_figure(fig)
        self.view.draw_contents()

    def clear(self):
        self.model.clear_figures()
        self.view.delete('all')
```

19.6.8 The Figure Class

The `Figure` class (an abstract superclass of the *Figure* class hierarchy) captures the elements which are common to graphic objects displayed within a drawing. The *x* and *y* values define the position of the figure, while the `size` attribute defines the size of the figure. The fill attributes defines the background colour used to fill the figure (if appropriate).

The Figure class defines a single abstract method `draw()` that must be implemented by all concrete subclasses. This method should define how the shape is drawn on the drawing panel.

```
class Figure:
    def __init__(self,
                 canvas,
                 x=0,
                 y=0,
                 size=None,
                 fill='blue'):
        self.canvas = canvas
        self.x = x
        self.y = y
        self.size = size
        self.fill = fill

    @abstractmethod
    def draw(self):
        pass
```

19.6.9 The Square Class

This is a subclass of `Figure` that specifies how to draw a square shape in a drawing. It implements the `draw()` method inherited from `Figure` using the `canvas.create_rectangle()` method.

```
class Square(Figure):
    def __init__(self, canvas, x, y, size):
        super().__init__(canvas=canvas, x=x, y=y, size=size)

    def draw(self):
        self.canvas.create_rectangle(self.x,
                                     self.y,
                                     self.x + self.size,
                                     self.y + self.size,
                                    fill=self.fill)
```

19.6.10 The Circle Class

This is another subclass of `Figure`. It implements the `draw()` method by drawing a circle (via the `create_oval()` method of the canvas. Note that the size attribute must be used to generate an appropriate radius.

```
class Circle(Figure):
    def __init__(self, canvas, x, y, size):
            super().__init__(canvas=canvas,  x=x,  y=y,  size=size,
fill='red')

    def draw(self):
        self.canvas.create_oval(self.x,
                                     self.y,
                                     self.x + self.size,
                                     self.y + self.size,
                                     fill=self.fill)
```

19.6.11 The Line Class

This is another subclass of `Figure`. In this very simple example, a default end point for the line is generated. Alternatively the program could look for a mouse released event and pick up the mouse at this location and use this as the end point of the line.

```
class Line(Figure):
    def __init__(self, canvas, x, y, size):
        super().__init__(canvas=canvas, x=x, y=y, size=size)
```

```
    def draw(self):
        self.canvas.create_line(self.x,
                                self.y,
                                self.x + self.size,
                                self.y + self.size)
```

19.6.12 The Text Class

This is also a subclass of `Figure`. A default value is used for the text to display; however a dialog could be presented to the user allowing them to input the text they wish to display:

```
class Text(Figure):
    def __init__(self, canvas, x, y, text_string='Text',
            font='Helvetica 15 bold', fill='black'):
        super().__init__(canvas=canvas, x=x, y=y, fill=fill)
        self.text_string = text_string
        self.font = font

    def draw(self):
        text = self.text_string
        self.canvas.create_text(self.x,
                                self.y,
                                text=text,
                                fill=self.fill,
                                font=self.font)
```

19.7 Reference

The following provides some background on the Model-View-Controller architecture in user interfaces.

1. G.E. Krasner, S.T. Pope, A cookbook for using the model-view controller user interface paradigm in smalltalk-80. JOOP **1**(3), 26–49 (1988)

19.8 Exercises

You could develop the PyDraw application further by adding the following features:

1. A delete option You can add a button labelled Delete to the window. It should set the mode to “delete”. The drawingPanel must be altered so that the mouseReleased method sends a delete message to the drawing. The drawing must find and remove the appropriate graphic object and send the changed message to itself.

2. A resize option This involves identifying which of the shapes has been selected and then either using a dialog to enter the new size or providing some option that allows the size for the shape to be indicated using the mouse.

Part III
Computer Games

Chapter 20
Introduction to Games Programming

20.1 Introduction

Games programming is performed by developers/coders who implement the logic that drives a game.

Historically games developers did everything; they wrote the code, designed the sprites and icons, handled the game play, dealt with sounds and music, generated any animations required etc. However, as the game industry has matured games companies have developed specific riles including Computer Graphics (CG) animators, artists, games developers and games engine and physics engine developers etc.

Those involved with code development may develop a physics engine, a games engine, the games themselves, etc. Such developers focus on different aspects of a game. For examples a game engine developer focusses on creating the framework within which the game will run. In turn a physics engine developer will focus on implementing the mathematics behind the physics of the simulated games world (such as the effect of gravity on characters and components within that world). In many cases there will also be developers working on the AI engine for a game. These developers will focus on providing facilities that allow the game or characters in the game to operate intelligently.

Those developing the actual game play will use these engines and frameworks to create the overall end result. It is they who give life to the game and make it an enjoyable (and playable) experience.

20.2 Games Frameworks and Libraries

There are many frameworks and libraries available that allow you to create anything from simple games to large complex role playing games with infinite worlds.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_20

One example is the Unity framework that can be used with the C# programming language. Another such framework is the Unreal engine used with the C++ programming language.

Python has also been used for games development with several well known games titles depending on it in one way or another. For example, Battlefield 2 by Digital Illusions CE is a military simulator first-person shooter game. Battlefield Heroes handles portions of the game logic involving game modes and scoring using Python.

Other games that use Python include Civilisation IV (for many of the tasks), Pirates of the Caribbean Online and Overwatch (which makes its choices with Python).

Python is also embedded as a scripting engine within tools such as Autodesk's Maya which is a computer animation toolkit that is often used with games.

20.3 Python Games Development

For those wanting to learn more about game development; Python has much to offer. There are many examples available online as well as several game oriented frameworks.

The frameworks/libraries available for games development in Python include:

- Arcade. This is a Python library for creating 2D style video games.
- pyglet is a windowing and multimedia library for Python that can also be used for games development.
- Cocos2d is a framework for building 2D games that is built on top of pyglet.
- pygame is probably the most widely used library for creating games within the Python world. There are also many extensions available for pygame that help to create a wide range of different types of games.

We will focus on pygame in the next two chapters in this book.

Other libraries of interest to Python games developers include:

- PyODE. This is an open-source Python binding for the Open Dynamics Engine which is an open-source physics engine.
- pymunk Pymunk is a easy-to-use 2D physics library that can be used whenever you need 2d rigid body physics with Python. It is very good when you need 2D physics in your game, demo or other application. It is built on top of the 2D physics library Chipmunk.
- pyBox2D pybox2d is a 2D physics library for your games and simple simulations. It's based on the Box2D library written in C++. It supports several shape types (circle, polygon, thin line segments) as well as a number of joint types (revolute, prismatic, wheel, etc.).
- Blender. This is a open-source 3D Computer Graphics software toolset used for creating animated films, visual effects, art, 3D printed models, interactive 3D applications and video games. Blender's features include 3D modelling, texturing,

raster graphics editing, rigging and skinning, etc. Python can be used as a scripting tool for creation, prototyping, game logic and more.
- Quake Army Knife which is an environment for developing 3D maps for games based on the Quake engine. It is written in Delphi and Python.

20.4 Using Pygame

In the next two chapters we will explore the core *pygame* library and how it can be used to develop interactive computer games. The next chapter explores pygame itself and the facilities it provides. The following chapter developers a simple video game in which the user moves a starship around avoiding meteors which scroll vertically down the screen.

20.5 Online Resources

For further information games programming and the libraries mentioned in this chapter see:

- https://unity.com/ the C# framework for games development.
- https://www.unrealengine.com for C++ games development.
- http://arcade.academy/ provides details on the Arcade games framework.
- http://www.pyglet.org/ for information on the piglet library.
- http://cocos2d.org/ is the home page for the Cocos2d framework.
- https://www.pygame.org for information on pygame.
- http://pyode.sourceforge.net/ for details of the PyODE bindings to the Open Dynamics Engine.
- http://www.pymunk.org/ provides information on pymunk.
- https://github.com/pybox2d/pybox2d which is a GitHub repository for pyBox2d.
- https://git.blender.org/gitweb/gitweb.cgi/blender.git GitHub repository for Blender.
- https://sourceforge.net/p/quark/code SourceForge repository for Quake Army Knife.
- https://www.autodesk.co.uk/products/maya/overview for information on Autodesks Maya computer animation software.

Chapter 21
Building Games with Pygame

21.1 Introduction

Pygame is a cross-platform, free and open-source Python library designed to make building multimedia applications such as games easy. Development of pygame started back in October 2000 with pygame version 1.0 being released six months later. The version of pygame discussed in this chapter is version 2.4.0. If you have a later version, check to see what changes have been made to see if they have any impact on the examples presented here.

You can install pygame using pip for example.

```
pip install pygame
```

The pygame library is built on top of the SDL library. SDL (or Simple Directmedia Layer) is a cross-platform development library designed to provide access to audio, keyboards, mouse, joystick and graphics hardware via OpenGL and Direct3D. To promote portability, pygame also supports a variety of additional backends including WinDIB, X11, Linux Frame Buffer, etc.

SDL officially supports Windows, Mac OS X, Linux, iOS and Android (although other platforms are unofficially supported). SDL itself is written in C, and pygame provides a wrapper around SDL. However, pygame adds functionality not found in SDL to make the creation of graphical or video games easier. These functions include vector maths, collision detection, 2D sprite scene graph management, MIDI support, camera, pixel array manipulation, transformations, filtering, advanced freetype font support, and drawing.

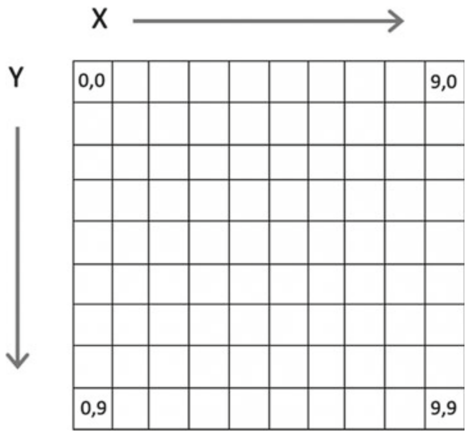
The remainder of this chapter introduces pygame, the key concepts; the key modules, classes and functions, and a very simple first pygame application. The next chapter steps through the development of a simple arcade style video game which illustrates how a game can be created using pygame.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_21

21.2 The Display Surface

The Display Surface (aka the display): this is the most important part of a pygame game. It is the main window display of your game and can be of any size; however, you can only have one display surface.

In many ways, the display surface is like a blank piece of paper on which you can draw. The surface itself is made up of pixels which are numbered from 0,0 in the top left hand corner with the pixel locations being indexed in the *x* axis and the *y* axis. This is shown below:



The above diagram illustrates how pixels within a surface are indexed. Indeed a surface can be used to draw lines, shapes (such as rectangles, squares, circles and elapses), display images, manipulate individual pixels, etc. Lines are drawn from one pixel location to another (e.g. from location 0,0 to location 9,0 which would draw a line across the top of the above display surface). Images can be displayed within the display surface given a starting point such as 1, 1.

The display surface is created by the `pygame.display.set_mode()` function. This function takes a tuple that can be used to specify the size of the display surface to be returned. For example:

```
display_surface = pygame.display.set_mode((400, 300))
```

This will create a display surface (window) of 400 by 300 pixels.

Once you have the display surface you can fill it with an appropriate background colour (the default is black); however, if you want a different background colour or want to clear everything that has previously been drawn on the surface, then you can use the surface's `fill()` method:

```
WHITE = (255, 255, 255)
display_surface.fill(WHITE)
```

The fill method takes a tuple that is used to define a colour in terms of red, green and blue (or RGB) colours. Although the above examples use a meaningful name for the tuple representing the RGB values used for white, there is of course no requirement to do this (although it is considered good practice).

To aid in performance, any changes you make to the display surface actually happen in the background and will not be rendered onto the actual display that the user sees until you call the `update()` or `flip()` methods on the surface. For example:

```
pygame.display.update()
pygame.display.flip()
```

The `update()` method will redraw the display with all changes made to the display in the background. It has an optional parameter that allows you to specify just a region of the display to update (this is defined using a `Rect` which represents a rectangular area on the screen). The `flip()` method always refreshes the whole of the display (and as such does exactly the same as the `update()` method with no parameters).

Another method, which is not specifically a display surface method, but which is often used when the display surface is created, provides a caption or title for the top-level window. This is the `pygame.display.set_caption()` function. For example:

```
pygame.display.set_caption('Hello World')
```

This will give the top-level window the caption (or title) 'Hello World'.

21.3 Events

Just as the graphical user interface systems described in earlier chapters have an event loop that allows the programmer to work out what the user is doing (in those cases this is typically selecting a menu item, clicking a button or entering data, etc.), pygame has an event loop that allows the game to work out what the player is doing. For example, the user may press the left or right arrow key. This is represented by an event.

21.3.1 Event Types

Each event that occurs has associated information such as the type of that event. For example:

- Pressing a key will result in a `KEYDOWN` type of event, while releasing a key will result in a KEYUP event type.

- Selecting the window close button will generate a `QUIT` event type, etc.
- Using the mouse can generate `MOUSEMOTION` events as well as `MOUSEBUTTONDOWN` and `MOUSEBUTTONUP` event types.
- Using a Joystick can generate several different types of event including `JOYAXISMOTION`, `JOYBALLMOTION`, `JOYBUTTONDOWN` and `JOYBUTTONUP`.

These event types tell you what occurred to generate the event. This means that you can choose which types of events you want to deal with and ignore other events.

21.3.2 Event Information

Each type of event object provides information associated with that event. For example a key-oriented event object will provide the actual key pressed, while a mouse-oriented event object will provide information on the position of the mouse, which button was pressed, etc. If you try an access an attribute on an event that does not support that attribute, then an error will be generated.

The following lists some of the attributes available for different event types:

- `KEYDOWN` and `KEYUP`, the event has a `key` attribute and a `mod` attribute (indicating if any other modifying keys such as shift are also being pressed).
- `MOUSEBUTTONUP` and `MOUSEBUTTONDOWN` have an attribute `pos` that holds a tuple indicating the mouse location in terms of *x* and *y* coordinates on the underlying surface. It also has a `button` attribute indicating which mouse was pressed.
- `MOUSEMOTION` has pos, rel and buttons attributes. The `pos` is a tuple indicating the *x* and *y* locations of mouse cursor. The `real` attribute indicates the amount of mouse movement, and `buttons` indicate the state of the mouse buttons.

As an example if we want to check for a keyboard event type and then check that the key pressed was the space bar, then we can write:

```
if event.type == pygame.KEYDOWN:
    # Check to see which key is pressed
   if event.key == pygame.K_SPACE:
     print('space')
```

This indicates that if it is a key pressed event and that the actual key was the space bar, then print the string 'space'.

There are many keyboard constants that are used to represent the keys on the keyboard and `pygame.K_SPACE` constant used above is just one of them.

All the keyboard constants are pre-fixed with 'K_' followed by the key or the name of the key, for example:

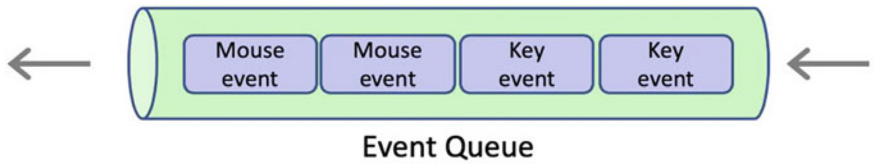
- K_TAB, K_SPACE, K_PLUS, K_0, K_1, K_AT, K_a, K_b, K_z, K_DELTE, K_DOWN, K_LEFT, K_RIGHT, K_LEFT, etc.

Further keyboard constants are provided for modifier states that can be combined with the above such as KMOD_SHIFT, KMOD_CAPS, KMOD_CTRL and KMOD_ALT.

21.3.3 The Event Queue

Events are supplied to a pygame application via the Event Queue.

The event queue is used to collect together events as they happen. For example, let us assume that a user clicks on the mouse twice and a key twice before a program has a chance to process them; then there will be four events in the event queue as shown below:



The application can then obtain an iterable from the event queue and process through the events in turn. While the program is processing these events, further events may occur and will be added to the event queue. When the program has finished processing the initial collection of events, it can obtain the next set of events to process.

One significant advantage of this approach is that no events are ever lost; that is if the user clicks the mouse twice while the program is processing a previous set of events, they will be recorded and added to the event queue. Another advantage is that the events will be presented to the program in the order that they occurred.

The `pygame.event.get()` function will read all the events currently on the event queue (removing them from the event queue). The method returns an `EventList` which is an iterable list of the events read. Each event can then be processed in turn. For example:

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        print('Received Quit Event:')
    elif event.type == pygame.MOUSEBUTTONDOWN:
        print('Received Mouse Event')
    elif event.type == pygame.KEYDOWN:
        print('Received KeyDown Event')
```

In the above code snippet an EventList is obtained from the event queue containing the current set of events. The for loop then processes each event in turn checking the type and printing an appropriate message.

You can use this approach to trigger appropriate behaviour such as moving an image around the screen or calculating the players score. However, be aware that if this behaviour takes too long it can make the game difficult to play (although the examples in this chapter and the next are simple enough that this is not a problem).

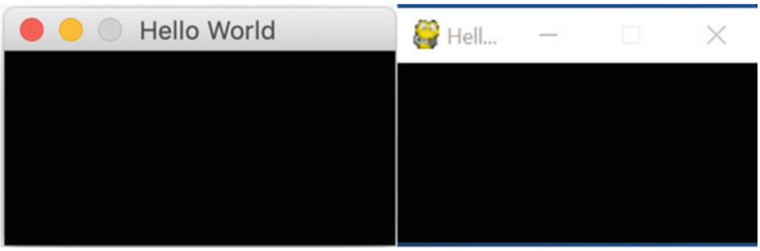
21.4 A First pygame Application

We are now at the point where we can put together what we have looked at so far and create a simple pygame application.

It is common to create a hello world style program when using a new programming language or using a new application framework, etc. The intention is that the core elements of the language or framework are explored in order to generate the most basic form of an application using the language or framework. We will therefore implement the most basic application possible using pygame.

The application we will create will display a pygame window, with a 'Hello World' title. We will then be able to quit the game. Although technically speaking this isn't a game, it does possess the basic architecture of a pygame application.

The simple HelloWorld game will initialise pygame and the graphical display. It will then have a main game playing loop that will continue until the user selects to quit the application. It will then shut down pygame. The display created by the program is shown below for both Mac and Windows operating systems:



To quit the program, click on the exit button for the windowing system you are using.

The simple HelloWorld *game* is given below:

```
import pygame

def main():
    print('Starting Game')

    print('Initialising pygame')
    pygame.init() # Required by every pygame application

    print('Initialising HelloWorldGame')
```

```
    pygame.display.set_mode((200, 100))
    pygame.display.set_caption('Hello World')

    print('Update display')
    pygame.display.update()

    print('Starting main Game Playing Loop')
    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                print('Received Quit Event:', event)
                running = False

    print('Game Over')
    pygame.quit()

if __name__ == '__main__':
    main()
```

There are several key steps highlighted by this example; these steps are

1. **Import pygame**. pygame is of course not one of the default modules available within Python. You must first import pygame into you code. The `import pygame` statement imports the `pygame` module into your code and makes the functions and classes in pygame available to you (note the capitalisation—pygame is not the same module name as `PyGame`). It is also common to find that programs import
   - `from pygame.locals import *`
   - This adds several constants and functions into the namespace of your program. In this very simple example we have not needed to do this.
2. **Initialise pygame**. Almost every pygame module needs to be initialised in some way, and the simplest way to do this is to call `pygame.init()`. This will do what is required to set the pygame environment up for use. If you forget to call this function, you will typically get an error message such as `pygame.error: video system not initialised` (or something similar). If you get such a method, check to see that you have called `pygame.init()`. Note that you can initialise individual pygame modules (e.g. the `pygame.font` module can be initialised using `pygame.font.init()`) if required. However `pygame.init()` is the most commonly used approach to setting up pygame.
3. **Set up the display**. Once you have initialised the pygame framework you can setup the display. In the above code example, the display is set up using the `pygame.display.set_mode()` function. This function takes a tuple specifying the size of the window to be created (in this case 200 pixels wide by 100 pixels high). Note that if you try and invoke this function by passing in two parameters instead of a tuple, then you will get an error. This function returns the drawing surface or screen/window that can be used to display items within the game such as icons, messages and shapes. As our example is so simple we

do not bother saving it into a variable. However, anything more complex than this will need to do so. We also set the window/frame's caption (or title). This is displayed in the title bar of the window.

4. **Render the display**. We now call the `pygame.display.update()` function. This function causes the current details of the display to be drawn. At the moment this is a blank window. However, it is common in games to perform a series of updates to the display in the background and then when the program is ready to update the display to call this function. This batches a series of updates and the causes the display to be refreshed. In a complex display it is possible to indicate which parts of the display need to be redrawn rather than redrawing the whole window. This is done by passing a parameter into the `update()` function to indicate the rectangle to be redrawn. However, our example is so simple we are OK with redrawing the whole window, and therefore we do not need to pass any parameters to the function.
5. **Main game playing loop**. It is common to have a main game playing loop that drives the processing of user inputs, modifies the state of the game and updates the display. This is represented above by the `while running:` loop. The local variable `running` is initialised to `True`. This means that the `while` loop ensures that the game continues until the user selects to quit the game at which point the `running` variable is set to `False` which causes the loop to exit. In many cases this loop will call `update()` to refresh the display. The above example does not do this as nothing is changed in the display. However the example developed later in this chapter will illustrate this idea.
6. **Monitor for events that drive the game**. As mentioned earlier the event queue is used to allow user inputs to be queued and then processed by the game. In the simple example shown above this is represented by a for loop that receives events using `pygame.event.get()` and then checking to see if the event is a pygame.`QUIT` event. If it is, then it sets the `running` flag to `False`, which will cause the main `while` loop of the game to terminate.
7. **Quit pygame once finished**. In pygame any module that has an `init()` function also has an equivalent `quit()` function that can be used to perform any cleanup operations. As we called `init()` on the `pygame` module at the start of our program we will therefore need to call `pygame.quit()` at the end of the program to ensure everything is tidied up appropriately.

The output generated from a sample run of this program is given below:

```
pygame 2.4.0 (SDL 2.26.4, Python 3.11.3)
Hello from the pygame community. https://www.pygame.org/contri
bute.html
Starting Game
Initialising pygame
Initialising HelloWorldGame
Update display
Starting main Game Playing Loop
```

21.5 Further Concepts

There are very many facilities in pygame that go beyond what we can cover in this book; however a few of the more common are discussed below.

Surfaces are a hierarchy. The top-level display surface may contain other surfaces that may be used to draw images or text. In turn containers such as panels may render surfaces to display images or text, etc.

Other types of surface. The primary display surface is not the only surface in pygame. For example, when an image, such as a PNG or JPEG image, is loaded into a game, then it is rendered onto a surface. This surface can then be displayed within another surface such as the display surface. This means that anything you can do to the display surface you can do with any other surface such as draw on it, put text on it, colour it and add another icon onto the surface.

Fonts. The `pygame.font.Font` object is used to create a font that can be used to render text onto a surface. The render method returns a surface with the text rendered on it that can be displayed within another surface such as the display surface. Note that you cannot write text onto an existing surface you must always obtain a new surface (using render) and then add that to an existing surface. The text can only be displayed in a single line, and the surface holding the text will be of the dimensions required to render the text. For example:

```
text_font = pygame.font.Font('freesansbold.ttf', 18)
text_surface = text_font.render('Hello World', antialias=True,
color=BLUE)
```

This creates a new font object using the specified font with the specified font size (in this case 18). It will then render the string ‘Hello World’ on to a new surface using the specified font and font size in blue. Specifying that antialias is true indicates that we would like to smooth the edges of the text on the screen.

Rectangles (or Rects). The `pygame.Rect` class is an object used to represent rectangular coordinates. A Rect can be created from a combination of the top left corner coordinates plus a width and height. For flexibility many functions that expect a Rect object can also be given a *Rectlike* list; this is a list that contains the data necessary to create a Rect object. Rects are very useful in a pygame game as they can be used to define the borders of a game object. This means that they can be used within games to detect if two objects have collided. This is made particularly easy because the Rect class provides several collision detection methods:

- `pygame.Rect.contains()` test if one rectangle is inside another.
- `pygame.Rect.collidepoint()` test if a point is inside a rectangle.
- `pygame.Rect.colliderect()` test if two rectangles overlap.
- `pygame.Rect.collidelist()` test if one rectangle in a list intersects.
- `pygame.Rect.collidelistall()` test if all rectangles in a list intersect.

- `pygame.Rect.collidedict()` test if one rectangle in a dictionary intersects.
- `pygame.Rect.collidedictall()` test if all rectangles in a dictionary intersect.

The class also provides several other utility methods such as `move()` which moves the rectangle and `inflate()` which can grow or shrink the rectangles size.

Drawing shapes. The `pygame.draw` module has numerous functions that can be used to draw lines and shapes onto a surface, for example:

```
pygame.draw.rect(display_surface, BLUE, [x, y, WIDTH, HEIGHT])
```

This will draw a filled blue rectangle (the default) onto the display surface. The rectangle will be located at the location indicated by *x* and *y* (on the surface). This indicates the top left hand corner of the rectangle. The width and height of the rectangle indicate its size. Note that these dimensions are defined within a list which is a structure referred to as being rect like (see below). If you do not want a filled rectangle (i.e. You just want the outline), then you can use the optional width parameter to indicate the thickness of the outer edge. Other methods available include

- `pygame.draw.polygon()` draw a shape with any number of sides.
- `pygame.draw.circle()` draw a circle around a point.
- `pygame.draw.ellipse()` draw a round shape inside a rectangle.
- `pygame.draw.arc()` draw a partial section of an ellipse.
- `pygame.draw.line()` draw a straight line segment.
- `pygame.draw.lines()` draw multiple contiguous line segments.
- `pygame.draw.aaline()` draw fine antialiased lines.
- `pygame.draw.aalines()` draw a connected sequence of antialiased lines.

Images. The `pygame.image` module contains functions for loading, saving and transforming images. When an image is loaded into pygame, it is represented by a surface object. This means that it is possible to draw, manipulate and process an image in exactly the same way as any other surface which provides a great deal of flexibility.

At a minimum the module only supports loading uncompressed BMP images but usually also supports JPEG, PNG, GIF (non-animated), BMP, TIFF as well as other formats. However, it only supports a limited set of formats when saving images; these are BMP, TGA, PNG and JPEG.

An image can be loaded from a file using:

```
image_surface = pygame.image.load(filename).convert()
```

This will load the image from the specified file onto a surface.

One thing you might wonder at is the use of the `convert()` method on the object returned from the `pygame.image.load()` function. This function returns a surface that is used to display the image contained in the file. We call the method `convert()` on this `Surface`, not to convert the image from a particular file format (such as PNG, or JPEG); instead this method is used to convert the pixel format used by the surface. If the pixel format used by the `Surface` is not the same as the display format, then it will need to be converted on the fly each time the image is displayed on the screen; this can be a fairly time-consuming (and unnecessary) process. We therefore do this once when the image is loaded which means that it should not hinder runtime performance and may improve performance significantly on some systems.

Once you have a surface containing an image it can be rendered onto another surface, such as the display surface using the `Surface.blit()` method. For example:

```
display_surface.blit(image_surface, (x, y))
```

Note that the position argument is a tuple specifying the *x* and *y* coordinates to the image on the display surface.

Strictly speaking the `blit()` method draws one surface (the source surface) onto another surface at the destination coordinates. Thus the target surface does not need to be the top-level display surface.

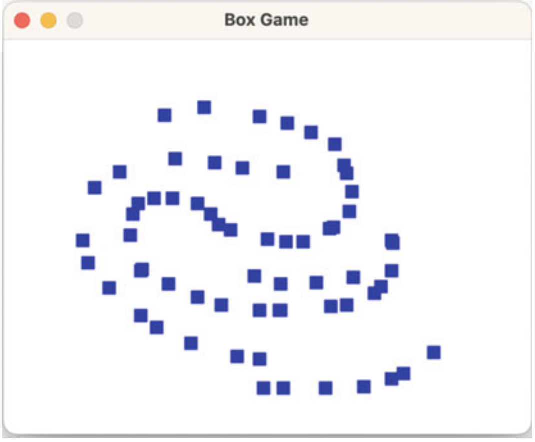
Clock. A `Clock` object is an object that can be used to track time. In particular it can be used to define the frame rate for the game, that is the number of frames rendered per second. This is done using the Clock.tick() method. This method should be called once (and only once) per frame. If you pass the optional `framerate` argument to the `tick()` the function, then pygame will ensure that the games refresh rate is slower than the given ticks per second. This can be used to help limit the runtime speed of a game. By calling `clock.tick(30)` once per frame, the program will never run at more than 30 frames per second.

21.6 A More Interactive pygame Application

The first pygame application we looked at earlier just displayed a window with the caption 'Hello World'. We can now extend this a little by playing with some of the features we have looked at above.

The new application will add some mouse event handling. This will allow us to pick up the location of the mouse when the user clicked on the window and draw a small blue box at that point.

If the user clicks the mouse multiple times, we will get multiple blue boxes being drawn. This is shown below.



This is still not much of a game but does make the pygame application more interactive.

The program used to generate this application is presented below:

```
import pygame

FRAME_REFRESH_RATE = 30
BLUE = (0, 0, 255)
BACKGROUND = (255, 255, 255) # White
WIDTH = 10
HEIGHT = 10

def main():
    print('Initialising PyGame')
    pygame.init() # Required by every PyGame application

    print('Initialising Box Game')
    display_surface = pygame.display.set_mode((400, 300))
    pygame.display.set_caption('Box Game')
    print('Update display')
    pygame.display.update()
    print('Setup the Clock')
    clock = pygame.time.Clock()
    # Clear the screen of current contents
    display_surface.fill(BACKGROUND)

    print('Starting main Game Playing Loop')
    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                print('Received Quit Event:', event)
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN:
                print('Received Mouse Event', event)
                x, y = event.pos
```

```
            pygame.draw.rect(display_surface, BLUE, [x, y, WIDTH,
HEIGHT])
        # Update the display
            pygame.display.update()

        # Defines the frame rate - the number of frames per second
        # Should be called once per frame (but only once)
        clock.tick(FRAME_REFRESH_RATE)

    print('Game Over')
    # Now tidy up and quit Python
    pygame.quit()

if __name__ == '__main__':
    main()
```

Note that we now need to record the display surface in a local variable so that we can use it to draw the blue rectangles. We also need to call the `pygame.display.update()` function each time round the main while loop so that the new rectangles we have drawn as part of the event processing for loop are displayed to the user.

We also set the frame rate each time round the main while loop. This should happen once per frame (but only once) and uses the clock object initialised at the start of the program.

21.7 Alternative Approach to Processing Input Devices

There are actually two ways in which inputs from a device such as a mouse, joystick or the keyboard can be processed. One approach is the event-based model described earlier. The other approach is the state-based approach.

Although the event-based approach has many advantages, it is has two disadvantages:

- Each event represents a single action, and continuous actions are not explicitly represented. Thus if the user presses both the *X* key and the *Z* key, then this will generate two events and it will be up to the program to determine that they have been pressed at the same time.
- It is also up to the program to determine that the user is still pressing a key (by noting that no `KEYUP` event has occurred).
- Both of these are possible but can be error prone.

An alternative approach is to use the state-based approach. In the state-based approach the program can directly check the state of an input device (such as a key or mouse or keyboard). For example, you can use `pygame.key.get_pressed()` which returns the state of all the keys. This can be used to determine if a specific key is being pressed at this moment in time. For example, `pygame.key.get_` `pressed()[pygame.K_SPACE]` can be used to check to see if the space bar is being pressed.

This can be used to determine what action to take. If you keep checking that the key is pressed, then you can keep performing the associated action. This can be very useful for continuous actions in a game such as moving an object.

However, if the user presses a key and then releases it before the program checks the state of the keyboard, then that input will be missed.

21.8 pygame Modules

There are numerous modules provided as part of pygame as well as associated libraries. Some of the core modules are listed below:

- `pygame.display` This module is used to control the display window or screen. It provides facilities to initialise and shutdown the display module. It can be used to initialise a window or screen. It can also be used to cause a window or screen to refresh, etc.
- `pygame.event` This module manages events and the event queue. For example `pygame.event.get()` retrieves events from the event queue, `pygame.event.poll()` gets a single event from the queue, and `pygame.event.peek()` tests to see if there are any event types on the queue.
- `pygame.draw` The draw module is used to draw simple shapes onto a `Surface`. For example, it provides functions for drawing a rectangle (`pygame.draw.rect`), a polygon, a circle, an ellipse, a line, etc.
- `pygame.font` The font module is used to create and render TrueType fonts into a new surface object. Most of the features associated with fonts are supported by the `pygame.font.Font` class. Free standing module functions allow the module to be initialised and shutdown, plus functions to access fonts such as `pygame.font.get_fonts()` which provides a list of the currently available fonts.
- `pygame.image` This module allows images to be saved and loaded. Note that images are loaded into a surface object (there is no image class unlike many other GUI-oriented frameworks).
- `pygame.joystick` The joystick module provides the joystick object and several supporting functions. These can be used for interacting with joysticks, gamepads and trackballs.
- `pygame.key` This module provides support for working with inputs from the keyboard. This allows the input keys to be obtained and modifier keys (such as control and shift) to be identified. It also allows the approach to repeating keys to be specified.
- `pygame.mouse` This module provides facilities for working with mouse input such as obtaining the current mouse position, the state of mouse buttons as well as the image to use for the mouse.

- `pygame.time` This is the pygame module for managing timing within a game. It provides the `pygame.time.Clock` class that can be used to track time.

21.9 Online Resources

There is a great deal of information available on pygame including:

- https://www.pygame.org The pygame home page.
- http://www.libsdl.org/ SDL (Simple Directmedia Layer) documentation.
- news://gmane.comp.python.pygame The official pygame news group.

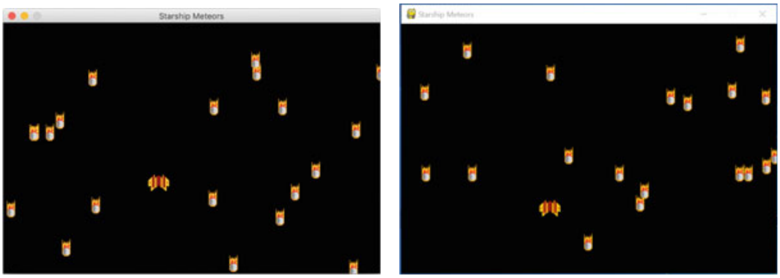
Chapter 22
StarshipMeteors Pygame

22.1 Introduction

In this chapter we will create a game in which you pilot a starship through a field of meteors. The longer you play the game, the larger the number of meteors you will encounter.

22.2 Creating a Spaceship Game

A typical display from the game is shown below for an Apple Mac and a Windows PC:

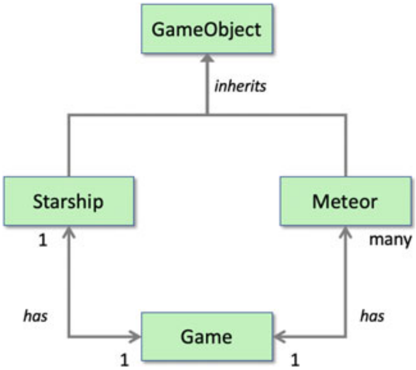


We will implement several classes to represent the entities within the game. Using classes is not a required way to implement a game, and it should be noted that many developers avoid the use of classes. However, using a class allows data associated with an object within the game to be maintained in one place; it also simplifies the

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_22

creation of multiple instances of the same object (such as the meteors) within the game.

The classes and their relationships are shown below:



This diagram shows that the `Starship` and `Meteor` classes will extend a class called `GameObject`.

In turn it also shows that the `Game` has a 1:1 relationship with the starship class. That is the `Game` holds a reference to one starship, and in turn the starship holds a single reference back to the `Game`.

In contrast the `Game` has a one to many relationship with the `Meteor` class. That is the game object holds references to many meteors, and each `Meteor` holds a reference back to the single `Game` object.

22.3 The Main Game Class

The first class we will look at will be the `Game` class itself.

The `Game` class will hold the list of meteors and the starship as well as the main game playing loop.

It will also initialise the main window display (e.g. by setting the size and the caption of the window).

In this case we will store the display surface returned by the `pygame.display.set_mode()` function in an attribute of the game object called `display_surface`. This is because we will need to use it later on to display the starship and the meteors.

We will also hold onto an instance of the `pygame.time.Clock()` class that we will use to set the frame rate each time round the main game playing `while` loop.

The basic framework of our game is shown below; this listing provides the basic `Game` class and the main method that will launch the game. The game also defines three global *constants* that will be used to define the frame refresh rate and the size of the display.

```
import pygame

# Set up Global 'constants'
FRAME_REFRESH_RATE = 30

DISPLAY_WIDTH = 600
DISPLAY_HEIGHT = 400

class Game:
    """ Represents the game itself, holds the main game playing 
loop """

  def __init__(self):
    print('Initialising PyGame')
    pygame.init()
    # Set up the display
      self.display_surface = pygame.display.set_mode((DISPLAY_
WIDTH, DISPLAY_HEIGHT))
    pygame.display.set_caption('Starship Meteors')
    # Used for timing within the program.
    self.clock = pygame.time.Clock()

  def play(self):
    is_running = True

    # Main game playing Loop
    while is_running:
      # Work out what the user wants to do
      for event in pygame.event.get():
        if event.type == pygame.QUIT:
          is_running = False
        elif event.type == pygame.KEYDOWN:
          if event.key == pygame.K_q:
            is_running = False

      # Update the display
      pygame.display.update()

      # Defines the frame rate
      self.clock.tick(FRAME_REFRESH_RATE)

    # Let pygame shutdown gracefully
    pygame.quit()

def main():
  print('Starting Game')
  game = Game()
  game.play()
  print('Game Over')

if __name__ == '__main__':
```

```
main()
```

The main `play()` method of the `Game` class has a loop that will continue until the user selects to quit the game. They can do this in one of two ways, either by pressing the 'q' key (represented by the `event.key` K_q) or by clicking on the window close button. In either case these events are picked up in the main event processing for loop within the main `while` loop method.

If the user does not want to quit the game, then the display is updated (refreshed) and then the `clock.tick()` (or frame) rate is set.

When the user selects to quit the game, then the main while loop is terminated (the `is_running` flag is set to `False`) and the `pygame.quit()` method is called to shut down pygame.

At the moment this not a very interactive game as it does not do anything except allow the user to quit. In the next section we will add in behaviour that will allow us to display the space ship within the display.

22.4 The GameObject Class

The `GameObject` class defines three methods:

The `load_image()` method can be used to load an image to be used to visually represent the specific type of game object. The method then uses the width and height of the image to define the width and height of the game object.

The `rect()` method returns a rectangle representing the current area used by the game object on the underlying drawing surface. This differs from the images own `rect()` which is not related to the location of the game object on the underlying surface. Rects are very useful for comparing the location of one object with another (e.g. when determining if a collision has occurred).

The `draw()` method draws the GameObjects' image onto the `display_surface` held by the game using the GameObjects current `x` and `y` coordinates. It can be overridden by subclasses if they wish to be drawn in a different way.

The code for the `GameObject` class is presented below:

```
class GameObject:

    def load_image(self, filename):
        self.image = pygame.image.load(filename).convert()
        self.width = self.image.get_width()
        self.height = self.image.get_height()

    def rect(self):
       """ Generates a rectangle representing the objects location
        and dimensions """
        return pygame.Rect(self.x, self.y, self.width, self.height)

    def draw(self):
        """ draw the game object at the
          current x, y coordinates """
```

```
self.game.display_surface.blit(self.image, (self.x, self.y))
```

The `GameObject` ô the `Starship` class and the `Meteor` class.

Currently there are only two types of elements: the starship and the meteors; but this could be extended in future to planets, comets, shooting stars, etc.

22.5 Displaying the Starship

The human player of this game will control a starship that can be moved around the display.

The starship will be represented by an instance of the class `Starship`. This class will extend the `GameObject` class that holds common behaviours for any type of element that is represented within the game.

The starship class defines its own `__init__()` method that takes a reference to the game that the starship is part of. This initialisation method sets the initial starting location of the starship as half the width of the display for the *x* coordinate and the display height minus 40 for the *y* coordinate (this gives a bit of a buffer before the end of the screen). It then uses the `load_image()` method from the `GameObject` parent class to load the image to be used to represent the `Starship`. This is held in a file called `starship.png`. For the moment we will leave the starship class as it is (however, we will return to this class so that we can make it into a movable object in the next section).

The current version of the starship class is given below:

```
class Starship(GameObject):
    """ Represents a starship"""

    def __init__(self, game):
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')
```

In the Game class we will now add a line to the `__init__()` method to initialise the starship object. This line is:

```
# Set up the starship
self.starship = Starship(self)
```

We will also add a line to the main while loop within the `play()` method just before we refresh the display. This line will call the `draw()` method on the starship object:

```
# Draw the starship
self.starship.draw()
```

This will have the effect of drawing the starship onto the windows drawing surface in the background before the display is refreshed.

When we now run this version of the StarshipMeteor game, we now see the starship in the display:

Of course at the moment the starship does not move; but we will address that in the next section.

22.6 Moving the Spaceship

We want to be able to move the starship about within the bounds of the display screen.

To do this, we need to change the starships `x` and `y` coordinates in response to the user pressing various keys.

We will use the arrow keys to move up and down the screen or to the left or right of the screen. To do this, we will define *four* methods within the starship class; these methods will move the starship up, down, left and right, etc.

The updated starship class is shown below:

```
class Starship(GameObject):
    """ Represents a starship"""

    def __init__(self, game):
        self.game = game
```

```
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')

    def move_right(self):
        """ moves the starship right across the screen """
        self.x = self.x + STARSHIP_SPEED
        if self.x + self.width > DISPLAY_WIDTH:
            self.x = DISPLAY_WIDTH - self.width

    def move_left(self):
        """ Move the starship left across the screen """
        self.x = self.x - STARSHIP_SPEED
        if self.x < 0:
            self.x = 0

    def move_up(self):
        """ Move the starship up the screen """
        self.y = self.y - STARSHIP_SPEED
        if self.y < 0:
            self.y = 0

    def move_down(self):
        """ Move the starship down the screen """
        self.y = self.y + STARSHIP_SPEED
        if self.y + self.height > DISPLAY_HEIGHT:
            self.y = DISPLAY_HEIGHT - self.height

    def __str__(self):
        return 'Starship(' + str(self.x) + ', ' + str(self.y) + ')'
```

This version of the starship class defines the various move methods. These methods use a new *global* value `STARSHIP_SPEED` to determine how far and how fast the starship moves. If you want to change the speed that the starship moves, then you can change this global value.

Depending upon the direction intended we will need to modify either the *x* or *y* coordinate of the Starship.

- If the starship moves to the left, then the `x` coordinate is reduced by `STARSHIP_SPEED`.
- if it moves to the right, then the `x` coordinate is increased by `STARSHIP_SPEED`.
- in turn if the starship moves up the screen, then the `y` coordinate is decremented by `STARSHIP_SPEED`.
- but if it moves down the screen, then the y coordinate is increased by `STARSHIP_SPEED`.

Of course we do not want our starship to fly off the edge of the screen, and so a test must be made to see if it has reached the boundaries of the screen. Thus tests are made to see if the `x` or `y` values have gone below zero or above the `DISPLAY_WIDTH` or `DISPLAY_HEIGHT` values. If any of these conditions are met, then the `x` or `y` values are reset to an appropriate default.

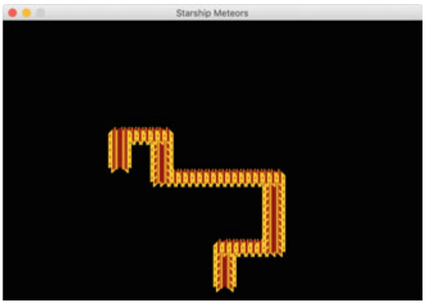
We can now use these methods with player input. This player input will indicate the direction that the player wants to move the starship. As we are using the left, right, up and down arrow keys for this we can extend the event processing loop that we have already defined for the main game playing loop. As with the letter q, the event keys are pre-fixed by the letter K and an underbar, but this time the keys are named K_LEFT, K_RIGHT, K_UP and K_DOWN.

When one of these keys is pressed, then we will call the appropriate move method on the starship object already held by the game object.

The main event processing for loop is now:

```
# Work out what the user wants to do
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        is_running = False
    elif event.type == pygame.KEYDOWN:
        # Check to see which key is pressed
        if event.key == pygame.K_RIGHT:
            # Right arrow key has been pressed
            # move the player right
            self.starship.move_right()
        elif event.key == pygame.K_LEFT:
            # Left arrow has been pressed
            # move the player left
            self.starship.move_left()
        elif event.key == pygame.K_UP:
            self.starship.move_up()
        elif event.key == pygame.K_DOWN:
            self.starship.move_down()
        elif event.key == pygame.K_q:
            is_running = False
```

However, we are not quite finished. If we try and run this version of the program, we will get a trail of starships drawn across the screen; for example:



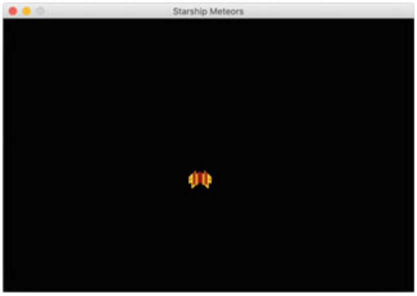
The problem is that we are redrawing the starship at a different position; but the previous image is still present.

We now have two choices: one is to merely fill the whole screen with black; effectively hiding anything that has been drawn so far; or alternatively we could just draw over the area used by the previous image position, which approach is adopted depends on the particular scenario represented by your game. As we will have a lot of meteors on the screen once we have added them; the easiest option is to overwrite everything on the screen before redrawing the starship. We will therefore add the following line:

```
# Clear the screen of current contents
self.display_surface.fill(BACKGROUND)
```

This line is added just before we draw the starship within the main game playing `while` loop.

Now when we move the starship, the old image is removed before we draw the new image:



One point to note is that we have also defined another global value `BACKGROUND` used to hold the background colour of the game playing surface. This is set to black as shown below:

```
# Define default RGB colours
BACKGROUND = (0, 0, 0)
```

If you want to use a different background colour, then change this global value.

22.7 Adding a Meteor Class

The `Meteor` class will also be a subclass of the `GameObject` class. However, it will only provide a `move_down()` method rather than the variety of move methods of the `Starship`.

It will also need to have a random x coordinate so that when a meteor is added to the game its starting position will vary. This random position can be generated using the `random.randint()` function using a value between 0 and the width of the drawing surface. The meteor will also start at the top of the screen so will have a different initial y coordinate to the starship. Finally, we also want our meteors to have different speeds; this can be another random number between 1 and some specified maximum meteor speed.

To support these, we need to add `random` to the modules being imported and define several new global values, for example:

```
import pygame, random

INITIAL_METEOR_Y_LOCATION = 10
MAX_METEOR_SPEED = 5
```

We can now define the `Meteor` class:

```
class Meteor(GameObject):
    """ represents a meteor in the game """

    def __init__(self, game):
        self.game = game
        self.x = random.randint(0, DISPLAY_WIDTH)
        self.y = INITIAL_METEOR_Y_LOCATION
        self.speed = random.randint(1, MAX_METEOR_SPEED)
        self.load_image('meteor.png')

    def move_down(self):
        """ Move the meteor down the screen """
        self.y = self.y + self.speed
        if self.y > DISPLAY_HEIGHT:
            self.y = 5

    def __str__(self):
        return 'Meteor(' + str(self.x) + ', ' + str(self.y) + ')'
```

The `__init__()` method for the `Meteor` class has the same steps as the starship; the difference is that the `x` coordinate and the speed are randomly generated. The image used for the meteor is also different as it is 'meteor.png'.

We have also implemented a `move_down()` method. This is essentially the same as the starships `move_down()`.

Note that at this point we could create a subclass of `GameObject` called `MoveableGameObject` (which extends GameObject) and push the move operations up into that class and have the `Meteor` and `Starship` classes extend that class. However we do not really want to allow meteors to move just anywhere on the screen.

We can now add the meteors to the `Game` class. We will add a new global value to indicate the number of initial meteors in the game:

```
INITIAL_NUMBER_OF_METEORS = 8
```

Next we will initialise a new attribute for the `Game` class that will hold a list of meteors. We will use a list here as we want to increase the number of meteors as the game progresses.

To make this process easy, we will use a list comprehension which allows a `for` loop to run with the results of an expression captured by the list:

```
# Set up meteors
self.meteors = [Meteor(self) for _ in range(0, INITIAL_NUMBER_OF_
METEORS)]
```

We now have a list of meteors that need to be displayed. We thus need to update the `while` loop of the `play()` method to draw not only the starship but also all the meteors:

```
# Draw the meteors and the starship
self.starship.draw()
for meteor in self.meteors:
    meteor.draw()
```

The end result is that a set of meteor objects are created at random starting locations across the top of the screen:

22.8 Moving the Meteors

We now want to be able to move the meteors down the screen so that the starship has some objects to avoid.

We can do this very easily as we have already implemented a `move_down()` method in the `Meteor` class. We therefore only need to add a for loop to the main game playing `while` loop that will move all the meteors. For example:

```
# Move the Meteors
for meteor in self.meteors:
   meteor.move_down()
```

This can be added after the event processing `for` loop and before the screen is refreshed/redrawn or updated.

Now when we run the game, the meteors move and the player can navigate the starship between the falling meteors.

22.9 Identifying a Collision

At the moment the game will play forever as there is no end state and no attempt to identify if a starship has collided with a meteor.

We can add meteor/starship collision detection using PyGame Rects. As mentioned in the last chapter, a `Rect` is a PyGame class used to represent rectangular coordinates. It is particularly useful as the `pygame.Rect` class provides several collision detection methods that can be used to test if one rectangle (or point) is inside another rectangle. We can therefore use one of the methods to test if the rectangle around the starship intersects with any of the rectangles around the meteors.

The `GameObject` class already provides a method `rect()` that will return a `Rect` object representing the objects current rectangle with respect to the drawing surface (essentially the box around the object representing its location on the screen).

Thus we can write a collision detection method for the `Game` class using the `GameObject` generated rects and the Rect class `colliderect()` method:

```
def _check_for_collision(self):
     """ Checks to see if any of the meteors have collided with
the starship """
```

```
    result = False
    for meteor in self.meteors:
        if self.starship.rect().colliderect(meteor.rect()):
            result = True
            break
    return result
```

Note that we have followed the convention here of preceding the method name with an underbar indicating that this method should be considered to provide to the class. It should therefore never be called by anything outside of the game class. This convention is defined in PEP 8 (Python Enhancement Proposal) but is not enforced by the language.

We can now use this method in the main `while` loop of the game to check for a collision:

```
# Check to see if a meteor has hit the ship
if self._check_for_collision():
    starship_collided = True
```

This code snippet also introduces a new local variable `starship_collided`. We will initially set this to `False` and is another condition under which the main game playing while loop will terminate:

```
is_running = True
starship_collided = False

# Main game playing Loop
while is_running and not starship_collided:
```

Thus the game playing loop will terminate if the user selects to quit or if the starship collides with a meteor.

22.10 Identifying a Win

We currently have a way to lose the game but we don't have a way to win the game! However, we want the player to be able to win the game by surviving for a specified period of time. We could represent this with a timer of some sort. However, in our case we will represent it as a specific number of cycles of the main game playing loop. If the player survives for this number of cycles, then they have won. For example:

```
# See if the player has won
if cycle_count == MAX_NUMBER_OF_CYCLES:
    print('WINNER!')
    break
```

In this case a message is printed out stating that the player won and then the main game playing loop is terminated (using the `break` statement).

The MAX_NUMBER_OF_CYCLES global value can be set as appropriate, for example:

```
MAX_NUMBER_OF_CYCLES = 1000
```

22.11 Increasing the Number of Meteors

We could leave the game at this point as it is now possible to both win and lose the game. However, there are a few things that can be easily added that will enhance the game playing experience. One of these is to increase the number of meteors on the screen making it harder as the game progresses.

We can do this using a NEW_METEOR_CYCLE_INTERVAL.

```
NEW_METEOR_CYCLE_INTERVAL = 40
```

When this interval is reached, we can add a new Meteor to the list of current meteors; it will then be automatically drawn by the game class. For example:

```
# Determine if new meteors should be added
if cycle_count % NEW_METEOR_CYCLE_INTERVAL == 0:
self.meteors.append(Meteor(self))
```

Now every NEW_METEOR_CYCLE_INTERVAL another meteor will be added at a random x coordinate to the game.

22.12 Pausing the Game

Another feature that many games have is the ability to pause the game. This can be easily added by monitoring for a pause key (this could be the letter *p* represented by the event_key pygame.K_p). When this is pressed, the game could be paused until the key is pressed again.

The pause operation can be implemented as a method _pause() that will consume all events until the appropriate key is pressed. For example:

```
def _pause(self):
    paused = True
    while paused:
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_p:
                    paused = False
                    break
```

In this method the outer while loop will loop until the paused local variable is set to False. This only happens when the '*p*' key is pressed. The break after the statement setting paused to `False` ensures that the inner `for` loop is terminated allowing the outer `while` loop to check the value of `paused` and terminate.

The `_pause()` method can be invoked during the game playing cycle by monitoring for the ‘p’ key within the event for loop and calling the `_pause()` method from there:

```
elif event.key == pygame.K_p:
    self._pause()
```

Note that again we have indicated that we don’t expect the `_pause()` method to be called from outside the game by pre-fixing the method name with an underbar (‘_’).

22.13 Displaying the Game Over Message

Pygame does not come with an easy way of creating a popup dialog box to display messages such as ‘You Won’ or ‘You Lost’ which is why we have used print statements so far. However, we could use a GUI framework such as wxPython to do this or we could display a message on the display surface to indicate whether the player has won or lost.

We can display a message on the display surface using the `pygame.font.Font` class. This can be used to create a `Font` object that can be rendered onto a surface that can be displayed onto the main display surface.

We can therefore add a method `_display_message()` to the game class that can be used to display appropriate messages:

```
def _display_message(self, message):
    """ Displays a message to the user on the screen """
    print(message)
    text_font = pygame.font.Font('freesansbold.ttf', 48)
    text_surface = text_font.render(message, True, BLUE, WHITE)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (DISPLAY_WIDTH / 2, DISPLAY_HEIGHT / 2)
    self.display_surface.fill(WHITE)
    self.display_surface.blit(text_surface, text_rectangle)
```

Again the leading underbar in the method name indicates that it should not be called from outside the game class.

We can now modify the main loop such that appropriate messages are displayed to the user, for example:

```
# Check to see if a meteor has hit the ship
if self._check_for_collision():
    starship_collided = True
    self._display_message('Collision: Game Over')
```

The result of the above code being run when a collision occurs is shown below:

22.14 The StarshipMeteors Game

The complete listing for the final version of the StarshipMeteors game is given below:

```
import pygame, random, time

FRAME_REFRESH_RATE = 30

DISPLAY_WIDTH = 600
DISPLAY_HEIGHT = 400
WHITE = (255, 255, 255)
BACKGROUND = (0, 0, 0)

INITIAL_METEOR_Y_LOCATION = 10
INITIAL_NUMBER_OF_METEORS = 8
MAX_METEOR_SPEED = 5
STARSHIP_SPEED = 10
MAX_NUMBER_OF_CYCLES = 1000
NEW_METEOR_CYCLE_INTERVAL = 40

class GameObject:

  def load_image(self, filename):
    self.image = pygame.image.load(filename).convert()
    self.width = self.image.get_width()
    self.height = self.image.get_height()

  def rect(self):
   """ Generates a rectangle representing the objects location
    and dimensions """
    return pygame.Rect(self.x, self.y, self.width, self.height)

  def draw(self):
    """ draw the game object at the
```

```python
        current x, y coordinates """
        self.game.display_surface.blit(self.image, (self.x, self.y))

class Starship(GameObject):
    """ Represents a starship"""

    def __init__(self, game):
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')

    def move_right(self):
        """ moves the starship right across the screen """
        self.x = self.x + STARSHIP_SPEED
        if self.x + self.width > DISPLAY_WIDTH:
            self.x = DISPLAY_WIDTH - self.width

    def move_left(self):
        """ Move the starship left across the screen """
        self.x = self.x - STARSHIP_SPEED
        if self.x < 0:
            self.x = 0

    def move_up(self):
        """ Move the starship up the screen """
        self.y = self.y - STARSHIP_SPEED
        if self.y < 0:
            self.y = 0

    def move_down(self):
        """ Move the starship down the screen """
        self.y = self.y + STARSHIP_SPEED
        if self.y + self.height > DISPLAY_HEIGHT:
            self.y = DISPLAY_HEIGHT - self.height

    def __str__(self):
        return 'Starship(' + str(self.x) + ', ' + str(self.y) + ')'

class Meteor(GameObject):
    """ represents a meteor in the game """

    def __init__(self, game):
        self.game = game
        self.x = random.randint(0, DISPLAY_WIDTH)
        self.y = INITIAL_METEOR_Y_LOCATION
        self.speed = random.randint(1, MAX_METEOR_SPEED)
        self.load_image('meteor.png')

    def move_down(self):
        """ Move the meteor down the screen """
        self.y = self.y + self.speed
        if self.y > DISPLAY_HEIGHT:
            self.y = 5

    def __str__(self):
```

```
        return 'Meteor(' + str(self.x) + ', ' + str(self.y) + ')'

class Game:
    """ Represents the game itself, holds the main game playing
loop """

    def __init__(self):
        pygame.init()
        # Set up the display
        self.display_surface = pygame.display.set_mode((DISPLAY_
WIDTH, DISPLAY_HEIGHT))
        pygame.display.set_caption('Starship Meteors')
        # Used for timing within the program.
        self.clock = pygame.time.Clock()
        # Set up the starship
        self.starship = Starship(self)
        # Set up meteors
        self.meteors = [Meteor(self) for _ in range(0, INITIAL_NUMBER_
OF_METEORS)]

    def _check_for_collision(self):
        """ Checks to see if any of the meteors have collided with
the starship """
        result = False
        for meteor in self.meteors:
            if self.starship.rect().colliderect(meteor.rect()):
                result = True
                break
        return result

    def _display_message(self, message):
        """ Displays a message to the user on the screen """
        text_font = pygame.font.Font('freesansbold.ttf', 48)
        text_surface = text_font.render(message, True, BLUE, WHITE)
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (DISPLAY_WIDTH / 2, DISPLAY_HEIGHT /
2)
        self.display_surface.fill(WHITE)
        self.display_surface.blit(text_surface, text_rectangle)

    def _pause(self):
        paused = True
        while paused:
            for event in pygame.event.get():
                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_p:
                        paused = False
                        break

    def play(self):
        is_running = True
        starship_collided = False
        cycle_count = 0

        # Main game playing Loop
        while is_running and not starship_collided:
```

```
            # Indicates how many times the main game loop has been
run
            cycle_count += 1

            # See if the player has won
            if cycle_count == MAX_NUMBER_OF_CYCLES:
                self._display_message('WINNER!')
                break

            # Work out what the user wants to do
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    is_running = False
                elif event.type == pygame.KEYDOWN:
                    # Check to see which key is pressed
                    if event.key == pygame.K_RIGHT:
                        # Right arrow key has been pressed
                        # move the player right
                        self.starship.move_right()
                    elif event.key == pygame.K_LEFT:
                        # Left arrow has been pressed
                        # move the player left
                        self.starship.move_left()
                    elif event.key == pygame.K_UP:
                        self.starship.move_up()
                    elif event.key == pygame.K_DOWN:
                        self.starship.move_down()
                    elif event.key == pygame.K_p:
                        self._pause()
                    elif event.key == pygame.K_q:
                        is_running = False

            # Move the Meteors
            for meteor in self.meteors:
                meteor.move_down()

            # Clear the screen of current contents
            self.display_surface.fill(BACKGROUND)

            # Draw the meteors and the starship
            self.starship.draw()
            for meteor in self.meteors:
                meteor.draw()

            # Check to see if a meteor has hit the ship
            if self._check_for_collision():
                starship_collided = True
                self._display_message('Collision: Game Over')

            # Determine if new meteors should be added
            if cycle_count % NEW_METEOR_CYCLE_INTERVAL == 0:
                self.meteors.append(Meteor(self))

            # Update the display
            pygame.display.update()
```

```
            # Defines the frame rate. The number is number of frames
per
           # second. Should be called once per frame (but only once)
            self.clock.tick(FRAME_REFRESH_RATE)

        time.sleep(1)
        # Let pygame shutdown gracefully
        pygame.quit()

def main():
    print('Starting Game')
    game = Game()
    game.play()
    print('Game Over')

if __name__ == '__main__':
    main()
```

22.15 Online Resources

There is a great deal of information available on PyGame including:

- https://www.pygame.org The PyGame home page.
- https://www.pygame.org/docs/tut/PygameIntro.html PyGame tutorial.
- https://www.python.org/dev/peps/pep-0008/ PEP8 Style Guide for Python Code.

22.16 Exercises

Using the example presented in this chapter add the following:

- Provide a score counter. This could be based on the number of cycles the player survives or the number of meteors that restart from the top of the screen, etc.
- Add another type of `GameObject`, this could be a shooting star that moves across the screen horizontally; perhaps using a random starting y coordinate.
- Allow the game difficulty to be specified at the start. This could affect the number of initial meteors, the maximum speed of a meteor, the number of shooting stars, etc.

Part IV
Testing

Chapter 23
Introduction to Testing

23.1 Introduction

This chapter considers the different types of tests that you might want to perform with the systems you develop in Python. It also introduces Test-Driven Development.

23.2 Types of Testing

There are at least two ways of thinking about testing:

1. It is the process of executing a program with the intent of finding errors/bugs (see Glenford Myers, The Art of Software Testing).
2. It is a process used to establish that software components fulfil the requirements identified for them that is that they do what they are supposed to do.

These two aspects of testing tend to have been emphasised at different points in the software lifecycle. Error testing is an intrinsic part of the development process, and an increasing emphasis is being placed on making testing a central part of software development (see Test-Driven Development).

It should be noted that it is extremely difficult—and in many cases impossible—to prove that software *works* and is completely *error-free*. The fact that a set of tests finds no defects does not prove that the software is error-free. 'Absence of evidence is not evidence of absence!'. This was discussed in the late 1960s and early 1970s by Dijkstra and can be summarised as:

> Testing shows the presence, not the absence of bugs

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_23

Testing to establish that software components fulfil their contract involves checking operations against their requirements. Although this does happen at development time, it forms a major part of Quality Assurance (QA) and User Acceptance testing. It should be noted that with the advent of Test-Driven Development, the emphasis on testing against requirements during development has become significantly higher.

There are of course many other aspects to testing; for example, Performance Testing which identifies how a system will perform as various factors that affect that system change, for example, as the number of concurrent requests increases, as the number of processors used by the underlying hardware changes, as the size of the database grows, etc.

However you view testing, the more the testing applied to a system, the higher the level of confidence that the system will work as required.

23.3 What Should Be Tested?

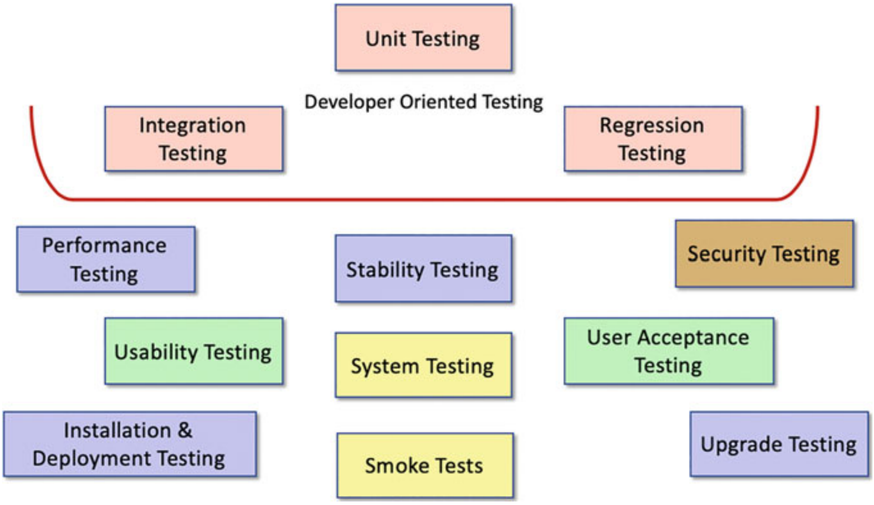
An interesting question is 'What aspects of your software system should be subject to testing?'.

In general, anything that is repeatable should be subject to formal (and ideally automated) testing. This includes (but is not limited to):

- The build process for all technologies involved.
- The deployment process to all platforms under consideration.
- The installation process for all runtime environments.
- The upgrade process for all supported versions (if appropriate).
- The performance of the system/servers as loads increase.
- The stability for systems that must run for any period of time (e.g. 24×7 systems).
- The backup processes.
- The security of the system.
- The recovery ability of the system on failure.
- The functionality of the system.
- The integrity of the system.

Notice that only the last two of the above list might be what are the commonly considered areas that would be subject to testing. However, to ensure the quality of the system under consideration, all of the above are relevant. In fact, testing should cover all aspects of the software development lifecycle and not just the QA phase. During requirements gathering testing is the process of looking for missing or ambiguous requirements. During this phase consideration should also be made with regard to how the overall requirements will be tested, in the final software system. Test planning should also look at all aspects of the software to test for functionality, usability, legal compliance, conformance to regulatory constraints, security, performance, availability, resilience, etc. Testing should be driven by the need to identify and reduce risk.

23.4 Types of Testing



As indicated above there are a number of different types of testing that are commonly used within industry. These types are.

- **Unit testing**, which is used to verify the behaviour of individual components.
- **Integration testing** that tests that when individual components are combined together to provide higher-level functional units that the combination of the units operates appropriately.
- **Regression testing**. When new components are added to a system, or existing components are changed, it is necessary to verify that the new functionality does not break any existing functionality. Such testing is known as regression testing.
- **Performance testing** is used to ensure that the systems' performance is as required and, within the design parameters, is able to scale as utilisation increases.
- **Stability testing** represents a style of testing which attempts to simulate system operation over an extended period of time. For example, for an online shopping application that is expected to be up and running 24 $\times$ 7 a stability test might ensure that with an average load that the system can indeed run 24 h a day for 7 days a week.
- **Security testing** ensures that access to the system is controlled appropriately given the requirements. For example, for an online shopping system there may be different security requirements depending upon whether you are browsing the store, purchasing some products, or maintaining the product catalogue.
- **Usability testing** which may be performed by a specialist usability group and may involve filming users while they use the system.

- **System testing** validates that the system as a whole actually meets the user requirements and conforms to required application integrity.
- **User acceptance testing** is a form of user-oriented testing where users confirm that the system does and behaves in the way they expect.
- **Installation, deployment and upgrade testing**. These three types of testing validate that a system can be installed and deployed appropriate including any upgrade processes that may be required.
- **Smoke tests** used to check that the core elements of a large system operate correctly. They can typically be run quickly and in a faction of the time taken to run the full system tests.

Key testing approaches are discussed in the remainder of this section.

23.4.1 Unit Testing

A unit can be as small as a single function or as large as a subsystem but typically is a class, object, self-contained library (API) or web page.

By looking at a small self-contained component, an extensive set of tests can be developed to exercise the defined requirements and functionality of the unit.

Unit testing typically follows a *white box* approach, (also called *Glass Box* or *Structural* testing), where the testing utilises knowledge and understanding of the code and its structure, rather than just its interface (which is known as the *black box* approach).

In *white box* testing, test coverage is measured by the number of code paths that have been tested. The goal in unit testing is to provide 100% coverage: to exercise every instruction, all sides of each logical branch, all called objects, handling of all data structures, normal and abnormal termination of all loops, etc. Of course this may not always be possible, but it is a goal that should be aimed for. Many automated test tools will include a code coverage measure so that you are aware of how much of your code has been exercised by any given set of tests.

Unit testing is almost always automated—there are many tools to help with this, perhaps the best-known being the xUnit family of test frameworks such as JUnit for Java and PyUnit for Python. The framework allows developers to:

- focus on testing the unit.
- simulate data or results from calling another unit (representative good and bad results).
- create data-driven tests for maximum flexibility and repeatability.
- rely on *mock* objects that represent elements outside the unit that it must interact with.

Having the tests automated means that they can be run frequently, at the very least after initial development and after each change that affects the unit.

Once confidence is established in the correct functioning of one unit, developers can then use it to help test other units with which it interfaces, forming larger units that can also be unit tested or, as the scale gets larger, put through integration testing.

23.4.2 Integration Testing

Integration testing is where several units (or modules) are brought together to be tested as an entity in their own right. Typically, integration testing aims to ensure that modules interact correctly and the individual unit developers have interpreted the requirements in a consistent manner.

An integrated set of modules can be treated as a unit and unit tested in much the same way as the constituent modules, but usually working at a "higher" level of functionality. Integration testing is the intermediate stage between unit testing and full system testing.

Therefore, integration testing focuses on the interaction between two or more units to make sure that those units work together successfully and appropriately. Such testing is typically conducted from the bottom-up but may also be conducted top-down using mocks or stubs to represent called or calling functions. An important point to note is that you should not aim to test everything together at once (so called *Big Bang* testing) as it is more difficult to isolate bugs in order that they can be rectified. This is why it is more common to find that integration testing has been performed in a bottom-up style.

23.4.3 System Testing

System testing aims to validate that the combination of all the modules, units, data, installation, configuration, etc., operates appropriately and meets the requirements specified for the whole system. Testing the system has a whole typically involving testing the top most functionality or behaviours of the system. Such behaviour-based testing often involves end-users and other stakeholders who are less technical. To support such tests, a range of technologies have evolved that allow a more *English* style for test descriptions. This style of testing can be used as part of the requirements gathering process and can lead to a behaviour-driven development (BDD) process. The Python module `pytest-bdd` provides a BDD style extension to the core PyTest framework.

23.4.4 Installation/Upgrade Testing

Installation testing is the testing of full, partial or upgrade install processes. It also validates that the installation and transition software needed to move to the new release for the product is functioning properly. Typically, it.

- verifies that the software may be completely uninstalled through its back-out process.
- determines what files are added, changed or deleted on the hardware on which the program was installed.
- determines whether any other programs on the hardware are affected by the new software that has been installed.
- determines whether the software installs and operates properly on all hardware platforms and operating systems that it is supposed to work on.

23.4.5 Smoke Tests

A smoke test is a test or suite of tests designed to verify that the fundamentals of the system work. Smoke tests may be run against a new deployment or a patched deployment in order to verify that the installation performs well enough to justify further testing. Failure to pass a smoke test would halt any further testing until the smoke tests pass. The name derives from the early days of electronics: If a device began to smoke after it was powered on, testers knew that there was no point in testing it further. For software technologies, the advantages of performing smoke tests include:

- Smoke tests are often automated and standardised from one build to another.
- Because smoke tests validate things that are expected to work, when they fail it is usually an indication that something fundamental has gone wrong (the wrong version of a library has been used) or that a new build has introduced a bug into core aspects of the system.
- If a system is built daily, it should be smoke tested daily.
- It will be necessary to periodically add to the smoke tests as new functionality is added to the system.

23.5 Automating Testing

The actual way in which tests are written and executed needs careful consideration. In general, we wish to automate as much of the testing process as is possible as this makes it easy to run the tests and also ensures not only that all tests are run but that they are run in the same way each time. In addition, once an automated test is set up it will typically be quicker to rerun that automated test than to manually repeat a series of tests. However, not all of the features of a system can be easily tested via an automated test tool, and in some cases the physical environment may make it hard to automate tests.

Typically, most unit testing is automated and most acceptance testing is manual. You will also need to decide which forms of testing must take place. Most software projects should have unit testing, integration testing, system testing and acceptance testing as a necessary requirement. Not all projects will implement performance or stability testing, but you should be careful about omitting any stage of testing and be sure it is not applicable.

23.6 Test-Driven Development

Test-Driven Development (or TDD) is a development technique whereby developers write test cases *before* they write any implementation code. The tests thus drive or dictate the code that is developed. The implementation only provides as much functionality as is required to pass the test, and thus the tests act as a specification of *what* the code does (and some argue that the tests are thus part of that specification and provide documentation of what the system is capable of).

TDD has the benefit that as tests must be written first, there are always a set of tests available to perform unit, integration, regression testing, etc. This is good as developers can find that writing tests and maintaining tests is boring and of less interest than the actual code itself, and this puts less emphasis into the testing regime than might be desirable. TDD encourages, and indeed requires, that developers maintain an exhaustive set of repeatable tests and that those tests are developed to the same quality and standards as the main body of code.

There are three rules of TDD as defined by Robert Martin; these are:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

This leads to the TDD cycle described in the next section.

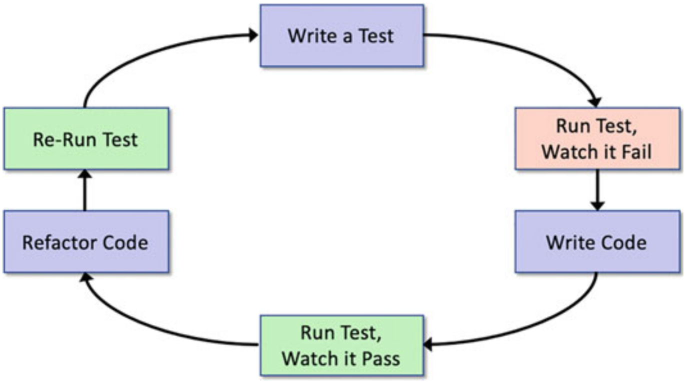
23.6.1 The TDD Cycle

There is a cycle to development when working in a TDD manner. The shortest form of this cycle is the TDD mantra:

Red/Green/Refactor

which relates to the unit testing suite of tools where it is possible to write a unit test. Within tools such as PyCharm, when you run a PyUnit or PyTest test a Test View is shown with red indicating that a test failed or green indicating that the test passed. Hence red/green, in other words write the test and let it fail, then implement the code to ensure it passes. The last part of this mantra is *refactor* which indicates once you have it working make the code cleaner, better and fitter by refactoring it. Refactoring is the process by which the behaviour of the system is not changed, but the implementation is altered to improve it.

The full TDD cycle is shown by the following diagram which highlights the test first approach of TDD:



The TDD mantra can be seen in the TDD cycle that is shown above and described in more detail below:

1. Write a single test.
2. Run the test and see it **fail**.
3. Implement *just enough* code to get the test to pass.
4. Run the test and see it **pass**.
5. **Refactor** for clarity and deal with any issue of reuse, etc.
6. Repeat for next test.

23.6.2 Test Complexity

The aim is to strive for simplicity in all that you do within TDD. Thus, you write a test that fails, then do just enough to make that test pass (but no more). Then you refactor the implementation code (i.e. change the internals of the unit under test) to improve the code base. You continue to do this until all the functionality for a unit has been completed. In terms of each test, you should again strive for simplicity with each test only testing one thing with only a single assertion per test (although this is the subject of a lot of debate within the TDD world).

23.6.3 Refactoring

The emphasis on refactoring within TDD makes it more than just testing or Test First Development. This focus on refactoring is really a focus on (re)design and incremental improvement. The tests provide the specification of what is needed as well as the verification that existing behaviour is maintained, but refactoring leads to better design software. Thus, without refactoring TDD is not TDD!

23.7 Design for Testability

Testability has a number of facets

1. Configurability. Set up the object under test to an appropriate configuration for the test.
2. Controllability. Control the input (and internal state).
3. Observability. Observe its output.
4. Verifiability. That we can verify that output in an appropriate manner.

23.7.1 Testability Rules of Thumb

> If you cannot test code, then change it so that you can!
> If your code is difficult to validate, then change it so that it isn't!
> Only one concrete class should be tested per Unit test and then *Mock the Rest*!
> If you code is hard to reconfigure to work with mocks, then make it so that your code can use Mocks!
> Design your code for testability!

23.8 Online Resources

See the following online resources for more information on testing and Test-Driven Development (TDD).

1. https://www.test-institute.org/Introduction_To_Software_Testing.php Introduction to Software Testing.
2. https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Testing Introduction to software Testing wiki book.

3. https://en.wikipedia.org/wiki/Test-driven_development Test-Driven Development wikipedia page.
4. http://agiledata.org/essays/tdd.html an introduction to Test-Driven Development.
5. https://medium.freecodecamp.org/learning-to-test-with-python-997ace2d8abe a simple introduction to TDD with Python.
6. http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd Robert Martins three rules for TDD.
7. http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata The Bowling Game Kata which presents a worked example of how TDD can be used to create a Ten Pin Bowls scoring keeping application.

23.9 Book Resources

1. *The Art of Software Testing*, G. J. Myers, C. Sandler and T. Badgett, John Wiley & Sons, 3rd Edition(Dec 2011), 1,118,031,962.

Chapter 24
PyTest Testing Framework

24.1 Introduction

There are several testing frameworks available for Python, although only one, unittest comes as part of the typical Python installation. Typical libraries include unittest (which is available within the Python distribution by default) and PyTest (which actually builds directly on top of unittest).

In this chapter we will look at PyTest and how it can be used to write unit tests in Python for both functions and classes.

24.2 What is PyTest?

PyTest is a testing library for Python; it is currently one of the most popular Python testing libraries (others include unittest and doctest). PyTest can be used for various levels of testing, although its most common application is as a unit testing framework. It is also often used as a testing framework within a TDD-based development project. In fact, it is used by Mozilla and Dropbox as their Python testing framework.

PyTest offers a large number of features and great flexibility in how tests are written and in how set up behaviour is defined. It automatically finds test based on naming conventions and can be easily integrated into a range of editors and IDEs including PyCharm.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_24

24.3 Setting up PyTest

You will probably need to set up PyTest so that you can use it from within your environment. If you are using the PyCharm editor, then you will need to add the PyTest module to the current PyCharm project and tell PyCharm that you want to use PyTest to run all tests for you. At the time of writing this is PyTest version 7.3.2—you can use pip to install PyTest.

24.4 A Simple PyTest Example

Something to test

To be able to explore PyTest we first need something to test; we will therefore define a simple `Calculator` class. The calculator keeps a running total of the operations performed; it allows a new value to be set, and then this value can be added to, or subtracted from, that accumulated total.

```
class Calculator:
    def __init__(self):
        self.current = 0
        self.total = 0

    def set(self, value):
        self.current = value

    def add(self):
        self.total += self.current

    def sub(self):
        self.total -= self.current

    def total(self):
        return self.total
```

Save this class into a file called `calculator.py`.

Writing a Test

We will now create a very simple PyTest unit test for our `Calculator` class. This test will be defined in a class called test_calculator.py.

You will need to *import* the calculator class we wrote above into your test_caclculator.py file (remember each file is a module in Python).

The exact import statement will depend on where you placed the calculator file relative to the test class. In this case the two files are both in the same directory and so we can write:

```
from calculator import Calculator
```

We will now define a test, the test should be pre-fixed with `test_` for PyTest to find them. In fact PyTest uses several conventions to find tests, which are:

- Search for `test_*.py` or `*_test.py` files.
- From those files, collect test items:
  - test_ pre-fixed test functions,
  - test_ pre-fixed test methods inside `Test` pre-fixed test classes (without an `__init__` method)

Note that we keep test files and the files containing the code to be tested separate; indeed in many cases they are kept in different directory structures. This means that there is no chance of developers accidentally using tests in production code, etc.

Now we will add to the file a function that defines a test. We will call the function `test_add_one`; it needs to start with `test_` due to the above convention. However, we have tried to make the rest of the function name descriptive, so that it is clear what it is testing. The function definition is given below:

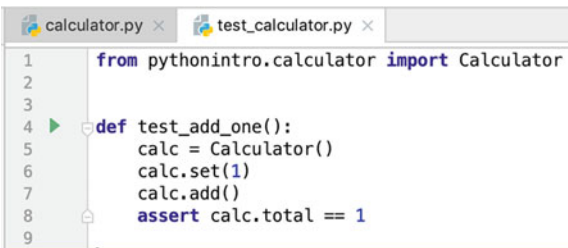
```
from calculator import Calculator

def test_add_one():
    calc = Calculator()
    calc.set(1)
    calc.add()
    assert calc.total == 1
```

The test function creates a new instance of the `Calculator` class and then calls several methods on it; to set up the value to add, then the call to the `add()` method itself, etc.

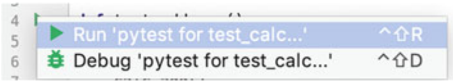
The final part of the test is the assertion. The assert verifies that the behaviour of the calculator is as expected. The PyTest `assert` statement works out what is being tested and what it should do with the result—including adding information to a test run report. It avoids the need to have to learn a load of *assertSomething* type methods (unlike some other testing frameworks).

Note that a test without an assertion is not a test; i.e. it does not test anything.

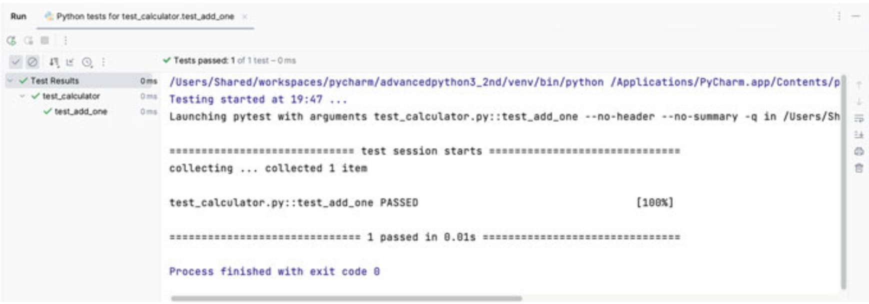
Many IDEs provide direct support for testing frameworks including PyCharm. For example, PyCharm will now detect that you have written a function with an assert statement in it and add a *Run Test* icon to the grey area to the left of the editor. This can be seen in the following picture where a green arrow has been added at line 4; this is the 'Run Test' button:



The developer can click on the green arrow to run the test. They will then be presented with the *Run* menu that is pre-configured to use PyTest for you:



If the developer now selects the Run option, this will use the PyTest runner to execute the test and collect information about what happened and present it in a PyTest output view at the bottom of the IDE:

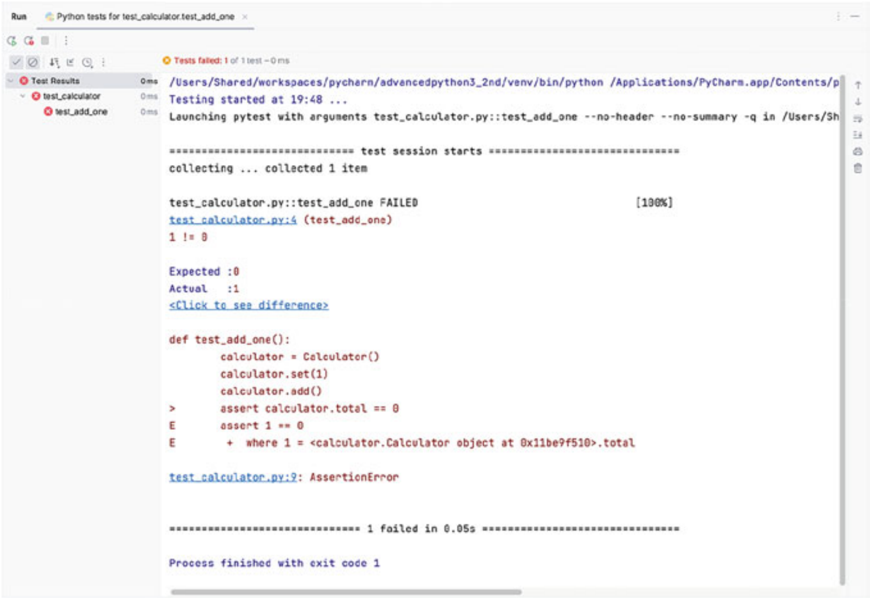


Here you can see a tree in the left-hand panel that currently holds the one test defined in the `test_calculator.py` file. This tree shows whether tests have passed or failed. In this case we have a green tick showing that the test passed.

To the right of this tree is the main output panel which shows the results of running the tests. In this case it shows that PyTest ran only one test and that this was the `test_add_one` test which was defined in `test_calculator.py` and that one test passed.

If you now change the assertion in the test to check to see that the result is 0, the test will fail. The IDE display will update accordingly.

The tree in the left-hand pane now shows the test as failed while the right-hand pane provides detailed information about the test that failed including where in the test the failed assertion was defined. This is very helpful when trying to debug test failures.



24.5 Working with PyTest

Testing Functions

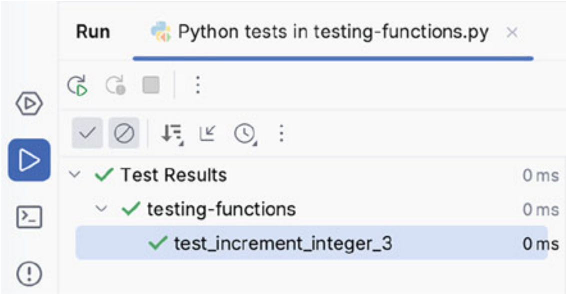
We can test *stand-alone* functions as well as classes using PyTest. For example, given the function `increment` below (which merely adds one to any number passed into it):

```python
def increment(x):
    return x + 1
```

We can write a PyTest test for this as follows:

```python
def test_increment_integer_3():
    assert increment(3) == 4
```

The only real difference is that we have not had to make an instance of a class:



Organising Tests

Tests can be grouped together into one or more files; PyTest will search for all files following the naming convention (file names that either start or end with '`test`') in specified locations:

- If no arguments are specified when PyTest is run, then the search for suitably named test files starts from the `testpaths` environment variable (if configured) or the current directory. Alternatively, command line arguments can be used in any combination of directories or filenames, etc.
- PyTest will recursively search down into subdirectories, unless they match `norecursedirs` environment variable.
- In those directories, it will search for files that match the naming conventions `test_*.py` or `*_test.py` files.

Tests can also be arranged within test files into test classes. Using test classes can be helpful in grouping tests together and managing the setup and tear down behaviours of separate groups of tests. However, the same effect can be achieved by separating the tests relating to different functions or classes into different files.

Test Fixtures

It is not uncommon to need to run some behaviour before or after each test or indeed before or after a group of tests. Such behaviours are defined within what is commonly known as test fixtures.

We can add specific code to run:

- at the beginning and end of a test class module of test code (setup_module/teardown_module).
- at the beginning and end of a test class (setup_class/teardown_class) or using the alternate style of the class level fixtures (setup/teardown).
- before and after a test function call (setup_function/teardown_function).
- before and after a test method call (setup_method/teardown_method).

To illustrate why we might use a fixture, let us expand our `Calculator` test:

```python
def test_initial_value():
    calc = Calculator()
    assert calc.total == 0

def test_add_one():
    calc = Calculator()
    calc.set(1)
    calc.add()
    assert calc.total == 1

def test_subtract_one():
    calc = Calculator()
    calc.set(1)
    calc.sub()
    assert calc.total == -1

def test_add_one_and_one():
    calc = Calculator()
    calc.set(1)
    calc.add()
    calc.set(1)
    calc.add()
    assert calc.total == 2
```

We now have four tests to run (we could go further but this is enough for now).

One of the issues with this set of tests is that we have repeated the creation of the `Calculator` object at the start of each test. While this is not a problem in itself it does result in duplicated code and the possibility of future issues in terms of maintenance if we want to change the way a calculator is created. It may also not be as efficient as reusing the calculator object for each test.

We can, however, define a fixture that can be run before each individual test function is executed. To do this we will write a new function and use the `pytest.fixture` decorator on that function. This marks the function as being special and that it can be used as a fixture on an individual function.

Functions that require the fixture should accept a reference to the fixture as an argument to the individual test function. For example, for a test to accept a fixture called `calculator`, it should have an argument with the fixture name, i.e. `calculator`. This name can then be used to access the object returned. This is illustrated below:

```python
import pytest
from calculator import Calculator

@pytest.fixture
def calculator():
    """Returns a Calculator instance"""
    return Calculator()

def test_initial_value(calculator):
    assert calculator.total == 0

def test_add_one(calculator):
    calculator.set(1)
    calculator.add()
```

```
        assert calculator.total == 1

    def test_subtract_one(calculator):
        calculator.set(1)
        calculator.sub()
        assert calculator.total == -1

    def test_add_one_and_one(calculator):
        calculator.set(1)
        calculator.add()
        calculator.set(1)
        calculator.add()
        assert calculator.total == 2
```

In the above code, each of the test functions accepts the `calculator` fixture that is used to instantiate the `Calculator` object. We have therefore *de-duplicated* our code; there is now only one piece of code that defines how a calculator object should be created for our tests. Note each test is supplied with a completely new instance of the `Calculator` object; there is therefore no chance of one test impacting on another test.

It is also considered good practice to add a *docstring* to your fixtures as we have done above. This is because PyTest can produce a list of all fixtures available along with their docstrings. From the command line this is done using:

```
> pytest fixtures
```

From the command line.

The PyTest fixtures can be applied to functions (as above), classes, modules, packages or sessions. The `scope` of a fixture can be indicated via the (optional) scope parameter to the fixture decorator. The default is "function" which is why we did not need to specify anything above. The scope determines at what point a fixture should be run. For example, the session will be run once for the test session, the module will be run once for the module (i.e. the fixture and anything it generates will be shared across all tests in the current module), the class scope indicates a fixture that is run for each new instance of a test class created, etc.

Another parameter to the fixture decorator is `autouse` which if set to `True` will activate the fixture for all tests that can see it. If it is set to `False` (which is the default), then an explicit reference in a test function (or method, etc.) is required to activate the fixture.

If we add some additional fixtures to our tests, we can see when they are run:

```
import pytest
from calculator import Calculator

    @pytest.fixture(scope='session', autouse=True)
    def session_scope_fixture():
        print('session_scope_fixture')

    @pytest.fixture(scope='module', autouse=True)
    def module_scope_fixture():
        print('module_scope_fixture')
```

```
    @pytest.fixture(scope='class', autouse=True)
    def class_scope_fixture():
        print('class_scope_fixture')

        @pytest.fixture
        def calculator():
            """Returns a Calculator instance"""
            print('calculator fixture')
            return Calculator()

def test_initial_value(calculator):
    assert calculator.total == 0

def test_add_one(calculator):
    calculator.set(1)
    calculator.add()
    assert calculator.total == 1

def test_subtract_one(calculator):
    calculator.set(1)
    calculator.sub()
    assert calculator.total == -1

def test_add_one_and_one(calculator):
    calculator.set(1)
    calculator.add()
    calculator.set(1)
    calculator.add()
    assert calculator.total == 2
```

If we run this version of the tests, then the output shows when the various fixtures are run:

```
session_scope_fixture
module_scope_fixture
class_scope_fixture
calculator fixture
.class_scope_fixture
calculator fixture
.class_scope_fixture
calculator fixture
.class_scope_fixture
calculator fixture
```

Note that higher scoped fixtures are instantiated first.

24.6 Parameterised Tests

One common requirement of a test is to run the same tests multiple times with several different input values. This can greatly reduce the number of tests that must be defined. Such tests are referred to as parametrised tests, with the parameter values for the test specified using the `@pytest.mark.parametrize` decorator.

```
@pytest.mark.parametrize('input1,input2,expected', [
      (3, 1, 4),
      (3, 2, 5),
])
def  test_calculator_add_operation(calculator,  input1,  input2,
expected):
     calculator.set(input1)
     calculator.add()
     calculator.set(input2)
     calculator.add()
     assert calculator.total == expected
```

This illustrates setting up a parametrised test for the `Calculator` in which two input values are added together and compared with the expected value. Note that the parameters are named in the decorator and then a list of tuples is used to define the values to be used for the parameters. In this case the `test_calculator_add_operation` will be run two passing in 3, 1 and 4 and then passing in 3, 2 and 5 for the parameters `input1`, `input2` and `expected`, respectively.

Testing for Exceptions

You can write tests that verify that an exception was raised. This is useful as testing negative behaviour is as important as testing positive behaviour. For example, we might want to verify that a particular exception is raised when we attempt to withdraw money from a bank account which will take us over our overdraft limit.

To verify the presence of an exception in PyTest, use the `with` statement and `pytest.raises`. This is a *context manager* that will verify on exit that the specified exception was raised. It is used as follows:

```
with pytest.raises(accounts.BalanceError):
     current_account.withdraw(200.0)
```

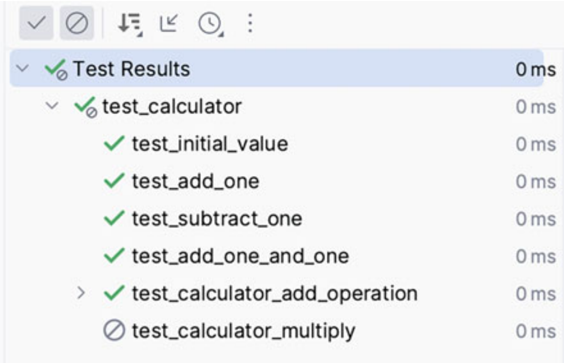
Ignoring tests

In some cases it is useful to write a test for functionality that has not yet been implemented; this may be to ensure that the test is not forgotten or because it helps to document what the item under test should do. However, if the test is run, then the test suite as a whole will fail because the test is running against behaviour that has yet to be written.

One way to address this problem is to decorate a test with the `@pytest.mark.skip` decorator:

```
@pytest.mark.skip(reason='not implemented yet')
def test_calculator_multiply(calculator):
    calculator.multiply(2, 3)
    assert calculator.total == 6
```

This indicates that PyTest should record the presence of the test but should not try to execute it. PyTest will then note that the test was skipped; for example in PyCharm this is shown using a circle with a line through it.



It is generally considered best practice to provide a reason why the test has been skipped so that it is easier to track. This information is also available when PyTest skips the test:

24.7 Online Resources

See the following online resources for information on PyTest:

- http://pythontesting.net/framework/PyTest/PyTest-introduction/ PyTest introduction.
- https://github.com/pluralsight/intro-to-PyTest An example based introduction to PyTest.
- https://docs.pytest.org/en/latest/ PyTest home page.
- https://docs.pytest.org/en/latest/#documentation PyTest documentation.

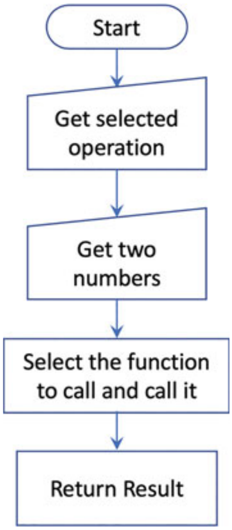
24.8 Exercises

Create a simple `Calculator` class that can be used for testing purposes. This simple calculator can be used to add, subtract, multiple and divide numbers.

This will be a purely command-driven application that will allow the user to specify.

- the operation to perform.
- the two numbers to use with that operation.

The `Calculator` object will then return a result. The same object can be used to repeat this sequence of steps. This general behaviour of the `Calculator` is illustrated below in flow chart form:



You should also provide a memory function that allows the current result to be added to or subtracted from the current memory total. It should also be possible to retrieve the value in memory and clear the memory.

Next write a PyTest set of tests for the `Calculator` class.

Think about what tests you need to write; remember you can't write tests for every value that might be used for an operation; but consider the boundaries, 0, − 1, 1, − 10, + 10, etc.

Of course you also need to consider the cumulative effect of the behaviour of the memory feature of the calculator; that is multiple memory adds or memory subtractions and combinations of these.

As you identify tests you may find that you have to update your implementation of the `Calculator` class. Have you taken into account all input options; for example dividing by zero, what should happen in these situations?

Chapter 25
Mocking for Testing

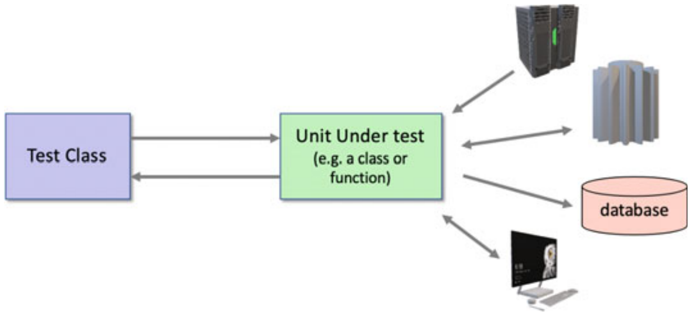
25.1 Introduction

Testing software systems is not an easy thing to do; the functions, objects, methods, etc., that are involved in any program can be complex things in their own right. In many cases they depend on and interact with other functions, methods and objects; very few functions and methods operate in isolation. Thus the success of failure of a function or method or the overall state of an object is dependent on other program elements.

However, in general it is a lot easier to test a single *unit* in isolation rather than to test it as part of a larger more complex system. For example, let us take a Python class as a single *unit* to be tested. If we can test this class on its own, we only have to take into account the state of the classes object and the behaviour defined for the class when writing our tests and determining appropriate outcomes.

However, if that class interacts with external systems such as external services, databases, third-party software and data sources, then the testing process becomes more complex:

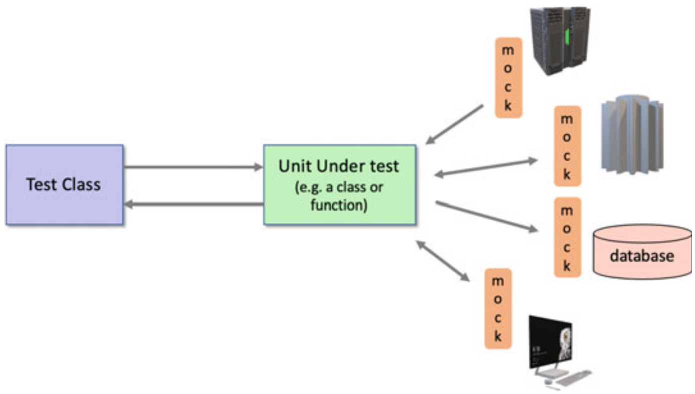
J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_25



It may now be necessary to verify data updates made to the database, or information sent to a remote service, etc., to confirm that the operation of a class's object is correct. This makes not only the software being tested more complex but it also makes the tests themselves more complex. This means that there is greater chance that the test will fail, that the tests will contain bugs or issues themselves and that the test will be harder for someone to understand and maintain. Thus a common objective when writing unit tests or subsystem tests is to be able to test elements/ units in isolation.

The question is how to do this when a function or method relies on other elements?

The key to decoupling functions, methods and objects from other program or system elements is to use mocks. These mocks can be used to decouple one object from another, one function from another and one system from another, thereby simplifying the testing environment. These mocks are only intended to be used for testing purposes; for example the above scenario could be simplified by mocking out each of the external systems as shown below:



Mocking is not a Python specific concept, and there are many mocking libraries available for many different languages. However, in this chapter we will be focussing on the unittest.mock library which has been part of the standard Python distribution since Python 3.3.

25.2 Why Mock?

A useful first question considered with regard to mocking in software testing is 'Why mock?'. That is, why bother with the concept of a mock in the first place; why not test with the *real thing*?

There are several answers to this, some of which are discussed below:

- **Testing in isolation is easier**. As mentioned in the introduction, testing a unit (whether that is a class, a function, a module, etc.) is easier in isolation and then when dependent on external classes, functions, modules, etc.
- **The real thing is not available**. In many cases it is necessary to mock out part of a system or an interface to another system because the *real thing* is just not available. This could be for several reasons including that it has not been developed yet. In the natural course of software development some parts of a system are likely to be developed and ready for testing before other parts. If one part relies on another part for some element of its operation, then the system that is not yet available can be mocked out. In other situations the development team or test team may not have access to the *real thing*. This may be because it is only available within a production context. For example, if a software development house is developing one subsystem, it may not have access to another subsystem as it is proprietary and only accessible once the software has been deployed within the client organisation.
- **Real elements can be time consuming**. We want our tests to run as quickly as possible and certainly within a Continuous Integration (CI) environment we want them to run fast enough that we can repeatedly test a system throughout the day. In some situations the *real thing* may take a significant amount of time to process the test scenario. As we want to test our own code we may not be worried about whether a system outside of our control operates correctly or not (at least at this level testing; it may still be a concern for integration and system testing). We can therefore improve the response times of our tests if we mock out the real system and replace it with a mock that provides much faster response times (possibly because it uses canned responses).
- **The real thing takes time to set up**. In a Continuous Integration (CI) environment, new bulls of system are regularly and repeatedly tested (e.g. whenever a change is made to their codebase). In such situations it may be necessary to configure and deploy the final system to a suitable environment to perform appropriate tests. If an external system is time consuming to configure, deploy and initialise, it may be more effective to mock that system out.

- **Difficult to emulate certain situations**. It can be difficult within a test scenario to emulate specific situations. These situations are often related to error or exceptional circumstances that should never happen within a correctly functioning environment. However, it may well be necessary to validate that if such a situation does occur, then the software can deal with that scenario. If these scanners are related to how external (the unit under test) system fails or operates incorrectly, then it may be necessary to mock out these systems to be able to generate the scenarios.
- **We want repeatable tests**. By their very nature when you run a test, you either want it to pass or fail each time it is run with the same inputs. You certainly do not want tests that pass sometimes and fail other times. This means that there is no confidence in the tests, and people often start ignoring failed tests. This situation can happen if the data provided by systems that a test depends on does not supply repeatable data. This can happen for several different reasons, but a common cause is because they return real data. Such real data may be subject to change, for example consider a system that uses a data feed for the current exchange rate between funds and dollars. If the associated test confirms that a trade when priced in dollars is correctly converted to funds using the current exchange rate, then that test is likely to generate a different result every time it is run. In this situation it would lie better to mock out the current exchange rate service so that a fixed/known exchange rate is used.
- **The Real System is not reliable enough**. In some cases the real system may not be reliable enough itself to allow for repeatable tests.
- **The Real System may not allow tests to be repeated**. Finally, the real system may not allow tests to be easily repeated. For example, a test which involves lodging a trade for a certain number of IBM shares with an trade order management system may not allow that trade, with those shares, for that customer to be run several times (as it would then appear to be multiple trades). However, for the purposes of testing we may want to test submitting such a trade in multiple different scenarios, multiple times. It may therefore be necessary to mock out the real order management system so that such tests can be written.

25.3 What is Mocking?

The previous section gave several reasons to use mocks; the next thing to consider then is what is a mock?

Mocks, both mock functions, methods and mock objects are things that:

- Possess the same *interface* as the real thing, whether they are mock functions, methods or whole objects. They thus take the same range and types of parameters and return similar information using similar types.
- Define behaviour that in some way represents/mimics real exemplar behaviour but typically in very controlled ways. This behaviour may be hard coded, may be

really on a set of rules or simplified behaviour and may be very simplistic or quite sophisticated in its own right.

They thus emulate the real system and from outside of the mock may actually appear to be the real system.

In many cases the term mock is used to cover a range of different ways in which the real thing can be emulated; each type of mock has its own characteristics. It is therefore useful to distinguish the different types of mocks as this can help determine the style of mock to be adopted in a particular test situation.

There are different types of mock include:

- **Test Stubs**. A test stub is typically a hand coded function, method or object used for testing purposes. The behaviour implemented by a test stub may represent a limited subset of the functionality of the real thing.
- **Fakes**. Fakes typically provide addition functionality compared with a test stub. Fakes may be considered to be a test specific version of the real thing, such as an in memory database used for testing rather than the real database. Such fakes typically still have some limitations on their functionality; for example when the tests are terminated all data is purged from the in memory database rather than stored permanently on disc.
- **Autogenerated Test Mocks**. These are typically generated automatically using a supporting framework. As part of the setup of the test the expectations are associated with the test mock. These expectations may specify the results to return for specific inputs as well as whether the test mock was called, etc.
- **Test Mock Spy**. If we are testing a particular unit and it returns the correct result, we might decide that we do not need to consider the internal behaviour of the unit. However, it is common to want to confirm that the test mock was invoked in the way we expected. This helps verify the internal behaviour of the unit under test. This can be done using a test mock spy. Such a test mock records how many times it was called and what the parameters used were (as well as other information). The test can then interrogate the test mock to validate that it was invoked as expected/as many times as expected/with the correct parameters, etc.

25.4 Common Mocking Framework Concepts

As has been mentioned there are several mocking frameworks around for not only Python but other languages such as Java, C# and Scala. All of these frameworks have a common core behaviour. This behaviour allows a mock function, method or object to be created based on the *interface* presented by the real thing. Of course unlike languages such as C# and Java Python do not have a formal interface concept, however this does not stop the mocking framework from still using the same idea.

In general once a mock has been created it is possible to define how that mock should appear to behave; in general this involves specifying the return result to use for a function or method. It is also possible to verify that the mock has been invoked as expected with the parameters expected.

The actual mock can be added to a test or a set of tests either programmatically or via some form of decorator. In either case for the duration of the test the mock will be used instead of the real thing.

Assertions can then be used to verify the results returned by the unit under test, while mock specific methods are typically used to verify (spy on) the methods defined on the mock.

25.5 Mocking Frameworks for Python

Due to Python's dynamic nature it is well suited to the construction of mock functions, methods and objects. In fact there are several widely used mocking frameworks available for Python including:

- **unittest.mock** The `unittest.mock` (included in the Python distribution from Python 3.3 onwards). This is the default mocking library provided with Python for creating mock objects in Python tests.
- **pymox** This is a widely used making framework. It is an open-source framework and has a more complete set of facilities for enforcing the interface of a mocked class.
- **Mocktest** This is another popular mocking framework. It has its own Domain Specific Language (DSL) to support mocking and a wide set of expectation matching behaviour for mock objects.

In the remainder of this chapter we will focus on the `unittest.mock` library as it is provided as part of the standard Python distribution.

25.6 The Unittest.Mock Library

The standard Python mocking library is the `unittest.mock` library. It has been included in the standard Python distribution since Python 3.3 and provides a simple way to define mocks for unit tests.

The key to the unites.mock library is the mock class and its subclass MagicMock. Mock and MagicMock objects can be used to mock functions, methods and even while classes. These mock objects can have canned responses defined so that when they are involved by the unit under test, they will respond appropriately. Existing objects can also have attributes or individual methods mocked allowing an object to be tested with a known state and specified behaviour.

To make it easy to work with mock objects, the library provides the `@unittest.mock.pactch()` decorator. This decorator can be used to replace real functions and objects with mock instances. The function behind the decorator can also be used as a context manager allowing it to be used in with-as statements providing for fine-grained control over the scope of the mock if required.

25.6.1 Mock and Magic Mock Classes

The unittest.mock library provides the `Mock` class and the `MagicMock` class. The `Mock` class is the base class for mock objects. The `MagicMock` class is a subclass of the `Mock` class. It is called the `MagicMock` class as it provides default implementations for several magic method such as.__len__(),.__str__(), and.__iter__().

As a simple example consider the following class to be tested:

```
class SomeClass():

    def _hidden_method(self):
        return 0

    def public_method(self, x):
        return self.hidden_method() + x
```

This class defines two methods; one is intended as part of the public interface of the class (the `public_method()`) and one is intended only for internal or private use (`the _hidden_method()`). Notice that the hidden method uses the convention of preceding its name by an underbar ('_').

Let us assume that we wish to test the behaviour of the `public_method()` and want to mock out the `_hidden_method()`.

We can do this by writing a test that will create a mock object and use this in place of the `real _hidden_method()`. We could probably use either the `Mock` class or the `MagicMock` class for this; however due to the additional functionality provided by the `MagicMock` class it is a common practice to use that class. We will therefore do the same.

The test to be created will be defined within a method within a test class. The names of the test method and the test class are by convention descriptive and thus will describe what is being tested, for example:

```
from unittest.mock import *

from unittest import TestCase
from unittest import main

class test_SomeClass_public_interface(TestCase):

    def test_public_method(self):
        test_object = SomeClass()
        # Set up canned response on mock method
          test_object._hidden_method = MagicMock(name = 'hidden_
method')
        test_object._hidden_method.return_value = 10
```

```
        # Test the object
        result = test_object.public_method(5)
         self.assertEqual(15, result,'return value from public_
method incorrect')
```

In this case note that the class being tested is initiated first. The MagicMock is then instantiated and assigned to the name of the method to be mocked. This in effect replaces that method for the test_object. The MagicMock object is given a name as this helps with treating any issues in the report generated by the unites framework. Following this the canned response from the mock version of the _hidden_method() is defined; it will always return the value 10.

At this point we have set up the mock to be used for the test and are now ready to run the test. This is done in the next line where the public_method() is called on the test_object with the parameter 5. The result is then stored.

The test then validated the result to ensure that it is correct; i.e. that the returned value is 15.

Although this is a very simple example it illustrates how a method can be mocked out using the MagicMock class.

25.6.2 The Patchers

The unittest.mock.patch(), unittest.mock.patch.object() and unittest.patch.dict() decorators can be used to simplify the creation of mock objects.

- The patch decorator takes a target for the patch and returns a MagicMock object in its place. It can be used as a TastCase method or class decorator. As a class decorator it decorates each test method in the class automatically. It can also be used as a context manager via the with and with-as statements.
- The patch.object decorator can be provided with either two or three arguments. When given three arguments, it will replace the object to be patched, with a mock for the given attribute/method name. When given two arguments the object to be patched is given a default MagicMock object for the specified attribute/function.
- The patch.dict decorator patches a dictionary or dictionary like object.

For example, we can rewrite the example presented in the previous section using the @patch.object decorator to provide the mock object for the _hidden_method() (it returns a MagicMock linked to SomeClass):

```
class test_SomeClass_public_interface(TestCase):

    @patch.object(SomeClass, '_hidden_method')
    def test_public_method(self, mock_method):
        # Set up canned response
        mock_method.return_value = 10
        # Create object to be tested
        test_object = SomeClass()
        result = test_object.public_method(5)
         self.assertEqual(15, result, 'return value from public_
method incorrect')
```

In the above code the `_hidden_method()` is replaced with a mock version for `SomeClass` within the `test_public_method()` method. Note that the mock version of the method is passed in as a parameter to the test method so that the canned response can be specified.

You can also use the `@patch()` decorator to mock a function from a module. For example, given some external module with a function `api_call`, we can mock that function out using the `@patch()` decorator:

```
@patch('external_module.api_call')
def test_some_func(self, mock_api_call):
```

You used `patch()` as a decorator and passed the target object's path. The target path was 'external_module.api_call' which consists of the module name and the function to mock.

25.6.3 Mocking Returned Objects

In the examples looked at so far the results returned from the mock functions or methods have been simple integers. However, in some cases the returned values must themselves be mocked as the real system would return a complex object with multiple attributes and methods.

The following example uses a `MagicMock` object to represent an object returned from a mocked function. This object has two attributes: one is a response code and the other is a JSON string. JSON stands for the JavaScript Object Notation and is a commonly used format in web services.

```
import external_module

from unittest.mock import *

from unittest import TestCase
from unittest import main
import json

def some_func():
    # Calls out to external API - which we want to mock
    response = external_module.api_call()
```

```
    return response

class test_some_func_calling_api(TestCase):

    @patch('external_module.api_call')
    def test_some_func(self, mock_api_call):
        # Sets up mock version of api_call
        mock_api_call.return_value = MagicMock(status_code=200,
                            response=json.dumps({'key': 'value'}))
            # Calls some_func() that calls the (mock) api_call()
function
        result = some_func()
      # Check that the result returned from some_func() is what
was expected
      self.assertEqual(result.status_code, 200, "returned status
code is not 200")
          self.assertEqual(result.response, '{"key": "value"}',
"response JSON incorrect")
```

In this example the function being tested is `some_func()`, but `some_func()` calls out to the mocked function `external_module.api_call()`. This mocked function returns a `MagicMock` object with a pre-specified `status_code` and `response`. The assertions then validate that the object returned by `some_func()` contains the correct status code and response.

25.6.4 Validating Mocks Have Been Called

Using `unittest.mock` it is possible to validate that a mocked function or method was called appropriately using `assert_called()`, `assert_called_with()` or `assert_called_once_with()` depending on whether the function takes parameters or not.

The following version of the `test_some_func_with_params()` test method verifies that the mock `api_call()` function was called with the correct parameter.

```
@patch('external_module.api_call_with_param')
def test_some_func_with_param(self, mock_api_call):
    # Sets up mock version of api_call
        mock_api_call.return_value = MagicMock(status_code=200,
response=json.dumps({'age': '23'}))
    result = some_func_with_param('Phoebe')
            # Check result returned from some_func() is what was
expected
        self.assertEqual(result.response, '{age": "23"}', 'JSON
result incorrect')
    # Verify that the mock_api_call was called with the correct
params
mock_api_call.api_call_with_param.assert_called_with('Phoebe')
```

If we wished to validate that it had only been called once, we could use the `assert_called_once_with()` method.

25.7 Mock and MagicMock Usage

25.7.1 Naming Your Mocks

It can be useful to give your mocks a name. The name is used when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

```
mock = MagicMock(name='foo')
```

25.7.2 Mock Classes

As well as mocking an individual method on a class it is possible to mock a whole class. This is done by providing the patch() decorator with the name of the class to patch (with no named attribute/method). In this case the while class is replaced by a `MagicMock` object. You must then specify how that class should behave.

```
import people

from unittest.mock import *
from unittest import TestCase
from unittest import main

class MyTest(TestCase):

    @patch('people.Person')
    def test_one(self, MockPerson):
        self.assertIs(people.Person, MockPerson)
        instance = MockPerson.return_value
        instance.calculate_pay.return_value = 250.0
        payroll = people.Payroll()
        result = payroll.generate_payslip(instance)
            self.assertEqual('You earned 250.0', result, 'payslip
incorrect')
```

In this example the people.Person class has been mocked out. This class has a method calculate_pay() which is being mocked here. The payroll class has a method `generate_payslip()` that expects to be given a person object. It then uses the information provided by the person objects `calculate_pay()` method to generate the string returned by the `generate_payslip()` method.

25.7.3 Attributes on Mock Classes

Attributes on a mock object can be easily defined; for example if we want to set an attribute on a mock object, then we can just assign a value to the attribute:

```
import people
from unittest.mock import *
from unittest import TestCase

class MyTest(TestCase):

    @patch('people.Person')
    def test_one(self, MockPerson):
        self.assertIs(people.Person, MockPerson)
        instance = MockPerson.return_value
        instance.age = 24
        instance.name = 'Adam'
        self.assertEqual(24, instance.age, 'age incorrect')
        self.assertEqual('Adam', instance.name, 'name incorrect')
```

In this case the attribute age and name have been added to the mock instance of the people.Person class.

If the attribute itself needs to be a mock object, then all that is required is to assign a `MagicMock` (or `Mock`) object to that attribute:

```
instance.address = MagicMock(name='Address')
```

25.7.4 Mocking Constants

It is very easy to mock out a constant; this can be done using the `@patch()` decorator and proving the name of the constant and the new value to use. This value can be a literal value such as 42 or 'Hello' or it can be a mock object itself (such as a `MagicMock` object). For example:

```
@patch('mymodule.MAX_COUNT', 10)
def test_something(self):
    # Test can now use mymodule.MAX_COUNT
```

25.7.5 Mocking Properties

It is also possible to mock Python properties. This is done again using the `@patch` decorator but using the `unittest.mock.PropertyMock` class and the `new_callable` parameter. For example:

```
@patch('mymoule.Car.wheels', new_callable=mock.PropertyMock)
def test_some_property(self, mock_wheels):
    mock_wheels.return_value = 6
    # Rest of test method
```

25.7.6 Raising Exceptions with Mocks

A very useful attribute that can be specified when a mock object is created is the `side_effect`. If you set this to an exception class or instance, then the exception will be raised when the mock is called, for example:

```
mock = Mock(side_effect=Exception('Boom!'))
mock()
```

This will result in the exception being raised when the mock() is invoked.

25.7.7 Applying Patch to Every Test Method

If you want to mock out the thing for every test in a test class, then you can decorate the whole class rather than each individual method. The effect of decorating the class is that the patch will be automatically applied to all test methods in the class (i.e. to all methods starting with the word 'test'). For example:

```
import people

from unittest.mock import *
from unittest import TestCase
from unittest import main

@patch('people.Person')
class MyTest(TestCase):

        def test_one(self, MockPerson):
         self.assertIs(people.Person, MockPerson)

        def test_two(self, MockSomeClass):
         self.assertIs(people.Person, MockSomeClass)

        def do_something(self):
         return 'something'
```

In the above test class, the tests `test_one` and `test_two` are supplied with the mock version of the `Person` class. However the `do_something()` method is not affected.

25.7.8 Using Patch as a Context Manager

The patch function can be used as a context manager. This gives fine-grained control over the scope of the mock object.

In the following example the `test_one()` method contains a `with-as` statement that we used to patch (mock) the person class as `MockPerson`. This mock class is only available within the `with-as` statement.

```
import people

from unittest.mock import *
from unittest import TestCase
from unittest import main

class MyTest(TestCase):

    def test_one(self):
        with patch('people.Person') as MockPerson:
            self.assertIs(people.Person, MockPerson)
            instance = MockPerson.return_value
            instance.calculate_pay.return_value = 250.0
            payroll = people.Payroll()
            result = payroll.generate_payslip(instance)
           self.assertEqual('You earned 250.0', result, 'payslip
incorrect')
```

25.8 Mock Where You Use It

The most common error made by people using the `unittest.mock` library is mocking in the wrong place. The rule is that you must mock out where you are going to use it; or to put it another way you must always mock the real thing where it is imported into, not where it's imported from.

25.9 Patch Order Issues

It is possible to have multiple patch decorators on a test method. However, the order in which you define the patch decorators is significant. The key to understanding what the order should be is to work backwards so that when the mocks are passed into the test method, they are presented to the right parameters. For example:

```
@patch('mymodule.sys')
@patch('mymodule.os')
@patch('mymodule.os.path')
def test_something(self,
                   mock_os_path,
                   mock_os,
                   mock_sys):
    # The rest of the test method
```

Notice that the last patch's mock is passed into the second parameter passed to the `test_something()` method (`self` is the first parameter to all methods). In turn the first patch's mock is passed into the last parameter. Thus the mocks are passed into the test method in the reverse order to that which they are defined in.

25.10 How Many Mocks?

An interesting question to consider is how many mocks should you use per test?

This is the subject or a lot of debate within the software testing community. The general rules of thumb around this topic are given below; however it should be borne in mind that these are guidelines rather than hard and fast rules.

- **Avoid more than 2 or 3 mocks per test**. You should avoid more than 2 to 3 mocks as the mocks themselves get harder to manage. Many also consider that if you need more than 2 to 3 mocks per test, then there are probably some underlying design issues that need to be considered. For example, if you are testing a Python class, then that class may have too many dependencies. Alternatively the class may have too many responsibilities and should be broken down into several independent classes; each with a distinct responsibility. Another cause might be that the class's behaviour may not be encapsulated enough and that you are allowing other elements to interact with the class in more informal ways (i.e. the interface between the class and other elements is not clean/exploit enough). The result is that it may be necessary to refactor your class before progressing with your development and testing.
- **Only mock you nearest neighbour.** You should only ever mock your nearest neighbour whether that is a function, method or object. You should try to avoid mocking dependencies of dependencies. If you find yourself doing this, then it will become harder to configure, maintain, understand and develop. It is also increasingly likely that you are testing the mocks rather than your own function, method or class.

25.11 Mocking Considerations

The following provide some rules of thumb to consider when using mocks with your tests:

- Don't over mock—if you do then you can end up just testing the mocks themselves.
- Decide what to mock, typical examples of what to mock include those elements that are not yet available, those elements that are not by default repeatable (such as live data feeds) or those elements of the system that are time consuming or complex.
- Decide where to mock such as the interfaces for the unit under test. You want to test the unit so any interface it has with another system, function, class might be a candidate for a mock.
- Decide when to mock so that you can determine the boundaries for the test.
- Decide how you will implement your mocks. For example you need to consider which mocking framework(s) you will use or how to mock larger components such as a database.

25.12 Online Resources

There is a great deal of information available on how to mock, when to mock and what mock libraries to use; however the following provides useful starting points for Python mocking:

- https://docs.python.org/3/library/unittest.mock.html The Python Standard Library documentation on the unitest.mock library.
- https://docs.python.org/3/library/unittest.mock-examples.html A set of examples you can use to explore mocking using unites.mock.
- https://pymox.readthedocs.io/en/latest/index.html Pymox is an alternative open-source mock object framework for Python.
- http://gfxmonk.net/dist/doc/mocktest/doc mocktest its yet another mocking library for Python.

25.13 Exercises

One of the reasons for mocking is to ensure that tests are repeatable. In this exercise we will mock out the use of a random number generated to ensure that our tests can be easily repeated.

The following program generates a deck of cards and randomly picks a card from the deck:

```
import random

def create_suite(suite):
    return [ (i, suite) for i in range(1, 14)]

def pick_a_card(deck):
    print('You picked')
    position = random.randint(0, 52)
    print(deck[position][0], "of", deck[position][1])
    return (deck[position])

# Set up the data
hearts = create_suite('hearts')
spades = create_suite('spades')
diamonds = create_suite('diamonds')
clubs = create_suite('clubs')

# Make the deck of cards
deck = hearts + spades + diamonds + clubs

# Randomly pick from the deck of cards
card = pick_a_card(deck)
```

Each time the program is run a different card is picked; for example in two consecutive runs the following output is obtained:

```
You picked
13 of clubs
You picked
1 of hearts
```

We now want to write a test for the pick_a_card() function. You should mock out the random.randint(0) function to do this.

Part V
File Input/Output

Chapter 26
Introduction to Files, Paths and IO

26.1 Introduction

The operating system is a critical part of any computer systems. It is comprised of elements that manage the processes that run on the CPU, how memory is utilised and managed, how peripheral devices are used (such as printers and scanners), it allows the computer system to communicate with other systems, and it also provides support for the file system used.

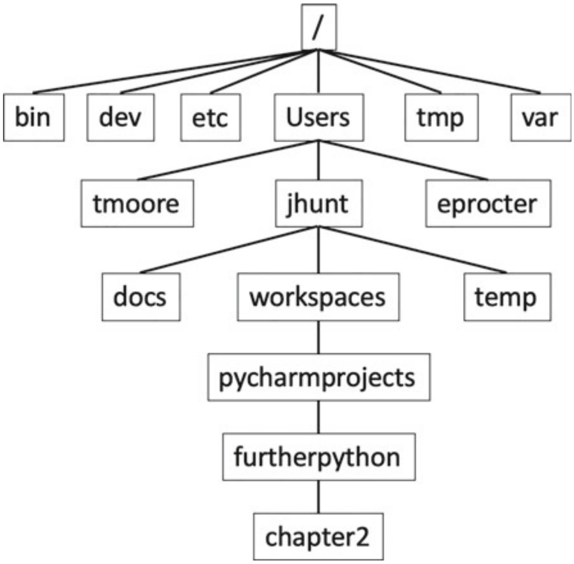
The file system allows programs to permanently store data. This data can then be retrieved by applications at a later date, potentially after the whole computer has been shut down and restarted.

The File Management System is responsible for managing the creation, access and modification of the long-term storage of data in files.

This data may be stored locally or remotely on discs, tapes, DVD drives, USB drives, etc.

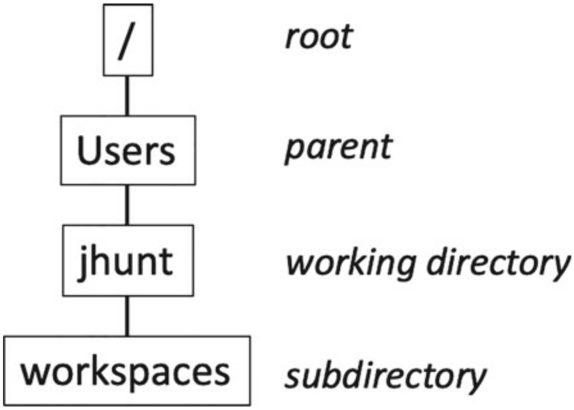
Although this was not always the case, most modern operating systems organise files into a hierarchical structure, usually in the form of an inverted tree. For example in the following diagram the root of the directory structure is shown as '/'. This root directory holds six subdirectories. In turn the users subdirectory holds three further directories and so on:

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_26



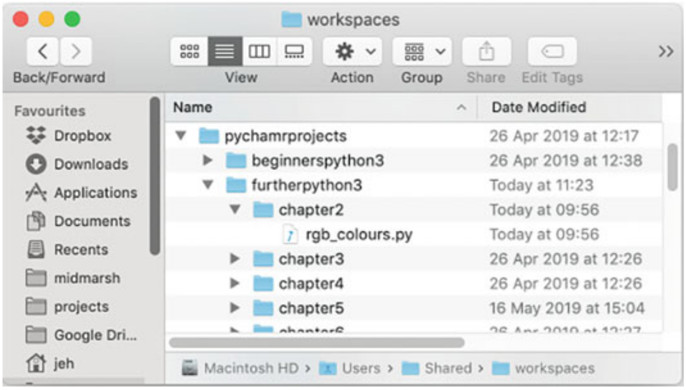
Each file is contained within a directory (also known as a folder on some operating systems such as Windows). A directory can hold zero or more files and zero or more directories.

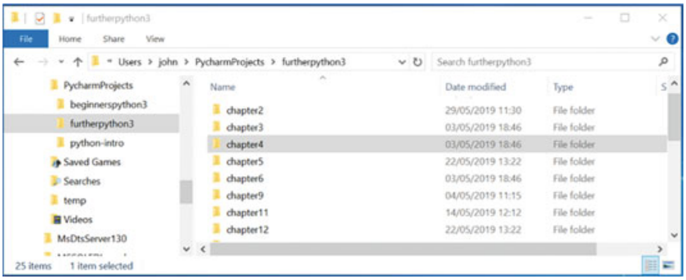
For any give directory there are relationships with other directories as shown below for the directory jhunt:



The root directory is the starting point for the hierarchical directory tree structure. A child directory of a given directory is known as a subdirectory. The directory that holds the given directory is known as the parent directory. At any one time, the directory within which the program or user is currently working is known as the *current working directory*.

A user or a program can move around this directory structure as required. To do this, the user can typically either issue a series of commands at a terminal or command window, such as DC to change directory or pwd to print the working directory. Alternatively graphical user interfaces (GUIs) to operating systems usually include some form of file manager application that allows a user to view the file structure in terms of a tree. The finder program for the Mac is shown below with a tree structure displayed for a `pycharmprojects` directory. A similar view is also presented for the Windows Explorer program.





26.2 File Attributes

A file will have a set of attributes associated with it such as the data that it was created, the date it was last updated/modified, how large the file is, etc. It will also typically have an attribute indicating who the owner of the file is. This may be the creator of the file; however the ownership of a file can be changed either from the command line or through the GUI interface. For example, on Linux and Mac OS X the command `chown` can be used to change the file ownership.

It can also have other attributes which indicate who can read, write or execute the file. In Unix style systems (such as Linux and Mac OS X) these access rights can be specified for the file owner, for the group that the file is associated with and for all other users.

The file owner can have rights specified for reading, writing and executing a file. These are usually represented by the symbols 'r', 'w' and 'x', respectively. For example the following uses the symbolic notation associated with Unix files and indicates that the file owner is allowed to read, write and execute a file:

```
-rwx------
```

Here the first dash is left blank as it is to do with special files (or directories), then the next set of three characters represent the permissions for the owner, the following set of three the permissions for all other users. As this example has `rwx` in the first group of three characters this indicates that the user can read 'r', write 'w' and execute 'x' the file. However the next six characters are all dashes indicating that the group and all other users cannot access the file at all.

The group that a file belongs to is a group that can have any number of users as members. A member of the group will have the access rights as indicated by the group settings on the file. As for the owner of a file these can be to read, write or execute the file. For example, if group members are allowed to read and execute a file, then this would be shown using the symbolic notation as:

```
----r-x---
```

Now this example indicates that only members of the group can read and execute the file; note that group members cannot write the file (they therefore cannot modify the file).

If a user is not the owner of a file, nor a member of the group that the file is part of, then their access rights are in the 'everyone else' category. Again this category can have read, write or execute permissions. For example, using the symbolic notation, if all users can read the file but are not able to do anything else, then this would be shown as:

```
-------r--
```

Of course a file can mix the above permissions together, so that an owner may be allowed to read, write and execute a file, the group may be able to read and execute the file, but all other users can only read the file. This would be shown as:

```
-rwxr-xr--
```

In addition to the symbolic notation there is also a numeric notation that is used with Unix style systems. The numeric notation uses three digits to represent the permissions. Each of the three *rightmost* digits represents a different component of the permissions: owner, group, and others.

Each of these digits is the sum of its component bits in the binary numeral system. As a result, specific bits add to the sum as it is represented by a numeral:

- The read bit adds 4 to its total (in binary 100),
- The write bit adds 2 to its total (in binary 010), and
- The execute bit adds 1 to its total (in binary 001).
- The following symbolic notations can be represented by an equivalent numeric notation:

Symbolic notation	Numeric notation	Meaning
rwx------	0700	Read, write, and execute only for owner
-rwxrwx---	0770	Read, write, and execute for owner and group
-rwxrwxrwx	0777	Read, write, and execute for owner, group, and others

Directories have similar attributes and access rights to files. For example, the following symbolic notation indicates that a directory (indicated by the 'd') has read and executed permissions for the directory owner and for the group. Other users cannot access this directory:

```
dr-xr-x---
```

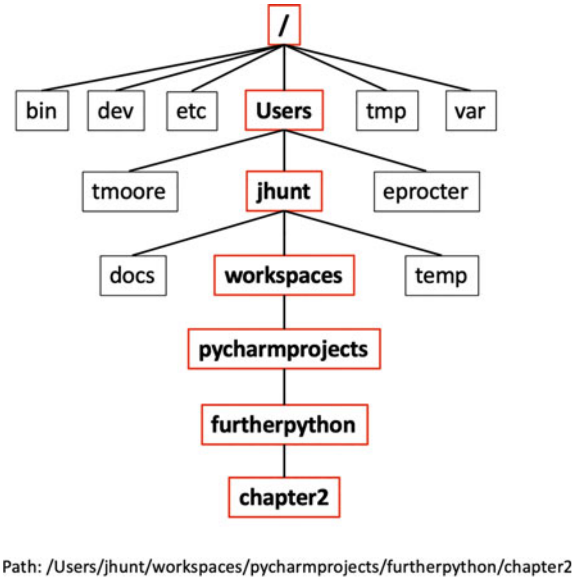
The permissions associated with a file or directory can be changed either using a command from a terminal or command window (such as chmod which is used to modify the permissions associated with a file or directory) or interactively using the file explorer style tool.

26.3 Paths

A path is a particular combination of directories that can lead to a specific subdirectory or file.

This concept is important as Unix/Linux/macOS X and Windows file systems represent an inverted tree of directories and files. It is thus important to be able to uniquely reference locations with the tree.

For example, in the following diagram the path /Users/jhunt/workspaces/pycharmprojects/furtherpython/chapter 2 is highlighted:

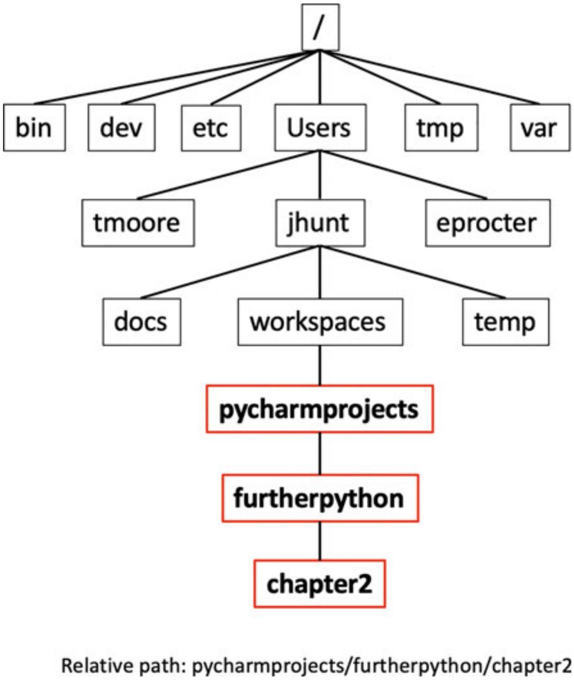


A path may be absolute or relative. An absolute path is one which provides a complete sequence of directories from the root of the file system to a specific subdirectory or file.

A relative path provides a sequence from the current working directory to a particular subdirectory or file.

The absolute path will work wherever a program or user is currently located within the directory tree. However, a relative path may only be relevant in a specific location.

For example, in the following diagram, the relative path pycharmprojects/furtherpython/chapter 2 is only meaningful relative to the directory workspaces:



Note that an absolute path starts from the root directory (represented by '`/`'), whereas a relative path starts from a particular subdirectory (such as `pychamprojects`).

26.4 File Input/Output

File input/output (often just referred to as File I/O) involves reading and writing data to and from files. The data being written can be in different formats.

For example a common format used in Unix/Linux and Windows systems is the ASCII text format. The ASCII format (or American Standard Code for Information Interchange) is a set of codes that represent various characters that is widely used by operating systems. The following table illustrates some of the ASCII character codes and what they represent:

Decimal code	Character	Meaning
42	*	Asterisk
43	+	Plus
48	0	Zero

(continued)

(continued)

Decimal code	Character	Meaning
49	1	One
50	2	Two
51	3	Three
65	A	Uppercase A
66	B	Uppercase B
67	C	Uppercase C
68	D	Uppercase D
97	a	Lowercase a
98	b	Lowercase b
99	c	Lowercase c
100	d	Lowercase d

ASCII is a very useful format to use for text files as they can be read by a wide range of editors and browsers. These editors and browsers make it very easy to create human readable files. However, programming languages such as Python often use a different set of character encodings such as a Unicode character encoding (such as UTF-8). Unicode is another standard for representing characters using various codes. Unicode encoding systems offer a wider range of possible character encodings than ASCII; for example the latest version of Unicode in May 2019, Unicode 12.1, contains a repertoire of 137,994 characters covering 150 modern and historic scripts, as well as multiple symbol sets and emojis.

However, this means that it can be necessary to translate ASCII into Unicode (e.g. UTF-8) and vice versa when reading and writing ASCII files in Python.

Another option is to use a binary format for data in a file. The advantage of using binary data is that there is little or no translation required from the internal representation of the data used in the Python program into the format stored in the file. It is also often more concise than an equivalent ASCII format, and it is quicker for a program to read and write and takes up less disc space, etc. However, the down side of a binary format is that it is not in an easily human readable format. It may also be difficult for other programs, particularly those written in other programming languages such as Java or C#, to read the data in the files.

26.5 Sequential Access versus Random Access

Data can be read from (or indeed written to) a file either sequentially or via a random access approach.

Sequential access to data in a file means that the program reads (or writes) data to a file sequentially, starting at the beginning of a file and processing the data an item at a time until the end of the file is reached. The read process only ever moves forward and only to the next item of data to read.

Random access to a data file means that the program can read (or write) data anywhere into the file at any time. That is the program can position itself at a particular point in the file (or rather a pointer can be positioned within the file), and it can then start to read (or write) at that point. If it is reading, then it will read the next data item relative to the pointer rather than the start of the file. If it is writing data, then it will write data from that point rather than at the end of the file. If there is already data at that point in the file, then it will be overwritten. This type of access is also known as direct access as the computer program needs to know where the data is stored within the file and thus goes directly to that location for the data. In some cases the location of the data is recorded in an index and thus is also known as indexed access.

Sequential file access has advantages when a program needs to access information in the same order each time the data is read. It is also faster to read or write all the data sequentially than via direct access as there is no need to move the file pointer around.

Random access files however are more flexible as data does not need to be written or read in the order in which it is obtained. It is also possible to jump to just the location for the data required and read that data (rather than needing to sequentially read through all the data to find the data items of interest).

26.6 Files and I/O in Python

In the remainder of this section of the book we will explore the basic facilities provided for reading and writing files in Python. We will also look at the underlying streams model for file I/O. After this we will explore the widely used CSV and Excel file formats and libraries available to support those. This section concludes by exploring the regular expression facilities in Python. While this last topic is not strictly part of file I/O it is often used to parse data read from files to screen out unwanted information.

26.7 Online Resources

See the following online resources for information on the topics in this chapter:

- https://en.wikipedia.org/wiki/ASCII Wikipedia page on ASCII.
- https://en.wikipedia.org/wiki/Unicode Wikipedia page on Unicode.
- https://en.wikipedia.org/wiki/UTF-8 Wikipedia page on UTF-8.

Chapter 27
Reading and Writing Files

27.1 Introduction

Reading data from and writing data to a file are very common within many programs. Python provides a large amount of support for working with files of various types. This chapter introduces you to the core file IO functionality in Python.

27.2 Obtaining References to Files

Reading from, and writing to, text files in Python is relatively straightforward. The built-in `open()` function creates a file object for you that you can use to read and/ or write data from and/or to a file.

The function requires as a minimum the name of the file you want to work with.

Optionally you can specify the access mode (e.g. read, write, append, etc.). If you do not specify a mode then the file is open in read-only mode. You can also specify whether you want the interactions with the file to be buffered which can improve performance by grouping data reads together.

The syntax for the `open()` function is

```
file_object = open(file_name, access_mode, buffering)
```

where

- `file_name` indicates the file to be accessed.
- `access_mode`: The `access_mode` determines the mode in which the file is to be opened, i.e. read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter, and the default file access mode is read (r).

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_27

- `buffering:` If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file.

The `access_mode` values are given in the following table.

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, it creates a new file for writing
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, it creates a new file for writing
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, it creates a new file for reading and writing
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing

The file object itself has several useful attributes such as

- `file.closed` returns `True` if the file has been closed (can no longer be accessed because the `close()` method has been called on it).
- `file.mode` returns the access mode with which the file was opened.
- `file.name` the name of the file.

The `file.close()` method is used to close the file once you have finished with it. This will flush any unwritten information to the file (this may occur because of buffering) and will close the reference from the file object to the actual underlying operating system file. This is important to do as leaving a reference to a file open can cause problems in larger applications as typically there are only a certain number of file references possible at one time, and over a long period of time these may all be used up resulting in future errors being thrown as files can no longer be opened.

The following short code snippet illustrates the above ideas:

```
file = open('myfile.txt', 'r+')
print('file.name:', file.name)
print('file.closed:', file.closed)
print('file.mode:', file.mode)
file.close()
print('file.closed now:', file.closed)
```

The output from this is:

```
file.name: myfile.txt
file.closed: False
file.mode: r+
file.closed now: True
```

27.3 Reading Files

Of course, having set up a file object we want to be able to either access the contents of the file or write data to that file (or do both). Reading data from a text file is supported by the `read()`, `readline()` and `readlines()` methods:

- The `read()` method will return the entire contents of the file as a single string.
- The `readline()` method reads the next line of text from a file. It returns all the text on one line up to and including the newline character. It can be used to read a file a line at a time.
- The `readlines()` method returns a list of all the lines in a file, where each item of the list represents a single line.

Note that once you have read some text from a file using one of the above operations then that line is not read again. Thus using `readlines()` would result in a further `readlines()` returning an empty list whatever the contents of the file.

The following illustrates using the `readlines()` method to read all the text in a text file into a program and then print each line out in turn:

```
file = open('myfile.txt', 'r')
lines = file.readlines()
for line in lines:
    print(line, end='')
file.close()
```

Notice that within the `for` loop we have indicated to the print function that we want the end character to be `''` rather than a newline; this is because the line string already possesses the newline character read from the file.

27.4 File Contents Iteration

As suggested by the previous example, it is very common to want to process the contents of a file one line at a time. In fact Python makes this extremely easy by making the file object support iteration. File iteration accesses each line in the file and makes that line available to the `for` loop. We can therefore write:

```
file = open('myfile.txt', 'r')
for line in file:
    print(line, end='')
file.close()
```

It is also possible to use the *list comprehension* to provide a very concise way to load and process lines in a file into a list. It is similar to the effect of `readlines()` but we are now able to pre-process the data before creating the list:

```
file = open('myfile.txt', 'r')
lines = [line.upper() for line in file]
file.close()
print(lines)
```

27.5 Writing Data to Files

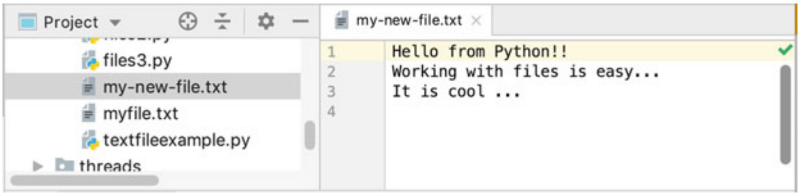
Writing a string to a file is supported by the `write()` method. Of course, the file object we create must have an access mode that allows writing (such as '`w`'). Note that the write method *does not* add a newline character (represented as `'\n'`) to the end of the string—you must do this manually.

An example short program to write a text file is given below:

```
print('Writing file')
f = open('my-new-file.txt', 'w')
f.write('Hello from Python!!\n')
f.write('Working with files is easy...\n')
f.write('It is cool ...\n')
f.close()
```

This creates a new file called `my-new-file.txt`. It then writes three strings to the file each with a newline character on the end; it then closes the file.

The effect of this is to create a new file called `myfile.txt` with three lines in it:



27.6 Using Files and with Statements

Like several other types where it is important to shut down resources, the file object class implements the *Context Manager Protocol* and thus can be used with the `with` statement. It is therefore common to write code that will open a file using the `with as` structure thus ensuring that the file will be closed when the block of code is finished with, for example:

```
with open('my-new-file.txt', 'r') as f:
    lines = file.readlines()
    for line in lines:
        print(line, end='')
```

27.7 The Fileinput Module

In some situations, you may need to read the input from several files in one go. You could do this by opening each file independently and then reading the contents and appending that contents to a list, etc. However, this is a common enough requirement that the `fileinput` module provides a function `fileinput.input()` that can take a list of files and treat all the files as a single input significantly simplifying this process, for example:

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
     for line in f:
         process(line)
```

Features provided by the `fileinput` module include

- Return the name of the file currently being read.
- Return the integer "file descriptor" for the current file.

- Return the cumulative line number of the line that has just been read.
- Return the line number in the current file. Before the first line has been read this returns 0.
- A Boolean function that indicates if the current line just read is the first line of its file.

Some of these are illustrated below:

```
with fileinput.input(files=('textfile1.txt', 'textfile2.txt'))
as f:
        line = f.readline()
        print('f.filename():', f.filename())
        print('f.isfirstline():', f.isfirstline())
        print('f.lineno():', f.lineno())
        print('f.filelineno():', f.filelineno())
        for line in f:
            print(line, end='')
```

27.8 Renaming Files

A file can be renamed using the `os.rename()` function. This function takes two arguments, the current filename and the new filename. It is part of the Python `os` module which provides methods that can be used to perform a range of file processing operations (such as renaming a file). To use the module, you will first need to import it. An example of using the rename function is given below:

```
import os
os.rename( 'myfileoriginalname.txt', 'myfilenewname.txt')
```

27.9 Deleting Files

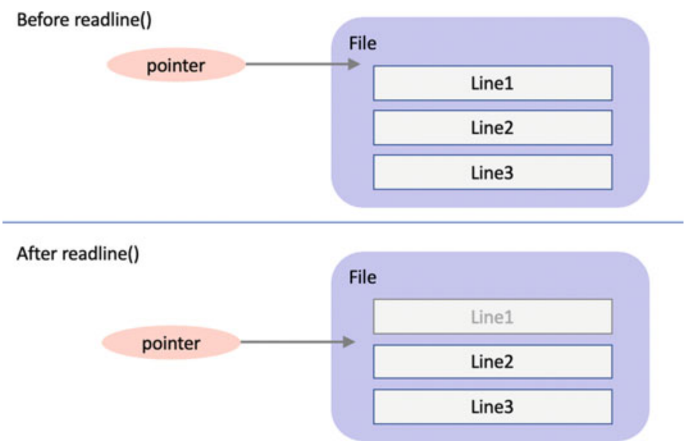
A file can be deleted using the `os.remove()` method. This method deletes the file specified by the filename passed to it. Again, it is part of the `os` module, and therefore this must be imported first:

```
import os
os.remove( 'somefilename.txt')
```

27.10 Random Access Files

All the examples presented so far suggest that files are accessed sequentially, with the first line read before the second and so on. Although this is (probably) the most common approach it is not the only approach supported by Python; it is also possible to use a random access approach to the contents within a file.

To understand the idea of random file access it is useful to understand that we can maintain a pointer into a file to indicate where we are in that file in terms of reading or writing data. Before anything is read from a file the pointer is before the beginning of the file, and reading the first line of text would, for example, advance the point to the start of the second line in the file, etc. This idea is illustrated below:



When randomly accessing the contents of a file the programmer manually moves the pointer to the location required and reads or writes text relative to that pointer. This means that they can move around in the file reading and writing data.

The random access aspect of a file is provided by the seek method of the file object:

- `file.seek (offset, whence)` this method determines where the next read or write operation (depending on the mode used in the `open()` call) takes place.

In the above the offset parameter indicates the position of the read/write pointer within the file. The move can also be forwards or backwards (represented by a negative offset).

The optional whence parameter indicates where the offset is relative to. The values used for whence are:

- 0 indicates that the offset is relative to start of file (the default).
- 1 means that the offset is relative to the current pointer position.
- 2 indicates the offset is relative to end of file.

Thus, we can move the pointer to a position relative to the start of the file, to the end of the file or to the current position.

For example, in the following sample code we create a new text file and write a set of characters into that file. At this point the pointer is positioned after the 'z' in the file. However, we then use seek() to move the point to the 10th character in the file and now write 'Hello'; next we reposition the pointer to the 6th character in the file and write out 'BOO'. We then close the file. Finally, we read all the lines from the file using a with as statement and the open() function, and from this we will see that the text is the file is now abcdef**BOO**j**HELLO**pqrstuvwxyz:

```
f = open('text.txt', 'w')
f.write('abcdefghijklmnopqrstuvwxyz\n')
f.seek(10,0)
f.write('HELLO')
f.seek(6, 0)
f.write ('BOO')
f.close()
with open('text.txt', 'r') as f:
    for line in f:
        print(line, end='')
```

27.11 Directories

Both Unix style and Windows operating systems are hierarchical structures comprising directories and files. The os module has several functions that can help with creating, removing and altering directories. These include:

- mkdir() This function is used to create a directory, and it takes the name of the directory to create as a parameter. If the directory already exists FileExistsError is raised.
- chdir() This function can be used to change the current working directory. This is the directory that the application will read from/write to by default.
- getcwd() This function returns a string representing the name of the current working directory.
- rmdir() This function is used to remove/delete a directory. It takes the name of the directory to delete as a parameter.
- listdir() This function returns a list containing the names of the entries in the directory specified as a parameter to the function (if no name is given the current directory is used).

A simple example illustrating the use of some of these functions is given below:

```
import os
print('os.getcwd(:', os.getcwd())
print('List contents of directory')
print(os.listdir())
print('Create mydir')
os.mkdir('mydir')
print('List the updated contents of directory')
print(os.listdir())
print('Change into mydir directory')
os.chdir('mydir')
print('os.getcwd(:', os.getcwd())
print('Change back to parent directory')
os.chdir('..')
print('os.getcwd(:', os.getcwd())
print('Remove mydir directory')
os.rmdir('mydir')
print('List the resulting contents of directory')
print(os.listdir())
```

Note that `'..'` is a short hand for the parent directory of the current directory and `'.'` is short hand for the current directory.

An example of the type of output generated by this program for a specific set up on a Mac is given below:

```
os.getcwd(:      /Users/Shared/workspaces/pycharm/pythonintro/
textfiles
List contents of directory
['my-new-file.txt',       'myfile.txt',       'textfile1.txt',
'textfile2.txt']
Create mydir
List the updated contents of directory
['my-new-file.txt',       'myfile.txt',       'textfile1.txt',
'textfile2.txt', 'mydir']
Change into mydir directory
os.getcwd(:      /Users/Shared/workspaces/pycharm/pythonintro/
textfiles/mydir
Change back to parent directory
os.getcwd(:      /Users/Shared/workspaces/pycharm/pythonintro/
textfiles
Remove mydir directory
List the resulting contents of directory
['my-new-file.txt',       'myfile.txt',       'textfile1.txt',
'textfile2.txt']
```

27.12 Temporary Files

During the execution of many applications it may be necessary to create a temporary file that will be created at one point and deleted before the application finishes. It is of course possible to manage such temporary files yourself; however, the `tempfile` module provides a range of facilities to simplify the creation and management of these temporary files.

Within the `tempfile` module `TemporaryFile`, `NamedTemporaryFile`, `TemporaryDirectory` and `SpooledTemporaryFile` are high-level file objects which provide automatic clean-up of temporary files and directories. These objects implement the *Context Manager Protocol.*

The `tempfile` module also provides the lower-level functions `mkstemp()` and `mkdtemp()` that can be used to create temporary files that require the developer to manage them and delete them at an appropriate time.

The high-level features for the `tempfile` module are:

- `TemporaryFile(mode = 'w + b')` Returns an anonymous file-like object that can be used as a temporary storage area. On completion of the managed context (via a `with` statement) or destruction of the file object, the temporary file will be removed from the filesystem. Note that by default all data is written to the temporary file in binary format which is generally more efficient.
- `NamedTemporaryFile(mode = 'w + b')` This function operates exactly as `TemporaryFile()` does, except that the file has a visible name in the filesystem.
- `SpooledTemporaryFile(max_size = 0, mode = 'w + b')` This function operates exactly as `TemporaryFile()` does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disc and operation proceeds as with `TemporaryFile()`.
- `TemporaryDirectory(suffix = None, prefix = None, dir = None)` This function creates a temporary directory. On completion of the context or destruction of the temporary directory object the newly created temporary directory and all its contents are removed from the filesystem.

The lower-level functions include:

- `mkstemp()` Creates a temporary file that is only readable or writable by the user who created it.
- `mkdtemp()` Creates a temporary directory. The directory is readable, writable and searchable only by the creating user ID.
- `gettempdir()` Returns the name of the directory used for temporary files. This defines the default value for the default temporary directory to be used with the other functions in this module.

An example of using the `TemporaryFile` function is given below. This code imports the `tempfile` module then prints out the default directory used for temporary files. It then creates a `TemporaryFile` object and prints its name and mode (the default mode is binary but for this example we have overwritten this so that plain text is used). We have then written a line to the file. Using seek we are repositioning ourselves at the start of the file and then reading the line we have just written.

```
import tempfile

print('tempfile.gettempdir():', tempfile.gettempdir())
temp = tempfile.TemporaryFile('w+')
print('temp.name:', temp.name)
print('temp.mode:', temp.mode)
temp.write('Hello world!')
temp.seek(0)
line = temp.readline()
print('line:', line)
```

The output from this when run on an Apple Mac is:

```
tempfile.gettempdir():
/var/folders/6n/8nrnt9f93pn66ypg9s5dq8y80000gn/T
temp.name: 4
temp.mode: w+
line: Hello world!
```

Note that the file name is '4' and that the temporary directory is not a meaningful name!

27.13 Working with Paths

The `pathlib` module provides a set of classes representing filesystem paths, that is paths through the hierarchy of directories and files within an operating systems file structure. It was introduced in Python 3.4. The core class in this module is the `Path` class.

A `Path` object is useful because it provides operations that allow you to manipulate and manage the path to a file or directory. The `Path` class also replicates some of the operations available from the `os` module (such as `mkdir`, `rename` and `rmdir`) which means that it is not necessary to work directly with the `os` module.

A path object is created using the `Path` constructor function; this function actually returns a specific type of `Path` depending on the type of operating system being used such as a `WindowsPath` or a `PosixPath` (for Unix style systems). The `Path()` constructor takes the path to create for example '`D:/mydir`' (on Windows) or '`/Users/user1/mydir`' on a Mac or '`/var/temp`' on Linux, etc.

You can then use several different methods on the Path object to obtain information about the path such as:

- `exists()` returns `True` of `False` depending on whether the path points to an existing file or directory.
- `is_dir()` returns `True` if the path points to a directory. `False` if it references a file. `False` is also returned if the path does not exist.
- `is_file()` returns `True` of the path points to a file, it returns `False` if the path does not exist or the path references a directory.

- `absolute()` A `Path` object is considered absolute if it has both a root and (if appropriate) a drive.
- `is_absolute()` returns a `Boolean` value indicating whether the Path is absolute or not.

An example of using some of these methods is given below:

```
from pathlib import Path

print('Create Path object for current directory')
p = Path('.')
print('p:', p)
print('p.exists():', p.exists())
print('p.is_dir():', p.is_dir())
print('p.is_file():', p.is_file())
print('p.absolute():', p.absolute())
```

Sample output produced by this code snippet is:

```
Create Path object for current directory
p: .
p.exists(): True
p.is_dir(): True
p.is_file(): False
p.absolute():     /Users/Shared/workspaces/pycharm/pythonintro/
textfiles
```

There are also several methods on the `Path` class that can be used to create and remove directories and files such as:

- `mkdir()` is used to create a directory path if it does not exist. If the path already exists, then a `FileExistsError` is raised.
- `rmdir()` removes this directory; the directory must be empty; otherwise an error will be raised.
- `rename(target)` renames this file or directory to the given target.
- `unlink()` removes the file referenced by the path object.
- `joinpath(*other)` appends elements to the path object, e.g. path.joinpath('/temp').
- `with_name(new_name)` returns a new path object with the name changed.
- The `'/'` operator can also be used to create new path objects from existing paths for example path / 'test' / 'output' which would append the directories test and out to the path object.

Two `Path` class methods can be used to obtain path objects representing key directories such as the current working directory (the directory the program is logically in at that point) and the home directory of the user running the program:

- `Path.cwd()` returns a new path object representing the current directory.
- `Path.home()` `returns` a new path object representing the user's home directory.

An example using several of the above features is given below. This example obtains a path object representing the current working directory and then appends ‘text’ to this. The result path object is then checked to see if the path exists (on the computer running the program), assuming that the path does not exist; it is created; and the `exists()` method is rerun.

```
p = Path.cwd()
print('Set up new directory')
newdir = p / 'test'
print('Check to see if newdir exists')
print('newdir.exists():', newdir.exists())
print('Create new dir')
newdir.mkdir()
print('newdir.exists():', newdir.exists())
```

The effect of creating the directory can be seen in the output:

```
Set up new directory
Check to see if newdir exists
newdir.exists(): False
Create new dir
newdir.exists(): True
```

A very useful method in the `Path` object is the `glob(pattern)` method. This method returns all elements within the path that meet the pattern specified.

For example `path.glob('*.py')` will return all the files ending `.py` within the current path.

Note that `'**/*.py'` would indicate the current directory and any subdirectory. For example, the following code will return all files where the file name ends with ‘`.txt`’ for a given path:

```
print('-'* 10)

for file in path.glob('*.txt'):
    print('file:', file)

print('-'* 10)
```

An example of the output generated by this code is:

```
----------
file: my-new-file.txt
file: myfile.txt
file: textfile1.txt
file: textfile2.txt
----------
```

Paths that reference a file can also be used to read and write data to that file. For example the `open()` method can be used to open a file that by default allows a file to be read:

- `open(mode='r') this` can be used to open the file referenced by the path object.

This is used below to read the contents of a file a line at a time (note that `with as` statement is used here to ensure that the file represented by the `Path` is closed):

```
p = Path('mytext.txt')
with p.open() as f:
    print(f.readline())
```

However, there are also some high-level methods available that allow you to easily write data to a file or read data from a file. These include the Path methods `write_text` and `read_text` methods:

- `write_text(data)` opens the file pointed to in text mode and writes the data to it and then closes the file.
- `read_text()` opens the file in read mode, reads the text and closes the file; it then returns the contents of the file as a string.

These are used below.

```
dir = Path('./test')
print('Create new file')
newfile = dir / 'text.txt'
print('Write some text to file')
newfile.write_text('Hello Python World!')
print('Read the text back again')
print(newfile.read_text())
print('Remove the file')
newfile.unlink()
```

which generates the following output:

```
Create new file
Write some text to file
Read the text back again
Hello Python World!
Remove the file
```

27.14 Online Resources

See the following online resources for information on the topics in this chapter:

- https://docs.python.org/3/tutorial/inputoutput.html for the Python Standard Tutorial on file input and output.
- https://pymotw.com/3/os.path/index.html for platform independent manipulation of filenames.
- https://pymotw.com/3/pathlib/index.html for information filesystem Path objects.
- https://pymotw.com/3/glob/index.html for filename pattern matching using glob.
- https://pymotw.com/3/tempfile/index.html for temporary filesystem objects.
- https://pymotw.com/3/gzip/index.html for information on reading and writing GNU Zip files.

27.15 Exercise

The aim of this exercise is to explore the creation of, and access to, the contents of a file.

You should write two programs, and these programs are outlined below:

1. Create a program that will write todays date into a file—the name of the file can be hard coded or supplied by the user. You can use the `datetime.today()` function to obtain the current date and time. You can use the `str()` function to convert this date time object into a string so that it can be written out to a file.
2. Create a second program to reload the date from the file and convert the string into a date object. You can use the `datetime.strptime()` function to convert a string into a date time object (see https://docs.python.org/3/library/datetime.html#datetime.datetime.strptime for documentation on this function). This functions takes a string containing a date and time in it and a second string which defines the format expected. If you use the approach outlined in step 1 above to write the string out to a file then you should find that the following defines an appropriate format to parse the date_str so that a date time object can be created:

```
datetime_object = datetime.strptime(date_str, '%Y-%m-%d %H:%M:%S.%f')
```

Chapter 28
Stream IO

28.1 Introduction

In this chapter we will explore the stream I/O model that under pins the way in which data is read from and written to data sources and sinks. One example of a data source or sink is a file but another might be a byte array.

This model is actually what sits underneath the file access mechanisms discussed in the previous chapter.

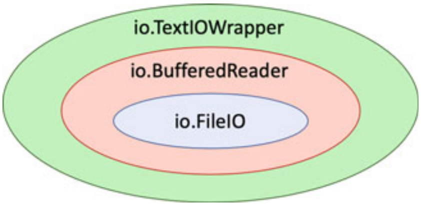
It is not actually necessary to understand this model to be able to read and write data to and from a file; however in some situations it is useful to have an understanding of this model so that you can modify the default behaviour when necessary.

The remainder of this chapter first introduces the stream model, discusses Python streams in general and then presents the classes provided by Python. It then considers what is the actual effect of using the open() function presented in the last chapter.

28.2 What is a Stream?

Streams are objects which serve as sources or sinks of data. At first this concept can seem a bit strange. The easiest way to think of a stream is as a conduit of data flowing from or into a pool. Some streams read data straight from the "source of the data", and some streams read data from other streams. These latter streams then do some "useful" processing of the data such as converting the raw data into a specific format. The following figure illustrates this idea.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_28



In the above figure the initial `FileIO` stream reads raw data from the actual data source (in this case a file). The `BufferedReader` then buffers the data reading process for efficiency. Finally the `TextIOWrapper` handles string encoding; that is it converts strings from the typical ASCII representation used in a file into the internal representation used by Python (which uses Unicode).

You might ask at this point why have a streams model at all, after all we read and wrote data to files without needing to know about streams in the last chapter? The answer is that a stream can read or write data to or from a source of data rather than just from a file. Of course a file can be a source of data but so can a socket, a pipe, a string, a web service, etc. It is therefore a more flexible data I/O model.

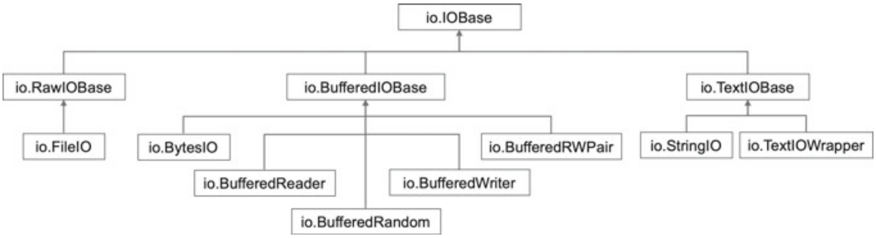
28.3 Python Streams

The Python `io` module provides Python's main facilities for dealing with data input and output. There are three main types of input/output; these are text I/O, binary I/O and raw I/.O. These categories can be used with various types of data source/sinks.

Whatever the category, each concrete stream can have a number of properties such as being *read-only*, *write-only* or *read-write*. It can also support sequential access or random access depending on the nature of the underlying data sink. For example, reading data from a socket or pipe is inherently sequential whereas reading data from a file can be performed sequentially or via a random access approach.

Whichever stream is used, however, they are aware of the type of data they can process. For example, attempting to supply a string to a binary write-only stream will raise a `TypeError`. As indeed will be presenting binary data to a text stream, etc.

As suggested by this there are a number of different types of stream provided by the Python `io` module, and some of these are presented below:



The abstract `IOBase` class is at the root of the stream IO class hierarchy. Below this class are stream classes for unbuffered and buffered IO and for text-oriented IO.

28.4 IOBase

This is the abstract base class for all I/O stream classes. The class provides many abstract methods that subclasses will need to implement.

The `IOBase` class (and its subclasses) all supports the iterator protocol. This means that an `IOBase` object (or an object of a subclass) can iterate over the input data from the underling stream.

`IOBase` also implements the Context Manager Protocol, and therefore it can be used with the `with` and `with-as` statements.

The `IOBase` class defines a core set of methods and attributes including:

- `close()` flush and close the stream.
- `closed` an attribute indicating whether the stream is closed.
- `flush()` flush the write buffer of the stream if applicable.
- `readable()` returns True if the stream can be read from.
- `readline(size=-1)` return a line from the stream. If size is specified at most size bytes will be read.
- `readline(hint=-1)` read a list of lines. If hint is specified then it is used to control the number of lines read.
- `seek(offset[, whence])` This method moves the current the stream position/pointer to the given offset. The meaning of the offset depends on the whence parameter. The default value for whence is SEEK_SET.
- SEEK_SET or 0: seek from the start of the stream (the default); offset must either be a number returned by TextIOBase.tell(), or zero. Any other offset value produces undefined behaviour.
- SEEK_CUR or 1: "seek" to the current position; offset must be zero, which is a no-operation (all other values are unsupported).
- SEEK_END or 2: seek to the end of the stream; offset must be zero (all other values are unsupported).
- `seekable()` does the stream support seek().
- `tell()` return the current stream position/pointer.

- `writeable()` returns true if data can be written to the stream.
- `writelines(lines)` write a list of lines to the stream.

28.5 Raw IO/UnBuffered IO Classes

Raw IO or unbuffered IO is provided by the `RawIOBase` and `FileIO` classes.

RawIOBase This class is a subclass of `IOBase` and is the base class for raw binary (aka unbuffered) I/O. Raw binary I/O typically provides low-level access to an underlying OS device or API and does not try to encapsulate it in high-level primitives (this is the responsibility of the Buffered I/O and Text I/O classes that can wrap a raw I/O stream). The class adds methods such as:

- `read(size=-1)` This method reads up to size bytes from the stream and returns them. If size is unspecified or -1 then all available bytes are read.
- `readall()` This method reads and returns all available bytes within the stream.
- `readint(b)` This method reads the bytes in the stream into a pre-allocated, writable bytes-like object b (e.g. into a byte array). It returns the number of bytes read.
- `write(b)` This method writes the data provided by `b` (a bytes -like object such as a byte array) into the underlying raw stream.

FileIO The `FileIO` class represents a raw unbuffered binary IO stream linked to an operating system level file. When the `FileIO` class is instantiated it can be given a file name and the mode (such as 'r' or 'w'). It can also be given a flag to indicate whether the file descriptor associated with the underlying OS level file should be closed or not.

This class is used for the low-level reading of binary data and is at the heart of all file-oriented data access (although it is often wrapped by another stream such as a buffered reader or writer).

28.6 Binary IO/Buffered IO Classes

Binary IO aka Buffered IO is a filter stream that wraps a lower-level `RawIOBase` stream (such as a FileIO stream). The classes implementing buffered IO all extend the `BufferedIOBase` class and are:

BufferedReader When reading data from this object, a larger amount of data may be requested from the underlying raw stream and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

BufferedWriter When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying RawIOBase object under various conditions, including:

- When the buffer gets too small for all pending data.
- When `flush()` is called.
- When the `BufferedWriter` object is closed or destroyed.

BufferedRandom A buffered interface to random access streams. It supports `seek()` and `tell()` functionality.

BufferedRWPair A buffered I/O object combining two unidirectional RawIOBase objects—one readable, the other writeable—into a single bidirectional endpoint.

Each of the above classes wraps a lower-level byte-oriented stream class such as the `io.FileIO` class, for example:

```
f = io.FileIO('data.dat')
br = io.BufferedReader(f)
print(br.read())
```

This allows data in the form of bytes to be read from the file 'data.dat'. You can of course also read data from a different source, such as an in memory `BytesIO` object:

```
binary_stream_from_file =
io.BufferedReader(io.BytesIO(b'starship.png'))
bytes = binary_stream_from_file.read(4)
print(bytes)
```

In this example the data is read from the `BytesIO` object by the `BufferedReader`. The `read()` method is then used to read the first 4 bytes, and the output is:

```
b'star'
```

Note the 'b' in front of both the string'starship.png' and the result 'star'. This indicates that the string literal should become a bytes literal in Python 3. Bytes literals are always prefixed with 'b' or 'B'; they produce an instance of the bytes type instead of the str type. They may only contain ASCII characters.

The operations supported by buffered streams include, for reading:

- `peek(n)` returns up to *n* bytes of data without advancing the stream pointer. The number of bytes returned may be less or more than requested depending on the amount of data available.
- `read(n)` returns n bytes of data as bytes, if n is not supplied (or is negative) the read all available data.
- `readl(n)` reads up to n bytes of data using a single call on the raw data stream.

The operations supported by buffered writers include:

- `write(bytes)` writes the bytes-like data and returns the number of bytes written.
- `flush()` This method forces the bytes held in the buffer into the raw stream.

28.7 Text Stream Classes

The text stream classes are the `TextIOBase` class and its two subclasses `TextIOWrapper` and `StringIO`.

TextIOBase This is the root class for all Text Stream classes. It provides a character and line-based interface to stream I/O. This class provides several additional methods to that defined in its parent class:

- `read(size=-1)` This method will return at most size characters from the stream as a single string. If size is negative or None, it will read all remaining data.
- `readline(size=-1)` This method will return a string representing the current line (up to a newline or the end of the data whichever comes first). If the stream is already at EOF, an empty string is returned. If size is specified, at most size characters will be read.
- `seek(offset, [, whence])` changes the stream position/pointer by the specified offset. The optional `whence` parameter indicates where the seek should start from:
  - SEEK_SET or 0: (the default) seek from the start of the stream.
  - SEEK_CUR or 1: seek to the current position; offset must be zero, which is a no-operation.
  - SEEK_END or 2: seek to the end of the stream; offset must be zero.
- `tell()` Returns the current stream position/pointer as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.
- `write(s)` This method will write the string s to the stream and return the number of characters written.

TextIOWrapper. This is a buffered text stream that wraps a buffered binary stream and is a direct subclass of TextIOBase. When a TextIOWrapper is created there are a range of options available to control its behaviour:

```
io.TextIOWrapper(buffer, encoding=None, errors=None, newline=None,
line_buffering=False, write_through=False)
```

where

1. `buffer` is the buffered binary stream.
2. `encoding` represents the text encoding used such as UTF-8.
3. `errors` defines the error handling policy such as *strict* or *ignore*.
4. `newline` controls how line endings are handled for example should they be ignored (None) or represented as a linefeed, carriage return or a newline/carriage return, etc.
5. `line_buffering` if `True` then `flush()` is implied when a call to write contains a newline character or a carriage return.

6. `write_through` if `True` then a call to write is guaranteed not to be buffered.

The `TextIOWrapper` is wrapped around a lower-level binary buffered I/O stream, for example:

```
f = io.FileIO('data.txt')
br = io.BufferedReader(f)
text_stream = io.TextIOWrapper(br, 'utf-8')
```

StringIO This is an in memory stream for text I/O. The initial value of the buffer held by the `StringIO` object can be provided when the instance is created, for example:

```
in_memory_text_stream = io.StringIO('to be or not to be that is
the question')
print('in_memory_text_stream', in_memory_text_stream)
print(in_memory_text_stream.getvalue())
in_memory_text_stream.close()
```

This generates:

```
in_memory_text_stream <_io.StringIO object at 0x10fdfaee8>
to be or not to be that is the question
```

Note that the underlying buffer (represented by the string passed into the `StringIO` instance) is discarded when the `close()` method is called.

The `getvalue()` method returns a string containing the entire contents of the buffer. If it is called after the stream was closed then an error is generated.

28.8 Stream Properties

It is possible to query a stream to determine what types of operations it supports. This can be done using the readable, seeable and writable methods. For example:

```
f = io.FileIO('myfile.txt')
br = io.BufferedReader(f)
text_stream = io.TextIOWrapper(br, encoding='utf-8')

print('text_stream', text_stream)
print('text_stream.readable():', text_stream.readable())
print('text_stream.seekable()', text_stream.seekable())
print('text_stream.writeable()', text_stream.writable())

text_stream.close()
```

The output from this code snippet is:

```
text_stream <_io.TextIOWrapper name='myfile.txt'encoding='utf-
8'>
text_stream.readable(): True
text_stream.seekable() True
text_stream.writeable() False
```

28.9 Closing Streams

All opened streams must be closed. However, you can close the top-level stream, and this will automatically close lower-level streams, for example:

```
f = io.FileIO('data.txt')
br = io.BufferedReader(f)
text_stream = io.TextIOWrapper(br, 'utf-8')
print(text_stream.read())
text_stream.close()
```

28.10 Returning to the Open() Function

If streams are so good then why don't you use them all the time? Well actually in Python 3 you do! The core open function (and indeed the `io.open()` function) both return a stream object. The actual type of object returned depends on the file mode specified, whether buffering is being used, etc. For example:

```
import io

# Text stream
f1 = open('myfile.txt', mode='r', encoding='utf-8')
print(f1)

# Binary IO aka Buffered IO
f2 = open('myfile.dat', mode='rb')
print(f2)

f3 = open('myfile.dat', mode='wb')
print(f3)

# Raw IO aka Unbufferedf IO
f4 = open('starship.png', mode='rb', buffering=0)
print(f4)
```

When this short example is run the output is:

```
<_io.TextIOWrapper name='myfile.txt'mode='r'encoding='utf-8'>
<_io.BufferedReader name='myfile.dat'>
<_io.BufferedWriter name='myfile.dat'>
<_io.FileIO name='starship.png'mode='rb'closefd=True>
```

As you can see from the output, four different types of object have been returned from the `open()` function. The first is a `TextIOWrapper`, the second a `BufferedReader`, the third a BufferedWriter and the final one is a `FileIO` object. This reflects the differences in the parameters passed into the open(0 function. For example, f1 references an `io.TextIOWrapper` because it must encode (convert) the input text into Unicode using the UTF-8 encoding scheme. While f2 holds an io.`BufferedReader` because the mode indicates that we want to read binary data while f3 holds a io.`BufferedWriter` because the mode used indicates we want to write binary data. The final call to open returns a `FileIO` because we have indicated that we do not want to buffer the data and thus we can use the lowest level of stream object.

In general the following rules are applied to determine the type of object returned based on the modes and encoding specified:

Class	Mode	Buffering
FileIO	binary	no
BufferedReader	'rb'	yes
BufferedWriter	'wb'	yes
BufferedRandom	'rb+ ' 'wb + ' 'ab + '	yes
TextIOWrapper	Any text	yes

Note that not all mode combinations make sense, and thus some combinations will generate an error.

In general you don't therefore need to worry about which stream you are using or what that stream does; not least because all the streams extend the `IOBase` class and thus have a common set of methods and attributes.

However, it is useful to understand the implications of what you are doing so that you can make better informed choices. For example, binary streams (that do less processing) are faster than Unicode-oriented streams that must convert from ASCII into Unicode.

Also understanding the role of streams in input and output can also allow you to change the source and destination of data without needing to rewrite the whole of your application. You can thus use a file or stdin for testing and a socket for reading data in production.

28.11 Online Resource

See the following online resource for information on the topics in this chapter:

1. https://docs.python.org/3/library/io.html This provides the Python Standard Library Guide to the core tools available for working with streams.

28.12 Exercise

Use the underlying streams model to create an application that will write binary data to a file. You can use the 'b' prefix to create a binary literal to be written, for example b'Hello World'.

Next create another application to reload the binary data from the file and print it out.

Chapter 29
Working with CSV Files

29.1 Introduction

This chapter introduces a module that supports the generation of Comma Separated Values (CSV) files.

29.2 CSV Files

The Comma Separated Values (CSV) format is the most common import and export format for spreadsheets and databases. However, CSV is not a precise standard with multiple different applications having different conventions and specific standards.

The Python `csv` module implements classes to read and write tabular data in CSV format. As part of this it supports the concept of a dialect which is a CSV format used by a specific application or suite of programs; for example, it supports an Excel dialect.

This allows programmers to say, "write this data in the format preferred by Excel", or "read data from this file which was generated by Excel", without knowing the precise details of the CSV format used by Excel.

Programmers can also describe the CSV dialects understood by other applications or define their own special-purpose CSV dialects.

The `csv` module provides a range of functions including:

- `csv.reader (csvfile, dialect = 'excel', **fmtparams)`
  Returns a reader object which will iterate over lines in the given csvfile. An optional dialect parameter can be given. This may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional `fmtparams` keyword arguments can be given to override individual formatting parameters in the current dialect.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_29

- `csv.writer (csvfile, dialect = 'excel', **fmtparams)` Returns a writer object responsible for converting the user’s data into delimited strings on the given csvfile. An optional dialect parameter provided. The `fmtparams` keyword arguments can be given to override individual formatting parameters in the current dialect.
- `csv.list_dialects()` Return the names of all registered dialects. For example on a macOS X the default list of dialects is [‘excel’, ‘excel-tab’, ‘unix’].

29.2.1 The CSV Writer Class

A CSV Writer is obtained from the `csv.writer()` function. The `csvwriter` supports two methods used to write data to the CSV file:

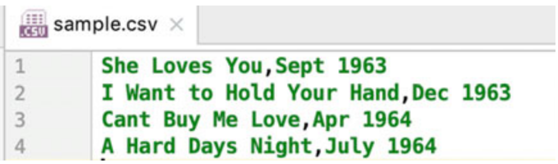
- `csvwriter.writerow(row)` Writes the row parameter to the writer’s file object, formatted according to the current dialect.
- `csvwriter.writerows(rows)` Write all elements in rows (an iterable of row objects as described above) to the writer’s file object, formatted according to the current dialect.
- Writer objects also have the following public attribute:
- `csvwriter.dialect` A read-only description of the dialect in use by the writer.

The following program illustrates a simple use of the `csv` module which creates a file called `sample.csv`.

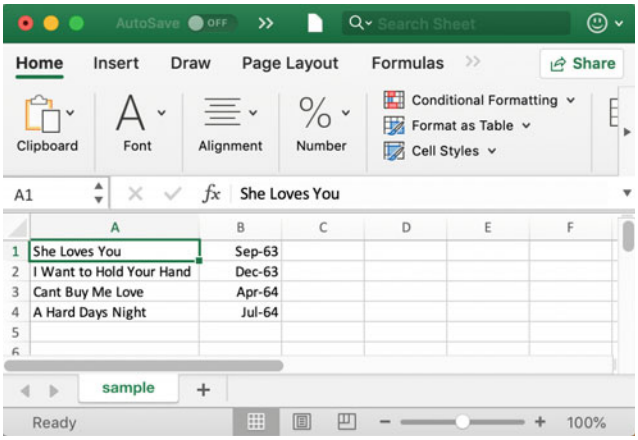
As we have not specified a dialect, the default ‘excel’ dialect will be used. The `writerow()` method is used to write each comma separate list of strings to the CSV file.

```
print('Crearting CSV file')
with open('sample.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(['She Loves You', 'Sept 1963'])
    writer.writerow(['I Want to Hold Your Hand', 'Dec 1963'])
    writer.writerow(['Cant Buy Me Love', 'Apr 1964'])
    writer.writerow(['A Hard Days Night', 'July 1964'])
```

The resulting file can be viewed as shown below:



However, as it is a CSV file, we can also open it in Excel:



29.2.2 The CSV Reader Class

A CSV Reader object is obtained from the `csv.reader()` function. It implements the iteration protocol.

If a csv reader object is used with a `for` loop then each time round the loop it supplies the next *row* from the CSV file as a list, parsed according to the current CSV dialect.

Reader objects also have the following public attributes:

- `csvreader.dialect` A read-only description of the dialect in use by the parser.
- `csvreader.line_num` The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.

The following provides a very simple example of reading a CSV file using a csv reader object:

```
print('Starting to read csv file')
with open('sample.csv', newline='') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        print(*row, sep=', ')

print('Done Reading')
```

The output from this program based on the sample.csv file created earlier is:

```
Starting to read csv file
She Loves You, Sept 1963
I Want to Hold Your Hand, Dec 1963
Cant Buy Me Love, Apr 1964
A Hard Days Night, July 1964
Done Reading
```

29.2.3 The CSV DictWriter Class

In many cases the first row of a CSV file contains a set of names (or keys) that define the fields within the rest of the CSV. That is the first row gives meaning to the columns and the data held in the rest of the CSV file. It is therefore very useful to capture this information and to structure the data written to a CSV file or loaded from a CSV file based on the keys in the first row.

The `csv.DictWriter` returns an object that can be used to write values into the CSV file based on the use of such named columns. The file to be used with the `DictWriter` is provided when the class is instantiated.

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name', 'result']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
writer.writerow({'first_name': 'Denise',
        'last_name': 'Smith',
        'result' : 54})
    writer.writerow({'first_name': 'Natalia',
        'last_name': 'Lewis',
        'result' : 63})
    writer.writerow({'first_name': 'Adam',
        'last_name': 'Davies',
        'result' : 72})
```

Note that when the `DictWriter` is created a list of the keys must be provided that are used for the columns in the CSV file.

The method `writeheader()` is then used to write the header row out to the CSV file.

The method `writerow()` takes a dictionary object that has keys based on the keys defined for the `DictWriter`. These are then used to write data out to the CSV (note the order of the keys in the dictionary is not important).

In the above example code the result of this is that a new file called `names.csv` is created which can be opened in Excel:

```
first_name,last_name,result
Denise,Smith,54
Natalia,Lewis,63
Adam,Davies,72
```

Of course, as this is a CSV file it can also be opened in a plain text editor as well.

29.2.4 The CSV DictReader Class

As well as the `csv.DictWriter` there is a `csv.DictReader`. The file to be used with the `DictReader` is provided when the class is instantiated. As with the `DictReader` the `DictWriter` class takes a list of keys used to define the columns in the CSV file. If the headings to be used for the first row can be provided although this is optional (if a set of keys are not provided, then the values in the first row of the CSV file will be used as the `fieldnames`).

The `DictReader` class provides several useful features including the `fieldnames` property that contains a list of the keys/headings for the CSV file as defined by the first row of the file.

The `DictReader` class also implements the iteration protocol, and thus it can be used in a for loop in which each row (after the first row) is returned in turn as a dictionary. The dictionary object representing each row can then be used to access each column value based on the keys defined in the first row.

An example is shown below for the CSV file created earlier:

```
import csv

print('Starting to read dict CSV example')

with open('names.csv', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    for heading in reader.fieldnames:
        print(heading, end=' ')

    print('\n------------------------------')

    for row in reader:
        print(row['first_name'], row['last_name'], row['result'])

print('Done')
```

This generates the following output:

```
Starting to read dict CSV example
first_name last_name result
------------------------------
Denise Smith 54
```

```
Natalia Lewis 63
Adam Davies 72
Done
```

29.3 Online Resources

See the following online resources for information on the topics in this chapter:

- https://docs.python.org/3/library/csv.html for the Python Standard documentation on CSV file reading and writing.
- https://pymotw.com/3/csv/index.html for the Python Module of the Week page on CSV files.
- https://pythonprogramming.net/reading-csv-files-python-3 for a tutorial on reading CSV files.

29.4 Exercises

In this exercise you will create a CSV file based on a set of transactions stored in a current account.

1. To do this first define a new Account class to represent a type of bank account.
2. When the class is instantiated you should provide the account number, the name of the account holder, an opening balance and the type of account (which can be a string representing 'current', 'deposit', 'investment', etc.). This means that there must be a __init__ method, and you will need to store the data within the object.
3. Provide three instance methods for the Account: deposit(amount), withdraw(amount) and get_balance(). The behaviour of these methods should be as expected, deposit will increase the balance, withdraw will decrease the balance, and get_balance() returns the current balance.

Your `Account` class should also keep a history of the transactions it is involved in.

A *Transaction* is a record of a deposit or withdrawal along with an amount.

Note that the initial amount in an account can be treated as an initial deposit.

The history could be implemented as a *list* containing an ordered sequence to transactions. A Transaction itself could be defined by a class with an action (deposit or withdrawal) and an amount.

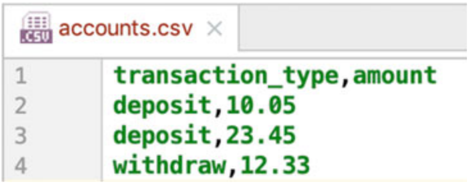
Each time a withdrawal or a deposit is made a new transaction record should be added to a transaction history list.

Next provide a function (which could be called something like `write_account_transactions_to_csv()`) that can take an account and then write each of the transactions it holds out to a CSV file, with each transaction type and the transaction amount separated by a comma.

The following sample application illustrates how this function might be used:

```
print('Starting')
acc = accounts.CurrentAccount('123', 'John', 10.05, 100.0)
acc.deposit(23.45)
acc.withdraw(12.33)

print('Writing Account Transactions')
write_account_transaction_to_csv('accounts.csv', acc)

print('Done')
```

The contents of the CSV file would then be:



Chapter 30
Working with Excel Files

30.1 Introduction

This chapter introduces the `openpyxl` module that can be used when working with Excel files. Excel is a software application developed by Microsoft that allows users to work with spreadsheets. It is a very widely used tool, and files using the Excel file format are commonly encountered within many organisations. It is in effect the industry standard for spreadsheets and as such is a very useful tool to have in the developers' toolbox.

30.2 Excel Files

Although CSV files are a convenient and simple way to handle data; it is very common to need to be able to read or write Excel files directly. To this end there are several libraries available in Python for this purpose. One widely used library is the *OpenPyXL* library. This library was originally written to support access to Excel 2010 files. It is an open-source project and is well documented.

The OpenPyXL library provides facilities for

- Reading and writing Excel workbooks.
- Creating/accessing Excel worksheets.
- Creating Excel formulas.
- Creating graphs (with support from additional modules).

As *OpenPyXL* is not part of the standard Python distribution you will need to install the library yourself using a tool such as Anaconda or pip (e.g. `pip install openpyxl`). Alternatively, if you are using PyCharm you will be able to add the OpenPyXL library to your project.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_30

30.3 The Openpyxl. Workbook Class

The key element in the `OpenPyXL` library is the `Workbook` class. This can be imported from the module:

```
from openpyxl import Workbook
```

A new instance of the (in memory) `Workbook` can be created using the `Workbook` class (note at this point it is purely a structure within the Python program and must be saved before an actual Excel file is created).

```
wb = Workbook()
```

30.4 The Openpyxl. WorkSheet Objects

A workbook is always created with at least one worksheet. You can get hold of the currently active worksheet using the `Workbook.active` property:

```
ws = wb.active
```

You can create additional worksheets using the workbooks' `create_sheet()` method:

```
ws = wb.create_sheet('Mysheet')
```

You can access or update the title of the worksheet using the `title` property:

```
ws.title = 'New Title'
```

The background colour of the tab holding this title is white by default. You can change this providing an RRGGBB colour code to the `worksheet.sheet_properties.tabColor` attribute, for example:

```
ws.sheet_properties.tabColor = "1072BA"
```

30.5 Working with Cells

It is possible to access the cells within a worksheet. A cell can be accessed directly as keys on the worksheet, for example:

```
ws['A1'] = 42
```

or

```
cell = ws['A1']
```

This returns a cell object; you can obtain the value of the cell using the value property, for example.

```
print(cell.value)
```

There is also the `Worksheet.cell()` method. This provides access to cells using row and column notation:

```
d = ws.cell(row=4, column=2, value=10)
```

A row of values can also be added at the current position within the Excel file using append:

```
ws.append([1, 2, 3])
```

This will add a row to the Excel file containing 1, 2 and 3.

Ranges of cells can be accessed using slicing:

```
cell_range = ws['A1':'C2']
```

Ranges of rows or columns can also be obtained:

```
col = ws['C']
col_range = ws['C:D']
row10 = ws[10]
row_range = ws[5:10]
```

The value of a cell can also be an Excel formula such as

```
ws['A3'] = '=SUM(A1, A2)'
```

A workbook is actually only a structure in memory; it must be saved to a file for permanent storage. These workbooks can be saved using the `save()` method. This method takes a filename and writes the `Workbook` out in Excel format.

```
workbook = Workbook()
...
workbook.save('balances.xlsx')
```

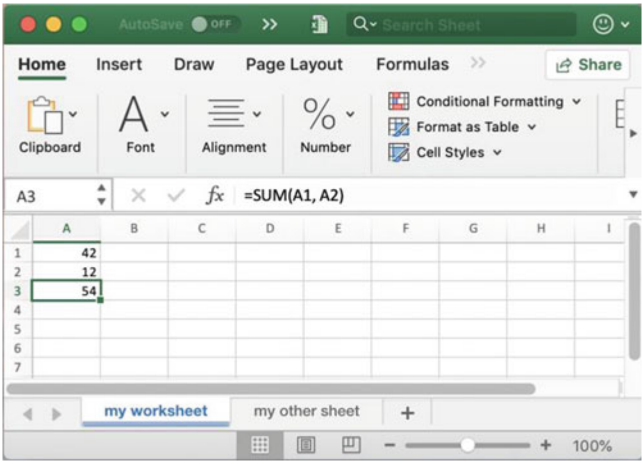
30.6 Sample Excel File Creation Application

The following simple application creates a `Workbook` with two worksheets. It also contains a simple Excel formula that sums the values held in to other cells:

```
from openpyxl import Workbook

def main():
    print('Starting Write Excel Example with openPyXL')
    workbook = Workbook()

    # Get the current active worksheet
```

```
    ws = workbook.active
    ws.title = 'my worksheet'
    ws.sheet_properties.tabColor = '1072BA'

    ws['A1'] = 42
    ws['A2'] = 12
    ws['A3'] = '=SUM(A1, A2)'

    ws2 = workbook.create_sheet(title='my other sheet')
    ws2['A1'] = 3.42
    ws2.append([1, 2, 3])
    ws2.cell(column=2, row=1, value=15)

    workbook.save('sample.xlsx')

    print('Done Write Excel Example')

if __name__ == '__main__':
    main()
```

The Excel file generated from this can be viewed in Excel as shown below:



30.7 Loading a Workbook from an Excel File

Of course, in many cases it is necessary not just to create Excel files for data export but also to import data from an existing Excel file. This can be done using the OpenPyXL `load_workbook()` function. This function opens the specified Excel file (in read-only mode by default) and returns a Workbook object.

```
from openpyxl import load_workbook

workbook = load_workbook(filename='sample.xlsx')
```

You can now access a list of sheets and their names and obtain the currently active sheet, etc. using properties provided by the `workbook` object:

- `workbook.active` returns the active worksheet object.
- `workbook.sheetnames` returns the names (strings) of the worksheets in this workbook.
- `workbook.worksheets` returns a list of worksheet objects.

The following sample application reads the Excel file created earlier in this chapter:

```
from openpyxl import load_workbook

def main():
    print('Starting reading Excel file using openPyXL')

    workbook = load_workbook(filename='sample.xlsx')
    print(workbook.active)
    print(workbook.sheetnames)
    print(workbook.worksheets)

    print('-' * 10)
    ws = workbook['my worksheet']
    print(ws['A1'])
    print(ws['A1'].value)
    print(ws['A2'].value)
    print(ws['A3'].value)

    print('-' * 10)
    for sheet in workbook:
        print(sheet.title)

    print('-' * 10)
    cell_range = ws['A1':'A3']
    for cell in cell_range:
        print(cell[0].value)
    print('-' * 10)

    print('Finished reading Excel file using openPyXL')

if __name__ == '__main__':
    main()
```

The output from this application is illustrated below:

```
Starting reading Excel file using openPyXL
<Worksheet "my worksheet">
['my worksheet', 'my other sheet']
[<Worksheet "my worksheet">, <Worksheet "my other sheet">]
----------
<Cell 'my worksheet'.A1>
42
12
=SUM(A1, A2)
----------
```

```
my worksheet
my other sheet
----------
42
12
=SUM(A1, A2)
----------
Finished reading Excel file using openPyXL
```

30.8 Online Resources

See the following online resources for information on the topics in this chapter:

- https://openpyxl.readthedocs.io/en/stable for documentation on the OpenPyXL Python to Excel library.

30.9 Exercises

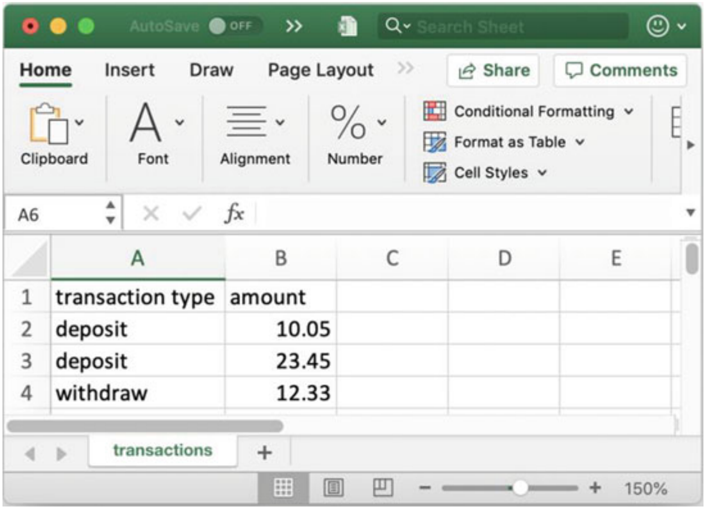
Using the Account class that you created in the last chapter; write the account transaction information to an Excel file instead of a CSV file.

To do this create a function called `write_account_transaction_to_excel()` that takes the name of the Excel file and the account to store. The function should then write the data to the file using the excel format.

The following sample application illustrates how this function might be used:

```
print('Starting')
acc = accounts.CurrentAccount('123', 'John', 10.05, 100.0)
acc.deposit(23.45)
acc.withdraw(12.33)

print('Writing Account Transactions')
write_account_transaction_to_excel('accounts.xlsx', acc)

print('Done')
```

The contents of the Excel file would then be:


AutoSave OFF
Search Sheet
Home
Insert
Draw
Page Layout
Share
Comments
Clipboard
Font
Alignment
Number
Conditional Formatting
Format as Table
Cell Styles
A6
fx
A
B
C
D
E
1
transaction type
amount
2
deposit
10.05
3
deposit
23.45
4
withdraw
12.33
transactions
150%

Chapter 31
Regular Expressions in Python

31.1 Introduction

Regular expression is a very powerful way of processing text while looking for recurring patterns; they are often used with data held in plain text files (such as log files), CSV files as well as Excel files. This chapter introduces regular expressions, discusses the syntax used to define a regular expression pattern and presents the Python `re` module and its use.

31.2 What Are Regular Expressions?

A regular expression (also known as a regex or even just re) is a sequence of characters (letters, numbers and special characters) that form a pattern that can be used to search text to see if that text contains sequences of characters that match the pattern.

For example, you might have a pattern defined as three characters followed by three numbers. This pattern could be used to look for such a pattern in other strings. Thus, the following strings either match (or contain) this pattern or they do not:

Abc123	Matches the pattern
A123A	Does not match the pattern
123AAA	Does not match the pattern

Regular expression is very widely used for finding information in files, for example.

- Finding all lines in a log file associated with a specific user or a specific operation.
- For validating input such as checking that a string is a valid email address or postcode/ZIP code, etc.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_31

- Support for regular expressions is wide spread within programming languages such as Java, C#, PHP and particularly Perl. Python is no exception and has the built-in module `re` (as well as additional third-party modules) that supports regular expressions.

31.3 Regular Expression Patterns

You can define a regular expression pattern using any ASCII character or number. Thus, the string 'John' can be used to define a regex pattern that can be used to match any other string that contains the characters 'J', 'o', 'h' and 'n'. Thus each of the following strings will match this pattern:

- 'John Hunt'
- 'John Jones'
- 'Andrew John Smith'
- 'Mary Helen John'
- 'John John John'
- 'I am going to visit the John'
- 'I once saw a film by John Wayne'

But the following strings would not match the pattern:

- 'Jon Davies' in this case because the spelling of John is different.
- 'John williams' in this case because the capital J does not match the lowercase j.
- 'David James' in this case because the string does not contain the string John!

Regular expressions (*regexs*) use special characters to allow more complex patterns to be described. For example, we can use the special characters '`[]`' to define a set of characters that can match. For example, if we want to indicate that the J may be a capital or a lowercase letter then we can write '[Jj]'—this indicates that either 'J' or 'j' can match the first.

- [Jj]ohn—this states that the pattern starts with either a capital J or a lowercase j followed by 'own'.

Now both 'john williams' and 'John Williams' will match this regex pattern.

31.3.1 Pattern Metacharacters

There are several special characters (often referred to as metacharacters) that have a specific meaning within a *regex* pattern; these are listed in the following table:

<table>
<tr><th>Character</th><th>Description</th><th>Example</th></tr>
<tr><td>[]</td><td>A set of characters</td><td>[a–d] characters in the sequence ‘a’ to ‘d’</td></tr>
<tr><td>\</td><td>Indicates a special sequence (can also be used to escape special characters)</td><td>‘\d’ indicates the character should be an integer</td></tr>
<tr><td>.</td><td>Any character with the exception of the newline character</td><td>‘J.hn’ indicates that there can be any character after the ‘J’ and before the ‘h’</td></tr>
<tr><td>^</td><td>Indicates a string must start with the following pattern</td><td>“^hello” indicates the string must start with ‘hello’</td></tr>
<tr><td>$</td><td>Indicates a string must end with the preceding pattern</td><td>“world$” indicates the string must end with ‘world’</td></tr>
<tr><td>*</td><td>Zero or more occurrences of the preceding pattern</td><td>“Python*” indicates we are looking for zero or more times Python is in a string</td></tr>
<tr><td>+</td><td>One or more occurrences of preceding pattern</td><td>“info+“ indicates that we must find info in the string at least once</td></tr>
<tr><td>?</td><td>Indicates zero or 1 occurrences of the preceding pattern</td><td>“John?” indicates zero or one instance of the ‘John’</td></tr>
<tr><td>{}</td><td>Exactly the specified number of occurrences</td><td>“John{3}” indicates we expect to see the ‘John’ in the string three times. “X{1,2}” indicates that there can be one or two Xs next to each other in the string</td></tr>
<tr><td>|</td><td>Either or</td><td>“True|OK” indicates we are looking for either True or OK</td></tr>
<tr><td>()</td><td>Groups together a regular expression; you can then apply another operator to the whole group</td><td>“(abc|xyz){2}” indicates that we are looking for the string abc or xyz repeated twice</td></tr>
</table>

31.3.2 Special Sequences

A special sequence is a combination of a ‘\’ (backslash) followed by a character combination which then has a special meaning. The following table lists the common special sequences used in regular expressions:

Sequence	Description	Example
\A	Returns a match if the following characters are at the beginning of the string	“\AThe” must start with ‘The’
\b	Returns a match where the specified characters are at the beginning or at the end of a word	“\bon” or “on\b” indicates a string must start or end with ‘on’

(continued)

(continued)

Sequence	Description	Example
\B	Indicates that the following characters must be present in a string but not at the start (or at the end) of a word	r"\Bon" or r"on\B" must not start or end with 'on'
\d	Returns a match where the string contains digits (numbers from 0 to 9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s~ "
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0 to 9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the following characters are present at the end of the string	"Hunt\Z"

31.3.3 Sets

A set is a sequence of characters inside a pair of square brackets which have specific meanings. The following table provides some examples.

Set	Description
[jeh]	Returns a match where one of the specified characters (j, e or h) is present
[a–x]	Returns a match for any lowercase character, alphabetically between a and x
[^zxc]	Returns a match for any character EXCEPT z, x and c
[0123]	Returns a match where any of the specified digits (0, 1, 2 or 3) is present
[0–9]	Returns a match for any digit between 0 and 9
[0–9][0–9]	Returns a match for any two-digit numbers from 00 and 99
[a–zA–Z]	Returns a match for any character alphabetically between a and z or A and Z

31.4 The Python re Module

The Python `re` module is the built-in module provided by Python for working with regular expressions.

You might also like to examine the third-party `regex` module (see https://pypi.org/project/regex) which is backwards compatible with the default `re` module but provides additional functionality.

31.5 Working with Python Regular Expressions

31.5.1 Using Raw Strings

An important point to note about many of the strings used to define the regular expression patterns is that they are preceded by an '`r`' for example `r'/bin/sh$'`.

The '`r`' before the string indicates that the string should be treated as a *raw* string.

A raw string is a Python string in which all characters are treated as exactly that individual characters. It means that backslash ('`\`') is treated as a literal character rather than as a special character that is used to *escape* the next character.

For example, in a standard string '`\n`' is treated as a special character representing a newline, thus if we wrote the following:

```
s = 'Hello \n world'
print(s)
```

We will get as output:

```
Hello
  World
```

However, if we prefix the string with an '`r`' then we are telling Python to treat it as a raw string. For example:

```
s = r'Hello \n world'
print(s)
```

The output is now

```
Hello \n world
```

This is important for regular expression as characters such as backslash ('`\`') are used within patterns to have a special regular expression meaning, and thus we do not want Python to process them in the normal way.

31.5.2 Simple Example

The following simple Python program illustrates the basic use of the re module. It is necessary to import the re module before you can use it.

```
import re

text1 = 'John williams'
pattern = '[Jj]ohn'
print('looking in', text1, 'for the pattern', pattern)

if re.search(pattern, text1):
    print('Match has been found')
```

When this program is run, we get the following output:

```
looking in John williams for the pattern [Jj]ohn
Match has been found
```

If we look at the code, we can see that the string that we are examining contains 'John williams' and that the pattern used with this string indicates that we are looking for a sequence of 'J' or 'j' followed by 'ohn'. To perform this test we use the re.search() function passing the regex pattern, and the text to test, as parameters. This function returns either None (which is taken as meaning *False* by the If statement) or a Match object (which always has a Boolean value of *True*). As of course 'John' at the start of text1 does match the pattern, the re.search() function returns a match object and we see the 'Match has been found' message is printed out.

Both the Match object and search() method will be described in more detail below; however, this short program illustrates the basic operation of a regular expression.

31.5.3 The Match Object

Match objects are returned by the search() and match() functions.

They always have a Boolean value of True.

The functions match() and search() return None when there is no match and a Match object when a match is found. It is therefore possible to use a match object with an if statement:

```
import re

match = re.search(pattern, string)
if match:
    process(match)
```

Match objects support a range of methods and attributes including:

- `match.re` The regular expression object whose `match()` or `search()` method produced this match instance.
- `match.string` The string passed to `match()` or `search()`.
- `match.start([group]) / match.end([group])` Returns the indices of the start and end of the substring matched by group.
- `match.group()` Returns the part of the string where there was a match.

31.5.4 The search() Function

The `search()` function searches the string for a match and returns a Match object if there is a match. The signature of the function is:

```
re.search(pattern, string, flags=0)
```

The meaning of the parameters is:

- `pattern:` This is the regular expression pattern to be used in the matching process.
- `string:` This is the string to be searched.
- `flags:` These (optional) flags can be used to modify the operation of the search.

The `re` module defines a set of flags (or indicators) that can be used to indicate any optional behaviours associated with the pattern. These flags include:

Flag	Description
`re.IGNORECASE`	Performs case-insensitive matching
`re.LOCALE`	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behaviour (\b and \B)
`re.MULTILINE`	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string)
`re.DOTALL`	Makes a period (dot) match any character, including a newline
`re.UNICODE`	Interprets letters according to the Unicode character set. This flag affects the behaviour of \w, \W, \b, \B
`re.VERBOSE`	Ignores whitespace within the pattern (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker

If there is more than one match, only the first occurrence of the match will be returned:

```
import re

line1 = 'The price is 23.55'
containsIntegers = r'\d+'

if re.search(containsIntegers, line1):
    print('Line 1 contains an integer')
else:
    print('Line 1 does not contain an integer')
```

In this case the output is

```
Line 1 contains an integer
```

Another example of using the `search()` function is given below. In this case the pattern to look for defines three alternative strings (that is the string must contain either Beatles, Adele or Gorillaz):

```
import re

# Alternative words
music = r'Beatles|Adele|Gorillaz'
request = 'Play some Adele'

if re.search(music, request):
    print('Set Fire to the Rain')
else:
    print('No Adele Available')
```

In this case we generate the output:

```
Set Fire to the Rain
```

31.5.5 The match() Function

This function attempts to match a regular expression pattern at the beginning of a string. The signature of this function is given below:

```
re.match(pattern, string, flags=0)
```

The parameters are:

- `pattern`: This is the regular expression to be matched.
- `string`: This is the string to be searched.
- `flags`: Modifier flags that can be used.

The `re.match()` function returns a `Match` object on success, `None` on failure.

31.5.6 The Difference Between Matching and Searching

Python offers two different primitive operations based on regular expressions:

- `match()` checks for a match only at the beginning of the string.
- `search()` checks for a match anywhere in the string.

31.5.7 The finadall() Function

The `findall()` function returns a list containing all matches. The signature of this function is:

```
re.findall(pattern, string, flags=0)
```

This function returns all non-overlapping matches of `pattern` in `string`, as a list of strings.

The `string` is scanned left to right, and matches are returned in the order found. If one or more groups are present in the pattern, then a list of groups is returned; this will be a list of tuples if the pattern has more than one group. If no matches are found, an empty list is returned.

An example of using the `findall()` function is given below. This example looks for a substring starting with two letters and followed by 'ai' and a single character. It is applied to a sentence and returns only the substring 'Spain' and 'plain'.

```
import re

str = 'The rain in Spain stays mainly on the plain'
results = re.findall('[a-zA-Z]{2}ai.', str)
print(results)
for s in results:
    print(s)
```

The output from this program is

```
['Spain', 'plain']
Spain
plain
```

31.5.8 The finditer() Function

This function returns an iterator yielding matched objects for the regular expression `pattern` in the `string` supplied. The signature for this function is:

```
re.finditer(pattern, string, flags=0)
```

The `string` is scanned left to right, and matches are returned in the order found. Empty matches are included in the result. Flags can be used to modify the matches.

31.5.9 The split() Function

The `split()` function returns a list where the string has been split at each match. The syntax of the `split()` function is

```
re.split(pattern, string, maxsplit=0, flags=0)
```

The result is to split a *string* by the occurrences of *pattern*. If capturing parentheses are used in the regular expression `pattern`, then the text of all groups in the `pattern` is also returned as part of the resulting list. If `maxsplit` is nonzero, at most `maxsplit` splits occur, and the remainder of the string is returned as the final element of the list. Flags can again be used to modify the matches.

```
import re

str = 'It was a hot summer night'
x = re.split('\s', str)
print(x)
```

The output is

```
['It', 'was', 'a', 'hot', 'summer', 'night']
```

31.5.10 The sub() Function

The `sub()` function replaces occurrences of the regular expression pattern in the string with the `repl` string.

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the regular expression pattern in string with `repl`, substituting all occurrences unless `max` is provided. This method returns the modified string.

```
import re

pattern = '(England|Wales|Scotland)'
input = 'England for football, Wales for Rugby and Scotland for
the Highland games'
print(re.sub(pattern, 'England', input ))
```

which generates:

```
England for football, England for Rugby and England for the Highland
games
```

You can control the number of replacements by specifying the `count` parameter: The following code replaces the first 2 occurrences:

```
import re

pattern = '(England|Wales|Scotland)'
input = 'England for football, Wales for Rugby and Scotland for
the Highland games'

x = re.sub(pattern, 'Wales', input, 2)
print(x)
```

which produces

```
Wales for football, Wales for Rugby and Scotland for the Highland
games
```

You can also find out how many substitutions were made using the `subn()` function. This function returns the new string and the number of substitutions in a tuple:

```
import re

pattern = '(England|Wales|Scotland)'
input = 'England for football, Wales for Rugby and Scotland for
the Highland games'

print(re.subn(pattern,'Scotland', input ))
```

The output from this is:

```
('Scotland for football, Scotland for Rugby and Scotland for the
Highland games', 3)
```

31.5.11 The compile() Function

Most regular expression operations are available as both module-level functions (as described above) and as methods on a compiled regular expression object.

The module-level functions are typically simplified or standardised ways to use the compiled regular expression. In many cases these functions are sufficient but if finer grained control is required then a compiled regular expression may be used.

```
re.compile(pattern, flags=0)
```

The `compile()` function compiles a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods as described below.

The expression's behaviour can be modified by specifying a *flags* value. V

The statements:

```
prog = re.compile(pattern)
result = prog.match(string)
```

are equivalent to.

```
result = re.match(pattern, string)
```

but using `re.compile()` and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

Compiled regular expression objects support the following methods and attributes:

- `Pattern.search(string, pos, endpos)` Scan through `string` looking for the first location where this regular expression produces a match and returns a corresponding Match object. Return `None` if no position in the string matches the pattern. Starting at `pos` if provided and ending at `endpos` if this is provided (otherwise process the whole string).
- `Pattern.match(string, pos, endpos)` If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding match object. Return `None` if the string does not match the pattern. The `pos` and `endpos` are optional and specify the start and end positions within which to search.
- `Pattern.split(string, maxsplit = 0)` Identical to the `split()` function, using the compiled pattern.
- `Pattern.findall(string[, pos[, endpos]])` Similar to the `findall()` function, but also accepts optional `pos` and `endpos` parameters that limit the search region like for `search()`.
- `Pattern.finditer(string[, pos[, endpos]])` Similar to the `finditer()` function, but also accepts optional `pos` and `endpos` parameters that limit the search region like for `search()`.
- `Pattern.sub(repl, string, count = 0)` Identical to the `sub()` function, using the compiled pattern.
- `Pattern.subn(repl, string, count = 0)` Identical to the `subn()` function, using the compiled pattern.

- `Pattern.pattern` the pattern string from which the pattern object was compiled.

An example of using the `compile()` function is given below. The pattern to be compiled is defined as containing 1 or more digits (0 to 9):

```
import re

line1 = 'The price is 23.55'
containsIntegers = r'\d+'
rePattern = re.compile(containsIntegers)

matchLine1 = rePattern.search(line1)
if matchLine1:
    print('Line 1 contains a number')
else:
    print('Line 1 does not contain a number')
```

The compiled pattern can then be used to apply methods such as `search()` to a specific string (in this case held in `line1`). The output generated by this is:

```
Line 1 contains a number
```

Of course the compiler pattern object supports a range of methods in addition to `search()` as illustrated by the spilt method:

```
p = re.compile(r'\W+')
s = '20 High Street'
print(p.split(s))
```

The output from this is

```
['20', 'High', 'Street']
```

31.6 Online Resources

See the Python Standard Library documentation for:

- https://docs.python.org/3/howto/regex.html Standard Library regular expression how to.
- https://pymotw.com/3/re/index.html the Python Module of the Week page for the `re` module.
- Other online resources include
- https://regexone.com An introduction to regular expressions.
- https://www.regular-expressions.info/tutorial.html a regular expressions tutorial.
- https://www.regular-expressions.info/quickstart.html regular expressions quick start.
- https://pypi.org/project/regex A well-known third-party regular expression module that extends the functionality offered by the built-in re module.

31.7 Exercises

Write a Python function to verify that a given string only contains letters (uppercase or lowercase) and numbers. Thus spaces and underbars ('_') are not allowed. An example of the use of this function might be:

```
print(contains_only_characters_and_numbers('John')) # True
print(contains_only_characters_and_numbers('John_Davies'))     #
False
print(contains_only_characters_and_numbers('42')) # True
print(contains_only_characters_and_numbers('John42')) # True
print(contains_only_characters_and_numbers('John 42')) # False
```

Write a function to verify a UK Postcode format (call it `verify_postcode`). The format of a postcode is two letters followed by 1 or 2 numbers, followed by a space, followed by one or two numbers and finally two letters. An example of a postcode is SY23 4ZZ; another postcode might be BB1 3PO; and finally we might have AA1 56NN (note this is a simplification of the UK Postcode system but is suitable for our purposes).

Using the output from this function you should be able to run the following test code:

```
# True
print("verify_postcode('SY23 3AA'):", verify_postcode('SY23
33AA'))
# True
print("verify_postcode('SY23 4ZZ'):", verify_postcode('SY23
4ZZ'))
# True
print("verify_postcode('BB1 3PO'):", verify_postcode('BB1 3PO'))
# False
print("verify_postcode('AA111 NN56'):", verify_postcode('AA111
NN56'))
# True
print("verify_postcode('AA1 56NN'):", verify_postcode('AA1
56NN'))
# False
print("verify_postcode('AA156NN'):", verify_
postcode('AA156NN'))
# False
print("verify_postcode('AA NN'):", verify_postcode('AA NN'))
```

Write a function that will extract the value held between two strings or characters such as '<' and '>'. The function should take three parameters, the start character, the end character and the string to process. For example, the following code snippet:

```
print(extract_values('<', '>', '<John>'))
print(extract_values('<', '>', '<42>'))
print(extract_values('<', '>', '<John 42>'))
print(extract_values('<', '>', 'The <town> was in the
<valley>'))
```

should generate output such as:

```
['John']
['42']
['John 42']
['town', 'valley']
```

Part VI
Database Access

Chapter 32
Introduction to Databases

32.1 Introduction

There are several different types of database system in common use today including Object databases, NoSQL databases and (probably the most common) Relational databases. This chapter focusses on Relational databases as typified by database systems such as Oracle, Microsoft SQL Server and MySQL. The database we will use in this book is MySQL.

32.2 What Is a Database?

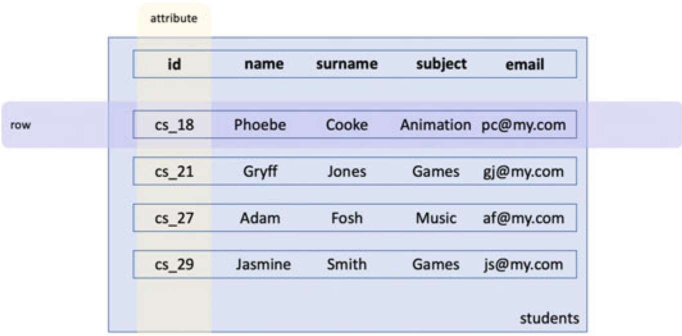
A database is essentially a way to store and retrieve data.

Typically, there is some form of query language used with the database to help select the information to retrieve such as SQL or Structured Query Language.

In most cases there is a structure defined that is used to hold the data (although this is not true of the newer NoSQL or non-relational unstructured databases such as CouchDB or MongoDB).

In Relational database the data is held in tables, where the *columns* define the properties or attributes of the data and each *row* defines the actual values being held, for example:

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_32



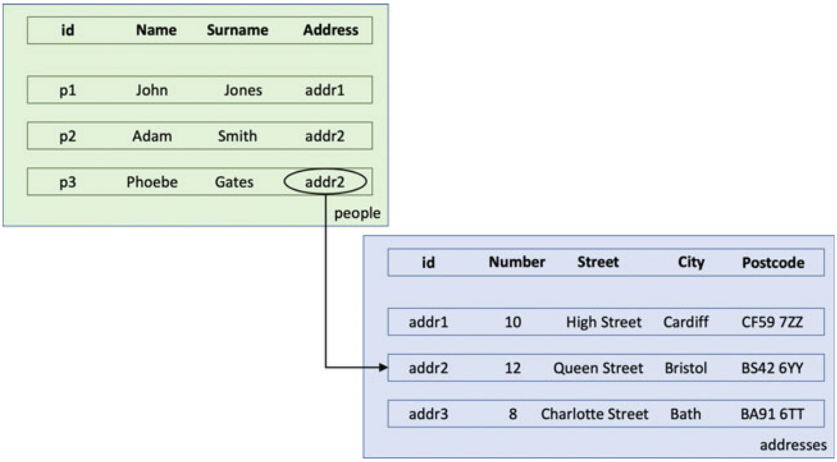
In this diagram there is a table called `students`; it is being used to hold information about students attending a meeting. The table has 5 attributes (or columns) defined for `id`, `name`, `surname`, `subject` and `email`.

In this case, the `id` is probably what is known as a *primary key*. The primary key is a property that is used to uniquely identify the student row; it cannot be omitted and must be unique (within the table). Obviously names and subjects may well be duplicated as there may be more than one student studying *Animation* or *Games* and students may have the same first name or surname. It is probable that the email column is also unique as students probably don't share an email address but again this may not necessarily be the case.

You might at this point wonder why the data in a Relational database is called *relational* and not tables or tabular? The reason is because of a topic known as relational algebra that underpins Relational database theory. Relational Algebra takes its name from the mathematical concept known as a *relation*. However, for the purposes of this chapter you don't need to worry about this and just need to remember that data is held in tables.

32.2.1 Data Relationships

When the data held in one table has a link or relationship to data held in another table then an index or key is used to link the values in one table to another. This is illustrated below for a table of *addresses* and a table of *people* who live in that address. This shows for example, that 'Phoebe Gates' lives at address 'addr2' which is 12 Queen Street, Bristol, BS42 6YY.



This is an example of a many to one (often written as many: 1) relationship; that is there are many people who can live at one address (in the above Adam Smith also lives at address ‘addr2’). In Relational databases there can be several different types of relationship such as:

- **One: one** where only one row in one table references one and only one row in another table. An example of a one to one relationship might be from a person to an order for a unique piece of jewellery.
- **One: many** this is the same as the above address example; however in this case the direction of the relationship is reversed (that is to say that one address in the *addresses* table can reference multiple persons in the *people* table).
- **Many: many** this is where many rows in one table may reference many rows in a second table. For example, many students may take a particular class and a student may take many classes. This relationship usually involves an intermediate (join) table to hold the associations between the rows.

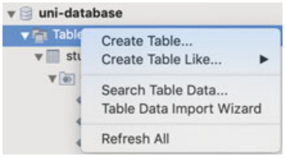
32.2.2 The Database Schema

The structure of a Relational database is defined using a Data Definition Language or Data Description Language (a DDL).

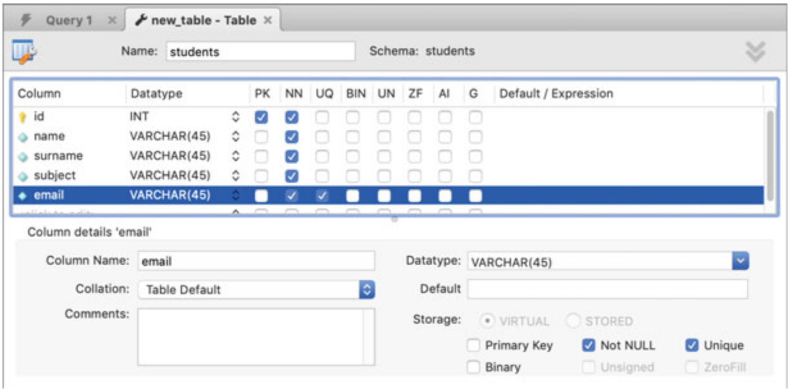
Typically, the syntax of such a language is limited to the semantics (meaning) required to define the structure of the tables. This structure is known as the database schema. Typically, the DDL has commands such as `CREATE TABLE`, `DROP TABLE` (to delete a table) and `ALTER TABLE` (to modify the structure of an existing table).

Many tools provided with a database allow you to define the structure of the database without getting too bound up in the syntax of the DDL; however, it is useful to be aware of it and to understand that the database can be created in this way. For example, we will use the MySQL database in this chapter. The MySQL Workbench is a tool that allows you to work with MySQL databases to manage and query the data held within a particular database instance. For references for MySQL and the MySQL Workbench see the links at the end of this chapter.

As an example, within the MySQL Workbench we can create a new table using a menu option on a database:



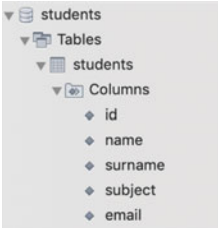
Using this we can interactively define the columns that will comprise the table:



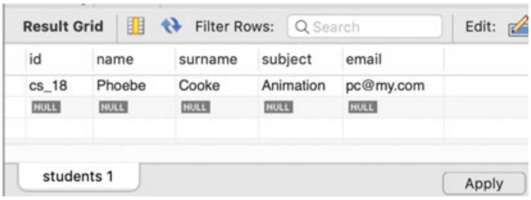
Here each column name, its type and whether it is the primary key (PK), not empty (or Not Null NN) or unique (UQ) have been specified. When the changes are applied, the tool also shows you the DDL that will be used to create the database:

```
CREATE TABLE `students`.`students` (
  `id` INT NOT NULL,
  `name` VARCHAR(45) NOT NULL,
  `surname` VARCHAR(45) NOT NULL,
  `subject` VARCHAR(45) NOT NULL,
  `email` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE INDEX `email_UNIQUE` (`email` ASC));
```

When this is applied a new table is created in the database as shown below:



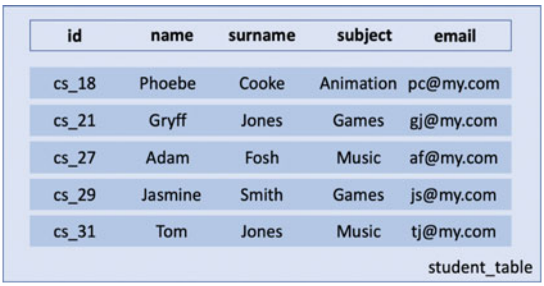
The tool also allows us to populate data into the table; this is done by entering data into a grid and hitting apply as shown below:



32.3 SQL and Databases

We can now use query languages to identify and return data held in the database often using specific criteria.

For example, let us say we want to return all the people who have the surname Jones from the following table:



id	name	surname	subject	email
cs_18	Phoebe	Cooke	Animation	pc@my.com
cs_21	Gryff	Jones	Games	gj@my.com
cs_27	Adam	Fosh	Music	af@my.com
cs_29	Jasmine	Smith	Games	js@my.com
cs_31	Tom	Jones	Music	tj@my.com

student_table

We can do this by specifying that data should be returned where the surname equals 'Jones'; in SQL this would look like:

```
SELECT * FROM students where surname='Jones';
```

The above SELECT statement states that all the properties (columns or attributes) in a row in the table students are to be returned where the surname equals 'Jones'. The result is that two rows are returned:

id	name	surname	subject	email
2	Gryff	Jones	Games	gj@my.com
5	Tom	Jones	Music	tj@my.com

Note we need to specify the table we are interested in and what data we want to return (the '*' after the select indicated we want all the data). If we were only interested in their first names then we could use:

```
SELECT name FROM students where surname='Jones';
```

This would return only the names of the students:

name
Gryff
Tom

32.4 Data Manipulation Language

Data can also be inserted into a table or existing data in a table can be updated. This is done using the Data Manipulation Language (DML).

For example, to insert data into a table we merely need to write an INSERT SQL statement providing the values to be added and how they map to the columns in the table:

```
INSERT INTO 'students' ('id', 'name', 'surname', 'subject',
'email') VALUES ('6', 'James', 'Andrews', 'Games', 'ja@my.com');
```

This would add the row 6 to the table `students` with the result that the table would now have an additional row:

id	name	surname	subject	email
1	Phoebe	Cooke	Animation	pc@my.com
2	Gryff	Jones	Games	gj@my.com
3	Adam	Fosh	Music	af@my.com
4	Jasmine	Smith	Games	js@my.com
5	Tom	Jones	Music	tj@my.com
6	James	Andrews	Games	ja@my.com

Updating an existing row is a little more complicated as it is first necessary to identify the row to be updated and then the data to modify. Thus an UPDATE statement includes a where clause to ensure the correct row is modified:

```
UPDATE 'students' SET 'email'='grj@my.com' WHERE 'id'='2';
```

The effect of this code is that the second row in the students table is modified with the new email address:

id	name	surname	subject	email
1	Phoebe	Cooke	Animation	pc@my.com
2	Gryff	Jones	Games	grj@my.com
3	Adam	Fosh	Music	af@my.com
4	Jasmine	Smith	Games	js@my.com
5	Tom	Jones	Music	tj@my.com
6	James	Andrews	Games	ja@my.com

32.5 Transactions in Databases

Another important concept within a database is that of a Transaction. A Transaction represents a unit of work performed within a database management system (or similar system) against a database instance and is independent of any other transaction.

Transactions in a database environment have two main purposes.

- To provide a unit of work that allows recovery from failures and keeps a database consistent even in cases of system failure, when execution stops (completely or partially). This is because either all the operations within a transaction are performed or none of them are. Thus, if one operation causes an error then all the changes being made by the transaction thus far are rolled back and none of them will have been made.
- To provide isolation between programs accessing a database concurrently. This means that the work being done by one program will not interact with another programs work.

A database transaction, by definition, must be atomic, consistent, isolated and durable:

- **Atomic** This indicates that a transaction represents an atomic unit of work; that is either all the operations in the transaction are performed or none of them are performed.
- **Consistent** Once completed the transaction must leave the data in a consistent state with any data constraints met (such as a row in one table must not reference a non-existent row in another table in a one to many relationship, etc.).
- **Isolated** This relates to the changes being made by concurrent transactions; these changes must be isolated from each other. That is, one transaction cannot see the changes being made by another transaction until the second transaction completes and all changes are permanently saved into the database.
- **Durable** This means that once a transaction completes then the changes it has made are permanently stored into the database (until some future transaction modifies that data).

Database practitioners often refer to these properties of database transactions using the acronym ACID (for Atomic, Consistent, Isolated, Durable).

Not all databases support transactions although all commercial, production quality databases such as Oracle, Microsoft SQL Server and MySQL do support transactions.

32.6 Further Reading

If you want to know more about databases and database management systems here are some online resources:

- https://en.wikipedia.org/wiki/Database which is the Wikipedia entry for databases and thus acts as a useful quick reference and jumping off point for other material.
- https://en.wikibooks.org/wiki/Introduction_to_Computer_Information_Systems/Database which provides a short introduction to databases.
- https://www.techopedia.com/6/28832/enterprise/databases/introduction-to-databases another useful starting point for delving deeper into databases.
- https://en.wikipedia.org/wiki/Object_database for information on Object databases.
- https://en.wikipedia.org/wiki/NoSQL for an introduction to No SQL or non-relational databases.
- https://www.mysql.com/ for the MySQL database.
- https://dev.mysql.com/downloads/workbench The MySQL Workbench home page.
- https://www.mongodb.com/ for the home page of the MongoDB site.
- http://couchdb.apache.org/ for the Apache Couch database.

If you want to explore the subject of database design (that is design of the tables and links between tables in a database) then these references may help:

- https://en.wikipedia.org/wiki/Database_design the Wikipedia entry for database design.

- https://www.udemy.com/cwdatabase-design-introduction/which covers most of the core ideas within database design.
- http://en.tekstenuitleg.net/articles/software/database-design-tutorial/intro.html which provides another tutorial that covers most of the core elements of database design.

If you wish to explore SQL more then see:

- https://en.wikipedia.org/wiki/SQL the Wikipedia site for SQL
- https://www.w3schools.com/sql/sql_intro.asp which is the W3 school material on SQL and as such an excellent resource.
- https://www.codecademy.com/learn/learn-sql which is a codecademy site for SQL.

Chapter 33
Python DB-API

33.1 Accessing a Database from Python

The standard for accessing a database in Python is the Python DB-API. This specifies a set of standard interfaces for modules that wish to allow Python to access a specific database. The standard is described in PEP 249 (https://www.python.org/dev/peps/pep-0249)—a PEP is a Python Enhancement Proposal.

Almost all Python database access modules adhere to this standard. This means that if you are moving from one database to another, or attempting to port a Python program from using one database to another, then the APIs you encounter should be very similar (although the SQL processed by different database can also differ). There are modules available for most common databases such as MySQL, Oracle and Microsoft SQL Server.

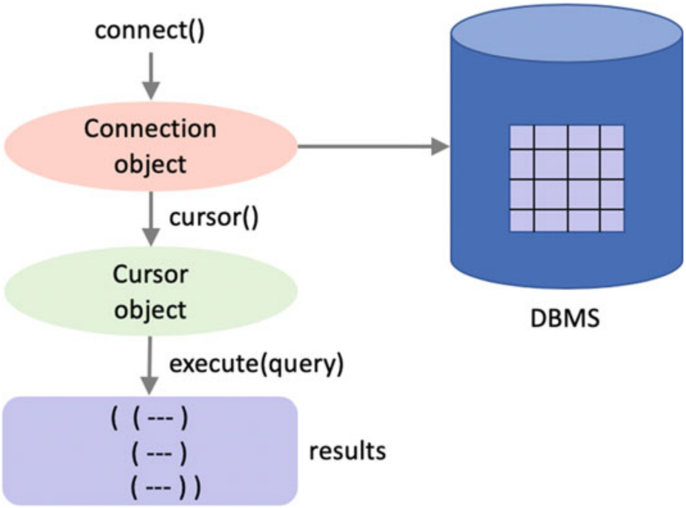
33.2 The DB-API

There are several key elements to the DB-API; these are:

- **The connect function**. The `connect()` function that is used to connect to a database and returns a Connection Object.
- **Connection Objects**. Within the DB-API access to a database is achieved through connection objects. These connection objects provide access to cursor objects.
- **Cursor objects** are used to execute SQL statements on the database.
- **The result of an execution**. These are the results that can be fetched as a sequence of sequences (such a tuple of tuples). The standard can thus be used to select, insert or update information in the database.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_33

These elements are illustrated below:



The standard specifies a set of functions and objects to be used to connect to a database. These include the connection function, the Connection Object and the Cursor object.

The above elements are described in more detail below.

33.2.1 The Connect Function

The connection function is defined as:

```
1. connect(parameters...)
```

It is used to make the initial connection to the database. The connection returns a Connection Object. The parameters required by the connection function are database dependent.

33.2.2 The Connection Object

The Connection Object is returned by the `connect()` function. The Connection object provides several methods including:

- `close()` used to close the connection once you no longer need it. The connection will be unusable from this point onwards.

- `commit()` used to commit a pending transaction.
- `rollback()` used to rollback all the changes made to the database since the last transaction commit (optional as not all databases provide transaction support).
- `cursor()` returns a new `Cursor` object to use with the connection.

33.2.3 The Cursor Object

The Cursor object is returned from the `connection.cusor()` method. A Cursor Object represents a database cursor, which is used to manage the context of a fetch operation or the execution of a database command. Cursors support a variety of attributes and methods:

- `cursor.execute(operation, parameters)` Prepare and execute a database operation (such as a query statement or an update command). Parameters may be provided as a sequence or mapping and will be bound to variables in the operation. Variables are specified in a database-specific notation.
- `cursor.rowcount` A read-only attribute providing the number of rows that the last `cursor.execute()` call returned (for select style statements) or affected (for update or insert style statements).
- `cursor.description` A read-only attribute providing information on the columns present in any results returned from a SELECT operation.
- `cursor.close()` Closes the cursor. From this point on the cursor will not be usable.

In addition, the Cursor object also provides several fetch style methods. These methods are used to return the results of a database query. The data returned is made up of a sequence of sequences (such as a tuple of tuples) where each inner sequence represents a single row returned by the SELECT statement. The fetch methods defined by the standard are:

- `cursor.fetchone()` Fetch the next row of a query result set, returning a single sequence, or `None` when no more data is available.
- `cursor.fetchall()` Fetch all (remaining) rows of a query result, returning them as a sequence of sequences.
- `cursor.fetchman(size)` Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a tuple of tuples). An empty sequence is returned when no more rows are available. The number of rows to fetch per call is specified by the parameter.

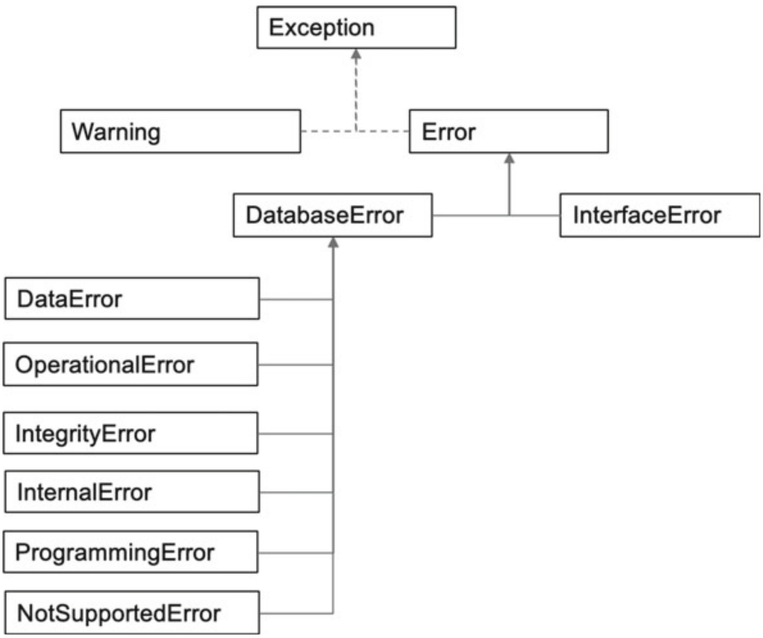
33.2.4 Mappings from Database Types to Python Types

The DB-API standard also specifies a set of mappings from the types used in a database to the types used in Python. For a full listing see the DB-API standard itself but the key mappings include:

`Date(year, month, day)`	Represents a database date
`Time(hour, minute, second)`	Represents a time database value
`Timestamp(year, month, day, hour, minute, second)`	Holds a database timestamp value
`String`	Used to represent string like database data (such as VARCHARs)

33.2.5 Generating Errors

The standard also specifies a set of Exceptions that can be thrown in different situations. These are presented below and in the following table:



The above diagram illustrates the inheritance hierarchy for the errors and warning associated with the standard. Note that the DB-API `Warning` and `Error` both extend the `Exception` class from standard Python; however, depending on the specific implementation there may be one or more additional classes in the hierarchy between these classes. For example, in the PyMySQL module there is a `MySQLError` class that extends `Exception` and is then extended by both `Warning` and `Error`.

Also note that `Warning` and `Error` have no relationship with each other. This is because Warnings are not considered Errors and thus have a separate class hierarchies. However, the `Error` is the root class for all database `Error` classes.

A description of each `Warning` or `Error` class is provided below.

`Warning`	Used to warn of issues such as data truncations during inserting, etc
`Error`	The base class of all other error exceptions
`InterfaceError`	Exception raised for errors that are related to the database interface rather than the database itself
`DatabaseError`	Exception raised for errors that are related to the database
`DataError`	Exception raised for errors that are due to problems with the data such as division by zero and numeric value out of range
`OperationalError`	Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, etc
`IntegrityError`	Exception raised when the relational integrity of the database is affected
`InternalError`	Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc
`ProgrammingError`	Exception raised for programming errors, e.g. table not found, syntax error in the SQL statement, wrong number of parameters specified, etc
`NotSupportedError`	Exception raised in case a method or database API was used which is not supported by the database, e.g. requesting a `.rollback()` on a connection that does not support transactions or has transactions turned off

33.2.6 Row Descriptions

The `Cursor` object has an attribute `description` that provides a sequence of sequences; each subsequence provides a description of one of the attributes of the data returned by a SELECT statement. The sequence describing the attribute is made up of up to seven items, these include:

- name representing the name of the attribute.
- type_code which indicates what Python type this attribute has been mapped to.
- display_size the size used to display the attribute.
- internal_size the size used internally to represent the value.
- precision if a real numeric value the precision supported by the attribute.
- scale indicates the scale of the attribute.
- null_ok indicates whether null values are acceptable for this attribute.

The first two items (name and type_code) are mandatory; the other five are optional and are set to None if no meaningful values can be provided.

33.3 Transactions in PyMySQL

Transactions are managed in PyMySQL via the database connection object. This object provides the following method:

- connection.commit() This causes the current transaction to commit all the changes made permanently to the database. A new transaction is then started.
- connection.rollback() This causes all changes that have been made so far (but not permanently stored in to the database, i.e. not committed) to be removed. A new transaction is then started.

The standard does not specify how a database interface should manage turning on and off transaction (not least because not all databases support transactions). However, MySQL does support transactions and can work in two modes; one supports the use of transactions as already described; the other uses an *autocommit* mode. In autocommit mode each command sent to the database (whether a SELECT statement or an INSERT/UPDATE statement) is treated as an independent transaction, and any changes are automatically committed at the end of the statement. This *autocommit* mode can be turned on in PyMySQL using:

- connection.autocommit(True) turns on autocommit (False to turn off autocommit which is the default).

Other associated methods include

- connection.get_autocommit() which returns a Boolean indicating whether autocommit is turned on or not.
- connection.begin() to explicitly begin a new transaction.

33.4 Online Resources

See the following online resources for more information on the Python Database API:

1. https://www.python.org/dev/peps/pep-0249/ Python Database API Specification V2.0.
2. https://wiki.python.org/moin/DatabaseProgramming Database Programming in Python.
3. https://docs.python-guide.org/scenarios/db/ Databases and Python.

Chapter 34
PyMySQL Module

34.1 The PyMySQL Module

The PyMySQL module provides access to a MySQL database from Python. It implements the Python DB-API v 2.0. This module is a pure Python database interface implementation meaning that it is portable across different operating systems; this is notable because some database interface modules are merely wrappers around other (native) implementations that may or may not be available on different operating systems. For example, a native Linux-based database interface module may not be available for the Windows operating system. If you are never going to switch between different operating systems, then this is not a problem of course.

To use the PyMySQL module you will need to install it on your computer. This will involve using a tool such as Anaconda or adding it to your PyCharm project. You can also use pip to install it:

```
> pip install PyMySQL
```

34.2 Working with the PyMySQL Module

To use the PyMySQL module to access a database you will need to follow these steps.

1. Import the module.
2. Make a connection to the host machine running the database and to the database you are using.
3. Obtain a cursor object from the connection object.
4. Execute some SQL using the `cursor.execute()` method.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_34

5. Fetch the result(s) of the SQL using the cursor object (e.g. `fetchall`, `fetchmany` or `fetchone`).
6. Close the database connection.

These steps are essentially boiler plate, code that is you will use them whenever you access a database via PyMySQL (or indeed any DB-API compliant module).

We will take each of these steps in turn.

34.2.1 Importing the Module

As the PyMySQL module is not one of the built-in modules provided by default with Python you will need to import the module into your code, for example using

```
import pymsql
```

Be careful with the case used here as the module name is `pymysql` in the code (if you try to import PyMySQL Python will not find it!).

34.2.2 Connect to the Database

Each database module will have their own specifics for connecting to the database server; these usually involve specifying the machine that the database is running on (as databases can be quite resource intensive, they are often run on a separate physical computer), the user to use for the connection and any security information required such as a password and the database instance to connect to. In most cases a database is looked after by a database management system (a DBMS) that can manage multiple database instances, and it is therefore necessary to specify which database instance you are interested in.

For MySQL, the MySQL database server is a DBMS that can indeed look after multiple database instances. The `pymysql.connect` function thus requires the following information when connecting to the database is:

- The name of the machine hosting the MySQL database server, e.g. dbserver.mydomain.com. If you want to connect to the same machine as your Python program is running on, then you can use *localhost*. This is a special name reserved for the local machine and avoids you needing to worry about the name of your local computer.
- The *user name* to use for the connection. Most databases limit access to their databases to named *users*. These are not necessary users such as humans that log into a system but rather entities that are allowed to connect to the database and perform certain operations. For example, one user may only be able to read data in the database where as another user is allowed to insert new data into the database. These users are authenticated by requiring them to provide a password.

- The password for the user.
- The database instance to connect to. As mentioned in the previous chapter a database management system (DMS) can manage multiple database instances, and thus it is necessary to say which database instance you are interested in.

For example:

```
# Open database connection
connection = pymysql.connect(host='localhost',
     user='user',
     password='password',
     database='uni-database')
print('connection:', connection)
```

In this case the machine we are connecting to is 'localhost' (that is the same machine as the Python program itself is running on), the user is represented by 'username' and 'password', and the database instance of interest is called 'uni-database'.

This returns a Connection object as per the DB-API standard.

34.2.3 Obtaining the Cursor Object

You can obtain the cursor object from the connection using the `cursor()` method:

```
# prepare a cursor object using cursor() method
cursor = connection.cursor()
```

34.2.4 Using the Cursor Object

Once you have obtained the cursor object you can use it to execute an SQL query or a DML insert, update or delete statement. The following example uses a simple select statement to select all the attributes in the students table for all rows currently stored in the students table:

```
# execute SQL query using execute() method.
cursor.execute('SELECT * FROM students')
```

Note that this method executes the `SELECT` statement but does not return the set of results directly. Instead the executed method returns an integer indicating the number of rows either affected by the modification or returned as part of the query. In the case of a `SELECT` statement the number returned can be used to determine which type of fetch method to use.

34.2.5 Obtaining Information About the Results

The Cursor Object can also be used to obtain information about the results to be fetched such as how many rows there are in the results and what the type is of each attribute in the results:

- `cusor.rowcount()` This is a read-only property that indicates the number of rows returned for a SELECT statement or rows affected for a UPDATE or INSERT statement.
- `cursor.description()` This is a read-only property that provides a description of each attribute in the results set. Each description provides the name of the attribute and an indication of the type (via a `type_code`) as well as further information on whether the value can be null or not and for numbers scale, precision and size information.

An example of using these two properties is given below:

```
print('cursor.rowcount', cursor.rowcount)
print('cursor.description', cursor.description)
```

A sample of the output generated by these lines is given below:

```
cursor.rowcount 6
cursor.description (('id', 3, None, 11, 11, 0, False), ('name',
253, None, 180, 180, 0, False), ('surname', 253, None, 180, 180, 0,
False), ('subject', 253, None, 180, 180, 0, False), ('email', 253,
None, 180, 180, 0, False))
```

34.2.6 Fetching Results

Now that a successful `SELECT` statement has been run against the database, we can *fetch* the results. The results are returned as a tuple of tuples. As mentioned in the last chapter there are several different fetch options available including `fetchone()`, `fetchmany(size)` and `fetchall()`. In the following example we use the `fetchall()` option as we know that there are only up to six rows that can be returned.

```
# Fetch all the rows and then iterate over the data
data = cursor.fetchall()
for row in data:
    print('row:', row)
```

In this case we loop through each tuple within the data collection and print that row out. However, we could just as easily have extracted the information in the tuple into individual elements. These elements could then be used to construct an object that could then be processed within an application, for example:

```
for row in data:
     id, name, surname, subject, email = row
     student = Student(id, name, surname, subject, email)
     print(student)
```

34.2.7 Close the Connection

Once you have finished with the database connection it should be closed.

```
# disconnect from server
connection.close()
```

34.3 Complete PyMySQL Query Example

A complete listing illustrating connecting up to the database, running a SELECT statement and printing out the results using a Student class is given below:

```
import pymysql

class Student:
    def __init__(self, student_id, name, surname, subject, email,
year):
        self.student_id = student_id
        self.name = name
        self.surname = surname
        self.subject = subject
        self.email = email
        self.year = year

    def __str__(self):
                return  f'Student[{self.student_id}]  {self.name}
{self.surname} - {self.subject} {self.email} {self.year}'

# Open database connection
connection = pymysql.connect(host='localhost',
    user='user',
    password='password',
    database='uni-database')

# prepare a cursor object using cursor() method
cursor = connection.cursor()

# execute SQL query using execute() method.
cursor.execute('SELECT * FROM students')

# Fetch all the rows
data = cursor.fetchall()
```

```
# Convert data into Student objects
for row in data:
    student_id, name, surname, subject, email, year = row
     student = Student(student_id, name, surname, subject, email,
year)
    print(student)

# disconnect from server
connection.close()
```

The output from this program, for the database created in the last chapter, is shown here:

```
Student[1] Phoebe Cooke - Animation pc@my.com 1
Student[2] Gryff Jones - Games grj@my.com 2
Student[3] Adam Fosh - Music af@my.com MSC
Student[4] Natalia Smith - Games js@my.com PHD
Student[5] Tom Jones - Music tj@my.com 3
Student[6] James Andrews - Games ja@my.com 2
Student[7] Phoebe Cooke - Animation pc1@my.com 1
Student[8] Jill Matthews - Law jm@my.com 2
```

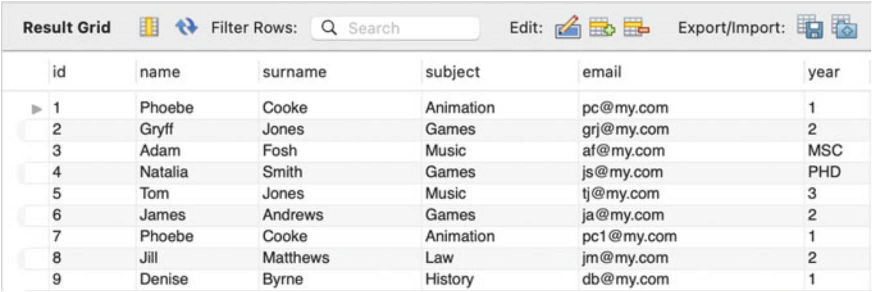
34.4 Inserting Data to the Database

As well as reading data from a database many applications also need to add new data to the database. This is done via the Data Manipulation Language (DML) INSERT statement. The process for this is very similar to running a query against the database using a SELECT statement; that is, you need to make a connection, obtain a cursor object and execute the statement. The one difference here is that you do not need to fetch the results.

```
import pymysql

# Open database connection
# Open database connection
connection = pymysql.connect(host='localhost',
                                                      user='user',
                                                     password='password',
                                                database='uni-database')
connection.autocommit(False)

# prepare a cursor object using cursor() method
cursor = connection.cursor()

try:
    # Execute INSERT command
    cursor.execute(
    "INSERT INTO students (id, name, surname, subject, email, year)
VALUES (9, 'Denise', 'Byrne', 'History', 'db@my.com', 1)")
    # Commit the changes to the database
```

```
        connection.commit()

    except:
        # Something went wrong
        # rollback the changes
        connection.rollback()

    # Close the database connection
    connection.close()
```

The result of running this code is that the database is updated with a seventh row for ‘Denise Byrne’. This can be seen in the MySQL Workbench if we look at the contents of the students table:



	id	name	surname	subject	email	year
▶	1	Phoebe	Cooke	Animation	pc@my.com	1
	2	Gryff	Jones	Games	grj@my.com	2
	3	Adam	Fosh	Music	af@my.com	MSC
	4	Natalia	Smith	Games	js@my.com	PHD
	5	Tom	Jones	Music	tj@my.com	3
	6	James	Andrews	Games	ja@my.com	2
	7	Phoebe	Cooke	Animation	pc1@my.com	1
	8	Jill	Matthews	Law	jm@my.com	2
	9	Denise	Byrne	History	db@my.com	1

There are a couple of points to note about this code example. The first is that we have used the double quotes around the string defining the INSERT command—this is because a double quotes string allows us to include single quotes within that string. This is necessary as we need to quote any string values passed to the database (such as ‘Denise’).

The second thing to note is that by default the PyMySQL database interface requires the programmer to decide when to commit or rollback a transaction. A transaction was introduced in the last chapter as an atomic unit of work that must either be completed or as a whole or rollback so that no changes are made. However, the way in which we indicate that a transaction is completed is by calling the `commit()` method on the database connection. In turn we can indicate that we want to rollback the current transaction by calling `rollback()`. In either case, once the method has been invoked a new transaction is started for any further database activity.

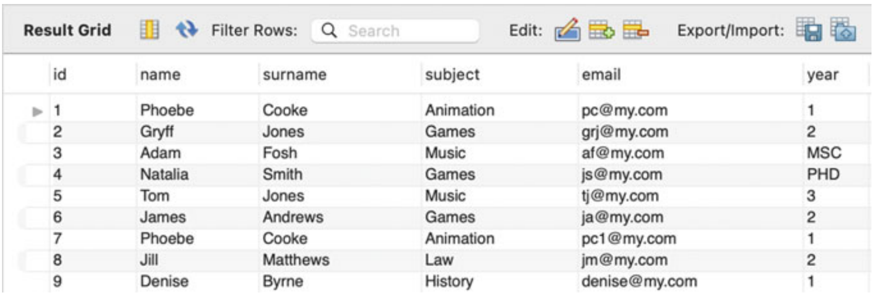
In the above code we have used a `try` block to ensure that if everything succeeds, we will commit the changes made, but if an exception is thrown (of any kind) we will rollback the transaction—this is a common pattern.

34.5 Updating Data in the Database

If we are able to insert new data into the database, we may also want to update the data in a database, for example to correct some information. This is done using the UPDATE statement which must indicate which existing row is being updated as well as what the new data should be.

```
import pymysql

# Open database connection
connection = pymysql.connect(host='localhost',
                             user='user',
                             password='password',
                             database='uni-database')
connection.autocommit(False)

# prepare a cursor object using cursor() method
cursor = connection.cursor()

try:
    # Execute UPDATE command
     cursor.execute("UPDATE students SET email = 'denise@my.com'
WHERE id = 9")

    # Commit the changes to the database
    connection.commit()
except:
    # rollback the changes if an exception / error
    connection.rollback()

# Close the database connection
connection.close()
```

In this example we are updating the student with id 7 such that their email address will be changed to 'denise@my.com'. This can be verified by examining the contents of the students table in the MySQL Workbench:



id	name	surname	subject	email	year
1	Phoebe	Cooke	Animation	pc@my.com	1
2	Gryff	Jones	Games	grj@my.com	2
3	Adam	Fosh	Music	af@my.com	MSC
4	Natalia	Smith	Games	js@my.com	PHD
5	Tom	Jones	Music	tj@my.com	3
6	James	Andrews	Games	ja@my.com	2
7	Phoebe	Cooke	Animation	pc1@my.com	1
8	Jill	Matthews	Law	jm@my.com	2
9	Denise	Byrne	History	denise@my.com	1

34.6 Deleting Data in the Database

Finally, it is also possible to delete data from a database, for example if a student leaves their course. This follows the same format as the previous two examples with the difference that the DELETE statement is used instead:

```
import pymysql

# Open database connection
connection = pymysql.connect(host='localhost',
                             user='user',
                             password='password',
                             database='uni-database')
connection.autocommit(False)

# prepare a cursor object using cursor() method
cursor = connection.cursor()

try:
    # Execute DELETE command
    cursor.execute("DELETE FROM students WHERE id = 9")

    # Commit the changes to the database
    connection.commit()
except:
    # rollback the changes if an exception / error
    connection.rollback()

# Close the database connection
connection.close()
```

In this case we have deleted the student with id 7. We can see that again in the MySQL Workbench by examining the contents of the students table after this code has run:

Result Grid Filter Rows: Search Edit: Export/Import:

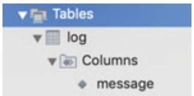
id	name	surname	subject	email	year
1	Phoebe	Cooke	Animation	pc@my.com	1
2	Gryff	Jones	Games	grj@my.com	2
3	Adam	Fosh	Music	af@my.com	MSC
4	Natalia	Smith	Games	js@my.com	PHD
5	Tom	Jones	Music	tj@my.com	3
6	James	Andrews	Games	ja@my.com	2
7	Phoebe	Cooke	Animation	pc1@my.com	1
8	Jill	Matthews	Law	jm@my.com	2

34.7 Creating Tables

It is not just data that you can add to a database; if you wish you can programmatically create new tables to be used with an application. This process follows exactly the same pattern as those used for INSERT, UPDATE and DELETE. The only difference is that the command sent to the database contains a CREATE statement with a description of the table to be created. This is illustrated below:

```
import pymysql

# Open database connection
connection = pymysql.connect(host='localhost',
                                            user='user',
                                            password='password',
                                        database='uni-database')
connection.autocommit(False)

# prepare a cursor object using cursor() method
cursor = connection.cursor()

try:
    # Execute CREATE command
     cursor.execute("CREATE TABLE log (message VARCHAR(100) NOT
NULL)")

    # Commit the changes to the database
    connection.commit()
except:
    # rollback the changes if an exception / error
    connection.rollback()

# Close the database connection
connection.close()
```

This creates a new table log within the uni-database; this can be seen by looking at the tables listed for the uni-database within the MySQL Workbench.



34.8 Online Resources

See the following online resources for more information on the Python database API:

1. https://pymysql.readthedocs.io/en/latest/ PyMySQL documentation site.
2. https://github.com/PyMySQL/PyMySQL GitHub repository for the PyMySQL library.

34.9 Exercises

In this exercise you will create a database and tables based on a set of transactions stored in a current account. You can use the account class you created in the CSV and Excel chapter for this.

You will need two tables, one for the account information and one for the transaction history.

The primary key of the account information table can be used as the foreign key for the transaction history table.

Then write a function that takes an Account object and populates the tables with the appropriate data.

To create the account information table you might use the following DDL:

```
CREATE TABLE acc_info (idacc_info INT NOT NULL, name VARCHAR(255)
NOT NULL, PRIMARY KEY (idacc_info))

While for the transactions table you might use:
                    CREATE TABLE transactions (idtransactions INT NOT
NULL, type VARCHAR(45) NOT NULL,
                     type VARCHAR(45) NOT NULL, amount VARCHAR(45) NOT
NULL, account INT
                   NOT NULL, PRIMARY KEY (idtransactions))"
```

Remember to be careful with integers and decimals if you are creating an SQL string such as:

```
statement = "INSERT into transactions (idtransactions, type,
amount, account) VALUES (" + str(id) + ", '" + action + "', " +
str(amount) + ", " + str(account_number) + ")"
```

Part VII
Logging

Chapter 35
Introduction to Logging

35.1 Introduction

Many programming languages have common logging libraries including Java and C#, and of course Python also has a logging module. Indeed the Python logging module has been part of the built-in modules since Python 2.3.

This chapter discusses why you should add logging to your programs, what you should (and should not) log and why just using the print() function is not sufficient.

35.2 Why Log?

Logging is typically a key aspect of any production application; this is because it is important to provide appropriate information to allow future investigation following some event or issue in such applications. These investigations include:

- **Diagnosing failures**; that is why did an application fail/crash.
- **Identifying unusual or unexpected behaviour**; which might not cause the application to fail but which may leave it in an unexpected state or where data may be corrupted, etc.
- **Identifying performance or capacity issues**; in such situations the application is performing as expected by it is not meeting some non-functional requirements associated with the speed at which it is operating or its ability to scale as the amount of data or the number of users grows.
- **Dealing with attempted malicious behaviour** in which some outside agent is attempting to affect the behaviour of the system or to acquire information which they should not have access to, etc. This could happen for example, if you are creating a Python web application and a user tries to hack into your web server.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_35

- **Regulatory or legal compliance**. In some cases records of program execution may be required for regulatory or legal reasons. This is particularly true of the financial sector where records must be kept for many years in case there is a need to investigate the organisations' or individuals' behaviour.

35.3 What is the Purpose of Logging?

In general there are therefore two general reasons to log what an application is doing during it operation:

- For diagnostic purposes so that recorded events/steps can be used to analyse the behaviour of the system when something goes wrong.
- Auditing purposes that allow for later analysis of the behaviour of the system for business, legal or regulatory purposes. For example, in this case to determine who did what with what and when.

Without such logged information it is impossible *after the event* to know what happened. For example, if all you know is that an application crashed (unexpectedly stopped executing) how can you determine what state the application was in, what functions, methods, etc. were being executed and which statements run?

Remember that although a developer may have been using an IDE to run their applications during development and may possibly been using the debugging facilities available that allow you to see what functions or methods, statements and even variable values are place; this is not how most production systems are run. In general a production Python system will be run either from a command line or possibly through a short cut (on a Windows box) to simplify running the program. All the user will know is that something failed or that the behaviour they expected didn't occur—if in fact they are aware of any issue at all!

Logs are therefore key to *after the event* analysis of failures, unexpected behaviour or for analysis of the operation of the system for business reasons.

35.4 What Should You Log?

One question that you might be considering at this point is 'what information should I log?'.

An application should log enough information so that post-event investigators can understand what was happening, when and where. In general this means that you will want to log the time of the log message, the module/filename, function name or method name executing, potentially the log level being used (see later) and in some cases the parameter values/state of the environment, program or class involved.

In many cases developers log the entry (and to a lesser extent) the exit from a function or method. However, it may also be useful to log what happens at branch points within a function or method so that the logic of the application can be followed.

All applications should log all errors/exceptions. Although care is needed to ensure that this is done appropriately. For example if an exception is caught and then re thrown several times it is not necessary to log it every time it is caught. Indeed doing this can make the log files much larger, cause confusion when the problem is being investigated and result in unnecessary overheads. One common approach is to log an exception where it is first raised and caught and not to log it after that.

35.5 What not to Log

The follow on question to consider is 'what information should I not log?'.

One general area not to log is any personal or sensitive information including any information that can be used to identify an individual. This sort of information is known as PII or Personally Identification Information.

Such information includes

- User ids and passwords.
- Email addresses.
- Data of birth, birth place.
- Personally identifiable financial information such as bank account details, credit card details.
- Biometric information.
- Medical/health information.
- Government issued personal information such as passport details, drivers' licence number, social security numbers, National Insurance numbers.
- Official organisational information such as professional registrations and membership numbers.
- Physical addresses, phone (land-line) numbers, mobile phone numbers.
- Verification elated information such as mother's maiden name, pets' names, high school, first school, favourite film.
- It also increasing includes online information relating to social media such as Facebook or LinkedIn accounts.

All of the above is sensitive information and much of it can be used to identify an individual; none of this information should be logged directly.

That does not mean that you cannot and shouldn't log that a user logged in; you may well need to do that. However, the information should at least be obfuscated and should not include any information not required. For example you may record that a user represented by some id attempted to log in at a specific time and whether they were successful or not. However, you should not log their password and may not log the actual userid—instead you may log an id that can be used to map to their actual userid.

You should also be careful about directly logging data input too an application directly into a log file. One way in which a malicious agent can attack an application (particularly a web application) is by attempting to send very large amounts of data to it (as part of a field or as a parameter to an operation). If the application blindly logs all data submitted to it, then the log files can fill up very quickly. This can result in the file store being used by the application filling up and causing potential problems for all software using the same file store. This form of attack is known as a log (or log file) injection attack and is well documented (see https://www.owasp.org/index.php/Log_Injection which is part of the well-respected Open Web Application Security Project).

Another point to note is that it is not merely enough to log an error. This is not error handling; logging an error does not mean you have handled it; only that you have noted it. An application should still decide how it should manage the error or exception.

In general you should also aim for empty logs in a production system; that is only information that needs to be logged in a production system should be logged (often information about errors, exceptions or other unexpected behaviour). However, during testing much more detail is required so that the execution of the system should be followed. It should therefore be possible to select how much information is logged depending on the environment the code is running in (that is within a test environment or within a production environment).

A final point to note is that it is important to log information to the correct place. Many applications (and organisations) log general information to one log file, errors and exceptions to another and security information to a third. It is therefore important to know where your log information is being sent and not to send information to the wrong log.

35.6 Why not Just Use Print?

Assuming that you want to log information in your application then next question is how should you do that? Through this book we have been using the Python `print()` function to print out information that indicates results generated by our code but also at times what is happening with a function or a method, etc.

Thus we need to consider whether using the `print()` function the best way to log information.

In actual fact, using `print()` to log information in a production system is almost never the right answer; this is for several reasons:

- The `print()` function by default writes strings out to the standard output (stdout) or standard error output (stderr) which by default directs output to the console/terminal. For example, when you run an application within an IDE, the output is displayed in the Console window. If you run an application from the command line then the output is directed back to that command/terminal window. Both of these are fine during development, but what if the program is not run from a command window, perhaps instead it is started up by the operating system automatically (as is typical of numerous services such as a print service or a web server). In this case there is no terminal/console window to send the data to; instead the data is just lost. As it happens the `stdout` and `stderr` output streams can be directed to a file (or files). However, this is typically done when the program is launched and may be easily omitted. In addition there is only the option of sending all `stdout` to a specific file or all error output to the `stderr`.

- Another issue with using the `print()` function is that all calls to print will be output. When using most loggers it is possible to specify the *log level* required. These different log levels allow different amounts of information to be generated depending upon the scenario. For example, in a well-tested reliable production system we may only want error related or critical information to be logged. This will reduce the amount of information we are collecting and reduce any performance impact introduced by logging into the application. However, during testing phases we may want a far more detailed level of logging.
- In other situations we may wish to change the log level being used for a running production system without needing to modify the actual code (as this has the potential to introduced errors into the code). Instead we would like to have the facility to externally change the way in which the logging system behaves, for example through a configuration file. This allows system administrators to modify the amount and the detail of the information being logged. It typically also allows the designation of the log information to be changed.
- Finally, when using the `print()` function a developer can use whatever format they like, they can include a timestamp on the message or not; they can include the module or function/method name or not they can include parameters of not. Using a logging system usually standardises the information generated along with the log message. Thus all log messages will have (or not have) a timestamp, or all messages will include (or not include) information on the function or method in which they were generated, etc.

35.7 Online Resources

For further information on logging see the following:

- https://en.wikipedia.org/wiki/Log_file A Wikipedia page on logging.
- https://www.codeproject.com/Articles/42354/The-Art-of-Logging An interesting article on the art of logging.
- www.owasp.org/index.php The Open Web Application Security Project (OWASP).

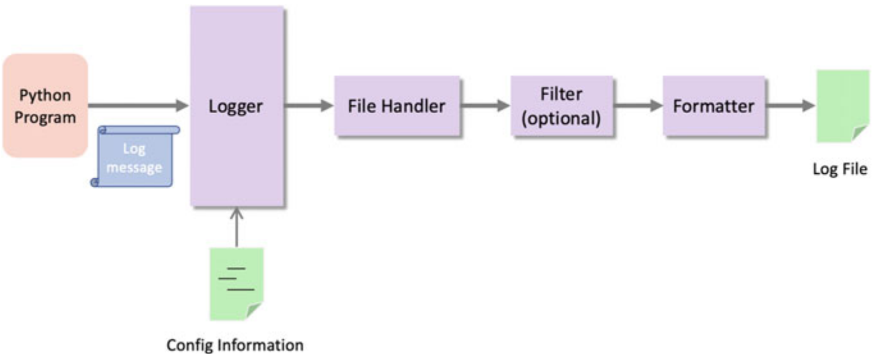
Chapter 36
Logging in Python

36.1 The Logging Module

Python has included a built-in logging module since Python 2.3. This module, the `logging` module, defines functions and classes which implement a flexible logging framework that can be used in any Python application/script or in Python libraries/modules.

Although different logging frameworks differ in the specific details of what they offer; almost all offer the same core elements (although different names are sometimes used). The Python logging module is no different, and the core elements that make up the logging framework and its processing pipeline are shown below (note that a very similar diagram could be drawn for logging frameworks in Java, Scala, C++, etc.).

The following diagram illustrates a Python program that uses the built-in Python logging framework to log messages to a file.



J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_36

The core elements of the logging framework (some of which are optional) are shown above and described below:

- **Log Message** The is the message to be logged from the application.
- **Logger** Provides the programmers entry point/interface to the logging system. The Logger class provides a variety of methods that can be used to log messages at different levels.
- **Handler** Handlers determine where to send a log message, default handlers include file handlers that send messages to a file and HTTP handlers that send messages to a web server.
- **Filter** This is an optional element in the logging pipeline. They can be used to further filter the information to be logged providing fine-grained control of which log messages are actually output (e.g. to a log file).
- **Formatter** These are used to format the log message as required. This may involve adding timestamps, module and function/method information, etc. to the original log message.
- **Configuration Information** The logger (and associated handlers, filters and formatters) can be configured either programmatically in Python or through configuration files. These configuration files can be written using key-value pairs or in a YAML file (which is a simple markup language). YAML stands for Yet Another Markup Language!

It is worth noting that much of the logging framework is hidden from the developer who really only sees the logger; the remainder of the logging pipeline is either configured by default or via log configuration information typically in the form of a log configuration file.

36.2 The Logger

The Logger provides the programmers interface to the logging pipeline.

A Logger object is obtained from the `getLogger()` function defined in the `logging` module. The following code snippet illustrates acquiring the default logger and using it to log an error message. Note that the `logging` module must be imported:

```
import logging
logger = logging.getLogger()
logger.error('This should be used with something unexpected')
```

The output from this short application is logged to the console as this is the default configuration:

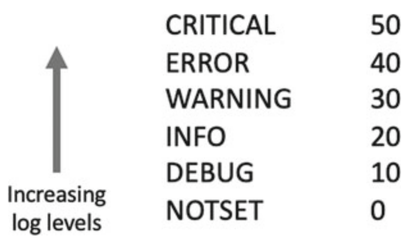
```
This should be used with something unexpected
```

36.3 Controlling the Amount of Information Logged

Log messages are actually associated with a log level. These log levels are intended to indicate the severity of the message being logged. There are six different log levels associated with the Python logging framework; these are:

- **NOTSET** At this level no logging takes place and logging is effectively turned off.
- **DEBUG** This level is intended to provide detailed information, typically of interest when a developer is diagnosing a bug or issues within an application.
- **INFO** This level is expected to provide less detail than the DEBUG log level as it is expected to provide information that can be used to confirm that the application is working as expected.
- **WARNING** This is used to provide information on an unexpected event or an indication of some likely problem that a developer or system administration might wish to investigate further.
- **ERROR** This is used to provide information on some serious issue or problem that the application has not been able to deal with and that is likely to mean that the application cannot function correctly.
- **CRITICAL** This is the highest level of issue and is reserved for critical situations such as ones in which the program can no longer continue executing.

The log levels are relative to one another and defined in a hierarchy. Each log level has a numeric value associated with it as shown below (although you should never need to use the numbers). Thus `INFO` is a higher log level than `DEBUG`, in turn `ERROR` is a higher log level than `WARNING`, `INFO`, `DEBUG`, etc.



Associated with the log level that a message is logged with, a logger also has a log level associated with it. The logger will process all messages that are at the loggers log level or above that level. Thus if a logger has a log level of `WARNING` then it will log all messages logged using the `warning`, `error` and `critical` log levels.

Generally speaking, an application will not use the `DEBUG` level in a production system. This is usually considered inappropriate as it is only intended for debug scenarios. The `INFO` level may be considered appropriate for a production system although it is likely to produce large amounts of information as it typically traces the execution of functions and methods. If an application has been well tested and verified then it is only really warnings and errors which should occur/be of concern. It is therefore not uncommon to default to the `WARNING` level for production systems (indeed this is why the default log level is set to `WARNING` within the Python logging system).

If we now look at the following code that obtains the default logger object and then uses several different logger methods, we can see the effect of the log levels on the output:

```
import logging

logger = logging.getLogger()

logger.debug('This is to help with debugging')
logger.info('This is just for information')
logger.warning('This is a warning!')
logger.error('This should be used with something unexpected')
logger.critical('Something serious')
```

The default log level is set to *warning*, and thus only messages logged at the warning level or above will be printed out:

```
This is a warning!
This should be used with something unexpected
Something serious
```

As can be seen from this, the messages logged at the debug and info level have been ignored.

However, the Logger object allows us to change the log level programmatically using the `setLevel()` method, for example `logger.setLevel(logging.DEBUG)` or via the `logging.basicConfig(level=logging.DEBUG)` function; both of these will set the logging level to `DEBUG`. Note that the log level must be set before the logger is obtained.

If we add one of the above approaches to setting the log level to the previous program we will change the amount of log information generated:

```
import logging

logging.basicConfig(level=logging.DEBUG)

logger = logging.getLogger()

logger.warning('This is a warning!')
logger.info('This is just for information')
logger.debug('This is to help with debugging')
logger.error('This should be used with something unexpected')
logger.critical('Something serious')
```

This will now output all the log messages as `debug` is the lowest logging level. We can of course turn off logging by setting the log level to `NOTSET`.

```
logger.setLevel(logging.NOTSET)
```

Alternatively you can set the Loggers `disabled` attribute to `True`:

```
logging.Logger.disabled = True
```

36.4 Logger Methods

The Logger class provides a number of methods that can be used to control what is logged including:

- `setLevel(level)` sets this loggers log level.
- `getEffectiveLevel()` returns this loggers log level.
- `isEnabledFor(level)` checks to see if this logger is enabled for the log level specified.
- `debug(message)` logs messages at the debug level.
- `info(message)` logs messages at the info level.
- `warning(message)` logs messages at the warning level.
- `error(message)` logs messages at the error level.
- `critical(message)` logs messages at the critical level.
- `exception(message)` logs a message at the error level. However, it can only be used within an exception handler and includes a stack trace of any associated exception, for example:

```
import logging

logger = logging.getLogger()

try:
    print('starting')
    x = 1 / 0
    print(x)
except:
    logger.exception('an exception message')

print('Done')
```

- `log(level, message)` logs messages at the log level specified as the first parameter.

In addition there are several methods that are used to manage handlers and filters:

- `addFilter(filter)` This method adds the specified filter to this logger.
- `removeFilter(filter)` The specified filter is removed from this logger object.
- `addHandler(handler)` The specified handler is added to this logger.
- `removeHandler(handler)` Removes the specified handler from this logger.

36.5 Default Logger

A default (or *root*) logger is always available from the logging framework.

This logger can be accessed via the functions defined in the `logging` module. These functions allow messages to be logged at different levels using methods such as `info()`, `error()`, `warning()` but without the need to obtain a reference to a logger object first. For example:

```
import logging

# Set the root logger level
logging.basicConfig(level=logging.DEBUG)

# Use root (default) logger
logging.debug('This is to help with debugging')
logging.info('This is just for information')
logging.warning('This is a warning!')
logging.error('This should be used with something unexpected')
logging.critical('Something serious')
```

This example sets the logging level for the root or default logger to `DEBUG` (the default is `WARNING`). It then uses the default logger to generate a range of log messages at different levels (from `DEBUG` up to `CRITICAL`). The output from this program is given below:

```
DEBUG:root:This is to help with debugging
INFO:root:This is just for information
WARNING:root:This is a warning!
ERROR:root:This should be used with something unexpected
CRITICAL:root:Something serious
```

Note that the format used by default with the root logger prints the log level, the name of the logger generating the output and the message. From this you can see that it is the root longer that is generating the output.

36.6 Module Level Loggers

Most modules will not use the root logger to log information; instead they will use a *named* or *module level* logger. Such a logger can be configured independently of the root logger. This allows developers to turn on logging just for a module rather than for a whole application. This can be useful if a developer wishes to investigate an issue that is located within a single module.

Previous code examples in this chapter have used the `getLogger()` function with no parameters to obtain a logger object, for example:

```
logger = logging.getLogger()
```

This is really just another way of obtaining a reference to the root logger which is used by the stand alone logging functions such as `logging.info()`, `logging.debug()` function, thus:

```
logging.warning('my warning')
```

and

```
logger = logging.getLogger()
logger.warning('my warning')
```

Have exactly the same effect; the only difference is that the first version involves less code.

However, it is also possible to create a *named* logger. This is a separate logger object that has its own name and can potentially have its own log level, handlers, formatters, etc. To obtain a named logger pass a *name* string into the `getLogger()` method:

```
logger1 = logging.getLogger('my logger')
```

This returns a logger object with the name ‘my logger’. Note that this may be a brand new logger object; however if any other code within the current system has previously requested a logger called ‘my logger’ then that logger object will be returned to the current code. Thus multiple calls to `getLogger()` with the same name will always return a reference to the same Logger object.

It is common practice to use the name of the module as the name of the logger; as only one module with a specific name should exist within any specific system. The name of the module does not need to be hard coded as it can be obtained using the `__name__` module attribute, it is thus common to see:

```
logger2 = logging.getLogger(__name__)
```

We can see the effect of each of these statements by printing out each logger:

```
logger = logging.getLogger()
print('Root logger:', logger)

logger1 = logging.getLogger('my logger')
print('Named logger:', logger1)

logger2 = logging.getLogger(__name__)
print('Module logger:', logger2)
```

When the above code is run the output is:

```
Root logger: <RootLogger root (WARNING)>
Named logger: <Logger my logger (WARNING)>
Module logger: <Logger __main__ (WARNING)>
```

This shows that each logger has their own name (the code was run in the main module, and thus the module name was `__main__`), and all three loggers have an effective log level of `WARNING` (which is the default).

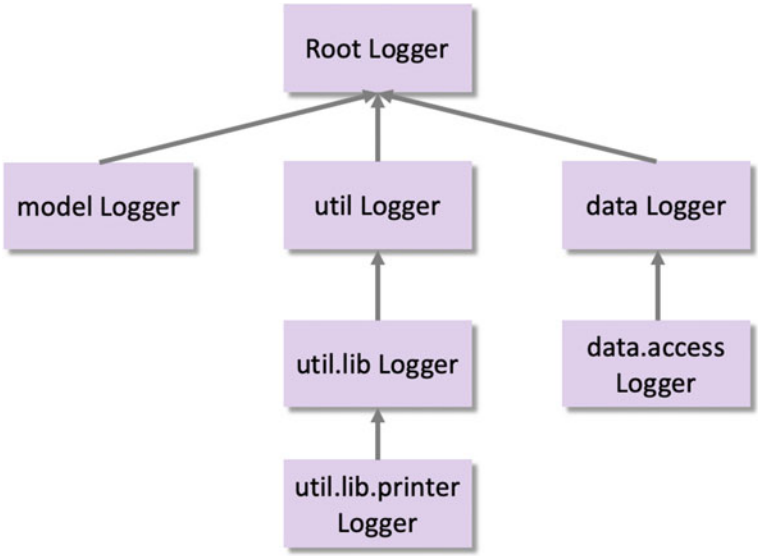
36.7 Logger Hierarchy

There is in fact a hierarchy of loggers with the root logger at the top of this hierarchy.

All *named* loggers are below the root logger.

The name of a logger can actually be a period-separated hierarchical value such as `util`, `util.lib` and `util.lib.printer`. Loggers that are further down the hierarchy are children of loggers further up the logger hierarchy.

For example given a logger called `lib`, then it will be below the `root` logger but above the logger with the name `util.lib`. This logger will in turn be above the logger called `util.lib.printer`. This is illustrated in the following diagram:



The logger name hierarchy is analogous to the Python package hierarchy and identical to it if you organise your loggers on a per-module basis using the recommended construction `logging.getLogger(__name__)`.

This hierarchy is important when considering the log level. If a log level has not been set for the current logger then it will look to its parent to see if that logger has a log level set. If it does that will be the log level used. This search back up the logger hierarchy will continue until either an explicit log level is found or the root logger is encountered which has a default log level of WARNING.

This is useful as it is not necessary to explicitly set the log level for every logger object used in an application. Instead it is only necessary to set the root log level (or for a module hierarchy an appropriate point in the module hierarchy). This can then be overridden where specifically required.

36.8 Formatters

The are two levels at which you can format the messages logged, and these are within the log message passed to a logging method (such as `info()` or `warn()`) and via the top-level configuration that indicates what additional information may be added to the individual log message.

36.8.1 Formatting Log Messages

The log message can have control characters that indicate what values should be placed within the message, for example:

```
logger.warning('%s is set to %d', 'count', 42)
```

This indicates that the format string expects to be given a string and a number. The parameters to be substituted into the format string follow the format string as a comma separated list of values.

36.8.2 Formatting Log Output

The logging pipeline can be configured to incorporate standard information with each log message. This can be done globally for all handlers. It is also possible to programmatically set a specific formatter on an individual handler; this is discussed in the next section.

To globally set the output format for log messages use the `logging.basicConfig()` function using the named parameter `format`.

The `format` parameter takes a string that can contain LogRecord attributes organised as you see fit. There is a comprehensive list of LogRecord attributes which can be referenced at https://docs.python.org/3/library/logging.html#logrecord-attributes. The key ones are:

- `args` a tuple listing the arguments used to call the associated function or method.
- `asctime` indicates the time that the log message was created.
- `filename` the name of the file containing the log statement.
- `module` the module name (the name portion of the filename).
- `funcName` the name of the function or method containing the log statement.
- `levelname` the log level of the log statement.
- `message` the log message itself as provided to the log method.

The effect of some of these is illustrated below.

```
import logging

logging.basicConfig(format='%(asctime)s %(message)s',
level=logging.DEBUG)

logger = logging.getLogger(__name__)

def do_something():
    logger.debug('This is to help with debugging')
    logger.info('This is just for information')
    logger.warning('This is a warning!')
    logger.error('This should be used with something unexpected')
    logger.critical('Something serious')
```

```
do_something()
```

The above program generates the following log statements:

```
2023-06-16 12:05:05,903 This is to help with debugging
2023-06-16 12:05:05,903 This is just for information
2023-06-16 12:05:05,903 This is a warning!
2023-06-16 12:05:05,903 This should be used with something unex-
pected
2023-06-16 12:05:05,903 Something serious
```

However, it might be useful to know the log level associated with the log statements, as well as the function that the log statements were called from. It is possible to obtain this information by changing the format string passed to the `logging.basicConfig()` function:

```
logging.basicConfig(format='%(asctime)s[%(levelname)s]
%(funcName)s: %(message)s', level=logging.DEBUG)
```

which will now generate the output within log level information and the function involved:

```
2023-06-16 12:05:24,307[DEBUG] do_something: This is to help with
debugging
2023-06-16 12:05:24,307[INFO] do_something: This is just for infor-
mation
2023-06-16 12:05:24,307[WARNING] do_something: This is a warning!
2023-06-16 12:05:24,307[ERROR] do_something: This should be used
with something unexpected
2023-06-16 12:05:24,307[CRITICAL] do_something: Something serious
```

We can even control the format of the date time information associated with the log statement using the `datafmt` parameter of the `logging.basicConfig()` function:

```
logging.basicConfig(format='%(asctime)s %(message)s',
datefmt='%m/%d/%Y %I:%M:%S %p', level=logging.DEBUG)
```

This format string uses the formatting options used by the `datetime.strptime()` function (see https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior) for information on the control characters, in this case.

- %m Month as a zero-padded decimal number, e.g. 01, 11, 12.
- %d Day of the month as a zero-padded decimal number, e.g. 01, 12, etc.
- %Y Year with century as a decimal number, e.g. 2020.
- %I Hour (12-h clock) as a zero-padded decimal number, e.g. 01, 10, etc.
- %M Minute as a zero-padded decimal number, e.g. 0, 01, 59, etc.
- %S Second as a zero-padded decimal number, e.g. 00, 01, 59, etc.
- %p Either AM or PM.

Thus the output generated using the above `datefmt` string is:

```
06/16/2023 12:05:43 PM This is to help with debugging
06/16/2023 12:05:43 PM This is just for information
06/16/2023 12:05:43 PM This is a warning!
06/16/2023 12:05:43 PM This should be used with something unexpected
06/16/2023 12:05:43 PM Something serious
```

To set a formatter on an individual handler see the next section.

36.9 Online Resources

For further information on the Python logging framework see the following:

- https://docs.python.org/3/library/logging.html The standard library documentation on the logging facilities in Python.
- https://docs.python.org/3/howto/logging.html A how to guide on logging from the Python standard library documentation.
- https://pymotw.com/3/logging/index.html Python Module of the Week logging page.

36.10 Exercises

This exercise will involve adding logging to the `Account` class you have been working on in this book.

You should add log methods to each of the methods in the class using either the `debug` or `info` methods. You should also obtain a module logger for the account classes.

Chapter 37
Advanced Logging

37.1 Introduction

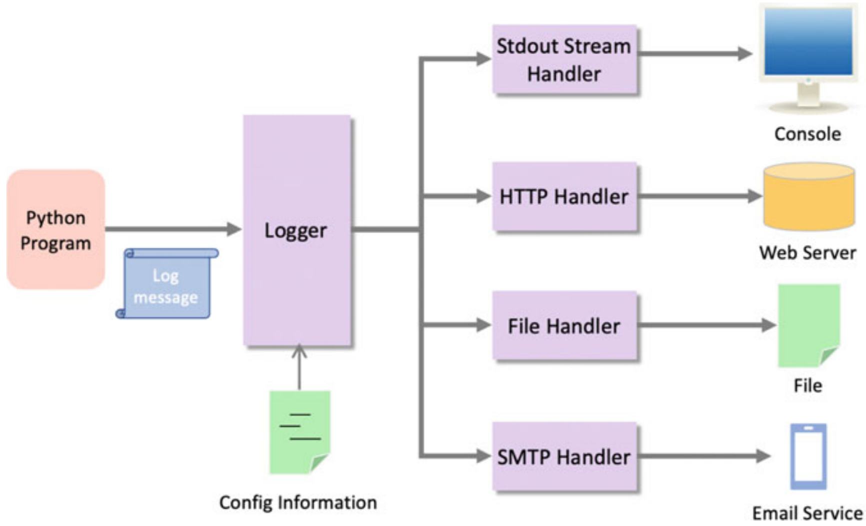
In this chapter we go further into the configuration and modification of the Python logging module. In particular we will look at Handlers (used to determine the destination of log messages), Filters which can be used by Handlers to provide finer grained control of log output and logger configuration files. We conclude the chapter by considering performance issues associated with logging.

37.2 Handlers

Within the logging pipeline, it is handlers that send the log message to their final destination.

By default the handler is set up to direct output to the console/terminal associated with the running program. However, this can be changed to send the log messages to a file, to an email service, to a web server, etc. Or indeed to any combination of these as there can be multiple handlers configured for a logger. This is shown in the diagram below:

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_37



In the above diagram the logger has been configured to send all log messages to four different handlers which allow a log message to be written to the console, to a web server to a file and to an email service. Such a behaviour may be required because:

- The web server will allow developers access to a web interface that allows them to see the log files even if they do not have permission to access a production server.
- The log file ensures that all the log data is permanently stored in a file within the file store.
- An email message may be sent to a notification system so that someone will be notified that there is an issue to be investigated.
- The console may still be available to the system administrators who may wish to look at the log messages generated.

The Python logging framework comes with several different handlers as suggested above and listed below:

- `logging.Stream Handler` sends messages to outputs such as `stdout`, `stderr`, etc.
- `logging.FileHandler` sends log messages to files. There are several varieties of File Handler in addition to the basic `FileHandler`; these include the `logging.handlers.RotatingFileHandler` (which will rotate log files based on a maximum file size) and `logging.handlers.TimeRotatingFileHandler` (which rotates the log file at specified time intervals, e.g. daily).
- `logging.handlers.SocketHandler` which sends messages to a TCP/IP socket where it can be received by a TCP server.

- `logging.handlers.SMTPHandler` that sends messages by the Simple Mail Transfer Protocol (SMTP) to a email server.
- `logging.handlers.SysLogHandler` that sends log messages to a Unix syslog program.
- `logging.handlers.NTEventLogHandler` that sends message to a Windows event log.
- `logging.handlers.HTTPHandler` which sends messages to a HTTP server.
- `logging.NullHandler` that does nothing with error messages. This is often used by library developers who want to include logging in their applications but expect developers to set up an appropriate handler when they use the library.

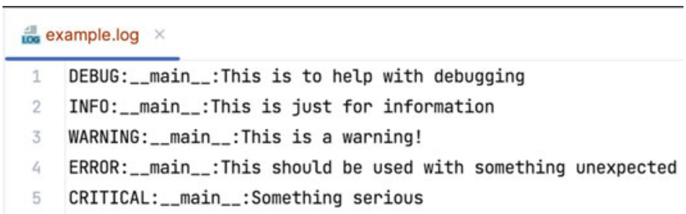
All of these handlers can be configured programmatically or via a configuration file.

37.2.1 Setting the Root Output Handler

The following example uses the `logging.basicConfig()` function to set up the root logger to use a `FileHandler` that will write the log messages to a file called ‘example.log’:

```
import logging

# Sets a file handler on the root logger to
# save log messages to the example.log file
logging.basicConfig(filename='example.log',level=logging.DEBUG)

# If no handler is explicitly set on the name logger
# it will delegate the messages to the parent logger to handle
logger = logging.getLogger(__name__)

logger.debug('This is to help with debugging')
logger.info('This is just for information')
logger.warning('This is a warning!')
logger.error('This should be used with something unexpected')
logger.critical('Something serious')
```

Note that if no handler is specified for a named logger then it delegates output to the parent (in this case the root) logger. The file generated for the above program is shown below:



As can be seen from this the default formatter is now configured for a `FileHandler`. This FileHandler adds the log message level before the log message itself.

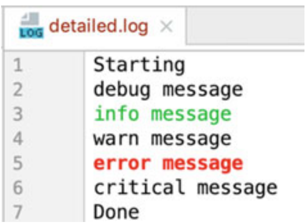
37.2.2 *Programmatically Setting the Handler*

It is also possible to programmatically create a handler and set it for the logger. This is done by instantiating one of the existing handler classes (or by subclassing an existing handler such as the root `Handler` class or the `FileHander`). The instantiated handler can then be added as a handler to the logger (remember the logger can have multiple handlers this is why the method is called `addHandler()` rather than something such as setHandler).

An example of explicitly setting the FileHandler for a logger is given below:

```
import logging

# Empty basic config turns off default console handler
logging.basicConfig()

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# create file handler which logs to the specified file
file_handler = logging.FileHandler('detailed.log')

# Add the handler to the Logger
logger.addHandler(file_handler)

# 'application' code
def do_something():
    logger.debug('debug message')
    logger.info('info message')
    logger.warning('warn message')
    logger.error('error message')
    logger.critical('critical message')

logger.info('Starting')
do_something()
logger.info('Done')
```

The result of running this code is that a log file is created with the logged messages:



Given that this is a lot more code than using the `basicConfig()` function; the question here might be 'Why bother?'. The answer is twofold:

- You can have different handlers for different loggers rather than setting the handler to be used centrally.
- Each handler can have its own format set so that logging to a file has a different format to logging to the console.

We can set the format for the handler by instantiating the `logging.Formatter` class with an appropriate format string. The formatter object can then be applied to a handler using the `setFormatter()` method on the handler object.

For example, we can modify the above code to include a formatter that is then set on the file handler as shown below.

```
# create file handler which logs to the specified file
file_handler = logging.FileHandler('detailed.log')

# Create formatter for the file_handler
formatter = logging.Formatter('%(asctime)s - %(funcName)s -
%(message)s')
file_handler.setFormatter(formatter)

logger.addHandler(file_handler)
```

The log file now generated is modified such that each message includes a timestamp, the function name (or module if at the module level) as well as the log message itself.

37.2.3 Multiple Handlers

As suggested in the previous section we can create multiple handlers to send log messages to different locations; for example from the console, to files and even email servers. The following program illustrates setting up both a *file handler* and a *console handler* for a module level logger.

To do this we create two handlers the file_handler and the console_handler. As a side effect we can also give them different log levels and different formatters. In this case the file_handler inherits the log level of the logger itself (which is DEBUG) while the console_handler has its log level set explicitly at WARNING. This means different amounts of information will be logged to the log file than the console output.

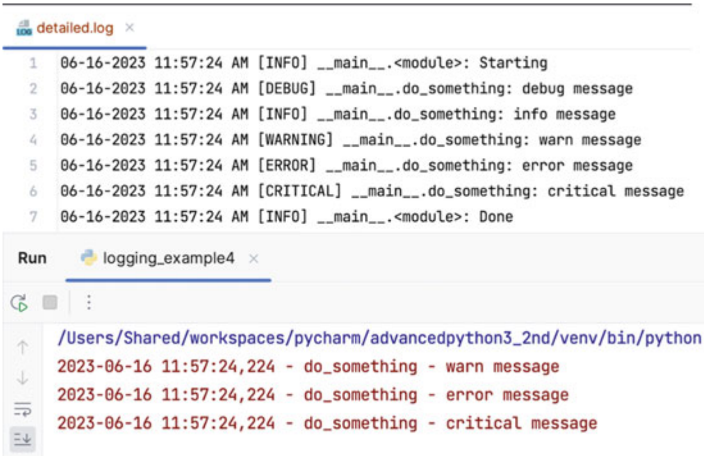
We have also set different formatters on each handler; in this case the log file handler's formatter provides more information than the console handlers formatter.

Both handlers are then added to the logger before it is used.

```
# Multiple Handlers and formatters
import logging

# Set up the default root logger to do nothing
logging.basicConfig(handlers=[logging.NullHandler()])

# Obtain the module level logger and set level to DEBUG
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Create file handler
file_handler = logging.FileHandler('detailed.log')

# Create console handler with a higher log level
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.WARNING)

# Create formatter for the file handler
fh_formatter = logging.Formatter('%(asctime)s [%(levelname)s]
%(name)s.%(funcName)s: %(message)s',
                              datefmt='%m-%d-%Y %I:%M:%S %p')
file_handler.setFormatter(fh_formatter)

# Create formatter for the console handler
console_formatter = logging.Formatter('%(asctime)s -
%(funcName)s - %(message)s')
console_handler.setFormatter(console_formatter)

# Add the handlers to logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)

# 'application' code
def do_something():
    logger.debug('debug message')
    logger.info('info message')
    logger.warning('warn message')
    logger.error('error message')
    logger.critical('critical message')

logger.info('Starting')
do_something()
```

```
logger.info('Done')
```

The output from this program is now split between the log file and the console out, as shown below:



37.3 Filters

Filters can be used by Handlers to provide finer grained control of the log output. A filter can be added to a logger using the `logger.addFilter()` method. A Filter can be created by extending the `logging.Filter` class and implementing the `filter()` method. This method takes a log record. This log record can be validated to determine if the record should be output or not. If it should be output then `True` is returned, if the record should be ignored `False` should be returned.

In the following example, a filter called `MyFilter` is defined that will filter out all log messages containing the string 'John'. It is added as a filter to the logger and then two log messages are generated.

```
import logging

class MyFilter(logging.Filter):

    def filter(self, record):
        if 'John' in record.msg:
            return False
        else:
            return True

logging.basicConfig(format='%(asctime)s %(message)s',
level=logging.DEBUG)
```

```
logger = logging.getLogger()
logger.addFilter(MyFilter())

logger.debug('This is to help with debugging')
logger.info('This is information on John')
```

The output shows that only the log message that does not contain the string 'John' is output:

```
2023-06-16 11:59:48,226 This is to help with debugging
```

37.4 Logger Configuration

All the examples so far in this chapter have used programmatic configuration of the logging framework. This is certainly feasible as the examples show, but it does require a code change if you wish to alter the logging level for any particular logger or to change where a particular handler is routing the log messages.

For most production systems a better solution is to use an external configuration file which is loaded when the application is run and is used to dynamically configure the logging framework. This allows system administrators and others to change the log level, the log destination, the log format, etc. without needing to change the code.

The logging configuration file can be written using several standard formats from the Java Script Object Notation (JSON), to Yet Another Markup Language (YAML) format, or as a set of key-value pairs in a `.conf` file. For further information on the different options available see the Python `logging` module documentation.

In this book we will briefly explore the YAML file format used to configure loggers.

```
version: 1
formatters:
 myformatter:
    format: '%(asctime)s [%(levelname)s] %(name)s.%(funcName)s:
%(message)s'
handlers:
 console:
   class: logging.StreamHandler
   level: DEBUG
   formatter: myformatter
   stream: ext://sys.stdout
loggers:
 myLogger:
level: DEBUG
handlers: [console]
propagate: no
root:
level: ERROR
handlers: [console]
```

The above YAML code is stored in a file called `logging.conf.yaml;` however you can call this file anything that is meaningful.

The YAML file always starts with a version number. This is an integer value representing the YAML schema version (currently this can only be the value 1). All other keys in the file are optional; they include:

- *formatters*—this lists one or more formatters; each formatter has a name which acts as a key and then a format value which is a string defining the format of a log message.
- *filters*—this is a lit of filter names and a set of filter definitions.
- *handlers*—this is a list of named handlers. Each handler definition is made up of a set of key-value pairs where the keys define the class used for the filter (mandatory), the log level of the filter (optional), the formatter to use with the handler (optional) and a list of filters to apply (optional).
- *loggers* provide one or more named loggers. Each logger can indicate the log level (optional) and a list of handlers (optional). The `propagate` option can be used to stop messages propagating to a parent logger (by setting it to `False`).
- *root* this is the configuration for the root logger.

This file can be loaded into a Python application using the `PyYAML` module. This module can be installed using pip:

```
pip install PyYAML
```

The version used at the time of writing is version 6.0.

This provides a YAML parser that can load a YAML file as a dictionary structure that can be passed to the `logging.config.dictConfig()` function. As this is a file it must be opened and closed to ensure that the resource is handled appropriately; it is therefore best managed using the `with-as` statement as shown below:

```
with open('logging.config.yaml', 'r') as f:
    config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)
```

This will open the YAML file in read-only mode and close it when the two statements have been executed. This snippet is used in the following application that loads the logger configuration from the YAML file:

```
import logging
import logging.config
import yaml

with open('logging.config.yaml', 'r') as f:
   config = yaml.safe_load(f.read())
   logging.config.dictConfig(config)

logger = logging.getLogger('myLogger')

# 'application' code
def do_something():
   logger.debug('debug message')
```

```
    logger.info('info message')
    logger.warning('warn message')
    logger.error('error message')
    logger.critical('critical message')

logger.info('Starting')
do_something()
logger.info('Done')
```

The output from this using the earlier YAML file is:

```
2023-06-16 12:02:23,863 [INFO] myLogger.<module>: Starting
2023-06-16 12:02:23,863 [DEBUG] myLogger.do_something: debug
message
2023-06-16 12:02:23,863 [INFO] myLogger.do_something: info
message
2023-06-16 12:02:23,863 [WARNING] myLogger.do_something: warn
message
2023-06-16 12:02:23,863 [ERROR] myLogger.do_something: error
message
2023-06-16 12:02:23,863 [CRITICAL] myLogger.do_something: crit-
ical message
2023-06-16 12:02:23,863 [INFO] myLogger.<module>: Done
```

37.5 Performance Considerations

Performance when logging should always be a consideration. In general you should aim to avoid performing any unnecessary work when logging is disabled (or disabled for the level being used). This may seem obvious but it can occur in several unexpected ways.

One example is string concatenation. If a message to be logged involves string concatenation; then that string concatenation will always be performed when a log method is being invoked. For example:

```
logger.debug('Count: ' + count + ', total: ' + total)
```

This will always result in the string being generated for `count` and `total` before the call is made to the debug function; even if the debug level is not turned on. However using a format string will avoid this. The formatting involved will only be performed if the string is to be used in a log message. You should therefore always use string formatting to populate log messages. For example:

```
logger.debug('Count: %d, total: %d', count, 42)
```

Another potential optimisation is to use the `logger.isEnabledFor(level)` method as a guard against running the log statement. This can be useful in situations where an associated operation must be performed to support the logging operation, and this operation is expensive. For example:

```
if logger.isEnabledFor(logging.DEBUG):
logger.debug('Message with %s, %s', expensive_func1(),
expensive_func2())
```

Now the two expensive functions will only be executed if the DEBUG log level is set.

37.6 Exercises

Using the logging you added to the Account class in the last chapter, you should load the log configuration information from a YAML file similar to that used in this chapter.

This should be loaded into the application program used to drive the account classes.

Part VIII
Concurrency and Parallelism

Chapter 38
Introduction to Concurrency and Parallelism

38.1 Introduction

In this chapter we will introduce the concepts of concurrency and parallelism. We will also briefly consider the related topic of distribution. After this we will consider process synchronisation, why object-oriented approaches are well suited to concurrency and parallelism before finishing with a short discussion of threads versus processes.

38.2 Concurrency

Concurrency is defined by the dictionary as.

> two or more events or circumstances happening or existing at the same time.

In Computer Science concurrency refers to the ability of different parts or units of a program, algorithm or problem to be *executed at the same time*, potentially on multiple processors or multiple cores.

Here a processor refers to the central processing unit (or CPU) or a computer while core refers to the idea that a CPU chip can have multiple cores or processors on it.

Originally a CPU chip had a single core. That is the CPU chip had a single processing unit on it. However, over time, to increase computer performance, hardware manufacturers added additional *cores* or processing units to chips. Thus a dual-core CPU chip has two processing units while a quad-core CPU chip has four processing units. This means that as far as the operating system of the computer is concerned, it has multiple CPUs on which it can run programs.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_38

Running processing at the same time, on multiple CPUs, can substantially improve the overall performance of an application.

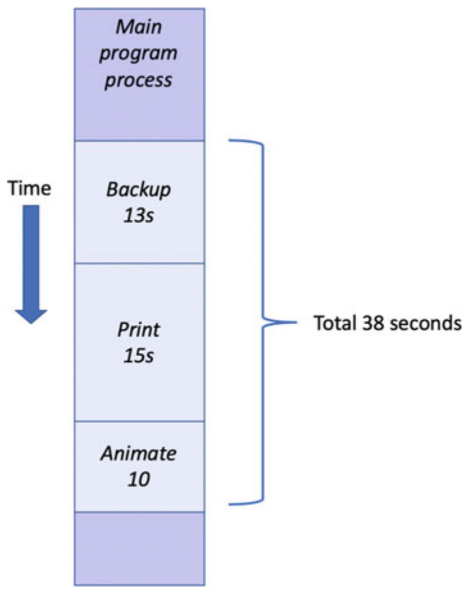
For example, let us assume that we have a program that will call three independent functions; these functions are:

- make a backup of the current data held by the program,
- print the data currently held by the program,
- run an animation using the current data.

Let us assume that these functions run sequentially, with the following timings:

- the *backup* function takes 13 s,
- the *print* function takes 15 s,
- the *animation* function takes 10 s.

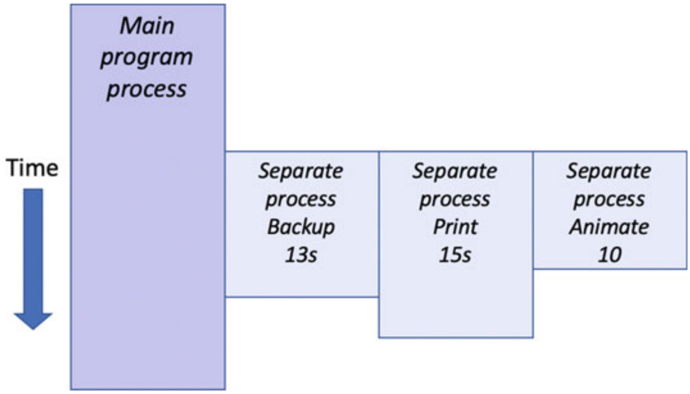
This would result in a total of 38 s to perform all three operations. This is illustrated graphically below:



However, the three functions are all completely *independent* of each other. That is they do not rely on each other for any results or behaviour; they do not need one of the other functions to complete before they can complete, etc. Thus we can run each function *concurrently*.

If the underlying operating system and program language being used support multiple processes, then we can potentially run each function in a separate process at the same time and obtain a significant speed up in overall execution time.

If the application starts all three functions at the same time, then the maximum time before the main process can continue will be 15 s, as that is the time taken by the longest function to execute. However, the main program may be able to continue as soon as all three functions are started as it also does not depend on the results from any of the functions; thus the delay may be negligible (although there will typically be some small delay as each process is set up). This is shown graphically below:



38.3 Parallelism

A distinction its often made in Computer Science between concurrency and *parallelism*.

In concurrency, separate independent tasks are performed potentially at the same time.

In parallelism, a large complex task is broken down into a set of *subtasks*. The subtasks represent part of the overall problem. Each subtask can be executed at the same time. Typically it is necessary to combine the results of the subtasks together to generate an overall result. These subtasks are also very similar if not functionally exactly the same (although in general each subtask invocation will have been supplied with different data).

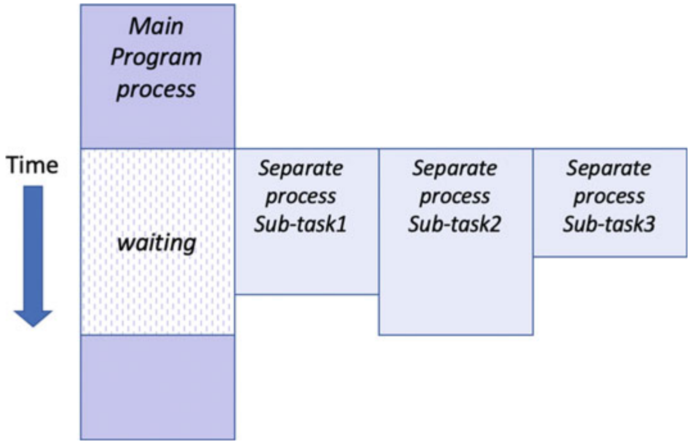
Thus parallelism is when *multiple copies* of the same functionality are run at the same time, but on different data.

Some examples of where parallelism can be applied include:

- A web search engine. Such a system may look at many, many web pages. Each time it does so it must send a request to the appropriate web site, receive the result and process the data obtained. These steps are the same whether it is the BBC

web site, Microsoft's web site or the web site of Cambridge University. Thus the requests can be run sequentially or in parallel.
- Image processing. A large image may be broken down into slices so that each slice can be analysed in parallel.

The following diagram illustrates the basic idea behind parallelism; a main program fires off three subtasks each of which runs in parallel. The main program then waits for all the subtasks to complete before combining together the results from the subtasks before it can continue.



38.4 Distribution

When implementing a concurrent or parallel solution, where the resulting processes run is typically an implementation detail. Conceptually these processes could run on the same processor, physical machine or on a remote or distributed machine. As such distribution, in which problems are solved or processes executed by sharing the work across multiple physical machines, is often related to concurrency and parallelism.

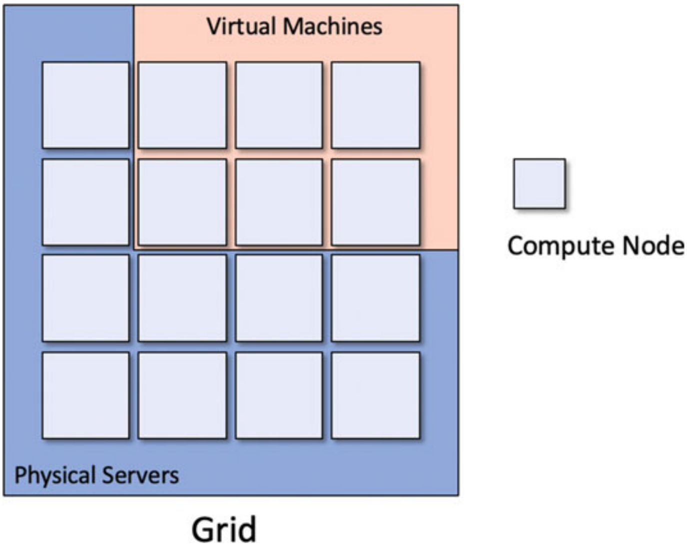
However, there is no requirement to distribute work across physical machines, indeed in doing so extra work is usually involved.

To distribute work to a remote machine, data and in many cases code, must be transferred and made available to the remote machine. This can result in significant delays in running the code remotely and may offset any potential performance advantages of using a physically separate computer. As a result many concurrent/parallel technologies default to executing code in a separate process on the same machine.

38.5 Grid Computing

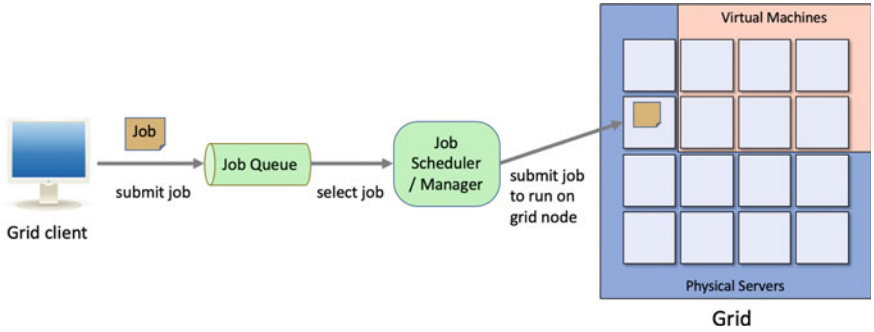
Grid Computing is based on the use of a network of loosely coupled computers, in which each computer can have a job submitted to it, which it will run to completion before returning a result.

In many cases the grid is made up of *heterogeneous* set of computers (rather than all computers being the same) and may be geographically dispersed. These computers may be comprised of both physical computers and virtual machines.



A *Virtual Machine* is a piece of software that emulates a whole computer and runs on some underlying hardware that is shared with other virtual machines. Each virtual machine thinks it is the only computer on the hardware; however the virtual machines all share the resources of the physical computer. Multiple virtual machines can thus run simultaneously on the same physical computer. Each virtual machine provides its own virtual hardware, including CPUs, memory, hard drives, network interfaces and other devices. The virtual hardware is then mapped to the real hardware on the physical machine which saves costs by reducing the need for physical hardware systems along with the associated maintenance costs, as well as reducing the power and cooling demands of multiple computers.

Within a grid, software is used to manage the grid nodes and to submit jobs to those nodes. Such software will receive the jobs to perform (programs to run and information about the environment such as libraries to use) from clients of the grid. These jobs are typically added to a job queue before a job scheduler submits them to a node within the grid. When any results are generated by the job they are collected from the node and returned to the client. This is illustrated below:



The use of grids can make distributing concurrent/parallel processes amongst a set of physical and virtual machines much easier.

38.6 Concurrency and Synchronisation

Concurrency relates to executing multiple tasks at the same time. In many cases these tasks are not related to each other such as printing a document and refreshing the user interface. In these cases, the separate tasks are completely independent and can execute at the same time without any interaction.

In other situations multiple concurrent tasks need to interact; for example, where one or more tasks produce data and one or more other tasks consume that data. This is often referred to as a *producer–consumer* relationship. In other situations, all parallel processes must have reached the same point before some other behaviour is executed.

Another situation that can occur is where we want to ensure that only one concurrent task executes a piece of sensitive code at a time; this code must therefore be protected from concurrent access.

Concurrent and parallel libraries need to provide facilities that allow for such synchronisation to occur.

38.7 Object Orientation and Concurrency

The concepts behind object-oriented programming lend themselves particularly well to the concepts associated with concurrency. For example, a system can be described as a set of discrete objects communicating with one another when necessary. In Python, only one object may execute at any one moment in time within a single interpreter. However, conceptually at least, there is no reason why this restriction should be enforced. The basic concepts behind object orientation still hold, even if each object executes within a separate independent process.

Traditionally a message send is treated like a procedural call, in which the calling object's execution is blocked until a response is returned. However, we can extend this model quite simply to view each object as a concurrently executable program, with activity starting when the object is created and continuing even when a message is sent to another object (unless the response is required for further processing). In this model, there may be very many (concurrent) objects executing at the same time. Of course, this introduces issues associated with resource allocation, etc. but no more so than in any concurrent system.

One implication of the concurrent object model is that objects are larger than in the traditional single execution thread approach, because of the overhead of having each object as a separate thread of execution. Overheads such as the need for a scheduler to handling these execution threads and resource allocation mechanisms mean that it is not feasible to have integers, characters, etc. as separate processes.

38.8 Threads V Processes

As part of this discussion it is useful to understand what is meant by a process. A process is an instance of a computer program that is being executed by the operating system. Any process has three key elements; the program being executed, the data used by that program (such as the variables used by the program) and the state of the process (also known as the execution context of the program).

A (Python) thread is a preemptive lightweight process.

A thread is considered to be *pre-emptive* because every thread has a chance to run as the main thread at some point. When a thread gets to execute then it will execute until.

- completion,
- until it is waiting for some form of I/O (Input/Output),
- sleeps for a period of time,
- it has run for 15 ms (the current threshold in Python 3).

If the thread has not completed when one of the above situations occurs, then it will give up being the executing thread and another thread will be run instead. This means that one thread can be interrupted in the middle of performing a series of related steps.

A thread is a considered a *lightweight* process because it does not possess its own address space and it is not treated as a separate entity by the host operating system. Instead, it exists within a single machine process using the same address space.

It is useful to get a clear idea of the difference between a thread (running within a single machine process) and a multiprocess system that uses separate processes on the underlying hardware.

38.9 Some Terminology

The world of concurrent programming is full of terminology that you may not be familiar with. Some of those terms and concepts are outlined below:

- **Asynchronous versus Synchronous invocations**. Most of the method, function or procedure invocations you will have seen in programming represent synchronous invocations. A synchronous method or function call is one which blocks the calling code from executing until it returns. Such calls are typically within a single thread of execution. Asynchronous calls are ones where the flow of control immediately returns to the callee, and the caller is able to execute in its own thread of execution. Allowing both the caller and the call to continue processing.
- **Non-blocking versus Blocking code**. Blocking code is a term used to describe the code running in one thread of execution, waiting for some activity to complete which causes one of more separate threads of execution to also be delayed. For example, if one thread is the producer of some data and other threads are the consumers of that data, then the consumer threads cannot continue until the producer generates the data for them to consume. In contrast, non-blocking means that no thread is able to indefinitely delay others.
- **Concurrent versus Parallel code**. Concurrent code and parallel code are similar, but different in one significant aspect. Concurrency indicates that two or more activities are both making progress even though they might not be executing at the same point in time. This is typically achieved by continuously swapping competing processes between execution and non-execution. This process is repeated until at least one of the threads of execution (Threads) has completed their task. This may occur because two threads are sharing the same physical processor with each is being given a short time period in which to progress before the other gets a short time period to progress. The two threads are said to be sharing the processing time using a technique known as time slicing. Parallelism on the other hand implies that there are multiple processors available allowing each thread to execute on their own processor simultaneously.

38.10 Online Resources

- See the following online resources for information on the topics in this chapter:
- https://en.wikipedia.org/wiki/Concurrency_(computer_science). Wikipedia page on concurrency.
- https://en.wikipedia.org/wiki/Virtual_machine. Wikipedia page on Virtual Machines.
- https://en.wikipedia.org/wiki/Parallel_computing. Wikipedia page on parallelism.

- http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html.Concurrency versus Parallelism tutorial.
- https://www.redbooks.ibm.com/redbooks/pdfs/sg246778.pdf. IBM Red Book on an Introduction to Grid Computing.

Chapter 39
Threading

39.1 Introduction

Threading is one of the ways in which Python allows you to write programs that multitask; that is *appearing* to do more than one thing at a time.

This chapter presents the threading module and uses a short example to illustrate how these features can be used.

39.2 Threads

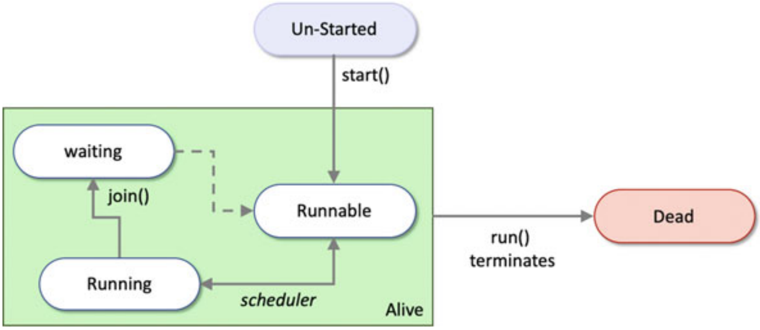
In Python the Thread class from the threading module represents an activity that is run in a separate thread of execution within a single process. These threads of execution are lightweight, pre-emptive execution threads. A thread is a *lightweight* process because it does not possess its own address space, and it is not treated as a separate entity by the host operating system; it is not a process. Instead, it exists within a single machine process using the same address space as other threads.

39.2.1 Thread States

When a thread object is first created it exists, but it is not yet schedulable; it must be started. Once it has been started it is then runnable; that is, it is eligible to be scheduled for execution. It may switch back and forth between running and being runnable under the control of the scheduler. The scheduler is responsible for managing multiple threads that all wish to grab some execution time.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_39

A thread object remains runnable or running until its `run()` method terminates; at which point it has finished its execution and it is now dead. All states between unstarted and dead are considered to indicate that the thread is alive (and therefore may run at some point). This is shown below:



A thread may also be in the waiting state; for example, when it is waiting for another thread to finish its work before continuing (possibly because it needs the results produced by that thread to continue). This can be achieved using the `join()` method and is also illustrated above. Once the second thread completes the waiting thread will again become runnable.

The thread which is currently executing is termed the active thread.

There are a few points to note about thread states:

- A thread is considered to be alive unless its `run()` method terminates after which it can be considered dead.
- A live thread can be running, runnable, waiting, etc.
- The `runnable` state indicates that the thread can be executed by the processor, but it is not currently executing. This is because an equal or higher priority process is already executing, and the thread must wait until the processor becomes free. Thus the diagram shows that the scheduler can move a thread between the running and runnable state. In fact, this could happen many times as the thread executes for a while, is then removed from the processor by the scheduler and added to the waiting queue, before being returned to the processor again at a later date.

39.2.2 Creating a Thread

There are two ways in which to initiate a new thread of execution:

- Pass a reference to a callable object (such as a function or method) into the Thread class constructor. This reference acts as the target for the Thread to execute.

- Create a subclass of the Thread class and redefine the `run()` method to perform the set of actions that the thread is intended to do.

We will look at both approaches.

As a thread is an object, it can be treated just like any other object: it can be sent messages, it can have instance variables and it can provide methods. Thus, the multithreaded aspects of Python all conform to the object-oriented model. This greatly simplifies the creation of multithreaded systems as well as the maintainability and clarity of the resulting software.

Once a new instance of a thread is created, it must be started. Before it is started, it cannot run, although it exists.

39.2.3 Instantiating the Thread Class

The `Thread` class can be found in the `threading` module and therefore must be imported prior to use. The class `Thread` defines a single constructor that takes up to six optional arguments:

```
class threading.Thread(group=None,
                       target=None,
                       name=None,
                       args=(),
                       kwargs={},
                       daemon=None)
```

The Thread constructor should always be called using *keyword* arguments; the meaning of these arguments is:

- `group` should be None; reserved for future extension when a ThreadGroup class is implemented.
- `target` is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called.
- `name` is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is an integer.
- `args` is the argument tuple for the target invocation. Defaults to `()`. If a single argument is provided the tuple is not required. If multiple arguments are provided then each argument is an element within the tuple.
- `kwargs` is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.
- `daemon` indicates whether this thread runs as a daemon thread or not. If not `None`, daemon explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.

Once a `Thread` is created it must be started to become eligible for execution using the `Thread.start()` method.

The following illustrates a very simple program that creates a `Thread` that will run the `simple_worker()` function:

```
from threading import Thread

def simple_worker():
    print('hello')

# Create a new thread and start it
# The thread will run the function simple_worker
t1 = Thread(target=simple_worker)
t1.start()
```

In this example, the thread `t1` will execute the function `simple_worker`. The main code will be executed by the *main* thread that is present when the program starts; there are thus two threads used in the above program; *main* and *t1*.

39.3 The Thread Class

The `Thread` class defines all the facilities required to create an object that can execute within its own lightweight process. The key methods are:

- `start()` Start the thread's activity. It must be called at *most* once per thread object. It arranges for the object's `run()` method to be invoked in a separate thread of control. This method will raise a `RuntimeError` if called more than once on the same thread object.
- `run()` Method representing the thread's activity. You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with positional and keyword arguments taken from the `args` and `kwargs` arguments, respectively. You should *not* call this method directly.
- `join(timeout = None)` Wait until the thread sent this message terminates. This blocks the calling thread until the thread whose `join()` method is called terminates. When the timeout argument is present and not None, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof). A thread can be join()ed many times.
- `name` A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor. Giving a thread a name can be useful for debugging purposes.
- `ident` The 'thread identifier' of this thread or `None` if the thread has not been started. This is a nonzero integer.
- `is_alive()` Return whether the thread is alive. This method returns `True` just before the `run()` method starts until just after the `run()` method terminates. The module function `threading.enumerate()` returns a list of all alive threads.

- `daemon` A Boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before `start()` is called, otherwise a `RuntimeError` is raised. Its default value is inherited from the creating thread. The entire Python program exits when no alive non-daemon threads are left.

  An example illustrating using some of these methods is given below:

```
from threading import Thread

def simple_worker():
    print('hello')

t1 = Thread(target=simple_worker)
t1.start()

print(t1.name)
print(t1.ident)
print(t1.is_alive())
```

  This produces:

```
Hello
Thread-1 (simple_worker)
123145398558720
True
```

The `join()` method can cause one thread to wait for another to complete. For example, if we want the *main* thread to wait until a thread completes before it prints the done message; then we can make it *join* that thread:

```
from threading import Thread
from time import sleep

def worker():
    for i in range(0, 10):
        print('.', end='', flush=True)
        sleep(1)

print('Starting')
# Create read object with reference to worker function
t = Thread(target=worker)
# Start the thread object

t.start()
# Wait for the thread to complete

t.join()

print('\nDone')
```

Now the 'Done' message should not be printed out until after the worker thread has finished as shown below:

```
Starting
..........
Done
```

39.4 The Threading Module Functions

There are a set of `threading` module functions which support working with threads; these functions include:

- `threading.active_count()` Return the number of Thread objects currently alive. The returned count is equal to the length of the list returned by `enumerate()`.
- `threading.current_thread()` Return the current Thread object, corresponding to the caller's thread of control. If the caller's thread of control was not created through the `threading` module, a dummy thread object with limited functionality is returned.
- `threading.get_ident()` Return the 'thread identifier' of the current thread. This is a nonzero integer. Thread identifiers may be recycled when a thread exits and another thread is created.
- `threading.enumerate()` Return a list of all Thread objects currently alive. The list includes daemon threads, dummy thread objects created by `current_thread()` and the *main* thread. It excludes terminated threads and threads that have not yet been started.
- `threading.main_thread()` Return the *main* Thread object.

39.5 Passing Arguments to a Thread

Many functions expect to be given a set of parameter values when they are run; these arguments still need to be passed to the function when they are run via a separate thread. These parameters can be passed to the function to be executed via the `args` parameter, for example:

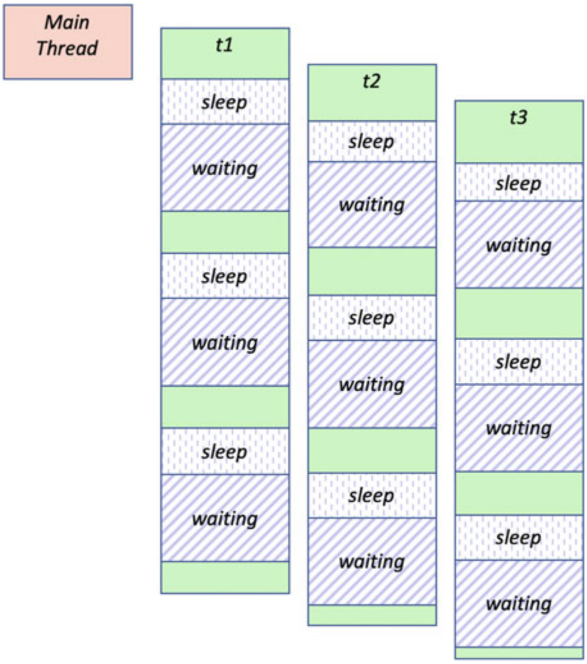
```
from threading import Thread
from time import sleep

def worker(msg):
     for i in range(0, 10):
          print(msg, end='', flush=True)
          sleep(1)

print('Starting')
t1 = Thread(target=worker, args='A')
t2 = Thread(target=worker, args='B')
t3 = Thread(target=worker, args='C')
t1.start()
t2.start()
t3.start()

print('Done')
```

In this example, the worker function takes a message to be printed 10 times within a loop. Inside the loop the thread will print the message and then sleep for a second.

This allows other threads to be executed as the thread must wait for the sleep timeout to finish before again becoming runnable.

Three threads t1, t2 and t3 are then created each with a different message. Note that the `worker()` function can be reused with each thread as each invocation of the function will have its own parameter values passed to it.

The three threads are then started. This means that at this point there is the *main* thread, and three worker threads that are runnable (although only one thread will run at a time). The three worker threads each run the `worker()` function printing out either the letter A, B or C ten times. This means that once started each thread will print out a string, sleep for 1 s and then wait until it is selected to run again, this is illustrated in the following diagram:



The output generated by this program is illustrated below:

```
Starting
ABCDone
BACBACBCABCABACCBACABCABCAB
```

Notice that the *main* thread is finished after the worker threads have only printed out a single letter each; however as long as there is at least one *non-daemon* thread running the program will not terminate; as none of these threads are marked as a daemon thread the program continues until the last thread has finished printing out the tenth of its letters.

Also notice how each of the threads gets a chance to run on the processor before it sleeps again; thus we can see the letters A, B and C all mixed in together.

39.6 Extending the Thread Class

The second approach to creating a `Thread` mentioned earlier was to subclass the `Thread` class. To do this you must.

1. Define a new subclass of `Thread`.
2. Override the `run()` method.
3. Define a new `__init__()` method that calls the parent class `__init__()` method to pass the required parameters up to the `Thread` class constructor.

This is illustrated below where the `WorkerThread` class passes the `name`, `target` and `daemon` parameters up to the Thread super class constructor.

```
from threading import Thread
from time import sleep

class WorkerThread(Thread):
    def __init__(self, daemon=None, target=None, name=None):
        super().__init__(daemon=daemon, target=target,
                         name=name)
    def run(self):
        for i in range(0, 10):
            print('.', end='', flush=True)
            sleep(1)
```

Once you have done this you can create an instance of the new `WorkerThread` class and then start that instance.

```
print('Starting')
t = WorkerThread()
t.start()
print('\nDone')
```

The output from this is:

```
Starting
.
Done
.........
```

Note that it is common to call any subclasses of the `Thread` class, *`Something`*`Thread`, to make it clear that it is a subclass of the `Thread` class and should be treated as if it was a Thread (which of course it is).

39.7 Daemon Threads

A thread can be marked as a *daemon* thread by setting the daemon property to true either in the constructor or latter via the accessor property.

For example:

```
from threading import Thread
from time import sleep

def worker(msg):
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)

print('Starting')

# Create a daemon thread
d = Thread(daemon=True, target=worker, args='C')
d.start()

sleep(5)
print('Done')
```

This creates a background daemon thread that will run the function `worker()`. Such threads are often used for housekeeping tasks (such as background data backups, etc.).

As mentioned above a daemon thread is not enough on its own to keep the current program from terminating. This means that the daemon thread will keep looping until the main thread finishes. As the main thread sleeps for 5 s that allows the daemon thread to print out about 5 strings before the main thread terminates. This is illustrated by the output below:

```
Starting
CCCCCDone
```

39.8 Naming Threads

Threads can be named, which can be very useful when debugging an application with multiple threads.

In the following example, three threads have been created; two have been explicitly given a name related to what they are doing while the middle one has been left with the default name. We then start all three threads and use the `threading.enumerate()` function to loop through all the currently live threads printing out their names:

```
import threading
from threading import Thread
from time import sleep

def worker(msg):
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)

t1 = Thread(name='worker', target=worker, args='A')
t2 = Thread(target=worker, args='B') # use default name e.g.
Thread-1
d = Thread(daemon = True, name='daemon', target=worker, args='C')

t1.start()
t2.start()
d.start()

print()
for t in threading.enumerate():
print(t.name)
```

The output from this program is given below:

```
ABC
MainThread
worker
Thread-1 (worker)
daemon
ACBACBACBACBACBABCACBCABABC
```

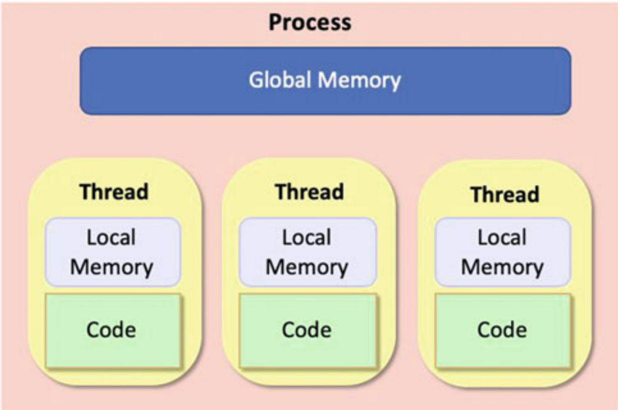
As you can see in addition to the worker thread and the daemon thread there is a `MainThread` (that initiates the whole program) and `Thread-1` which is the thread referenced by the variable `t2` and uses the default thread name.

39.9 Thread Local Data

In some situations each Thread requires its own copy of the data it is working with; this means that the shared (heap) memory is difficult to use as it is inherently shared between all threads.

To overcome this Python provides a concept known as *Thread Local* data.

Thread local data is data whose values are associated with a thread rather than with the shared memory. This idea is illustrated below:



To create thread local data it is only necessary to create an instance of `threading.local` (or a subclass of this) and store attributes into it. The instances will be thread specific, meaning that one thread will not see the values stored by another thread.

For example:

```
from threading import Thread, local, current_thread
from random import randint

def show_value(data):
    try:
        val = data.value
    except AttributeError:
        print(current_thread().name, '- No value yet')
    else:
        print(current_thread().name, '- value =', val)

def worker(data):
     show_value(data)
     data.value = randint(1, 100)
     show_value(data)

print(current_thread().name, '- Starting')

# Create thread local data object
local_data = local()
show_value(local_data)

for i in range(2):
    t = Thread(name='W'+ str(i),
               target=worker, args=[local_data])
```

```
    t.start()

show_value(local_data)
print(current_thread().name, '- Done')
```

The output from this is.

```
MainThread          - Starting
MainThread          - No value yet
W0    - No value yet
W0    - value = 43
W1    - No value yet
W1    - value = 75
MainThread - No value yet
MainThread - Done
```

The example presented above defines two functions.

- The first function attempts to access a value in the thread local data object. If the value is not present an exception is raised (AttributeError). The show_value() function catches the exception or successfully processes the data.
- The worker function calls show_value() twice, once before it sets a value in the local data object and once after. As this function will be run by separate threads, the current_thread name is printed by the show_value() function.

The main function creates a local data object using the local() function from the threading library. It then calls show_value() itself. Next it creates two threads to execute the worker function in passing the local_data object into them; each thread is then started. Finally, it calls show_value() again.

As can be seen from the output one thread cannot see the data set by another thread in the local_data object (even when the attribute name is the same).

39.10 Timers

The Timer class represents an action (or task) to run after a certain amount of time has elapsed. The Timer class is a subclass of Thread and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their `start()` method. The timer can be stopped (before its action has begun) by calling the `cancel()` method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user as another thread may be running when the timer wishes to start.

The signature of the `Timer` class constructor is:

```
Timer(interval, function, args=None, kwargs=None)
```

An example of using the `Timer` class is given below:

```
from threading import Timer

def hello():
    print('hello')

print('Starting')
t = Timer(5, hello)
t.start()

print('Done')
```

In this case the `Timer` will run the hello function after an initial delay of 5 s. The output is:

```
Starting
Done
hello
```

However, when you run this you will notice that the hello is printed some 5 s after the 'Done' string is printed.

39.11 The Global Interpreter Lock

The Global Interpreter Lock (or the GIL) is a global lock within the underlying CPython interpreter that was designed to avoid potential deadlocks between multiple tasks. It is designed to protect access to Python objects by preventing multiple threads from executing at the same time.

For the most part you do not need to worry about the GIL as it is at a lower level than the programs you will be writing.

However, it is worth noting that the GIL is controversial because it prevents multithreaded Python programs from taking full advantage of multiprocessor systems in certain situations.

This is because in order to execute a thread must obtain the GIL and only one thread at a time can hold the GIL (that is the lock it represents). This means that Python acts like a single CPU machine; only one thing can run at a time. A Thread will only give up the GIL if it sleeps, has to wait for something (such as some I/O) or it has held the GIL for a certain amount of time. If the maximum time that a thread can hold the GIL has been met the scheduler will release the GIL from that thread (resulting it stopping execution and now having to wait until it has the GIL returned to it) and will select another thread to gain the GIL and start to execute.

It is thus impossible for standard Python threads to take advantage of the multiple CPUs typically available on modern computer hardware.

One solution to this is to use the Python `multiprocessing` library described in the next chapter.

39.12 Online Resources

See the following online resources for information on the topics in this chapter:

- https://docs.python.org/3/library/threading.html. The Python standard Library documentation on Threading.
- https://pymotw.com/3/threading/. The Python Module of the Week page on Threading.
- https://pythonprogramming.net/threading-tutorial-python/. Tutorial on Python's Threading module.

39.13 Exercise

Create a function called `printer()` that takes a message and a maximum value to use for a period to sleep.

Within the function create a loop that iterates 10 times. Within the loop.

- generate a random number from 0 to the max period specified and then sleep for that period of time. You can use the `random.randint()` function for this.
- Once the sleep period has finished print out the message passed into the function.
- Then loop again until this has been repeated 10 times.

Now create five threads to run five invocations of the function you produced above and start all five threads. Each thread should have a different `max_sleep` time.

An example program to run the printer function five times via a set of Threads is given below:

```
t1 = Thread(target=printer, args=('A', 10))
t2 = Thread(target=printer, args=('B', 5))
t3 = Thread(target=printer, args=('C', 15))
t4 = Thread(target=printer, args=('D', 7))
```

```
t5 = Thread(target=printer, args=('E', 2))

t1.start()
t2.start()
t3.start()
t4.start()
t5.start()
```

An example of the sort of output this could generate is given below:

```
BAEAEABEDAEAEBEDCECBEEEADCDBBDABCADBBDABADCDCDCCCC
```

Chapter 40
MultiProcessing

40.1 Introduction

The `multiprocessing` library supports the generation of separate (operating system level) processes to execute behaviour (such as functions or methods) using an API that is similar to the Threading API presented in the last chapter.

It can be used to avoid the limitation introduced by the Global Interpreter Lock (the GIL) by using separate operating system processes rather than lightweight threads (which run within a single process).

This means that the `multiprocessing` library allows developers to fully exploit the multiple processor environment of modern computer hardware which typically has multiple processor cores allowing multiple operations/behaviours to run in parallel; this can be very significant for data analytics, image processing, animation and games applications.

The multiprocessing library also introduces some new features, most notably the `Pool` object for parallelising execution of a callable object (e.g., functions and methods) that has no equivalent within the Threading API.

40.2 The Process Class

The Process class is the `multiprocessing` library's equivalent to the `Thread` class in the `threading` library. It can be used to run a callable object such as a function in a separate process. To do this it is necessary to create a new instance of the `Process` class and then call the `start()` method on it. Methods such as `join()` are also available so that one process can wait for another process to complete before continuing, etc.

The main difference is that when a new `Process` is created it runs within a separate process on the underlying operating systems (such as Window, Linux or Mac

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_40

OS). In contrast a Thread runs within the same process as the original program. This means that the process is managed and executed directly by the operating system on one of the processors that are part of the underlying computer hardware.

The up side of this is that you are able to exploit the underlying parallelism inherent in the physical computer hardware. The downside is that a Process takes more work to set up than the lighter weight Threads.

The constructor for the Process class provides the same set of arguments as the Thread class, namely:

```
class multiprocessing.Process(group=None,
                              target=None,
                              name=None,
                              args=(),
                              kwargs={},
                              daemon=None)
```

- group should always be None; it exists solely for compatibility with the Threading API.
- target is the callable object to be invoked by the run() method. It defaults to None, meaning nothing is called.
- name is the process name.
- args is the argument tuple for the target invocation.
- kwargs is a dictionary of keyword arguments for the target invocation.
- daemon argument sets the process daemon flag to True or False. If None (the default), this flag will be inherited from the creating process.

As with the Thread class, the Process constructor should always be called using keyword arguments.

The Process class also provides a similar set of methods to the Thread class:

- start() Start the process's activity. This must be called at *most once* per process object. It arranges for the object's run() method to be invoked in a separate process.
- join([timeout]) If the optional argument timeout is None (the default), the method blocks until the joined process terminates. If timeout is a positive number, it blocks at most timeout seconds. Note that the method returns None if its process terminates or if the method times out.
- is_alive() Return whether the process is alive. Roughly, a process object is alive from the moment the start() method returns until the child process terminates.

The process class also has several attributes:

- `name` The process's name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name. It can be useful for debugging purposes.
- `daemon` The process's daemon flag, a Boolean value. This must be set before `start()` is called. The default value is inherited from the creating process. When a process exits, it attempts to terminate all of its daemonic child processes. Note that a daemonic process is not allowed to create child processes.
- `pid` Return the process ID. Before the process is spawned, this will be `None`.
- `exitcode` The process exit code. This will be `None` if the process has not yet terminated. A negative value -N indicates that the child was terminated by signal N.

In addition to these methods and attributes, the `Process` class also defines additional process related methods including:

- `terminate()` Terminate the process.
- `kill()` Same as `terminate()` except that on Unix the SIGKILL signal is used instead of the SIGTERM signal.
- `close()` Close the `Process` object, releasing all resources associated with it. `ValueError` is raised if the underlying process is still running. Once `close()` returns successfully, most of the other methods and attributes of the `Process` object will raise a `ValueError`.

40.3 Working with the Process Class

The following simple program creates three `Process` objects; each runs the function `worker()`, with the string arguments A, B and C respectively. These three process objects are then started using the `start()` method.

```
from multiprocessing import Process, set_start_method
from time import sleep

def worker(msg):
    for _ in range(0, 10):
       print(msg, end='', flush=True)
       sleep(1)

if __name__ == '__main__':
  print('Starting')
  set_start_method('spawn')

  t2 = Process(target=worker, args='A')
  t3 = Process(target=worker, args='B')
  t4 = Process(target=worker, args='C')

  t2.start()
  t3.start()
```

```
    t4.start()

    print('Done')
```

It is essentially the same as the equivalent program for threads but with the `Process` class being used instead of the `Thread` class.

The output from this application is given below:

```
Starting
Done
ABCABCABCABCABCABCABCACBACBACB
```

The main difference between the `Thread` and `Process` versions is that the Process version runs the worker function in separate processes, whereas in the Thread version all the Threads share the same process.

However, there are two additional points to note, the first is that we have had to set the start method. In this case the value is set to 'spawn'. This is because when the start(0 method is called on a process, there are three different approaches/methods that can be used to start the underlying process, these are:

- 'spawn' The parent process starts a fresh Python interpreter process (that is it spawns a new process in OS terminology).
- 'fork' The parent process uses os.fork() to fork the Python interpreter.
- 'forkserver' In this case a server process is started. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process.

The set_start_method() should be used to set approach.

The second thing to note is that we are using the `if __name__=='__main__':` pattern or idiom in this program. If you don't do this you will get a RuntimeError indicating that the context has already been set. This is related to the file be parsed each time by each process, and attempting to run the processes multiple times, whereas we only want the main body of the program to be run once with each subprocess only needing to load the definition of the worker function. This is ensured using the `if __name__=='__main__'` pattern. If you fail to do this then you will get an output similar to:

```
Starting
Done
Starting
Starting
Starting
Traceback (most recent call last):
   File "<string>", line 1, in <module>
   …
   raise RuntimeError('context has already been set')
   raise RuntimeError('context has already been set')
   raise RuntimeError('context has already
been set')RuntimeError
: context has already been set
RuntimeError: context has already been set
RuntimeError: context has already been set
```

40.4 Alternative Ways to Start a Process

Returning to the different ways to spawn a process, we should consider the underlying operating system and what these mean to it:

- '**spawn**' The parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process objects run() method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using fork or forkserver. Available on Unix and Windows.
- '**fork**' The parent process uses os.fork() to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Available only on Unix type operating systems.
- '**forkserver**' In this case a server process is started. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded so it is safe for it to use `os.fork()`. No unnecessary resources are inherited. Available on Unix style platforms which support passing file descriptors over Unix pipes.

The `set_start_method()` should be used to set the start method (and this should only be set once within a program).

This is illustrated below, where the *spawn* start method is specified:

```
from multiprocessing import Process
from multiprocessing import set_start_method
from time import sleep
import os

def worker(msg):
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)

def main():
    print('Starting')
    print('Root application process id:', os.getpid())
    set_start_method('spawn')
    t = Process(target=worker, args='A')
    t.start()

    print('Done')

if __name__ == '__main__':
    main()
```

The output from this is shown below:

```
Starting
```

```
Root application process id: 23417
Done
module name: __mp_main__
parent process: 23417
process id: 23419
AAAAAAAAAA
```

Note that the parent process and current process ids are printed out for the `worker()` function, while the `main()` method prints out only its own id. This shows that the *main* application process id is the same as the worker process parents' id.

Alternatively, it is possible to use the `get_context()` method to obtain a context object. Context objects have the same API as the `multiprocessing` module and allow you to use multiple start methods in the same program, for example:

```
ctx = multiprocessing.get_context('spawn')
q = ctx.Queue()
p = ctx.Process(target=foo, args=(q,))
```

40.5 Using a Pool

Creating Processes is expensive in terms of computer resources. It would therefore be useful to be able to reuse processes within an application. The Pool class provides such reusable processes.

The `Pool` class represents a pool of worker processes that can be used to perform a set of concurrent, parallel operations. The `Pool` provides methods which allow tasks to be offloaded to these worker processes.

The `Pool` class provides a constructor which takes a number of arguments:

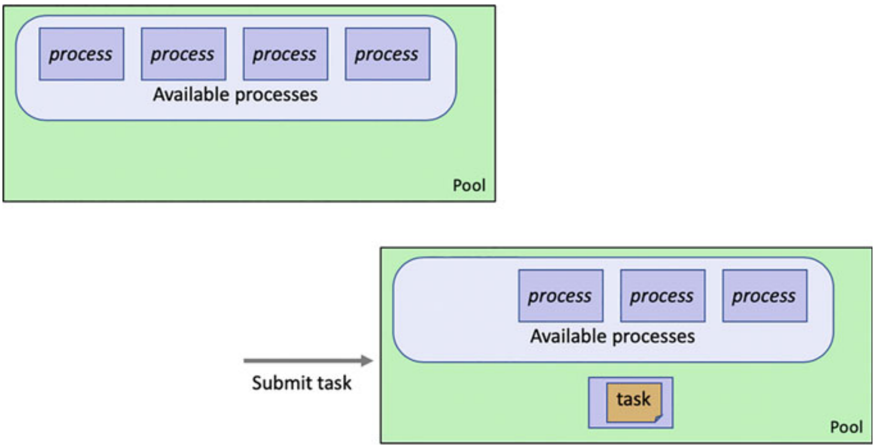
```
class multiprocessing.pool.Pool(processes,
                               initializer, initargs,
                               maxtasksperchild,
                               context)
```

These represent:

- `processes` is the number of worker processes to use. If processes is None then the number returned by `os.cpu_count()` is used.
- `initializer` If `initializer` is not `None` then each worker process will call `initializer(*initargs)` when it starts.
- `maxtasksperchild` is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default `maxtasksperchild` is `None`, which means worker processes will live as long as the pool.
- `context` can be used to specify the context used for starting the worker processes. Usually a pool is created using the function `multiprocessing.Pool()`. Alternatively the pool can be created using the `Pool()` method of a context object.

The `Pool` class provides a range of methods that can be used to submit work to the worker processes managed by the pool. Note that the methods of the `Pool` object should only be called by the process which created the pool.

The following diagram illustrates the effect of submitting some work or task to the pool. From the list of available processes, one process is selected and the task is passed to the process. The process will then execute the task. On completion any results are returned and the process is returned to the available list. If when a task is submitted to the pool there are no available processes then the task will be added to a wait queue until such time as a process is available to handle the task.



The simplest of the methods provided by the `Pool` for work submission is the `map` method:

```
pool.map(func, iterable, chunksize=None)
```

This method returns a list of the results obtained by executing the function in parallel against each of the items in the `iterable` parameter.

- The `func` parameter is the callable object to be executed (such as a function or a method).
- The `iteratable` is used to pass in any parameters to the function.
- This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting `chunksize` to a positive integer. The method blocks until the result is ready.

The following sample program illustrates the basic use of the `Pool` and the `map()` method.

```
from multiprocessing import Pool

def worker(x):
    print('In worker with: ', x)
    return x * x

def main():
    with Pool(processes=4) as pool:
        print(pool.map(worker, [0, 1, 2, 3, 4, 5]))

if __name__ == '__main__':
    main()
```

Note that the `Pool` object must be closed once you have finished with it; we are therefore using the '`with as`' statement described earlier in this book to handle the `Pool` resource cleanly (it will ensure the `Pool` is closed when the block of code within the `with as` statement is completed).

The output from this program is.

```
In worker with: 0
In worker with: 1
In worker with: 2
In worker with: 3
In worker with: 4
In worker with: 5
[0, 1, 4, 9, 16, 25]
```

As can be seen from this output the `map()` function is used to run six different instances of the `worker()` function with the values provided by the list of integers. Each instance is executed by a *worker* process managed by the `Pool`.

However, note that the `Pool` only has 4 worker processes, this means that the last two instances of the worker function must wait until two of the worker Processes have finished the work they are doing and can be reused. This can act as a way of throttling, or controlling, how much work is done in parallel.

A variant on the `map()` method is the `imap_unordered()` method. This method also applies a given function to an iterable but does not attempt to maintain the order of the results. The results are accessible via the iterable returned by the function. This may improve the performance of the resulting program.

The following program modified the `worker()` function to return its result rather than print it. These results are then accessible by iterating over them as they are produced via a `for` loop:

```
from multiprocessing import Pool

def worker(x):
    print('In worker with: ', x)
    return x * x

def main():
    with Pool(processes=4) as pool:
        for result in pool.imap_unordered(worker,
                                          [0, 1, 2, 3, 4, 5]):
            print(result)

if __name__ == '__main__':
    main()
```

As the new method obtains results as soon as they are available, the order in which the results are returned may be different, as shown below:

```
In worker with: 0
In worker with: 1
In worker with: 3
In worker with: 2
In worker with: 4
In worker with: 5
0
1
9
16
4
25
```

A further method available on the `Pool` class is the `Pool.apply_async()` method. This method allows operations/functions to be executed asynchronously allowing the method calls to return immediately. That is as soon as the method call is made, control is returned to the calling code which can continue immediately. Any results to be collected from the asynchronous operations can be obtained either by providing a callback function or by using the blocking `get()` method to obtain a result.

Two examples are shown below, the first uses the blocking `get()` method. This method will wait until a result is available before continuing. The second approach uses a *callback* function. The callback function is called when a result is available; the result is passed into the function. Note that if we do not wait for the pool to finish what it is doing then the callback function might never be called as the program (as it is so short) may well have terminated. We there join() the pool which will block the main program until the pool closes after it finishes all the current tasks.

```
from multiprocessing import Pool
```

```
def collect_results(result):
    print('In collect_results: ', result)

def worker(x):
    print('In worker with: ', x)
    return x * x

def main():
    with Pool(processes=2) as pool:
        # get based example
        res = pool.apply_async(worker, [6])
        print('Result from async: ', res.get(timeout=1))

    # callback based example
    pool = Pool(processes=2)
    pool.apply_async(worker, args=[4], callback=collect_results)
    pool.close()
    # Need to wait for asynchronous process to complete
    pool.join()

if __name__ == '__main__':
    main()
```

The output from this is:

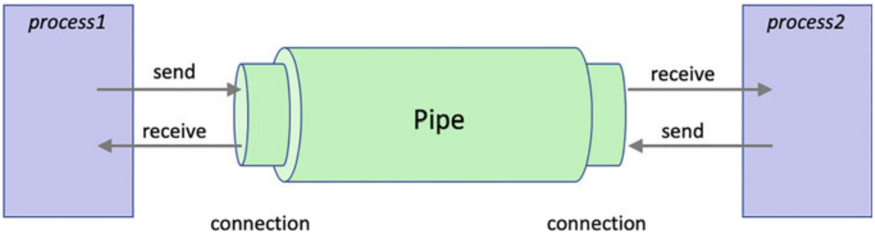
```
In worker with: 6
Result from async: 36
In worker with: 4
In collect_results: 16
```

40.6 Exchanging Data Between Processes

In some situations it is necessary for two processes to exchange data. However, the two process objects do not share memory as they are running in separate operating system level *processes*. To get around this the multiprocessing library provides the `Pipe()` function.

The `Pipe()` function returns a pair of `connection.Connection` objects connected by a pipe which by default is duplex (two-way).

The two connection objects returned by `Pipe()` represent the two ends of the pipe. Each connection object has `send()` and `recv()` methods (amongst others). This allows one process to send data via the `send()` method of one end of the connection object. In turn a second process can receive that data via the `receive()` method of the other connection object. This is illustrated below:



Once a program has finished with a connection is should be closed using `close()`.

The following program illustrates how pipe connections are used:

```
from multiprocessing import Process, Pipe
from time import sleep

def worker(conn):
    print('Worker - started now sleeping for 1 second')
    sleep(1)
    print('Worker - sending data via Pipe')
    conn.send('hello')
    print('Worker - closing worker end of connection')
    conn.close()

def main():
    print('Main - Starting, creating the Pipe')
    main_connection, worker_connection = Pipe()
    print('Main - Setting up the process')
    p = Process(target=worker, args=[worker_connection])
    print('Main - Starting the process')
    p.start()
    print('Main - Wait for a response from the child process')
    print(main_connection.recv())
    print('Main - closing parent process end of connection')
    parent_connection.close()
    print('Main - Done')

if __name__ == '__main__':
    main()
```

The output from this `Pipe` example is:

```
Main - Starting, creating the Pipe
Main - Setting up the process
Main - Starting the process
Main - Wait for a response from the child process
Worker - started now sleeping for 1 second
Worker - sending data via Pipe
Worker - closing worker end of connection
hello
Main - closing parent process end of connection
Main - Done
```

Note that data in a pipe may become corrupted if two processes try to read from or write to the same end of the pipe at the same time. However, there is no risk of corruption from processes using different ends of the pipe at the same time.

40.7 Sharing State Between Processes

In general, if it can be avoided, then you should not share state between separate processes. However, if it is unavoidable then the `mutiprocessing` library provides two ways in which state (data) can be shared, these are shared memory (as supported by `multiprocessing.Value` and `multiprocessing.Array`) and server process.

40.7.1 Process Shared Memory

Data can be stored in a shared memory map using a `multiprocessing.Value` or `multiprocessing.Array`. This data can be accessed by multiple processes.

The constructor for the `multiprocessing.Value` type is:

```
multiprocessing.Value(typecode_or_type, *args, lock=True)
```

where

- `typecode_or_type` determines the type of the returned object: it is either a ctypes type or a one character typecode. For example, ‘d’ indicates a double precision float and ‘i’ indicates a signed integer.
- `*args` is passed on to the constructor for the type.
- `lock` If lock is `True` (the default) then a new recursive lock object is created to synchronise access to the value. If lock is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be *process-safe.*

The constructor for `multiprocessing.Array` is.

```
multiprocessing.Array(typecode_or_type,
                      size_or_initializer,
                      lock=True)
```

where

- `typecode_or_type` determines the type of the elements of the returned array.
- `size_or_initializer` If size_or_initializer is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, size_or_initializer is a sequence which is used to initialise the array and whose length determines the length of the array.
- If `lock` is True (the default) then a new lock object is created to synchronise access to the value. If lock is False then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

An example using both the `Value` and `Array` type is given below:

```
from multiprocessing import Process, Value, Array

def worker(n, a):
      n.value = 3.1415927
      for i in range(len(a)):
           a[i] = -a[i]

def main():
     print('Starting')
     num = Value('d', 0.0)
     arr = Array('i', range(10))
     p = Process(target=worker, args=(num, arr))
     p.start()
     p.join()
     print(num.value)
     print(*arr)
     print('Done')

if __name__ == '__main__':
     main()
```

40.8 Online Resources

See the following online resources for information on multiprocessing:

- https://docs.python.org/3/library/multiprocessing.html. The Python standard Library documentation on MultiProcessing.
- https://pymotw.com/3/multiprocessing. The Python Module of the Week page on MultiProcessing
- https://pythonprogramming.net/multiprocessing-python-intermediate-python-tutorial. Tutorial on Python’s MultiProcessing module.

40.9 Exercises

Using the `factorial()` function that you wrote earlier in the book, run multiple factorial calculations in parallel.

Collect all the results together in a list and print that list out.

You can use whichever approach you like to running multiple processes although a `Pool` could be a good approach to use.

Your program should compute the factorials of 5, 8, 10, 15, 3, 6, and 4 in parallel.

Chapter 41
Inter Thread/Process Synchronisation

41.1 Introduction

In this chapter we will look at several facilities supported by both the `threading` and `multiprocessing` libraries that allow for synchronisation and cooperation between threads or processes.

In the remainder of this chapter we will look at some of the ways in which Python supports synchronisation between multiple threads and processes. Note that most of the libraries are mirrored between threading and multiprocessing so that the same basic ideas hold for both approaches with in the main very similar APIs. However, you should not mix and match threads and processes. If you are using Threads then you should only use facilities from the threading library. In turn if you are using Processes than you should only use facilities in the multiprocessing library. The examples given in this chapter will use one or other of the technologies but are relevant for both approaches.

41.2 Using a Barrier

Using a `threading.Barrier` (or `multiprocessing.Barrier`) is one of the simplest ways in which the execution of a set of Threads (or Processes) can be synchronised.

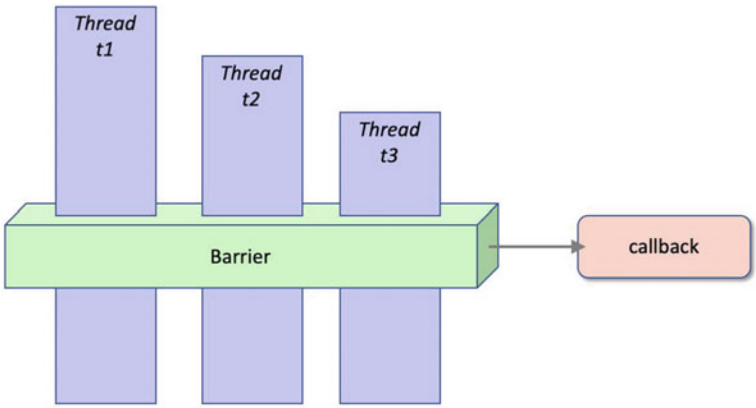
The threads or processes involved in the barrier are known as the *parties* that are taking part in the barrier.

Each of the *parties* in the barrier can work independently until it reaches the barrier point in the code.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_41

The barrier represents an end point that all parties must reach before any further behaviour can be triggered. At the point that all the parties reach the barrier it is possible to optionally trigger a *post-phase* action (also known as the barrier callback). This post-phase action represents some behaviour that should be run when all parties reach the barrier but before allowing those parties to continue. The post-phase action (the callback) executes in a single thread (or process). Once it is completed then all the parties are unblocked and may continue.

This is illustrated in the following diagram. Threads t1, t2 and t3 are all involved in the barrier. When thread t1 reaches the barrier it must wait until it is released by the barrier. Similarly when t2 reaches the barrier it must wait. When t3 finally reaches the barrier the callback is invoked. Once the callback has completed the barrier releases all three threads which are then able to continue.



An example of using a Barrier object is given below. Note that the function being invoked in each Thread must also cooperate in using the barrier as the code will run up to the `barrier.wait()` method and then wait until all other threads have also reached this point before being allowed to continue.

The `Barrier` is a class that can be used to create a barrier object. When the Barrier class is instantiated, it can be provided with three parameters:

```
Barrier(parties, action=None, timeout=None)
```

where

- `parties` the number of individual parties that will participate in the Barrier.
- `action` is a callable object (such as a function) which, when supplied, will be called after all the parties have entered the barrier and just prior to
- releasing them all.
- `timeout` If a 'timeout' is provided, it is used as the default for all subsequent `wait()` calls on the barrier.

Thus, in the following code.

```
b = Barrier(3, action=callback)
```

Indicates that there will be three parties involved in the `Barrier` and that the `callback` function will be invoked when all three reach the barrier (however the timeout is left as the default value `None`).

The `Barrier` object is created outside of the Threads (or Processes) but must be made available to the function being executed by the Thread (or Process). The easiest way to handle this is to pass the barrier into the function as one of the parameters; this means that the function can be used with different barrier objects depending upon the context.

An example using the `Barrier` class with a set of Threads is given below:

```
from threading import Barrier, Thread
from time import sleep
from random import randint

def print_it(msg, barrier):
    print('print_it for:', msg)
    for i in range(0, 10):
       print(msg, end='', flush=True)
       sleep(1)
    sleep(randint(1, 6))
    print('Wait for barrier with:', msg)
    barrier.wait()
    print('Returning from print_it:', msg)

def callback():
    print('Callback Executing')

print('Main - Starting')

b = Barrier(3, callback)
t1 = Thread(target=print_it, args=('A', b))
t2 = Thread(target=print_it, args=('B', b))
t3 = Thread(target=print_it, args=('C', b))
t1.start()
t2.start()
t3.start()

print('Main - Done')
```

The output from this is:

```
Main - Starting
print_it for: A
print_it for: B
print_it for: C
ABC
Main - Done
ABCACBACBABCACBCABACBACBBAC
Wait for barrier with: B
Wait for barrier with: A
```

```
Wait for barrier with: C
Callback Executing
Returning from print_it: A
Returning from print_it: B
Returning from print_it: C
```

From this you can see that the `print_it()` function is run three times concurrently; all three invocations reach the `barrier.wait()` statement but in a different order to that in which they were started. Once the three have reached this point the `callback` function is executed before the `print_it()` function invocations can proceed.

The `Barrier` class itself provides several methods used to manage or find out information about the barrier:

Method	Description
wait(timeout = None)	Wait until all threads have notified the barrier (unless timeout is reached) – returns the number of threads that passed the barrier
reset()	Return barrier to default state
abort()	Put the barrier into a broken state
parties	Return the number of threads required to pass the barrier
n_waiting	Number of threads currently waiting

A `Barrier` object can be reused any number of times for the same number of Threads.

The above example could easily be changed to run using `Process` by altering the `import` statement and creating a set of Processes instead of Threads. We will also sent the start method this time to 'fork' as 'spawn' can cause problems on some versions of Python on macOS. We will also use the if __name__=='main' pattern:

```
from multiprocessing import Barrier, Process, set_start_method
from time import sleep
from random import randint

def print_it(msg, barrier):
    print('print_it for:', msg)
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)
    sleep(randint(1, 6))
    print('Wait for barrier with:', msg)
    barrier.wait()
    print('Returning from print_it:', msg)

def callback():
    print('\nCallback Executing')
```

```
if __name__ == '__main__':
    print('Main - Starting')

    set_start_method('fork')
    barrier = Barrier(3, callback)
    p1 = Process(target=print_it, args=('A', barrier))
    p2 = Process(target=print_it, args=('B', barrier))
    p3 = Process(target=print_it, args=('C', barrier))
    p1.start()
    p2.start()
    p3.start()

    print('Main - Done')
```

Note that you should only use threads with a threading.Barrier. In turn you should only use Processes with a multiprocessing.Barrier.

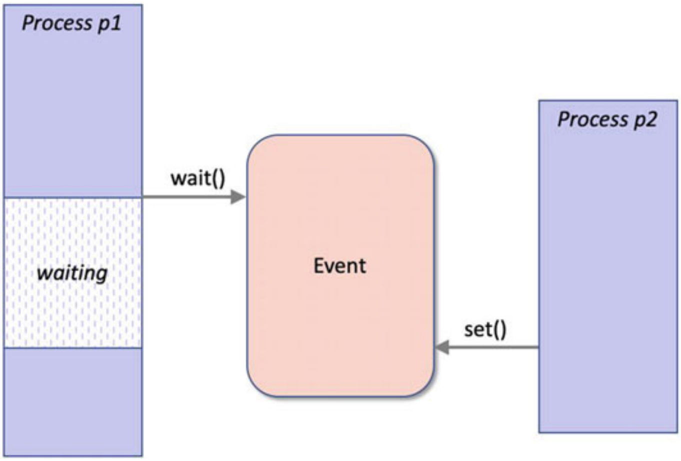
41.3 Event Signalling

Although the point of using multiple Threads or Processes is to execute separate operations concurrently, there are times when it is important to be able to allow two or more Threads or Processes to cooperate on the timing of their behaviour. The `Barrier` object presented above is a relatively high-level way to do this; however, in some cases finer grained control is required. The `threading.Event` or `multiprocessing.Event` classes can be used for this purpose.

An `Event` manages an internal flag that callers can either `set()` or `clear()`. Other threads can `wait()` for the flag to be `set()`, effectively blocking their own progress until allowed to continue by the `Event`. The internal flag is initially set to `False` which ensures that if a task gets to the `Event` before it is *set* then it must *wait*.

You can infact invoke wait with an optional timeout. If you do not include the optional timeout then `wait()` will wait forever while `wait(timeout)` will wait up to the timeout given in seconds. If the time out is reached, then the `wait` method returns `False`; otherwise `wait` returns `True`.

As an example, the following diagram illustrates two processes sharing an event object. The first process runs a function that waits for the event to be set. In turn the second process runs a function that will set the event and thus release the waiting process.



The following program implements the above scenario:

```
from multiprocessing import Process, Event, set_start_method
from time import sleep

def wait_for_event(event):
    print('wait_for_event - Entered and waiting')
    event_is_set = event.wait()
    print('wait_for_event - Event is set: ', event_is_set)

def set_event(event):
    print('set_event - Entered but about to sleep')
    sleep(5)
    print('set_event - Waking up and setting event')
    event.set()
    print('set_event - Event set')

if __name__ == '__main__':
    print('Main - Starting')

    set_start_method('fork')

    # Create the event object
    event = Event()

    # Start a Process to wait for the event notification
    p1 = Process(target=wait_for_event, args=[event])
    p1.start()

    # Set up a process to set the event
    p2 = Process(target=set_event, args=[event])
    p2.start()

    # Wait for the first process to complete
    p1.join()

    print('Done')
```

The output from this program is:

```
Main - Starting
wait_for_event - Entered and waiting
set_event - Entered but about to sleep
set_event - Waking up and setting event
set_event - Event set
wait_for_event - Event is set: True
Done
```

To change this to use Threads we would merely need to change the import and to create two Threads:

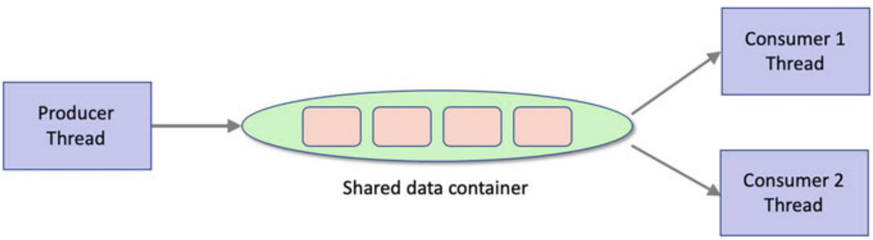
```python
from threading import Thread, Event
...
print('Starting')
event = Event()
t1 = Thread(target=wait_for_event, args=[event])
t1.start()
t2 = Thread(target=set_event, args=[event])
t2.start()
t1.join()
print('Done')
```

41.4 Synchronising Concurrent Code

It is not uncommon to need to ensure that critical regions of code are protected from concurrent execution by multiple Threads or Processes. These blocks of code typically involve the modification of, or access to, shared data. It is therefore necessary to ensure that only one Thread or Process is updating a shared object at a time and that consumer threads or processes are blocked while this update is occurring.

This situation is most common where one or more Threads or Processes are the producers of data and one or more other Threads or Processes are the consumers of that data.

This is illustrated in the following diagram.



In this diagram the *Producer* is running in its own Thread (although it could also run in a separate Process) and places data onto some common shared data container. Subsequently a number of independent *Consumers* can consume that data when it is available and when they are free to process the data. However, there is no point in the consumers repeatedly checking the container for data as that would be a waste of resources (for example in terms of executing code on a processor and of context switching between multiple Threads or Processes).

We therefore need some form of notification or synchronisation between the Producer and the Consumer to manage this situation.

Python provides several classes in the `threading` (and also in the `multiprocessing`) library that can be used to manage critical code blocks. These classes include `Lock`, `Condition` and `Semaphore`.

41.5 Python Locks

The `Lock` class defined (both in the `threading` and the `multiprocessing` libraries) provides a mechanism for synchronising access to a block of code. The `Lock` object can be in one of two states *locked* and *unlocked* (with the initial state being *unlocked*). The `Lock` grants access to a single thread at a time; other threads must wait for the `Lock` to become free before progressing.

The `Lock` class provides two basic methods for acquiring the lock (`acquire()`) and releasing (`release()`) the lock.

- When the state of the `Lock` object is unlocked, then `acquire()` changes the state to locked and returns immediately.
- When the state is locked, `acquire()` blocks until a call to `release()` in another thread changes it to unlocked, then the `acquire()` call resets it to locked and returns.
- The `release()` method should only be called in the *locked* state; it changes the state to *unlocked* and returns immediately. If an attempt is made to release an unlocked lock, a `RuntimeError` will be raised.
- An example of using a Lock object is shown below:

```
from threading import Thread, Lock

class SharedData(object):
      def __init__(self):
          self.value = 0
          self.lock = Lock()

      def read_value(self):
          try:
             print('read_value Acquiring Lock')
             self.lock.acquire()
             return self.value
          finally:
```

```
            print('read_value releasing Lock')
            self.lock.release()

    def change_value(self):
        print('change_value acquiring lock')
        with self.lock:
            self.value = self.value + 1
        print('change_value lock released')
```

The SharedData class presented above uses locks to control access to critical blocks of code, specifically to the read_value() and the change_value() methods. The Lock object is held internally to the ShareData object, and both methods attempt to acquire the lock before performing their behaviour but must then release the lock after use.

The read_value() method does this explicitly using try: finally: blocks while the change_value() method uses a with statement (as the Lock type supports the *Context Manager Protocol*). Both approaches achieve the same result, but the with statement style is more concise.

The SharedData class is used below with two simple functions. In this case the SharedData object has been defined as a *global* variable, but it could also have been passed into the reader() and updater() functions as an argument. Both the reader and updater functions loop, attempting to call the read_value() and change_value() methods on the shared_data object.

As both methods use a lock to control access to the methods, only one thread can gain access to the locked area at a time. This means that the reader() function may start to read data before the updater() function has changed the data (or vice versa).

This is indicated by the output where the *reader* thread accesses the value '0' twice before the *updater* records the value '1'. However, the updater() function runs a second time before the reader gains access to locked block of code which is why the value 2 is missed. Depending upon the application this may or may not be an issue.

```
shared_data = SharedData()

def reader():
    while True:
        print(shared_data.read_value())

def updater():
    while True:
        shared_data.change_value()

print('Starting')

t1 = Thread(target=reader)
t2 = Thread(target=updater)

t1.start()
t2.start()
```

```
print('Done')
```

The output from this is:

```
Starting
read_value Acquiring Lock
read_value releasing Lock
0
read_value Acquiring Lock
read_value releasing Lock
0
Done
change_value acquiring lock
change_value lock released
1
change_value acquiring lock
change_value lock released
change_value acquiring lock
change_value lock released
3
change_value acquiring lock
change_value lock released
4
```

`Lock` objects can only be acquired once; if a thread attempts to acquire a lock on the same `Lock` object more than once then a `RuntimeError` is thrown.

If it is necessary to re-acquire a lock on a `Lock` object then the `threading.RLock` class should be used. This is a re-entrant lock and allows the same Thread (or Process) to acquire a lock multiple times. The code must however release the lock as many times as it has acquired it.

41.6 Python Conditions

Conditions can be used to synchronise the interaction between two or more Threads or Processes. Conditions objects support the concept of a notification model; ideal for a shared data resource being accessed by multiple consumers and producers.

A `Condition` can be used to notify one or all of the waiting Threads or Processes that they can proceed (for example to read data from a shared resource). The methods available that support this are:

- `notify()` notifies one waiting thread which can then continue
- `notify_all()` notifies all waiting threads that they can continue
- `wait()` causes a thread to wait until it has been notified that it can continue.

A `Condition` is always associated with an *internal* lock which must be acquired and released before the `wait()` and `notify()` methods can be called. The `Condition` supports the *Context Manager Protocol* and can therefore be used via a `with` statement (which is the most typical way to use a `Condition`) to obtain this lock. For example, to obtain the condition lock and call the wait method we might write:

```
with condition:
    condition.wait()
    print('Now we can proceed')
```

The condition object is used in the following example to illustrate how a producer thread and two consumer threads can cooperate. A `DataResource` class has been defined which will hold an item of data that will be shared between a consumer and a set of producers. It also (internally) defines a `Condition` attribute. Note that this means that the `Condition` is completely internalised to the `DataResource` class; external code does not need to know, or be concerned with, the `Condition` and its use. Instead external code can merely call the `consumer()` and `producer()` functions in separate Threads as required.

The `consumer()` method uses a `with` statement to obtain the (internal) lock on the `Condition` object before waiting to be notified that the data is available. In turn the `producer()` method also uses a `with` statement to obtain a lock on the condition object before generating the data attribute value and then notifying anything *waiting* on the condition that they can proceed. Note that although the consumer method obtains a lock on the condition object; if it has to wait it will release the lock and re obtain the lock once it is notified that it can continue. This is a subtly that is often missed.

```
from threading import Thread, Condition, current_thread
from time import sleep
from random import randint

class DataResource:

    def __init__(self):
        print('DataResource - Initialising the empty data')
        self.data = None
        print('DataResource - Setting up the Condition object')
        self.condition = Condition()

    def consumer(self):
        """wait for the condition and use the resource"""
        print('DataResource - Starting consumer method in', current_
thread().name)
        with self.condition:
            self.condition.wait()
            print('DataResource - Resource is available to',
current_thread().name)
            print('DataResource - Data read in', current_
thread().name, ':', self.data)

    def producer(self):
```

```
        """set up the resource to be used by the consumer"""
        print('DataResource - Starting producer method')
        with self.condition:
            print('DataResource - Producer setting data')
            self.data = randint(1, 100)
           print('DataResource - Producer notifying all waiting
threads')
            self.condition.notify_all()

print('Main - Starting')
print('Main - Creating the DataResource object')
resource = DataResource()

print('Main - Create the Consumer Threads')
c1 = Thread(target=resource.consumer)
c1.name = 'Consumer1'
c2 = Thread(target=resource.consumer)
c2.name = 'Consumer2'
print('Main - Create the Producer Thread')
p = Thread(target=resource.producer)

print('Main - Starting consumer threads')
c1.start()
c2.start()
sleep(1)

print('Main - Starting producer thread')
p.start()

print('Main - Done')
```

The output from an example run of this program is:

```
Main - Starting
Main - Creating the DataResource object
DataResource - Initialising the empty data
DataResource - Setting up the Condition object
Main - Create the Consumer Threads
Main - Create the Producer Thread
Main - Starting consumer threads
DataResource - Starting consumer method in Consumer1
DataResource - Starting consumer method in Consumer2
Main - Starting producer thread
DataResource - Starting producer method
DataResource - Producer setting data
Main - Done
DataResource - Producer notifying all waiting threads
DataResource - Resource is available to Consumer1
DataResource - Data read in Consumer1 : 36
DataResource - Resource is available to Consumer2
DataResource - Data read in Consumer2 : 36
```

41.7 Python Semaphores

The Python `Semaphore` class implements Dijkstra counting semaphore model.

In general, a semaphore is like an integer variable; its value is intended to represent a number of available resources of some kind. There are typically two operations available on a semaphore; these operations are `acquire()` and `release()`; although in some libraries Dijkstra's original names of `p()` and `v()` are used, these operation names are based on the original Dutch phrases.

- The `acquire()` operation subtracts one from the value of the semaphore, unless the value is 0, in which case it blocks the calling thread until the semaphore's value increases above 0 again.
- The `signal()` operation adds one to the value, indicating a new instance of the resource has been added to the pool.

Both the `threading.Semaphore` and the `multiprocessing.Semaphore` classes also support the *Context Management Protocol.*

An optional parameter used with the Semaphore constructor gives the initial value for the internal counter; it defaults to 1. If the value given is less than 0, `ValueError` is raised.

The following example illustrates 5 different Threads all running the same `worker()` function. The `worker()` function attempts to acquire a semaphore; if it does then it continues into the `with` statement block; if it doesn't, it waits until it can acquire it. As the semaphore is initialised to 2 there can only be two threads that can acquire the `Semaphore` at a time.

The sample program, however, starts up five threads; this therefore means that the first 2 running Threads will acquire the semaphore and the remaining thee will have to wait to acquire the semaphore. Once the first two release the semaphore a further two can acquire it and so on.

```
from threading import Thread, Semaphore, current_thread
from time import sleep

def worker(semaphore):
    with semaphore:
        print(current_thread().name + " - entered")
        sleep(0.5)
        print(current_thread().name + " - exiting")

print('MainThread - Starting')

semaphore = Semaphore(2)
for i in range(0, 5):
        thread = Thread(name='T' + str(i), target=worker,
args=[semaphore])
    thread.start()

print('MainThread - Done')
```

The output from a run of this program is given below:

```
MainThread - Starting
T0 - entered
T1 - entered
MainThread - Done
T0 - exiting
T2 - entered
T1 - exiting
T3 - entered
T2 - exiting
T4 - entered
T3 - exiting
T4 - exiting
```

41.8 The Concurrent Queue Class

As might be expected the model where a producer Thread or Process generates data to be processed by one or more Consumer Threads or Processes is so common that a higher level abstraction is provided in Python than the use of Locks, Conditions or Semaphores; this is the blocking queue model implemented by the `threading.Queue` or `multiprocessing.Queue` classes.

Both these `Queue` classes are Thread and Process safe. That is they work appropriately (using internal locks) to manage data access from concurrent Threads or Processes.

An example of using a `Queue` to exchange data between a worker process and the main process is shown below.

The *worker* process executes the `worker()` function sleeping, for 2 s before putting a string 'Hello World' on the queue. The main application function sets up the queue and creates the process. The queue is passed into the process as one of its arguments. The process is then started. The main process then waits until data is available on the queue via the (blocking) `get()` methods. Once the data is available it is retrieved and printed out before the main process terminates.

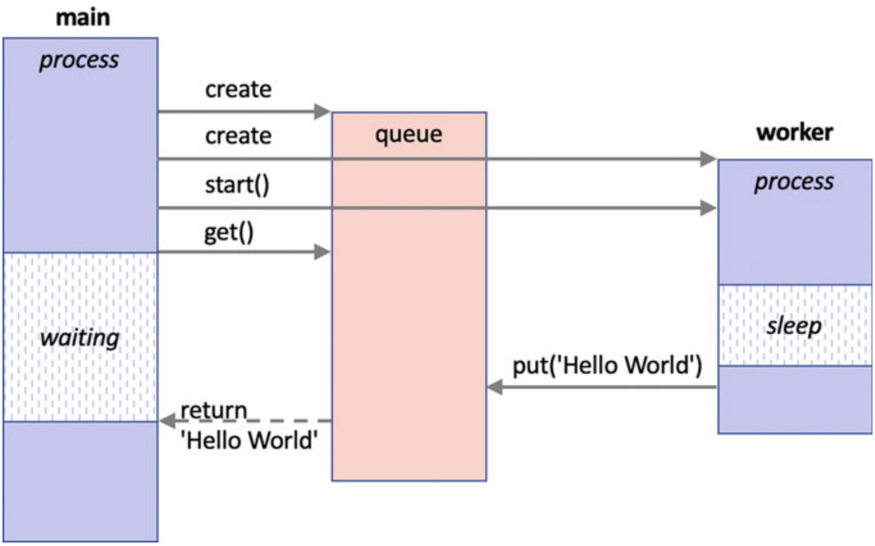
```
from multiprocessing import Process, Queue
from time import sleep
def worker(queue):
    print('Worker - going to sleep')
    sleep(2)
    print('Worker - woken up and putting data on queue')
    queue.put('Hello World')

def main():
    print('Main - Starting')
    queue = Queue()
    p = Process(target=worker, args=[queue])
    print('Main - Starting the process')
```

```
    p.start()
    print('Main - waiting for data')
    print(queue.get())
    print('Main - Done')

if __name__ == '__main__':
    main()
```

The output from this is shown below:

```
Main - Starting
Main - Starting the process
Main - wait for data
Worker - going to sleep
Worker - woken up and putting data on queue
Hello World
Main - Done
```

However, this does not make it that clear how the execution of the two processes interweaves. The following diagram illustrates this graphically:



In the above diagram the main process waits for a result to be returned from the queue following the call to the `get()` method; as it is waiting it is not using any system resources. In turn the worker process sleeps for two seconds before putting some data onto the queue (via `put('Hello World')`). After this value is sent to the `Queue` the value is returned to the main process which is woken up (moved out of the waiting state) and can continue to process the rest of the main function.

41.9 Online Resources

See the following online resources for information discussed in this chapter:

- https://docs.python.org/3/library/threading.html for information on Thread based barriers, locks, conditions, semaphores and events.
- https://docs.python.org/3/library/multiprocessing.html for information on Process based barriers, locks, conditions, semaphores and events.
- https://en.wikipedia.org/wiki/Semaphore_programming Semaphore programming model.

41.10 Exercises

The aim of this exercise is to implement a concurrent version of your `Stack`-based container/collection.

It should be possible to safely add data to your stack and *pop* data off the stack using multiple Threads.

It should follow a similar pattern to the Queue class described above but support the First In Last Out (FILO) behaviour of a stack and be usable with any number of producer and consumer threads (you can ignore processes for this exercise).

The key to implementing the `Stack` is to remember that no data can be read from the stack until there is some data to access; it is therefore necessary to wait for data to become available and then to read it. However, it is a producer thread that will provide that data and then inform any waiting threads that there is not data available. You can implement this in any way you wish; however a common solution is to use a `Condition`.

To illustrate this idea, the following test program can be used to verify the behaviour of your `Stack`:

```
from stack.Stack import Stack
from time import sleep
from threading import Thread

def producer(stack):
    for i in range(0,6):
        data = 'Task' + str(i)
        print('Producer pushing:', data)
        stack.push(data)
        sleep(2)

def consumer(label, stack):
    while True:
        print(label, 'stack.pop():', stack.pop())

print('Create shared stack')
stack = Stack()
print('Stack:', stack)
```

```
print('Creating and starting consumer threads')
consumer1 = Thread(target=consumer, args=('Consumer1', stack))
consumer2 = Thread(target=consumer, args=('Consumer2', stack))
consumer3 = Thread(target=consumer, args=('Consumer3', stack))
consumer1.start()
consumer2.start()
consumer3.start()

print('Creating and starting producer thread')
producer = Thread(target=producer, args=[stack])
producer.start()
```

The output generated from this sample program (which includes print statements from the `Stack`) is given below:

```
Create shared stack
Stack: Stack: []
Creating and starting consumer threads
Creating and starting producer thread
Producer pushing: Task0
Consumer1 stack.pop(): Task0
Producer pushing: Task1
Consumer2 stack.pop(): Task1
Producer pushing: Task2
Consumer3 stack.pop(): Task2
Producer pushing: Task3
Consumer1 stack.pop(): Task3
Producer pushing: Task4
Consumer2 stack.pop(): Task4
Producer pushing: Task5
Consumer3 stack.pop(): Task5
```

Chapter 42
Futures

42.1 Introduction

A future is a thread (or process) that promises to return a value in the future once the associated behaviour has completed. It is thus a *future* value. It provides a very simple way of firing off behaviour that will either be time consuming to execute or which may be delayed due to expensive operations such as Input/Output and which could slow down the execution of other elements of a program. This chapter discusses futures in Python.

42.2 The Need for a Future

In a normal method or function invocation, the method or function is executed in line with the invoking code (the *caller*) having to wait until the function or method (the *callee*) returns. Only after this is the caller able to continue to the next line of code and execute that. In many (most) situations this is exactly what you want as the next line of code may depend on a result returned from the previous line of code etc.

However, in some situations the next line of code is independent of the previous line of code. For example, let us assume that we are populating a user interface (UI). The first line of code may read the name of the user from some external data source (such as a database) and then display it within a field in the UI. The next line of code may then add todays data to another field in the UI. These two lines of code are independent of each other and could be run concurrently/in parallel with each other.

In this situation we could use either a Thread or a Process to run the two lines of code independently of the caller, thus achieving a level of concurrency and allowing the caller to carry onto the third line of code etc.

However, neither the Thread or the Process by default provide a simple mechanism for obtaining a result from such an independent operation. This may not be a problem

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_42

as operations may be self-contained; for example they may obtain data from the database or from today's date and then updated a UI. However, in many situations the calculation will return a result which needs to be handled by the original invoking code (the caller). This could involve performing a long running calculation and then using the result returned to generate another value or update another object etc.

A Future is an abstraction that simplifies the definition and execution of such concurrent tasks. Futures are available in many different languages including Python but also Java, Scala, C++, etc. When using a Future; a callable object (such as a function) is passed to the Future which executes the behaviour either as a separate Thread or as a separate Process and then can return a result once it is generated. The result can either be handled by a call back function (that is invoked when the result is available) or by using a operation that will wait for a result to be provided.

42.3 Futures in Python

The `concurrent.futures` library was introduced into Python in version 3.2 (and is also available in Python 2.5 onwards). The `concurrent.futures` library provides the Future class and a high level API for working with Futures.

The `concurrent.futures.Future` class encapsulates the asynchronous execution of a callable object (e.g. a function or method).

The `Future` class provides a range of methods that can be used to obtain information about the state of the future, retrieve results or cancel the future:

- `cancel()` Attempt to cancel the Future. If the Future is currently being executed and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.
- `cancelled()` Returns `True` if the Future was successfully cancelled.
- `running()` Return s`True` if the Future is currently being executed and cannot be cancelled.
- `done()` Returns `True` if the Future was successfully cancelled or finished running.
- `result(timeout=None)` Return the value returned by the Future. If the Future hasn't yet completed then this method will wait up to timeout seconds. If the call hasn't completed in timeout seconds, then a `TimeoutError` will be raised. timeout can be an `int` or `float`. If timeout is not specified or `None`, there is no limit to the wait time. If the future is cancelled before completing then the `CancelledError` will be raised. If the call raised, this method will raise the same exception.

It should be noted, however, that Future instances should not be created directly, rather they should be created via the submit method of an appropriate *executor*.

42.3.1 Future Creation

Futures are created and executed by Executors. An `Executor` provides two methods that can be used to execute a `Future` (or Futures) and one to shut down the executor.

At the root of the executor class hierarchy is the `concurrent.futures.Executor` abstract class. It has two subclasses:

- the `ThreadPoolExecutor` and
- the `ProcessPoolExecutor`.

The `ThreadPoolExecutor` uses *threads* to execute the futures while the `ProcessPoolExecutor` uses separate *processes*. You can therefore choose how you want the Future to be executed by specifying one or other of these executors.

42.3.2 Simple Example Future

To illustrate these ideas, we will look at a very simple example of using a Future.

To do this we will use a simple worker function; similar to that used in the previous chapters:

```
from time import sleep

# define function to be used with future
def worker(msg):
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)
    return i
```

The only difference with this version of `worker` is that it also returns a result which is the number of times that the worker printed out the message.

We can of course invoke this method inline as follows:

```
res = worker('A')
print(res)
```

We can make the invocation of this method into a Future. To do this we use a `ThreadPoolExecutor` imported from the `concurrent.futures` module. We will then submit the `worker` function to the pool for execution. This returns a reference to a `Future` which we can use to obtain the result:

```
from time import sleep
from concurrent.futures import ThreadPoolExecutor

print('Setting up the ThreadPoolExecutor')
pool = ThreadPoolExecutor(1)

# Submit the function ot the pool to run
# concurrently - obtain a future from pool
```

```
print('Submitting the worker to the pool')
future = pool.submit(worker, 'A')

print('Obtained a reference to the future object', future)

# Obtain the result from the future - wait if necessary
print('future.result():', future.result())

print('Done')
```

The output from this is:

```
Setting up the ThreadPoolExecutor
Submitting the worker to the pool
AAObtained a reference to the future object <Future at
0x1086ea8d0 state=running>
AAAAAAAAfuture.result(): 9
Done
```

Notice how the output from the main program and the worker is interwoven with two ‘A’s being printed out before the message starting ‘Obtained a…’.

In this case a new `ThreadPoolExecutor` is being created with one thread in the pool (typically there would be multiple threads in the pool, but one is being used here for illustrative purposes).

The `submit()` method is then used to submit the function worker with the parameter ‘A’ to the `ThreadPoolExecutor` for it to schedule execution of the function. The `submit()` method returns a Future object.

The main program then waits for the future object to return a result (by calling the `result()` method on the future). This method can also take a timeout.

To change this example to use Processes rather than Threads all that is needed is to change the pool executor to a `ProcessPoolExecutor`:

```
from multiprocessing import set_start_method
from time import sleep
from concurrent.futures import ProcessPoolExecutor

def worker(msg):
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)
    return i

if __name__ == '__main__':
    print('Starting')

    set_start_method('spawn')

    print('Setting up the ProcessPoolExecutor')
    pool = ProcessPoolExecutor(1)

    print('Submitting the worker to the pool')
    future = pool.submit(worker, 'A')

    print('Obtained a reference to the future object', future)
```

```
    print('future.result():', future.result())

    print('Done')
```

The output from this program is very similar to the last one:

```
Starting
Setting up the ThreadPoolExecutor
Submitting the worker to the pool
Obtained a reference to the future object <Future at 0x109178630
state=running>
AAAAAAAAAfuture.result(): 9
Done
```

The only difference is that in this particular run the message starting ‘Obtained a..’ is printed out before any of the ‘A’s are printed; this may be due to the fact that a Process initially takes longer to set up than a Thread.

42.4 Running Multiple Futures

Both the `ThreadPoolExecutor` and the `ProcessPoolExecutor` can be configured to support multiple Threads/Processes via the pool. Each task that is submitted to the pool will then run within a separate Thread/Process. If more tasks are submitted than there are Threads/Processes available, then the submitted task will wait for the first available Thread/Process and then be executed. This can act as a way of managing the amount of concurrent work being done.

For example, in the following example, the `worker()` function is submitted to the pool four times, but the pool is configured to use threads. Thus the fourth worker will need to wait until one of the first three completes before it is able to execute:

```
from concurrent.futures import ThreadPoolExecutor

print('Starting...')
pool = ThreadPoolExecutor(3)
future1 = pool.submit(worker, 'A')
future2 = pool.submit(worker, 'B')
future3 = pool.submit(worker, 'C')
future4 = pool.submit(worker, 'D')
print('\nfuture4.result():', future4.result())
print('All Done')
```

When this runs we can see that the Futures for A, B and C all run concurrently but D must wait until one of the others finishes:

```
Starting...
ABCACBCABCBABCACBACABCBACABCBADDDDDDDDDD
future4.result(): 9
All Done
```

The main thread also waits for `future4` to finish as it requests the result which is a blocking call that will only return once the future has completed and generates a result.

Again, to use Processes rather than Threads all we need to do is to replace the `ThreadPoolExecutor` with the `ProcessPoolExecutor` and add the required Process structure such as `set_start_method` and the if statement on the module name:

```
from concurrent.futures import ProcessPoolExecutor
from multiprocessing import set_start_method
from time import sleep

def worker(msg):
    for i in range(0,10):
        print(msg,end='', flush=True)
        sleep(1)
    return i

if __name__ == '__main__':
    print('Starting')
    set_start_method('spawn')

    pool = ProcessPoolExecutor(3)
    future1 = pool.submit(worker, 'A')
    future2 = pool.submit(worker, 'B')
    future3 = pool.submit(worker, 'C')
    future4 = pool.submit(worker, 'D')
    print('\nfuture4.result():', future4.result())
    print('All Done')
```

42.4.1 Waiting for All Futures to Complete

It is possible to wait for all futures to complete before progressing. In the previous section it was assumed that `future4` would be the last future to complete; but in many cases it may not be possible to know which *future* will be the last to complete. In such situations it is very useful to be able to wait for all the futures to complete before continuing. This can be done using the `concurrent.futures.wait` function. This function takes a collection of futures and optionally a `timeout` and a `return_when` indicator.

```
wait(fs, timeout=None, return_when=ALL_COMPLETED)
```

where:

- `timeout` can be used to control the maximum number of seconds to wait before returning. timeout can be an `int` or `float`. If timeout is not specified or `None`, there is no limit to the wait time.

- `return_when` indicates when this function should return. It must be one of the following constants:
  - `FIRST_COMPLETED` The function will return when any future finishes or is cancelled.
  - `FIRST_EXCEPTION` The function will return when any future finishes by raising an exception. If no future raises an exception, then it is equivalent to `ALL_COMPLETED`.
  - `ALL_COMPLETED` The function will return when all futures finish or are cancelled.

The `wait()` function returns two sets `done` and `not_done`. The first set contains the futures that completed (finished or were cancelled) before the wait completed. The second set, the not_dones, contains uncompleted futures.

We can use the `wait()` function to modify out previous example so that we no longer rely on `future4` finishing last:

```
from concurrent.futures import ProcessPoolExecutor
from concurrent.futures import wait
from multiprocessing import set_start_method
from time import sleep

def worker(msg):
    for i in range(0,10):
        print(msg,end='', flush=True)
        sleep(1)
    return i

if __name__ == '__main__':
    set_start_method('spawn')
    print('Starting...setting up pool')
    pool = ProcessPoolExecutor(3)
    futures = []

    print('Submitting futures')
    future1 = pool.submit(worker, 'A')
    futures.append(future1)
    future2 = pool.submit(worker, 'B')
    futures.append(future2)
    future3 = pool.submit(worker, 'C')
    futures.append(future3)
    future4 = pool.submit(worker, 'D')
    futures.append(future4)

    print('Waiting for futures to complete')
    wait(futures)

    print('\nAll Done')
```

The output from this is:

```
Starting...setting up pool
Submitting futures
Waiting for futures to complete
```

```
ABCABCABCABCABCABCBCACBACBABCADDDDDDDDDD
All Done
```

Note how each future is added to the list of futures which is then passed to the `wait()` function.

42.4.2 Processing Results as Completed

What if we want to process each of the results returned by our collection of futures? We could loop through the futures list in the previous section once all the results have been generated. However, this means that we would have to wait for them all to complete before processing the list.

In many situations we would like to process the results as soon as they are generated without being concerned if that is the first, third, last or second etc.

The `concurrent.futures.as_completed()` function does preciously this; it will serve up each future in turn as soon as they are completed; with all futures eventually being returned but without guaranteeing the order (just that as soon as a future is finished generating a result it will be immediately available).

For example, in the following example, the `is_even()` function sleeps for a random number of seconds (ensuring that different invocations of this function will take different durations) then calculates a result:

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from time import sleep
from random import randint

def is_even(n):
    print('Checking if', n , 'is even')
    sleep(randint(1, 5))
    return str(n) + ''+ str(n % 2 == 0)

print('Started')
data = [1, 2, 3, 4, 5, 6]
pool = ThreadPoolExecutor(5)
futures = []

for v in data:
    futures.append(pool.submit(is_even, v))

for f in as_completed(futures):
    print(f.result())

print('Done')
```

The second `for` loop will loop through each future as they complete printing out the result from each, as shown below:

```
Started
Checking if 1 is even
```

```
Checking if 2 is even
Checking if 3 is even
Checking if 4 is even
Checking if 5 is even
Checking if 6 is even
1 False
4 True
5 False
3 False
2 True
6 True
Done
```

As you can see from this output although the six futures were started in sequence the results returned are in a different order (with the returned order being 1, 4, 5, 3, 2 and finally 6).

42.5 Processing Future Results Using a Callback

An alternative to the `as_complete()` approach is to provide a function that will be called once a result has been generated. This has the advantage that the main program is never paused; it can continue doing whatever is required of it.

The function called once the result is generated is typically known as a callback function; that is the future calls *back* to this function when the result is available.

Each future can have a separate call back as the function to invoke is set on the future using the `add_done_callback()` method. This method takes the name of the function to invoke.

For example, in this modified version of the previous example, we specify a call back function that will be used to print the futures result. This call back function is called `print_future_result()`. It takes the future that has completed as its argument:

```
from concurrent.futures import ThreadPoolExecutor
from time import sleep
from random import randint

def is_even(n):
    print('Checking if', n, 'is even')
    sleep(randint(1, 5))
    return str(n) + ''+ str(n % 2 == 0)

def print_future_result(future):
    print('In callback Future result: ', future.result())

print('Started')
data = [1, 2, 3, 4, 5, 6]

pool = ThreadPoolExecutor(5)
```

```
for v in data:
    future = pool.submit(is_even, v)
    future.add_done_callback(print_future_result)

print('Done')
```

When we run this, we can see that the call back function is called after the main thread has completed. Again, the order is unspecified as the `is_even()` function still sleeps for a random amount of time.

```
Started
Checking if 1 is even
Checking if 2 is even
Checking if 3 is even
Checking if 4 is even
Checking if 5 is even
Done
In callback Future result:  1 False
Checking if 6 is even
In callback Future result:  5 False
In callback Future result:  4 True
In callback Future result:  3 False
In callback Future result:  2 True
In callback Future result:  6 True
```

42.6 Online Resources

See the following online resources for information on Futures:

- https://docs.python.org/3/library/concurrent.futures.html The Python standard Library documentation on Futures.
- https://pymotw.com/3/concurrent.futures The Python Module of the Week page on Futures.
- https://www.blog.pythonlibrary.org/2016/08/03/python-3-concurrency-the-concurrent-futures-module an alternative tutorial on Python Futures.

42.7 Exercises

In mathematics, the factorial of a positive integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example,

```
5! = 5 x 4 x 3 x 2 x 1 = 120
```

Note that the value of 0! is 1,

Write a `Future` that will calculate the factorial of any number with the result being printed out via a call back function.

There are several ways in which the factorial value can be calculated either using a for loop or a recursive function. In either case sleep for a millisecond between each calculation.

Start multiple Futures for different factorial values and see which comes back first.

Chapter 43
Concurrency with AsyncIO

43.1 Introduction

The Async IO facilities in Python are relatively recent additions originally introduced in Python 3.4 and evolving up to and including Python 3.7. They are comprised of two new keywords `async` and `await` (introduced in Python 3.7) and the Async IO Python package.

In this chapter we first discuss Asynchronous IO before introducing the `async` and `await` keywords. We then present Async IO Tasks, how they are created used and managed.

43.2 Asynchronous IO

Asynchronous IO (or Async IO) is a language agnostic concurrent programming model (or paradigm) that has been implemented in several different programming language (such as C# and Scala) as well as in Python.

Asynchronous IO is another way in which you can build concurrent applications in Python. It is in many ways an alternative to the facilities provided by the Threading library in Python. However, whereas the Threading library is more susceptible to issues associated with the GIL (The Global Interpreter Lock) which can affect performance, the Async IO facilities are better insulated from this issue.

The way in which Async IO operates is also lighter weight then the facilities provide day the `multiprocessing` library since the asynchronous tasks in Async IO run within a single process rather than requiring separate processes to be spawned on the underlying hardware.

Async IO is therefore another alternative way of implementing concurrent solutions to problems. It should be noted that it does not build on either Threading or

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_43

Multi Processing; instead Async IO is based on the idea of cooperative multitasking. These cooperating tasks operate asynchronously; by this we mean that the tasks:

- are able to operate separately from other tasks,
- are able to wait for another task to return a result when required,
- and are thus able to allow other tasks to run while they are waiting.

The IO (Input/Output) aspect of the name Async IO is because this form of concurrent program is best suited to I/O bound tasks.

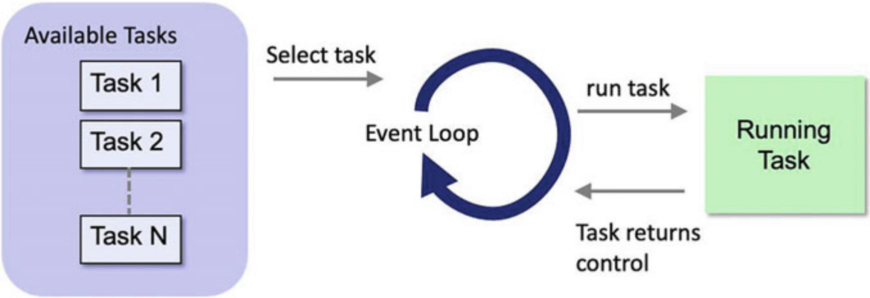
In an I/O bound task a program spends most of its time sending data to, or reading data from, some form of external device (for example a database or set of files, etc.). This communication is time consuming and means that the program spends most of its time waiting for a response from the external device.

One way in which such I/O bound applications can (appear to) speed up is to overlap the execution of different tasks; thus, while one task is waiting for a database to respond with some data, another task can be writing data to a log file etc.

43.3 Async IO Event Loop

When you are developing code using the Async IO facilities you do not need to worry about how the internals of the Async IO library work; however at least at the conceptual level it is useful to understand one key concept; that of the Async IO *Event Loop*; this loop controls how and when each task gets run. For the purposes of this discussion a task represents some work that can be run independently of other pieces of work.

The Event Loop knows about each task to be run and what the state of the task currently is (for example whether it is waiting for something to happen/complete). It selects a task that is *ready to run* from the list of available tasks and executes it. This task has complete control of the CPU until it either completes its work or hands back control to the Event Loop (for example, because it must now wait for some data to be supplied from a database). The Event Loop now checks to see if any of the waiting tasks are ready to continue executing and makes a note of their status. The Event Loop then selects another task that is ready to run and starts that task off. This loop continues until all the tasks have finished. This is illustrated below:



An important point to note in the above description is that a task does not give up the processor unless it decides to, for example by having to wait for something else. They never get interrupted in the middle of an operation; this avoids the problem that two threads might have when being time sliced by a separate scheduler as they may both be sharing the same resource. This can greatly simplify your code.

43.4 The Async and Await Keywords

The `async` keyword, introduced in Python 3.7, is used to mark a function as being something that uses the `await` keyword (we will come back to this below as there is one other use of the `async` keyword). A function that uses the `await` keyword can be run as a separate task and can give up control of the processor when it calls `await` against another `async` function and must *wait* for that function to complete. The invoked `async` function can then run as a separate task, etc.

To invoke an `async` function it is necessary to start the Async IO Event Loop and for that function to be treated as a task by the Event Loop. This is done by calling the `asyncio.run()` method and passing in the root async function.

The `asyncio.run()` function was introduced in Python 3.7 (older versions of Python such as Python 3.6 required you to explicitly obtain a reference to the Event Loop and to run the root async function via that). One point to note about this function is that it has been marked as being provisional in Python 3.7. This means that future versions of Python may or may not support the function or may modify the function in some way. You should therefore check the documentation for the version of Python you are using to see whether the run method has been altered or not.

43.4.1 Using Async and Await

We will examine a very simple Async IO program from the top down. The `main()` function for the program is given below:

```
print('Main - Starting')
asyncio.run(do_something())
print('Main - Done')
```

The `main()` function is the entry point for the program and calls:

```
asyncio.run(do_something())
```

This starts the Async IO Event Loop running and results in the `do_something()` function being wrapped up in a `Task` that is managed by the loop. Note that you do not explicitly create a Task in Async IO; they are always created by some function; however, it is useful to be aware of Tasks as you can interact with them to check their status or to retrieve a result.

The `do_something()` function is marked with the keyword `async`:

```
async def do_something():
    print('do_something - will wait for worker')
    result = await worker()
    print('do_something - result:', result)
```

As previously mentioned this indicates that it can be run as a separate Task and that it can use the keyword `await` to wait for some other function or behaviour to complete. In this case the `do_something()` asynchronous function must wait for the `worker()` function to complete.

The `await` keyword does more than merely indicate that the `do_something()` function must wait for the worker to complete. It triggers another `Task` to be created that will execute the `worker()` function and releases the processor allowing the Event Loop to select the next task to execute (which may or may not be the task running the `worker()` function). The status of the `do_something` task is now *waiting* while the status of the `worker()` task is *ready* (to run).

The code for the worker task is given below:

```
async def worker():
    print('worker - will take some time')
    time.sleep(3)
    print('worker - Done it')
    return 42
```

The `async` keyword again indicates that this function can be run as a separate task. However, this time the body of the function does not use the `await` keyword. This is because this is a special case known as an Async IO *coroutine* function. This is a function that returns a value from a Task (it is related to the idea of a standard Python coroutine which is a data consumer).

Sadly, Computer Science has many examples where the same term has been used for different things as well as examples where different terms have been used for the same thing. In this case to avoid confusion just stick with Async IO coroutines are functions marked with async that can be run as a separate task and *may* call await.

The full listing for the program is given below:

```
import asyncio
import time

async def worker():
    print('worker - will take some time')
    time.sleep(3)
    print('worker - done it')
    return 42

async def do_something():
    print('do_something - will wait for worker')
    result = await worker()
    print('do_something - result:', result)

print('Main - Starting')
asyncio.run(do_something())
print('Main - Done')
```

When this program is executed the output is:

```
Main - Starting
do_something - will wait for worker
worker - will take some time
worker - done it
do_something - result: 42
Main – Done
```

When this is run there is a pause between the two worker printouts as it sleeps, the final output is thus:

```
Main - Starting
do_something - will wait for worker
worker - will take some time
worker - done it
do_something - result: 42
Main - Done
```

Although it is not completely obvious here, the `do_something()` function was run as one task, this task then waited when it got to the `worker()` function which was run as another Task. Once the worker task completed the do_something task could continue and complete its operation. Once this happened the Async IO Event Loop could then terminate as no further tasks were available.

43.5 Async IO Tasks

Tasks are used to execute functions marked with the `async` keyword concurrently. Tasks are never created directly instead they are created implicitly via the keyword `await` or through functions such as `asyncio.run` described above or `asyncio.create_task()`, `asyncio.gather()` and `asyncio.as_completed()`. These additional task creation functions are described below:

- `asyncio.create_task()` This function takes a function marked with `async` and wraps it inside a Task and schedules it for execution by the Async IO Event Loop. This function was added in Python 3.7.
- `asyncio.gather(*aws)` This function runs all the async functions passed to it as separate Tasks. It gathers the results of each separate task together and returns them as a list. The order of the results corresponds to the order of the async functions in the `aws` list.
- `asyncio.as_completed(aws)` Runs each of the async functions passed to it

A Task object supports several useful methods

- `cancel()` cancels a running task. Calling this method will cause the Task to throw a `CancelledError` exception.
- `cancelled()` returns `True` if the Task has been cancelled.
- `done()` returns `True` if the task has completed, raised an exception or was cancelled.
- `result()` returns the result of the Task if it is done. If the Tasks result is not yet available, then the method raises the `InvalidStateError` exception.
- `exception()` return an exception if one was raised by the Task. If the task was cancelled then raises the `CancelledError` exception. If the task is not yet *done*, then raises an `InvalidStateError` exception.

It is also possible to add a callback function to invoke once the task has completed (or to remove such a function if it has been added):

- `add_done_callback(callback)` Add a callback to be run when the Task is done.
- `remove_done_callback(callback)` Remove callback from the callbacks list.

Note that the method is called 'add' rather than 'set' implying that there can be multiple functions called when the task has completed (if required).

The following example illustrates some of the above:

```
import asyncio

async def worker():
    print('worker - will take some time')
    await asyncio.sleep(1)
    print('worker - Done it')
```

```
    return 42

def print_it(task):
    print('print_it result:', task.result())

async def do_something():
    print('do_something - create task for worker')
    task = asyncio.create_task(worker())
    print('do_something - add a callback')
    task.add_done_callback(print_it)
    await task
    # Information on task
    print('do_something - task.cancelled():',
task.cancelled())
    print('do_something - task.done():', task.done())
    print('do_something - task.result():', task.result())
    print('do_something - task.exception():',
task.exception())
    print('do_something - finished')

print('Main - Starting')
asyncio.run(do_something())
print('Main - Done')
```

In this example, the `worker()` function is wrapped within a task object that is returned from the `asyncio.create_task(worker())` call.

A function (`print_it()`) is registered as a callback on the task using the `asyncio.create_task(worker())` function. Note that the worker is passed the task that has completed as a parameter. This allows it to obtain information from the task such as any result generated.

In this example the async function `do_something()` explicitly waits on the task to complete. Once this happens several different methods are used to obtain information about the task (such as whether it was cancelled or not).

One other point to note about this listing is that in the `worker()` function we have added an `await` using the `asyncio.sleep(1)` function; this allows the worker to sleep and wait for the triggered task to complete; it is an Async IO alternative to `time.sleep(1)`.

The output from this program is:

```
Main - Starting
do_something - create task for worker
do_something - add a callback
worker - will take some time
worker - Done it
print_it result: 42
do_something - task.cancelled(): False
do_something - task.done(): True
do_something - task.result(): 42
do_something - task.exception(): None
do_something - finished
Main - Done
```

43.6 Running Multiple Tasks

In many cases it is useful to be able to run several tasks concurrently. There are two options provided for this: the `asyncio.gather()` and the `asyncio.as_completed()` function; we will look at both in this section.

43.6.1 Collating Results from Multiple Tasks

It is often useful to collect all the results from a set of tasks together and to continue only once all the results have been obtained. When using Threads or Processes this can be achieved by starting multiple Threads or Processes and then using some other object such as a Barrier to wait for all the results to be available before continuing. Within the Async IO library all that is required is to use the `asyncio.gather()` function with a list of the async functions to run, for example:

```
import asyncio
import random

async def worker():
    print('Worker - will take some time')
    await asyncio.sleep(1)
    result = random.randint(1,10)
    print('Worker - Done it')
    return result

async def do_something():
    print('do_something - will wait for worker')
    # Run three calls to worker concurrently and collect results
    results = await asyncio.gather(worker(), worker(), worker())
    print('results from calls:', results)

print('Main - Starting')
asyncio.run(do_something())
print('Main - Done')
```

In this program the `do_something()` function uses .

```
results = await asyncio.gather(worker(), worker(), worker())
```

To run three invocations of the `worker()` function in three separate Tasks and to wait for the results of all three to be made available before they are returned as a list of values and stored in the `results` variable.

This makes is very easy to work with multiple concurrent tasks and to collate their results.

Note that in this code example the worker `async` function returns a random number between 1 and 10.

The output from this program is:

```
Main - Starting
```

```
do_something - will wait for worker
Worker - will take some time
Worker - will take some time
Worker - will take some time
Worker - Done it
Worker - Done it
Worker - Done it
results from calls: [5, 3, 4]
Main - Done
```

As you can see from this all three of the worker invocations are started but then release the processor while they sleep. After this the three tasks wake up and complete before the results are collected together and printed out.

43.6.2 Handling Task Results as They Are Made Available

Another option when running multiple Tasks is to handle the results as they become available, rather than wait for all the results to be provided before continuing. This option is supported by the `asyncio.as_completed()` function. This function returns an iterator of async functions which will be served up as soon as they have completed their work.

The for-loop construct can be used with the iterator returned by the function; however within the for loop the code must call await on the async functions returned so that the result of the task can be obtained. For example:

```
async def do_something():
    print('do_something - will wait for worker')
    # Run three calls to worker concurrently and collect results
    for async_func in asyncio.as_completed((worker('A'),
                                            worker('B'),
                                            worker('C'))):
        result = await async_func
        print('do_something - result:', result)
```

Note that the `asyncio.as_completed()` function takes a container such as a tuple of async functions.

We have also modified the worker function slightly so that a label is added to the random number generated so that it is clear which invocation of the worker function return which result:

```
async def worker(label):
    print('Worker - will take some time')
    await asyncio.sleep(1)
    result = random.randint(1,10)
    print('Worker - Done it')
    return label + str(result)
```

When we run this program

```
print('Main - Starting')
asyncio.run(do_something())
print('Main - Done')
```

The output is

```
Main - Starting
do_something - will wait for worker
Worker - will take some time
Worker - will take some time
Worker - will take some time
Worker - Done it
Worker - Done it
Worker - Done it
do_something - result: C2
do_something - result: A1
do_something - result: B10
Main - Done
```

As you can see from this, the results are not returned in the order that the tasks are created, task 'C' completes first followed by 'A' and 'B'. This illustrates the behaviour of the `asyncio.as_completed()` function.

43.7 Online Resources

See the following online resources for information on Futures:

- https://docs.python.org/3/library/asyncio-task.html The Python standard Library documentation on AsyncIO.
- https://pymotw.com/3/asyncio The Python Module of the Week page on AsyncIO.
- https://pythonprogramming.net/asyncio-basics-intermediate-python-tutorial An AsyncIO tutorial.

43.8 Exercises

- This exercise will use the facilities in the AsyncIO library to calculate a set of factorial numbers.

The factorial of a positive integer is the product of all positive integers less than or equal to n. For example,

```
5! = 5 x 4 x 3 x 2 x 1 = 120
```

Note that the value of 0! is 1,

Create an application that will use the `async` and `await` keywords to calculate the factorials of a set of numbers. The factorial function should await for 0.1 of a second (using `asyncio.sleep(0.1)` each time round the loop used to calculate the factorial of a number.

You can use with `asyncio.as_completed()` or `asyncio.gather()` to collect the results up.

You might also use a list comprehension to create the list of calls to the factorial function.

The main function might look like:

```
def main():
    print('Main - Starting')
    asyncio.run(calculate_factorials([5, 7, 3, 6]))
    print('Main - Done')

if __name__ == '__main__':
    main()
```

Chapter 44
Performance Monitoring and Profiling

44.1 Introduction

This chapter considers performance monitoring and profiling of Python programs. It considers why you might want to monitoring performance and memory, and the difference between monitoring and profiling. It then presents tools available for both monitoring and behaviour and memory profiling.

44.2 Why Monitor Performance and Memory?

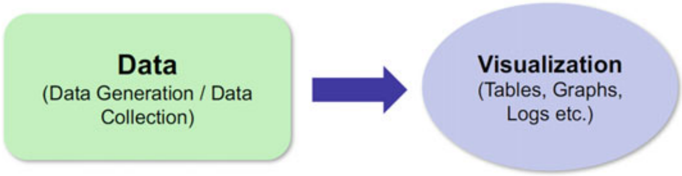
The aim of both performance monitoring and profiling is to collect data on the behaviour of a body of code. This may involve:

- counting the number of times a function is called,
- collecting statistics on memory usage,
- recording the amount of time spent in one or more functions.

This is an important activity as it can help to avoid unnecessary optimisations and instead help to identify actual bottlenecks, memory leaks or inefficient data structures and their use.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_44

There are typically two aspects to performance monitoring and profiling, and these are data collection and visualisation of the results:



Both aspects are supported in Python although open-source/free solutions tend to be less graphical and more report file based.

44.3 Performance Monitoring and Profiling

Performance monitoring and profiling are related concepts but serve different purposes in the context of software analysis.

- **Performance Monitoring** relates to monitoring various aspects of a running application. This can include CPU usage, memory consumption, disk I/O, network activity and other application or system-level metrics. It therefore aims to provide higher-level indicators about the behaviour of a system from the perspective of its use/utilisation of various system and application specific resources. The aim is to help to identify application or system bottlenecks or resource constraint. Performance monitoring is typically carried out using tools and utilities that collect and analyse the application and system-level metrics over time. The focus is on monitoring the health and resource utilisation of the system rather than analysing specific code-level performance.
- **Profiling** is a set of techniques used to analyse in detail the runtime behaviour and characteristics of a specific body of code or application or script. Profiling involves gathering detailed metrics about the function or method calls, memory usage, execution time, execution paths and other relevant data to help identify specific bottlenecks, memory leaks or other performance issues within the codebase. Thus, profiling has a focus that is on the behaviour of the code under consideration rather than the overall system performance. For example, a memory profiling will typically identify how much memory is being used by a particular class and its objects but not necessarily where those objects were created.

In summary, performance monitoring looks at the application and system-level metrics to assess the overall health and resource utilisation of a system, while profiling analyses the behaviour and performance of specific code sections or functions to optimise their execution. Both performance monitoring and profiling are valuable techniques in understanding and improving the performance of a software system, but they operate at different levels of granularity and provide distinct insights.

An important point to note is that profiling may itself introduce overheads or otherwise impact system resources, and thus, it is recommended to profile representative scenarios and avoid profiling in production environments unless necessary.

44.4 Performance Monitoring

There are several built-in modules that support aspects of performance monitoring. In additional there are third-party tools (both those that are open source and free as well as paid for commercial tools) that provide more sophisticated facilities.

The built-in tools include:

- **The time module**. This built-in module can be used to obtain useful time stamps that can be used to determine the time taken for your code to perform different actions.
- **The timeit module**. The `timeit` module provides features that can be used to measure the execution time of code snippets. The `timeit` module is especially useful when comparing the performance of different approaches or algorithms.

There are several third-party tools available for Python performance monitoring. Some popular options include psutils, Py-Spy, New Relic, Datadog, and Prometheus. These tools offer more advanced monitoring and profiling features, including real-time monitoring, visualisation, and integration with monitoring platforms.

There are further tools to consider such as PyCharm Profiler and snakeviz. These tools provide more advanced profiling features, such as visualisations, flame graphs and real-time monitoring.

44.4.1 The Time Module

You can use the `time` module to add behaviour to your code that can be used to generate performance information. This represents *instrumentation* added to the code, that is the actual code base is modified in order to add to the code instruments that can be used to generate monitoring information.

For example, you can use the time module to create a decorator that can be used to time functions. To do this we can use the `time.time()` function to generate a time stamp that can be used to produce the time taken to run a function:

```
from time import time
# decorator to time functions
def timer(func):
    def func_timer(*args, **kwargs):
```

```
        start = time()
        result = func(*args, **kwargs)
        end = time()
        print(func.__name__, 'executed in', end - start)
        return result
    return func_timer

@timer
def sample_function():
    result = 0
    for i in range(100000):
        result = result + (i * i * i)
    return result

sample_function()
```

The output from this code is:

```
sample_function executed in 0.008463859558105469
```

In the above code the `timer()` decorator uses the `time.time()` function to take a timestamp before a function is executed and then after the function finishes. The difference is then reported as the amount of time taken to run the function.

44.4.2 The Timeit Module

It is also possible to use the built-in `timeit` module. This can be used to measure the execution time of snippets of code. The `timeit` module provides the `timeit` and `repeat` functions and the `Timer` class. The `timeit` function takes a function to execute and the number of times to run that function in order to get an indication of its execution time. The syntax for the `timeit` function is:

```
timeit(stmt, setup, timer, number)
```

The other parameters to the `timeit` function are the `setup` parameter which provides any setup code required for the snippet and the `timer` parameter which specifies the timer function to use. By default, the `timeit.default_timer()` function is used, which provides the most accurate timing across different platforms.

The `timeit()` function returns the execution time of the `stmt` in seconds as a floating-point number. This can then be used to generate information, on for example, the average time taken to run the function.

The `repeat()` function is very similar to the `timeit()` function. The difference is that it returns a list of the execution times as indicated by the repeat parameter. The syntax for the `repeat()` functions is:

```
repeat(stmt, setup, timer, repeat, number):
```

Here the repeat parameter indicates how many times to repeat the test.

For example, to test the performance of some function sample_function using timeit and repeat we can write:

```
from timeit import timeit, Timer, repeat

TIMES_TO_RUN_FUNCTION = 1000

def sample_function():
    result = 0
    for i in range(100000):
        result = result + (i * i * i)
    return result

# Using the timer function
time_taken = timeit(sample_function, number=TIMES_TO_RUN_
FUNCTION)
print(f'total time: {time_taken}, average time: {time_taken /
TIMES_TO_RUN_FUNCTION}')

print('-' * 25)
# Using the repeat function
print(repeat(sample_function, repeat=5,
number=TIMES_TO_RUN_FUNCTION))
```

The output from this code is:

```
total time: 8.464858236999135, average time: 0.008464858236999134
- - - - - - - - - - - - -
[8.49980784500076, 8.38068166100129, 8.414855341005023,
8.5682273949933, 8.524605954007711]
```

As you can see from this the `repeat()` function performs the same basic function as the `timeit()` function but repeated in this case 5 times.

The final element of the `timeit` module is the `Timer` class. This class is the class that sits behind the `timeit` function. The `timeit` function is intended to provide a simplified interface to the functionality offered by the `Timer` class. Thus the `Timer` class provides a more flexible and object-oriented approach. Instead of passing the code snippet directly to the `timeit` function, you create an instance of the `Timer` class and specify the code to be timed.

As an example, we can create an instance of the Timer class to time the `sample_function` defined above. We can then call the `timeit()` method on the `Timer` instance in a similar manner to the way in which we called the standalone `timeit()` function. For example:

```
print('-' * 25)
timer = Timer(sample_function)
print(timer.timeit(number=1000))
The output from this code is:
```

The primary advantage of using the `Timer` class is that you can create and reuse the `Timer` object, adjust the setup code and customise the measurement process

```
- - - - - - - - - - - - -
8.36629470900516
```

The `timeit` module has both a command line and a programmatic interface. The above examples utilise the programmatic interface. However we can use the command line interface to interactively time the execution of code. This command line interface can be used as follows:

```
python -m timeit 'statement to time'
```

The command line options available include:

- -n N, -number = N how many times to execute 'statement'
- -s S, -setup = S statement to be executed once initially (default pass)
- -p, -process measure process time, not wallclock time
- -u, -unit = U specify a time unit for timer output; can select nsec, usec, msec or sec
- -v, -verbose print raw timing results; repeat for more digits precision
- -h, -help print a short usage message and exit.

44.4.3 The Psutil Module

The psutil module is a Python cross-platform library used to access system details and process utilities. This module provides facilities that allow you to obtain system-level performance metrics including CPU usage memory usage, Disk I/O and network statistics.

The psutil library can be installed using pip, for example:

```
pip install psutil
```

An example of using the `psutil` library is given below. This code uses the `psutil.cpu_percent()` and `psutil.virtual_memory()` functions. The first function returns information on the CPU utilisation as a percentage. If the `percpu` parameter is `True`, it records the data per CPU on the underlying machine. If the `interval` parameter is > `0` then compares system CPU times elapsed before and after the interval (blocking). The second function, `psutil.virtual_memory()`, returns a set of statistics about system memory usage as a `namedtuple` such as the *total physical memory available* as well as *current available memory* all in bytes.

The code sample is:

```
import psutil as psutil

CPU_usage = psutil.cpu_percent(interval=1, percpu=True)
ram = psutil.virtual_memory()

print(f'CPU: {CPU_usage} ')
print(f'RAM: {ram}')
```

The output from this is:

```
CPU: [26.7, 2.0, 25.3, 1.0, 15.2, 0.0, 11.0, 1.0]
RAM:       svmem(total=17179869184,       available=7709667328,
percent=55.1,         used=9346764800,          free=141074432,
active=7553294336, inactive=7421161472, wired=1793470464)
```

44.5 Python Profiling

Profiling a Python program involves measuring its execution time, memory usage, and identifying performance bottlenecks. There are several built-in tools available with Python to help with profiling; we will discuss cProfile below. There are also other third-party libraries available for profiling in Python including line_profiler and memory_profiler as well as py-spy.

44.5.1 The cProfile Module

The built-in `cProfile` module provides profiling facilities for Python. It is implemented in C and has a minimal impact on your own code.

An alternative is the `profile` module which is implemented in pure python and is therefore cross-platform (in situations where **cProfile** is not available for a specific platform).

There are two ways in which **cProfile** can be used, it can be used either to *instrument* your code (that is add profiling statements to your codebase) or via the *command line* in which case the `-m` option is used with the `python` command to run the `cProfile` module against a whole Python program. For example:

```
python -m cProfile my_program.py
```

This will display the profiling results, including the number of function calls, total time spent in each function, and more.

44.5.2 The Line_Profiler Module

This module allows a developer to profile individual lines of code. It is not a built-in module and as such it is necessary to install it into your runtime environment, for example using pip:

```
pip install line_profiler
```

Once you have done this you can use the `@profile` decorator to monitor functions. Running the program with the `kernprof` script generates the profiling results:

```
kernprof -l my_program.py
```

This will create a `my_program.py.lprof` file containing the line-by-line profiling information.

44.5.3 The Memory_Profiler Module

The memory_profiler module is the memory version of the line_profiler. It allows you to analyse the memory used by a body of code. As with the line_profiler it is necessary to install it into your runtime environment, for example:

```
pip install memory_profiler`
```

as with the line_profiler you can then decorate a target function with @profile. Running the program with the following command generates a set of profiling results:

```
python -m memory_profiler
```

for example:

```
python -m memory_profiler my_program.py
```

The output generates will include the memory usage at different points in the program's execution. The memory_profiler is an open-source project.

44.5.4 Additional Third-Party Libraries

There are several commonly used third-party libraries used for memory profiling including:

- **py-spy**. As well as being used for performance monitoring py-spy can also be used to provide memory profiling information.
- **objgraph**: It is a Python library that specialises in visualising and debugging object references and memory usage.
- **pympler**: It is a library that provides tools for memory profiling and monitoring in Python.

44.6 Profiling with cProfile

The profiling libraries `cProfile` and `Profile` provide for deterministic profiling. It is recommended to use `cProfile` as it is newer than the profile library and is implemented in C and is less invasive with lower overheads.

To illustrate the use of `cProfile` we will use the programmatic interface offered by the cProfile module. To do this we will define a function `fibonacci()` and another function `calculate()`. The `calculate()` function runs the `fibonacci()` function `n` times sleeping for 0.5 ms between each invocation. In the main body of the code we then use `cProfile` to run the calculate function. The results from the analysis are stored into an output file called `profile_results.profile`. Finally the program uses the stats. The Stats class is used for creating reports from data generated by the profiling library classes. It is a "friend" of those classes, and imports data either by direct access to members of the cProfile class, or by reading in a dictionary that was emitted from the Profile class. In this case we are loading data from the file created when we ran the `cProfile.run()` function into the `Stats` instance.

Following this we use some of the functionality on this class to remove directory names from the raw data, sort the statistics by time and then print them out.

```
import cProfile
import pstats

from random import randint
from time import sleep

def fibonacci(n):
    if n < 0:
        print("Incorrect input")
    # First Fibonacci number is 0
    elif n == 1:
        return 0
    # Second Fibonacci number is 1
    elif n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

def calculate(n):
    print('In calculate')
    for _ in range(0, n):
        fibonacci(randint(10,30))
        sleep(0.5)
    print('Done calculate')

print('Running cProfile')
cProfile.run('calculate(5)', 'profile_results.profile')
print('Done profiling')

p = pstats.Stats('profile_results.profile')
p.strip_dirs().sort_stats('time').print_stats()
```

The output from this code is given below (although an additional file the 'profile_results.profile' us created as a side effect of running the program):

```
Running cProfile
In calculate
Done calculate
```

```
Done profiling
Fri May 19 15:17:16 2023     profile_results.profile

         271225 function calls (59 primitive calls) in 2.618 seconds

    Ordered by: internal time


   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    2.511    0.502    2.511    0.502 {built-in method time.sleep}
 271171/5    0.107    0.000    0.107    0.021 main3.py:7(fibonacci)
        1    0.000    0.000    2.618    2.618 main3.py:19(calculate)
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        5    0.000    0.000    0.000    0.000
random.py:235(_randbelow_with_getrandbits)
        1    0.000    0.000    2.618    2.618 {built-in method builtins.exec}
        5    0.000    0.000    0.000    0.000 random.py:284(randrange)
        5    0.000    0.000    0.000    0.000 random.py:358(randint)
       15    0.000    0.000    0.000    0.000 {built-in method _operator.index}
        8    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random'
objects}
        1    0.000    0.000    2.618    2.618 <string>:1(<module>)
        5    0.000    0.000    0.000    0.000 {method 'bit_length' of 'int' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'
objects}
```

This report illustrates the amount of time that was spent within each function, the number of times a function was called and the cumulative time spent in a function, etc. The headings in the report mean:

- **ncalls**: for the number of calls.
- **tottime**: for the total time spent in the given function (and excluding time made in calls to subfunctions).
- **percall**: is the quotient of tottime divided by ncalls.
- **cumtime**: is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.
- **percall**: is the quotient of cumtime divided by primitive calls.
- **filename:lineno(function)**: provides the respective data of each function.

From this we can see that the built-in function `time.sleep` was called 5 times, and we spent a total of 2.511 ms within that function, etc.

It is also possible use the cProfiler in command line mode on a whole Python program or script. To do this we can use the following command style:

```
python -m cProfile [-o outputfile] [-s sortorder] some_program.py
```

where

- -o defines the profile results file instead of standard out
- -s specifies one of the sort_stats() sort values to sort the output by for example 'name'
- -s only applies when –o is not supplied.

44.7 Memory Profiling

At the most basic level you can find out the size of an object using the `sys.getsizeof()` function. This function will tell you in terms of byte how much space something takes up. For example:

```
import sys

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

print('int', sys.getsizeof(1), 'bytes')
print('float:', sys.getsizeof(1.0))
print('tuple:', sys.getsizeof((1, 2, 3)))
print('list:', sys.getsizeof([1, 2, 3]))
print('set:', sys.getsizeof({1, 2, 3}))
print('string:', sys.getsizeof('Hello World'))
print('Person:', sys.getsizeof(Person('Phoebe', 22)))
```

In the above program we are using sys.getsizeof() to determine the size of an int, float, tuple, list, set, string and a Person instance. The actual values returned from this will depend on the version of Python you are using, the more recent versions have reduced the memory overhead of things such as objects. For Python 3.11 the results returned on macOS were:

```
int 28 bytes
float: 24
tuple: 64
list: 88
set: 216
string: 60
Person: 56
```

However, the above is quite primitive in terms of the functionality available and being able to generate an appropriate report on memory usage. For this the module/ library such as the pimple is a better bet. To use the pympler library you need to first install it into your own runtime using for example pip, for example using:

```
pip install pympler
```

Once you have done this you can now memory profile in your own application.

Some simple examples use the `asizeof.asizeof()` function, the tracker and `ClassTracker` class (which tracks the lifetime of an object). For example:

```
from pympler import asizeof

class Person:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
```

```
        return f'Person({self.name})'

data = [1, 2, (3, 4), 'Denise', True, Person('John')]
print(f'asizeof.asizeof(data): {asizeof.asizeof(data)}')

print(f'asizeof.asized(obj,
detail=1).format():\n{asizeof.asized(data, detail=1).format()}')
```

The output from this is:

```
asizeof.asizeof(data): 840
asizeof.asized(obj, detail=1).format():
[1, 2, (3, 4), 'Denise', True, Person(John)] size=840 flat=104
    Person(John) size=464 flat=56
    (3, 4) size=120 flat=56
    'Denise' size=56 flat=56
    1 size=32 flat=32
    2 size=32 flat=32
    True size=32 flat=32
```

The code above illustrates how the `asizeof` module can be used to investigate how much memory certain Python objects consume. In contrast to sys.getsizeof, asizeof sizes objects recursively. You can use one of the `asizeof` functions to get the size of these objects and all associated referents. For example `asizeof.asizeof` or `asizeof.asized`.

Tracking the lifetime of objects of certain classes can be achieved with the ClassTracker. This gives insight into instantiation patterns and helps to understand how specific objects contribute to the memory footprint over time. In the following code we

```
from pympler import classtracker
class Person:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f'Person({self.name})'

tracker = classtracker.ClassTracker()
# Set up which class should be tracked
tracker.track_class(Person)

# Capture statistics
tracker.create_snapshot()
p1 = Person('John')
p2 = Person('Denise')
p3 = Person('Phoebe')
tracker.create_snapshot()

# Generate the report
tracker.stats.print_summary()
```

This code can generate a report on the number of instances of the class Person created and the amount of memory they consumed. The output is:

```
---- SUMMARY ------------------------------------------------------------------
                                          active      0     B       average   pct
  __main__.Person2                              0      0     B       0     B    0%
                                          active      0     B       average   pct
  __main__.Person2                              3      1.20 KB    410     B    0%
-------------------------------------------------------------------------------
```

This shows that initially there were no instance also the class Person present. When the second snapshot was taken there were now three instances of the class and they took up 120 kB of memory.

44.8 Online Resources

- The memory_profiler is an open-source project; for more information see https://pypi.org/project/memory-profiler.
- The documentation for pympler can be found at https://pympler.readthedocs.io/en/latest/.

Part IX
Reactive Programming

Chapter 45
Reactive Programming Introduction

45.1 Introduction

In this chapter we will introduce the concept of Reactive Programming. Reactive Programming is a way of writing programs that allow the system to *reactive to* data being published to it. We will look at the RxPy library which provides a Python implementation of the ReactiveX approach to Reactive programming.

45.2 What Is a Reactive Application?

A Reactive Application is one that must react to data; typically either to the presence of new data, or to changes in existing data. The Reactive manifesto presents the key characteristics of Reactive Systems as:

- **Responsive**. This means that such systems respond in a timely manner. Here of course timely will differ depending upon the application and domain; in one situation a second may be timely in another it may be far too slow.
- **Resilient**. Such systems stay responsive in the face of failure. The systems must therefore be designed to handle failure gracefully and continue to work appropriately following the failure.
- **Elastic**. As the workload grows the system should continue to be responsive.
- **Message Driven**. Information is exchanged between elements of a reactive system using messages. This ensures loose coupling, isolation and location transparency between these components.

As an example, consider an application that lists a set of *Equity Stock Trade* values based on the latest market stick price data. This application might present the current value of each trade within a table. When new market stock price data is published,

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_45

then the application must update the value of the trade within the table. Such an application can be described as being reactive.

Reactive Programming is a programming style (typically supported by libraries) that allows code to be written that follow the ideas of reactive systems. Of course just because part of an application uses a Reactive Programming library does not make the whole application reactive; indeed it may only be necessary for part of an application to exhibit reactive behaviour.

45.3 The ReactiveX Project

ReactiveX is the best-known implementation of the Reactive Programming paradigm.

ReactiveX is based on the *Observer-Observable* design pattern. However it is an extension to this design pattern as it extends the pattern such that the approach supports sequences of data and/or events and adds operators that allow developers to compose sequences together declaratively while abstracting away concerns associated with low-level threads, synchronisation, concurrent data structures and non-blocking I/O.

The ReactiveX project has implementations for many languages including RxJava, RxScala and RxPy; this last is the version we are looking at as it is for the Python language.

RxPy is described as:

> A library for composing asynchronous and event-based programs using Observable collections and query operator functions in Python

45.4 The Observer Pattern

The Observer Pattern is one of the Gang of Four set of patterns. The Gang of Four Patterns (as originally described in the 1995 book entitled '*Design patterns: elements of reusable object-oriented software*' published by Addison-Wesley) are so-called because this book on design patterns was written by four very famous authors, namely Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

The *Observer* pattern provides a way of ensuring that a set of objects are notified whenever the state of another object changes. It has been widely used in a number of languages (such as Smalltalk and Java) and can also be used with Python.

The intent of the *Observer* pattern is to manage a one to many relationship between an object and those objects interested in the state, and in particular state changes, of that object. Thus when the objects state changes, the interested (dependent) objects are notified of that change and can take whatever action is appropriate.

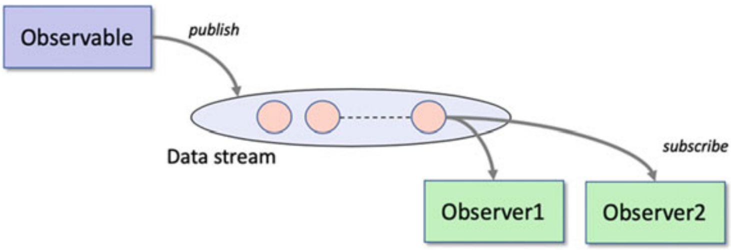
There are two key roles within the Observer Pattern, these are the Observable and the Observer roles.

- **Observable**. This is the object that is responsible for notifying other objects that a change in its state has occurred
- **Observer**. An Observer is an object that will be notified of the change in state of the Observable and can take appropriate action (such as triggering a change in their own state or performing some action).

In addition the state is typically represented explicitly:

- **State**. This role may be played by an object that is used to share information about the change in state that has occurred within the Observable. This might be as simple as a String indicating the new state of the Observable or it might be a data-oriented object that provides more detailed information.

These are illustrated in the following figure.



In the above figure, the Observable object publishes data to a Data Stream. The data in the Data Stream is then sent to each of the Observers registered with the Observable. In this way data is broadcast to all Observers of an Observable.

It is common for an Observable to only publish data once there is an Observer available to process that data. The process of registering with an Observable is referred to as subscribing. Thus an Observable will have zero or more subscribers (Observers).

If the Observable publishes data at a faster rate than can be processed by the Observer then the data is queued via the Data Stream. This allows the Observer to process the data received one at a time at its own pace; without any concern for data loss (as long as sufficient memory is available for the data stream).

45.5 Hot and Cold Observables

Another concept that it is useful to understand is that of Hot and Cold Observables.

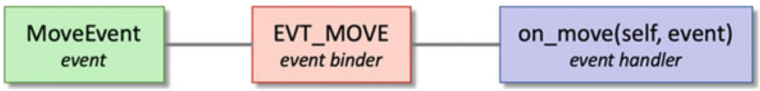
- Cold Observables are lazy Observables. That is, a Cold Observable will only publish data if at least one Observer is subscribed to it.
- Hot Observables, by contrast, publish data whether there is an Observer subscribed or not.

The most common situation in which a Hot Observable is created is when the source producer represents data that may be irrelevant if not processed immediately or may be superseded by subsequent data. For example, data published by a Stock Market Price data feed would fall into this category. When an Observable wraps around this data feed it can publish that data whether or not an Observer is subscribed.

It is important to know whether you have a Hot or Cold Observable because this can impact on what you can assume about the data supplied to the Observers and thus how you need to design your application. If it is important that no data is lost then care is needed to ensure that the subscribers are in place before a Hot Observable starts to publish data (whereas this is not a concern for a Cold Observable).

45.6 Differences Between Event Driven Programming and Reactive Programming

In Event Driven programming, an event is generated in response to something happening; the event then represents this with any associated data. For example, if the user clicks the mouse then an associated `MouseClickEvent` might be generated. This object will usually hold information about the x and y coordinates of the mouse along with which button was clicked, etc. It is then possible to associate some behaviour (such as a function or a method) with this event so that if the event occurs, then the associated operation is invoked and the event object is provided as a parameter. This is certainly the approach used in the `wxPython` library presented earlier in this book:



From the above diagram, when a MoveEvent is generated the `on_move()` method is called and the event is passed into the method.

In the Reactive Programming approach, an Observer is associated with an Observable. Any data generated by the Observable will be received and handled by the Observer. This is true whatever that data is, as the Observer is a handler of data generated by the Observable rather than a handler of a specific type of data (as with the Event driven approach).

Both approaches could be used in many situations. For example, we could have a scenario in which some data is to be processed whenever a stock price changes.

This could be implemented using a `StockPriceChangeEvent` associated with a `StockPriceEventHandler`. It could also be implemented via `StockPriceChangeObserverable` and a `StockPriceChangeObserver`. In either case one element handles the data generated by another element. However, the RxPy library simplifies this process and allows the Observer to run in the same thread as, or a separate thread from, the Observable with just a small change to the code.

45.7 Advantages of Reactive Programming

There are several advantages to the use of a Reactive Programming library these include:

- **It avoids multiple callback methods**. The problems associated with the use of callbacks are sometimes referred to as *callback hell*. This can occur when there are multiple callbacks, all defined to run in response to some data being generated or some operation completing. It can be hard to understand, maintain and debug such systems.
- **Simpler asynchronous, multithreaded execution**. The approach adopted by RxPy makes it very easy to execute operations/behaviour within a multithreaded environment with independent asynchronous functions.
- **Available Operators**. The RxPy library comes prebuilt with numerous operators that make processing the data produced by an Observable much easier.
- **Data Composition**. It is straight forward to compose new data streams (Observables) from data supplied by two or more other Observables for asynchronous processing.

45.8 Disadvantages of Reactive Programming

It is easy to over complicate things when you start to chain operators together. If you use too many operators, or too complex a set of functions with the operators, it can become hard to understand what is going on.

Many developers think that Reactive programming is inherently multithreaded; this is not necessarily the case; in fact RxPy (the library explored in the next two chapters) is single threaded by default. If an application needs the behaviour to execute asynchronously then it is necessary to explicitly indicate this.

Another issue for some Reactive programming frameworks is that it can become memory intensive to store streams of data so that Observers can processes that data when they are ready.

45.9 The RxPy Reactive Programming Framework

The RxPy library is a part of the larger ReactiveX project and provides an implementation of ReactiveX for Python. It is built on the concepts of Observables, Observers, Subjects and operators. In this book we use RxPy version 3.

In the next chapter we will discuss Observables, Observers, Subjects and subscriptions using the RxPy library. The following chapter will explore various RxPy operators.

45.10 Online Resources

See the following online resources for information on reactive programming:

- https://www.reactivemanifesto.org/ The Reactive Manifesto.
- http://reactivex.io/ The ReactiveX home page.
- https://en.wikipedia.org/wiki/Design_Patterns Wikipedia page on Design Patterns book.

Chapter 46
RxPy Observables, Observers and Subjects

46.1 Introduction

In this chapter we will discuss Observables, Observers and Subjects. We also consider how observers may or may not run concurrently. In the remainder of this chapter we look at RxPy version 3 which is a major update from RxPy version 1 (you will therefore need to be careful if you are looking on the web for examples as some aspects have changed; most notably the way in which operators are chained).

46.2 RxPy Library

You will need to install the Rx library into your environment using for example pip:

```
pip install rx
```

At the time of writing the current version is 3.2.0, and this is the version that has been used to test the examples in this chapter.

46.3 Observables in RxPy

An `Observable` is a Python class that publishes data so that it can be processed by one or more Observers (potentially running in separate threads).

An `Observable` can be created to publish data from static data or from dynamic sources. Observables can be chained together to control how and when data is published, to transform data before it is published and to restrict what data is actually published.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_46

For example, to create an `Observable` from a list of values we can use the `rx.from_list()` function. This function (also known as an RxPy operator) is used to create the new `Observable` object:

```
import rx
Observable = rx.from_list([2, 3, 5, 7])
```

46.4 Observers in RxPy

We can add an Observer to an Observable using the `subcribe()` method. This method can be supplied with a `lambda` function, a named function or an object whose class implements the Observer protocol.

For example, the simplest way to create an Observer is to use a *lambda* function:

```
# Subscribe a lambda function
observable.subscribe(lambda value: print('Lambda Received', value))
```

When the Observable publishes data the `lambda` function will be invoked. Each data item published will be supplied independently to the function. The output from the above subscription for the previous Observable is:

```
Lambda Received 2
Lambda Received 3
Lambda Received 5
Lambda Received 7
```

We can also used a standard or named function as an Observer:

```
def prime_number_reporter(value):
    print('Function Received', value)

# Subscribe a named function
observable.subscribe(prime_number_reporter)
```

Note that it is only the name of the function that is used with the `subscribe()` method (as this effectively passes a reference to the function into the method).

If we now run this code using the previous Observable we get:

```
Function Received 2
Function Received 3
Function Received 5
Function Received 7
```

In actual fact the `subscribe()` method takes four optional parameters. These are:

- `on_next` [Optional] Action to invoke for each data item generated by the Observable.

- `on_error` [Optional] Action to invoke upon exceptional termination of the Observable sequence.
- `on_completed` [Optional] Action to invoke upon graceful termination of the Observable sequence.
- `Observer` [Optional] The object that is to receive notifications. You may subscribe using an Observer or callbacks, not both.

Each of the above can be used as positional parameters or as keyword arguments, for example:

```
# Use lambdas to set up all three functions
observable.subscribe(
    on_next = lambda value: print('Received on_next', value),
    on_error = lambda exp: print('Error Occurred', exp),
    on_completed = lambda: print('Received completed
notification')
)
```

The above code defines three lambda functions that will be called depending upon whether data is supplied by the Observable, if an error occurs or when the data stream is terminated. The output from this is:

```
Received on_next 2
Received on_next 3
Received on_next 5
Received on_next 7
Received completed notification
```

Note that the on_error function is not run as no error was generated in this example.

The final optional parameter to the subscribe() method is an *Observer* object. An Observer object can implement the Observer protocol which has the following methods `on_next()`, `on_completed()` and `on_error()`, for example:

```
class PrimeNumberObserver:
    def on_next(self, value):
        print('Object Received', value)

    def on_completed(self):
        print('Data Stream Completed')

    def on_error(self, error):
        print('Error Occurred', error)
```

Instances of this class can now be used as an Observer via the `subscribe()` method:

```
# Subscribe an Observer object
observable.subscribe(PrimeNumberObserver())
```

The output from this example using the previous Observable is:

```
Object Received 2
Object Received 3
Object Received 5
Object Received 7
Data Stream Completed
```

Note that the `on_completed()` method is also called; however the `on_errror()` method is not called as there were no exceptions generated.

The Observer class must ensure that the methods implemented adhere to the Observer protocol (i.e. That the signatures of the `on_next()`, `on_completed()` and on_error() methods are correct).

46.5 Multiple Subscribers/Observers

An Observer can have multiple Observers subscribed to it. In this case each of the Observers is sent all of the data published by the Observable. Multiple Observers can be registered with an Observable by calling the subscribe method multiple times. For example, the following program has four subscribes as well as on_error and on_completed function registered:

```
# Create an observable using data in a list
observable = rx.from_list([2, 3, 5, 7])

class PrimeNumberObserver:
    """ An Observer class """

    def on_next(self, value):
        print('Object Received', value)

    def on_completed(self):
        print('Data Stream Completed')

    def on_error(self, error):
        print('Error Occurred', error)

def prime_number_reporter(value):
        print('Function Received', value)

print('Set up Observers / Subscribers')

# Subscribe a lambda function
observable.subscribe(lambda  value:  print('Lambda Received',
value))
# Subscribe a named function
observable.subscribe(prime_number_reporter)
# Subscribe an Observer object
observable.subscribe(PrimeNumberObserver())

# Use lambdas to set up all three functions
```

```
observable.subscribe(
     on_next=lambda value: print('Received on_next', value),
     on_error=lambda exp: print('Error Occurred', exp),
     on_completed=lambda: print('Received completed
notification')
)
```

The output from this program is:

```
Create the Observable object
Set up Observers / Subscribers
Lambda Received 2
Lambda Received 3
Lambda Received 5
Lambda Received 7
Function Received 2
Function Received 3
Function Received 5
Function Received 7
Object Received 2
Object Received 3
Object Received 5
Object Received 7
Data Stream Completed
Received on_next 2
Received on_next 3
Received on_next 5
Received on_next 7
Received completed notification
```

Note how each of the subscribe is sent all of the data before the next subscriber is sent their data (this is the default single-threaded RxPy behaviour).

46.6 Subjects in RxPy

A *subject* is both an Observer and an Observable. This allows a subject to receive an item of data and then to republish that data or data derived from it.

For example, imagine a subject that receives stock market price data published by an external (to the organisation receiving the data) source. This subject might add a timestamp and source location to the data before republishing it to other internal Observers.

However, there is a subtle difference that should be noted between a *Subject* and a plain *Observable*. A subscription to an Observable will cause an independent execution of the Observable when data is published. Notice how in the previous section all the messages were sent to a specific Observer before the next Observer was sent any data at all.

However, a Subject shares the publication action with all of the subscribers, and they will therefore all receive the same data item in a chain before the next data item.

In the class hierarchy the `Subject` class is a direct subclass of the `Observer` class.

The following example creates a `Subject` that enriches the data it receives by adding a timestamp to each data item. It then republishes the data item to any Observers that have subscribed to it.

```
import rx
from rx.subject import Subject
from datetime import datetime

source = rx.from_list([2, 3, 5, 7])

class TimeStampSubject(Subject):

    def on_next(self, value):
        print('Subject Received', value)
        super().on_next((value, datetime.now()))

    def on_completed(self):
        print('Data Stream Completed')
        super().on_completed()

    def on_error(self, error):
        print('In Subject - Error Occurred', error)
        super().on_error(error)

def prime_number_reporter(value):
    print('Function Received', value)

print('Set up')

# Create the Subject
subject = TimeStampSubject()

# Set up multiple subscribers for the subject
subject.subscribe(prime_number_reporter)
subject.subscribe(lambda value: print('Lambda Received', value))
subject.subscribe(
    on_next = lambda value: print('Received on_next', value),
    on_error = lambda exp: print('Error Occurred', exp),
    on_completed = lambda: print('Received completed
notification')
)

# Subscribe the Subject to the Observable source
source.subscribe(subject)
```

```
print('Done')
```

Note that in the above program the Observers are added to the Subject before the Subject is added to the source Observable. This ensures that the Observers are subscribed before the Subject starts to receive data published by the Observable. If the Subject was subscribed to the Observable before the Observers were subscribed to the Subject, then all the data could have been published before the Observers were registered with the Subject.

The output from this program is:

```
Set up
Subject Received 2
Function Received (2, datetime.datetime(2023, 6, 16, 14, 19, 57,
276451))
Lambda Received (2, datetime.datetime(2023, 6, 16, 14, 19, 57,
276451))
Received on_next (2, datetime.datetime(2023, 6, 16, 14, 19, 57,
276451))
Subject Received 3
Function Received (3, datetime.datetime(2023, 6, 16, 14, 19, 57,
276481))
Lambda Received (3, datetime.datetime(2023, 6, 16, 14, 19, 57,
276481))
Received on_next (3, datetime.datetime(2023, 6, 16, 14, 19, 57,
276481))
Subject Received 5
Function Received (5, datetime.datetime(2023, 6, 16, 14, 19, 57,
276504))
Lambda Received (5, datetime.datetime(2023, 6, 16, 14, 19, 57,
276504))
Received on_next (5, datetime.datetime(2023, 6, 16, 14, 19, 57,
276504))
Subject Received 7
Function Received (7, datetime.datetime(2023, 6, 16, 14, 19, 57,
276527))
Lambda Received (7, datetime.datetime(2023, 6, 16, 14, 19, 57,
276527))
Received on_next (7, datetime.datetime(2023, 6, 16, 14, 19, 57,
276527))
Data Stream Completed
Received completed notification
Done
```

As can be seen from this output the number 2 is received by all of the Observers once the Subject has added the timestamp.

Finally, note that the `rx.subject` was previously known as the `rx.subjects` module.

46.7 Observer Concurrency

By default RxPy uses a single-threaded model; that is Observables and Observers execute in the same thread of execution. However, this is only the default as it is the simplest approach.

It is possible to indicate that when a Observer subscribes to an Observable that it should run in a separate thread using the `scheduler` keyword parameter on the `subscribe()` method. This keyword is given an appropriate scheduler such as the rx.scheduler.NewThreadScheduler (formerly known as `rx.concurrency.NewThreadScheduler`). This scheduler will ensure that the Observer runs in a separate thread.

To see the difference look at the following two programs. The main difference between the programs is the use of specific schedulers:

```
import rx

# Set up an observable
observable = rx.from_list([2, 3, 5])

# Subscribe three observers
observable.subscribe(lambda v: print('Lambda1 Received', v))
observable.subscribe(lambda v: print('Lambda2 Received', v))
observable.subscribe(lambda v: print('Lambda3 Received', v))
```

The output from this first version is given below:

```
Lambda1 Received 2
Lambda1 Received 3
Lambda1 Received 5
Lambda2 Received 2
Lambda2 Received 3
Lambda2 Received 5
Lambda3 Received 2
Lambda3 Received 3
Lambda3 Received 5
```

The `subscribe()` method takes an optional keyword parameter called `scheduler` that allows a scheduler object to be provided.

Now if we specify a few different schedulers we will see that the effect is to run the Observers concurrently with the resulting output being interwoven:

```
import rx
from rx.scheduler import NewThreadScheduler,
ThreadPoolScheduler, ImmediateScheduler

observable = rx.from_list([2, 3, 5])

observable.subscribe(lambda  v:  print('Lambda1  Received',  v),
scheduler=ThreadPoolScheduler(3))
observable.subscribe(lambda  v:  print('Lambda2  Received',  v),
scheduler=ImmediateScheduler())
```

```
observable.subscribe(lambda v: print('Lambda3 Received', v),
scheduler=NewThreadScheduler())

# As the observable runs in a separate thread we need
# to ensure that the main thread does not terminate
input('Press enter to finish')
```

Note that we have to ensure that the main thread running the program does not terminate (as all the Observables are now running in their own threads) by waiting for user input. The output from this version is:

```
Lambda2 Received 2
Lambda1 Received 2
Lambda2 Received 3
Lambda2 Received 5
Lambda1 Received 3
Lambda1 Received 5
Press enter to finish
Lambda3 Received 2
Lambda3 Received 3
Lambda3 Received 5
```

By default the scheduler keyword on the `subscribe()` method defaults to `None` indicating that the current thread will be used for the subscription to the Observable.

Also note that the `rx.scheduler` module was previously known as `rx.concurrency`.

46.7.1 Available Schedulers

To support different scheduling strategies the RxPy library provides several different schedulers.

The following schedulers are available in RxPy:

- `rx.scheduler.ImmediateScheduler` This schedules an action for immediate execution.
- `rx.scheduler.CurrentThreadScheduler` This schedules activity for the current thread.
- `rx.scheduler.TimeoutScheduler` This scheduler works via a timed callback.
- `rx.scheduler.NewThreadScheduler` creates a scheduler for each unit of work on a separate thread.
- `x.scheduler.ThreadPoolScheduler`. This is a scheduler that utilises a thread pool to execute work. This scheduler can act as a way of throttling the amount of work carried out concurrently.

The `rx.scheduler.eventloop` module also defines the following schedulers:

- `IOLoopScheduler` A scheduler that schedules work via the Tornado I/O main event loop.
- `AsyncIOScheduler` A scheduler that schedules work with AsyncIO.

Note that the package formerly known as `rx.concurrency.mainloopscheduler` has been split into two parts, `rx.scheduler.mainloop` and `rx.scheduler.eventloop`.

46.8 Online Resources

See the following online resources for information on RxPy:

- https://github.com/ReactiveX/RxPY The RxPy Git hub repository.
- https://rxpy.readthedocs.io/en/latest/ Documentation for the RxPy library.
- https://rxpy.readthedocs.io/en/latest/operators.html Lists of the available RxPy operators.

46.9 Exercises

Given the following set of tuples representing stock /. Equity prices:

```
stocks = (('APPL', 12.45), ('IBM', 15.55), ('MSFT', 5.66), ('APPL',
13.33))
```

Write a program that will create an Observable based on the stocks data.

Next subscribe three different observers to the Observable. The first should print out the stock price, the second should print out the name of the stock, and the third should print out the entire tuple.

Chapter 47
RxPy Operators

47.1 Introduction

In this chapter we will look at the types of operators provided by RxPy that can be applied to the data emitted by an Observable.

47.2 Reactive Programming Operators

Behind the interaction between an Observable and an Observer is a data stream. That is the Observable supplies a data stream to an Observer that consumes/processes that stream. It is possible to apply an operator to this data stream that can be used to to filter, transform and generally refine how and when the data is supplied to the Observer.

The operators are mostly defined in the `rx.operators` module, for example `rx.operators.average()`. However it is common to use an alias for this such that the `operators` module is called `op`, such as

```
from rx import operators as op
```

This allows for a short-hand form to be used when referencing an operator, such as `op.average()`.

Many of the RxPy operators execute a function which is applied to each of the data items produced by an Observable. Others can be used to create an initial Observable (indeed you have already seen these operators in the form of the `from_list()` operator). Another set of operators can be used to generate a result based on data produced by the Observable (such as the `sum()` operator).

In fact RxPy provides a wide variety of operators, and these operators can be categorised as follows:

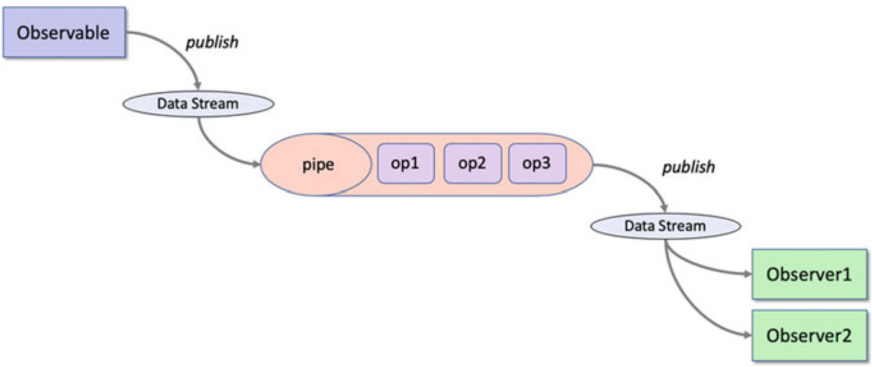
J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_47

- Creational,
- Transformational,
- Combinatorial,
- Filters,
- Error handlers,
- Conditional and Boolean operators,
- Mathematical,
- Connectable.

Examples of some of these categories are presented in the rest of this section.

47.3 Piping Operators

To apply an operator other than a creational operator to an Observable it is necessary to create a pipe. A pipe is essentially a series of one or more operations that can be applied to the data stream generated by the Observable. The result of applying the pipe is that a new data stream is generated that represents the results produced following the application of each operator in turn. This is illustrated below:



To create a pipe the `Observable.pipe()` method is used. This method takes a comma delimited list of one or more operators and returns a data stream. Observers can then subscribe to the pipe's data stream. This can be seen in the examples given in the rest of this chapter for transformations, filters, mathematical operators, etc.

47.4 Creational Operators

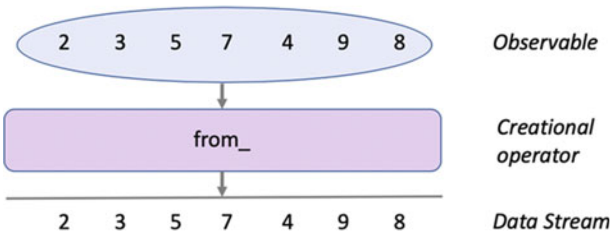
You have already seen an example of a creational operator in the examples presented earlier in this chapter. This is because the `rx.from_list()` operator is an example of a creational operator. It is used to create a new Observable based on data held in a *list like* structure.

A more generic version of `from_list()` is the `from_()` operator. This operator takes an iterable and generates an Observable based on the data provided by the iterable. Any object that implements the iterable protocol can be used including user defined types. There is also an operator `from_iterable()`. All three operators do the same thing and you can choose which to use based on which provides the most semantic meaning in your context.

All three of the following statements have the same effect:

```
source = rx.from_([2, 3, 5, 7])
source = rx.from_iterable([2, 3, 5, 7])
source = rx.from_list([2, 3, 5, 7])
```

This is illustrated pictorially below:



Another creational operator is the rx.range() operator. This operator generates an observable for a range of integer numbers. The range can be specified with our without a starting value and with or within an increment. However the maximum value in the range must always be provided, for example:

```
obs1 = rx.range(10)
obs2 = rx.range(0, 10)
obs3 = rx.range(0, 10, 1)
```

47.5 Transformational Operators

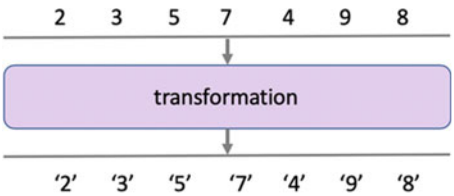
There are several transformational operators defined in the rx.operators module including `rx.operators.map()` and `rx.operators.flat_map()`.

The `rx.operators.map()` operator applies a function to each data item generated by an Observable.

The `rx.operators.flat_map()` operator also applies a function to each data item but then applies a *flatten* operation to the result. For example, if the result is a list of lists then `flat_map` will flatten this into a single list. In this section we will focus on the `rx.operators.map()` operator.

The `rx.operators.map()` operator allows a function to be applied to all data items generated by an Observable. The result of this function is then returned as the result of the map() operators Observable. The function is typically used to perform some form of transformation to the data supplied to it. This could be adding one to all integer values, converting the format of the data from XML to JSON, enriching the data with additional information such as the time the data was acquired and who the data was supplied by etc.

In the example given below we are transforming the set of integer values supplied by the original `Observable` into strings. In the diagram these strings include quotes around them to highlight they are in fact a string:



This is quite typical of the use of a transformation operator; that is to change the data from one format to another or to add information to the data.

The code used to implement this scenario is given below. Note the use of the `pipe()` method to apply the operator to the data stream generated by the Observable:

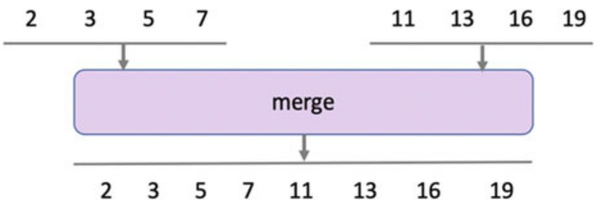
```
# Apply a transformation to a data source to convert
# integers into strings
import rx
from rx import operators as op

# Set up a source with a map function
source = rx.from_list([2, 3, 5, 7]).pipe(
    op.map(lambda value: "'" + str(value) + "'")
)

# Subscribe a lambda function
source.subscribe(lambda value: print('Lambda Received',
                                     value,
                                     ' is a string ',
                                     isinstance(value, str)))
```

The output from this program is:

```
Lambda Received '2' is a string True
Lambda Received '3' is a string True
Lambda Received '5' is a string True
Lambda Received '7' is a string True
```

47.6 Combinatorial Operators

Combinatorial operators combine together multiple data items in some way. One example of a combinatorial operator is the `rx.merge()` operator. This operator merges the data produced by two Observables into a single Observable data stream. For example:



In the above diagram two Observables are represented by the sequence 2, 3, 5, 7 and the sequence 11, 13, 16, 19. These Observables are supplied to the merge operator that generates a single Observable that will supply data generated from both of the original Observables. This is an example of an operator that does not take a function but instead takes two Observables.

The code representing the above scenario is given below:

```
# An example illustrating how to merge two data sources
import rx

# Set up two sources
source1 = rx.from_list([2, 3, 5, 7])
source2 = rx.from_list([10, 11, 12])

# Merge two sources into one
rx.merge(source1, source2)\
     .subscribe(lambda v: print(v, end=','))
```

Notice that in this case we have subscribed directly to the Observable returned by the merge() operator and have not stored this in an intermediate variable (this was a design decision and either approach is acceptable).

The output from this program is presented below:

```
2,3,5,7,10,11,12,
```

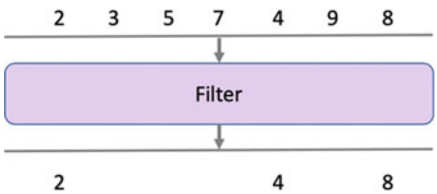
Notice from the output the way in which the data held in the original Observables is intertwined in the output of the Observable generated by the merge() operator.

47.7 Filtering Operators

There are several operators in this category including `rx.operators.filter()`, `rx.operators.first()`, `rx.operators.last()` and `rx.operators.distinct()`.

The `filter()` operator only allows those data items to pass through that pass some test expression defined by the function passed into the filter. This function must return `True` or `False`. Any data item that causes the function to return `True` is allowed to pass through the filter.

For example, let us assume that the function passed into `filter()` is designed to only allow even numbers through. If the data stream contains the numbers 2, 3, 5, 7, 4, 9 and 8 then the `filter()` will only emit the numbers 2, 4 and 8. This is illustrated below:



The following code implements the above scenario:

```
# Filter source for even numbers
import rx
from rx import operators as op

# Set up a source with a filter
source = rx.from_list([2, 3, 5, 7, 4, 9, 8]).pipe(
    op.filter(lambda value: value % 2 == 0)
)

# Subscribe a lambda function
source.subscribe(lambda value: print('Lambda Received', value))
```

In the above code the `rx.operators.filter()` operator takes a lambda function that will verify if the current value is even or not (note this could have been a named function or a method on an object, etc.). It is applied to the data stream generated by the Observable using the pipe() method. The output generated by this example is:

```
Lambda Received 2
Lambda Received 4
Lambda Received 8
```

The `first()` and `last()` operators emit only the first and last data item published by the Observable.

The `distinct()` operator suppresses duplicate items being published by the Observable. For example, in the following list used as the data for the Observable, the numbers 2 and 3 are duplicated:

```
# Use distinct to suppress duplicates
source = rx.from_list([2, 3, 5, 2, 4, 3, 2]).pipe(
    op.distinct()
)

# Subscribe a lambda function
source.subscribe(lambda value: print('Received', value))
```

However, when the output is generated by the program all duplicates have been suppressed:

```
Received 2
Received 3
Received 5
Received 4
```

47.8 Mathematical Operators

Mathematical and aggregate operators perform calculations on the data stream provided by an Observable. For example, the `rx.operators.average()` operator can be used to calculate the average of a set of numbers published by an Observable. Similarly `rx.operators.max()` can select the maximum value, `rx.operators.min()` the minimum value and `rx.operators.sum()` will total all the numbers published, etc.

An example using the `rx.operators.sum()` operator is given blow:

```
# Example of summing all the values in a data stream
import rx
from rx import operators as op
```

```
# Set up a source
observable = rx.from_list([2, 3, 5, 7])
# Apply sum function to initial source
observable2 = observable.pipe(
    op.sum()
)

# Subscribe to the result generated by sum
observable2.subscribe(lambda v: print(v))
```

The output from the `rx.operators.sum()` operator is the total of the data items published by the Observable (in this case the total of 2, 3, 5 and 7). The Observer function that is subscribed to the `rx.operators.sum()` operators Observable will print out this value:

```
17
```

However, in some cases it may be useful to be notified of the intermediate running total as well as the final value so that other operators down the chain can react to these subtotals. This can be achieved using the `rx.operators.scan()` operator. The `rx.operators.scan()` operator is actually a transformational operator but can be used in this case to provide a mathematical operation. The `scan()` operator applies a function to each data item published by an Observable and generates its own data item for each value received. Each generated value is passed to the next invocation of the `scan()` function as well as being published to the `scan()` operators Observable data stream. The running total can thus be generated from the previous subtotal and the new value obtained. This is shown below:

```
import rx
from rx import operators as op
# Rolling or incremental sum
rx.from_([2, 3, 5, 7]).pipe(
        op.scan(lambda subtotal, i: subtotal+i)
).subscribe(lambda v: print(v))
```

The output from this example is:

```
2
5
10
17
```

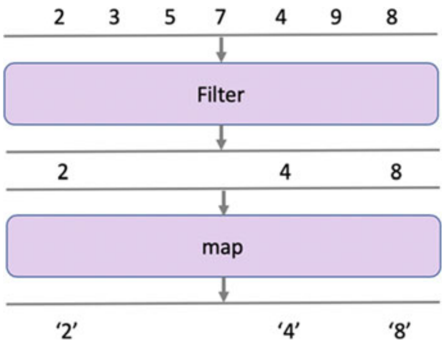
This means that each subtotal is published as well as the final total.

47.9 Chaining Operators

An interesting aspect of the RxPy approach to data stream processing is that it is possible to apply multiple operators to the data stream produced by an Observable.

The operators discussed earlier actually return another Observable. This new Observable can supply its own data stream based on the original data stream and the result of applying the operator. This allows another operator to be applied in sequence to the data produced by the new Observable. This allows the operators to be chained together to provide sophisticated processing of the data published by the original Observable.

For example, we might first start off by filtering the output from an Observable such that only certain data items are published. We might then apply a transformation in the form of a `map()` operator to that data, as shown below:



Note the order in which we have applied the operators; we first filter out data that is not of interest and then apply the transformation. This is more efficient than apply the operators the other way around as in the above example we do not need to transform the odd values. It is therefore common to try and push the filter operators as high up the chain as possible.

The code used to generate the chained set of operators is given below. In this case we have used lambda functions to define the `filter()` function and the `map()` function. The operators are applied to the Observable obtained from the list supplied. The data stream generated by the Observable is processed by each of the operators defined in the pipe. As there are now two operators the pipe contains both operators and acts a pipe down which the data flows.

The list used as the initial source of the Observables data contains a sequence of event and odd numbers. The `filter()` function selects only even numbers and the `map()` function transforms the integer values into strings. We then subscribe an Observer function to the Observable produced by the transformational `map()` operator.

```
# Example of chaining operators together
import rx
from rx import operators as op
```

```
# Set up a source with a filter
source = rx.from_list([2, 3, 5, 7, 4, 9, 8])
pipe = source.pipe(
     op.filter(lambda value: value % 2 == 0),
     op.map(lambda value: "'" + str(value) + "'")
)

# Subscribe a lambda function
pipe.subscribe(lambda value: print('Received', value))
```

The output from this application is given below:

```
Received '2'
Received '4'
Received '8'
```

This makes it clear that only the three even numbers (2, 4 and 8) are allowed through to the `map()` function.

47.10 Online Resources

See the following online resources for information on RxPy:

- https://rxpy.readthedocs.io/en/latest/ Documentation for the RxPy library.
- https://rxpy.readthedocs.io/en/latest/operators.html Lists of the available RxPy operators.

47.11 Exercises

Given the following set of tuples representing stock /. Equity prices:

```
stocks = (('APPL', 12.45), ('IBM', 15.55), ('MSFT', 5.66), ('APPL',
13.33))
```

Provide solutions to the following:

- Select all the 'APPL' stocks.
- Select all stocks with a price over 15.00
- Find the average price of all 'APPL' stocks.

Now use the second set of tuples and merge them with the first set of stock prices:

```
stocks2 = (('GOOG', 8.95), ('APPL', 7.65), ('APPL', 12.45),
('MSFT', 5.66), ('GOOG', 7.56), ('IBM', 12.76))
```

Convert each tuple into a list and calculate how much 25 shares in that stock would be, print this out as the result).

- Find the highest value stock.
- Find the lowest value stock.
- Only publish unique data times (i.e. suppress duplicates).

Part X
Network Programming

Chapter 48
Introduction to Sockets and Web Services

48.1 Introduction

In the following two chapters we will explore socket-based and web service approaches to interprocess communications. These processes may be running on the same computer or different computers on the same local area network or may be geographically far apart. In all cases information is sent by one program running in one process to another program running in a separate process via internet sockets.

48.2 Sockets

Socket-based communication, or rather Internet Protocol (IP) sockets provide a programming interface to the network protocol stack that is managed by the underlying operating system. Using such an API means that the programmer is abstracted away from the low-level details of how data is exchanged between processes on (potentially) different computers and can instead focus on the higher level aspects of their solution.

There are a number of different types of IP socket available; however the focus in this book is on *Stream Sockets*. A stream socket uses the Transmission Control Protocol (TCP) to send messages. Such a socket is often referred to as a TCP/IP socket.

TCP provides for ordered and reliable transmission of data across the connection between two devices (or hosts). This can be important as TCP guarantees that for every message sent; that every message will not only arrive at the receiving host, but that the messages will arrive in the correct order.

A common alternative to the TCP is the User Datagram Protocol (or UDP). UDP does not provide any delivery guarantees (that is messages can be lost or may arrive out of order). However, UDP is a simpler protocol and can be particularly useful for

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_48

broadcast systems, where multiple clients may need to receive the data published by a server host (particularly if data loss is not an issue).

48.3 Web Services

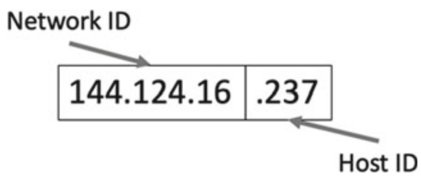
A Web Service is a service offered by a host computer that can be invoked by a remote client using the Hypertext Transfer Protocol (HTTP). HTTP can be run over any reliable stream transport protocol, although it is typically used over TCP/IP. It was originally designed to allow data to be transferred between a HTTP server and a web browser so that the data could be presented in a human readable form to a user. However, when used with a web service it is used to support program to program communication between a client and a server using machine-readable data formats. Currently this format is most typically Java Script Object Notation (JSON) although in the past eXtensible Markup Language (XML) was often used.

48.4 Addressing Services

Every device (host) connected to the internet has a unique identity (we are ignoring private networks here). This unique identity is represented as an IP address. Using an IP address we can connect a socket to a specific host anywhere on the internet. It is therefore possible to connect to a whole range of device types in this way from printers to cash tills to fridges as well as servers, mainframes and PCs, etc.

IP addresses have a common format such as `144.124.16.237`. An IP version 4 address is always a set of four numbers separated by full stops. Each number can be in the range 0–255, so the full range of IP addresses is from 0.0.0.0 to 255.255.255.255.

An IP address can be divided up into two parts; the part indicating the network on which the host is connected and the host's ID, for example:



Thus:

- The Network ID elements of the IP address identify the specific network on which the host is currently located.

- The Host ID is the part of the IP address that specifies a specificities device on the network (such as your computer).

On any given network there may be multiple hosts, each with their own host ID but with a shared network ID. For example, on a private home network there may be:

- `192.168.1.1` Natalia's laptop
- `192.168.1.2` Adam's PC
- `192.168.1.3` Home Printer
- `192.168.1.4` Smart TV.

In many ways the network id and host id elements of an IP address are like the postal address for a house on a street. The street may have a name, for example Coleridge Avenue and there may be multiple houses on the street. Each house has a unique number; thus 10 Coleridge Avenue is uniquely differentiated from 20 Coleridge Avenue by the house number.

At this point you may be wondering where the URLs you see in your web browser come into play (such as www.bbc.co.uk). These are textual names that actually map to an IP address. The mapping is performed by something called a Domain Name System (or DNS) server. A DNS server acts as a lookup service to provide the actual IP address for a particular textual URL name. The presence of an english textual version of a host address is because humans are better at remembering (a hopefully) meaningful name rather than what might appear to be a random sequence of numbers.

There are several web sites that can be used to see these mappings (and one is given at the end of this chapter). Some examples of how the english textual name maps to an IP address are given below:

- www.aber.ac.uk maps to 144.124.16.237
- www.uwe.ac.uk maps to 164.11.132.96
- www.bbc.net.uk maps to 212.58.249.213
- www.gov.uk maps to 151.101.188.144.

Note that these mappings were correct at the time of writing; they can change as new entries can be provided to the DNS servers causing a particular textual name to map to a different physical host.

48.5 Localhost

There is a special IP address which is usually available on a host computer and is very useful for developers and testers. This is the IP address:

```
127.0.0.1
```

It is also known as *localhost* which is often easier to remember.

Localhost (and `127.0.0.1`) is used to refer to the computer you are currently on when a program is run; that is it is your local host computer (hence the name localhost).

For example, if you start up a socket server on your local computer and want a client socket program, running on the same computer, to connect to the server program; you can tell it to do so by getting it to connect to localhost.

This is particularly useful when either you don't know the IP address of your local computer or because the code may be run on multiple different computers each of which will have their own IP address. This is particularly common if you are writing test code that will be used by developers when running their own tests on different developer (host) machines.

We will be using localhost in the next two chapters as a way of specifying where to look for a server program.

48.6 Port Numbers

Each internet device/host can typically support multiple processes. It is therefore necessary to ensure that each process has its own channel of communications. To do this each host has available to it multiple ports that a program can connect too. For example port 80 is often reserved for HTTP web servers, while port 25 is reserved for SMTP servers. This means that if a client wants to connect to a HTTP server on a particular computer then it must specify port 80 not port 25 on that host.

A port number is written after the IP address of the host and separated from the address by a colon, for example:

- www.aber.ac.uk:80 indicates port 80 on the host machine which will typically be running a HTTP server, in this case for Aberystwyth University.
- localhost:143 this indicates that you wish to connect to port 143 which is typically reserved for an IMAP (Internet Message Access Protocol) server on your local machine.
- www.uwe.ac.uk:25 this indicates port 25 on a host running at the University of the West of England, Bristol. Port 25 is usually reserved for Simple Mail Transfer Protocol (SMTP) servers.

Port numbers in the IP system are 16 bit numbers in the range 0–65,536. Generally, port numbers below 1024 are reserved for predefined services (which means that you should avoid using them unless you wish to communicate with one of those services such as telnet, SMTP mail, ftp, etc.). Therefore it is typically to choose a port number above 1024 when setting up your won services.

48.7 IPv4 Versus IPv6

What we have described in this chapter in terms of IP addresses is in fact based on the Internet Protocol version 4 (aka IPv4). This version of the Internet Protocol was developed during the 1970s and published by the Internet Engineering Task Force (IETF) in September 1981 (replacing an earlier definition published in January 1980). This version of the standard uses 32 binary bits for each element of the host address (hence the range of 0 to 255 for each of their parts of the address). This provides a total of 4.29 billion possible unique addresses. This seemed a huge amount in 1981 and certainly enough for what was imagined at the time for the internet.

Since 1981 the internet has become the backbone to not only the World Wide Web itself, but also to the concept of the Internet of Things (in which every possible device might be connected to the internet from your fridge, to your central heating system to your toaster). This potential explosion in internet addressable devices/hosts leads in the mid-1990s to concerns about the potential lack of internet addresses using IPv4. The IETF therefore designed a new version of the Internet Protocol; Internet Protocol version 6 (or IPv6). This was ratified as an Internet Standard in July 2017.

IPv6 uses a 128 bit address for each element in a hosts address. It also uses eight number groups (rather than 4) which are separated by a colon. Each number group has four hexadecimal digits.

The following illustrates what an IPv6 address looks like:

```
2001:0DB8:AC10:FE01:EF69:B5ED:DD57:2CLE
```

Uptake of the IPv6 protocol has been slower than was originally expected; this is in part because the IPv4 and IPv6 have not been designed to be interoperable but also because the utilisation of the IPv4 addresses has not been as fast as many originally feared (partly due to the use of private networks). However, over time this is likely to change as more organisations move over to using the IPv6.

48.8 Sockets and Web Services in Python

The next two chapters discuss how sockets and web services can be implemented in Python. The first chapter discusses both general sockets and HTTP server sockets. The second chapter looks at how the Flask library can be used to create web services that run over HTTP using TCP/IP sockets.

48.9 Online Resources

See the following online resources for information

- https://en.wikipedia.org/wiki/Network_socket Wikipedia page on Sockets.
- https://en.wikipedia.org/wiki/Web_service Wikipedia page on Web Services.
- https://codebeautify.org/website-to-ip-address Provides mappings from URLs to IP addresses.
- https://en.wikipedia.org/wiki/IPv4 Wikipedia page on IPv4.
- https://en.wikipedia.org/wiki/IPv6 Wikipedia page on IPv6.
- https://www.techopedia.com/definition/28503/dns-server For an introduction to DNS.

Chapter 49
Sockets in Python

49.1 Introduction

A *socket* is an end point in a communication link between separate processes. In Python, sockets are objects which provide a way of exchanging information between two processes in a straightforward and platform independent manner.

In this chapter we will introduce the basic idea of socket communications and then present a simple socket server and client application.

49.2 Socket to Socket Communication

When two operating system level processes wish to communicate, they can do so via sockets. Each process has a socket which is connected to the other sockets. One process can then write information out to the socket, while the second process can read information in from the socket.

Associated with each socket are two streams, one for input and one for output. Thus, to pass information from one process to another, you write that information out to the output stream of one socket object and read it from the input stream of another socket object (assuming the two sockets are connected).

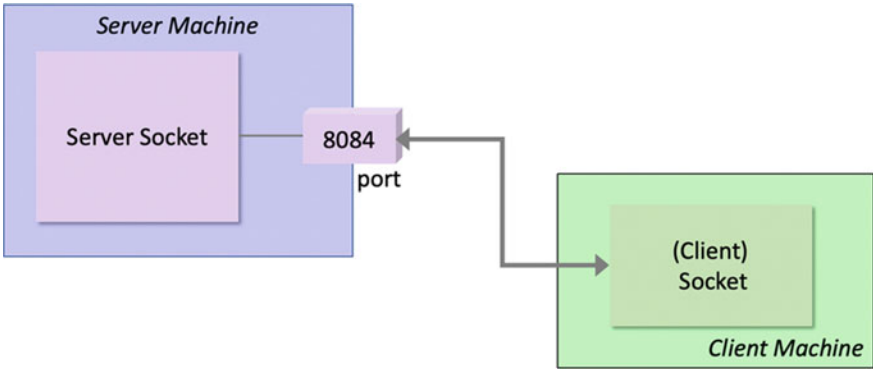
Several different types of sockets are available; however in this chapter we will focus on TCP/IP sockets. Such a socket is a connection-oriented socket that will provide a guarantee of delivery of data (or notification of the failure to deliver the data). TCP/IP, or the Transmission Control Protocol/Internet Protocol, is a suite of communication protocols used to interconnect network devices on the Internet or in a private Intranet. TCP/IP actually specifies how data is exchanged between programs over the Internet by providing end-to-end communications that identify how the data should be broken down into packets, addressed, transmitted, routed and received at the destination.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_49

49.3 Setting up a Connection

To set up the connection, one process must be running a program that is waiting for a connection, while the other must try to connect up to the first program. The first is referred to as a server socket while the second just as a socket.

For the second process to connect to the first (the server socket) it must know what machine the first is running on and which port it is connected to.



For example, in the above diagram the server socket connects to port 8084. In turn the client socket connects to the machine on which the server is executing and to port number 8084 on that machine. Nothing happens until the server socket accepts the connection. At that point the sockets are connected, and the socket streams are bound to each other. This means that the server's output stream is connected to the client socket input stream and vice versa.

49.4 An Example Client Server Application

49.4.1 The System Structure

The above diagram illustrates the basic structure of the system we are trying to build. There will be a server object running on one machine and a client object running on another. The client will connect up to the server using sockets in order to obtain information.

The actual application being implemented in this example is an address book look up application. The addresses of employees of a company are held in a dictionary. This dictionary is set up in the server program but could equally be held in a database, etc. When a client connects up to the server, it can obtain an employees' office address.

49.4.2 Implementing the Server Application

We shall describe the server application first. This is the Python application program that will service requests from client applications. To do this, it must provide a server socket for clients to connect to. This is done by first binding a server socket to a port on the server machine. The server program must then listen for incoming connections.

The listing presents the source code for the `Server` program.

```
import socket

def main():
    # Setup names and offices
    addresses = {'JOHN': 'C45',
                 'DENISE': 'C44',
                 'PHOEBE': 'D52',
                 'ADAM': 'B23'}

    print('Starting Server')
    print('Create the socket')
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    print('Bind the socket to the port')
    server_address = ('localhost', 8084)
    print('Starting up on', server_address)
    sock.bind(server_address)

    # specifies the number of connections
    # before refusing new connections.
    print('Listen for incoming connections')
    sock.listen(1)
    while True:
        print('Waiting for a connection')
        connection, client_address = sock.accept()
        try:
            print('Connection from', client_address)
            while True:
                data = connection.recv(1024).decode()
                print('Received: ', data)
                if data:
                    key = str(data).upper()
                    response = addresses[key]
                    print('sending data back to the client: ',
                              response)
                    connection.sendall(response.encode())
                else:
                    print('No more data from', client_address)
                    break
        finally:
            connection.close()

if __name__ == '__main__':
    main()
```

The server in the above listing sets up the `addresses` to contain a dictionary of the names and addresses.

It then waits for a client to connect to it. This is done by creating a socket and binding it to a specific port (in this case port 8084) using:

```
print('Create the socket')
sock = socket.socket(socket.AF_INET,
                     socket.SOCK_STREAM)
print('Bind the socket to the port')
server_address = ('localhost',
                  8084)
```

The construction of the socket object is discussed in more detail in the next section. Note that you could use `socket.gethostname()` instead of 'localhost' as this only works if both the server and the line tare on the same machine—which is fine for testing but unlikely to work in the real world. For example you could rewrite the above as:

```
server_address = (socket.gethostname(),
                  8084)
```

Next the server listens for a connection from a client. Note that the `sock.listen()` method takes the value 1 indicating that it will handle one connection at a time.

An infinite loop is then set up to run the server. When a connection is made from a client, both the connection and the client address are made available. While there is data available from the client, it is read using the recv function. Note that the data received from the client is assumed to be a string. This is then used as a key to look the address up in the address dictionary.

Once the address is obtained it can be sent back to the client. In Python 3 it is necessary to `decode()` and `encoded()` the string format the raw data transmitted via the socket streams.

Note you should always *close* a socket when you have finished with it.

49.4.3 Socket Types and Domains

When we created the socket class above, we passed in two arguments to the socket constructor:

```
socket(socket.AF_INET, socket.SOCK_STREAM)
```

To understand the two values passed into the `socket()` constructor, it is necessary to understand that sockets are characterised according to two properties: their *domain* and their *type*.

The *domain* of a socket essentially defines the communications protocols that are used to transfer the data from one process to another. It also incorporates how sockets are named (so that they can be referred to when establishing the communication).

Two *standard* domains are available on Unix systems; these are AF_UNIX which represents intra-system communications, where data is moved from process to process through kernel memory buffers. AF_INET represents communication using the TCP/IP protocol suite; in which processes may be on the same machine or on different machines.

- A socket's *type* indicates how the data is transferred through the socket. There are essentially two options here.
- Datagram which sockets support a *message-based* model where no connection is involved, and communication is not guaranteed to be reliable.
- Stream sockets that support a *virtual circuit* model, where data is exchanged as a byte stream and the connection is reliable.

Depending on the domain, further socket types may be available, such as those that support message passing on a reliable connection.

49.4.4 Implementing the Client Application

The client application is essentially a very simple program that creates a link to the server application. To do this, it creates a socket object that connects to the servers' host machine, and in our case this socket is connected to port 8084.

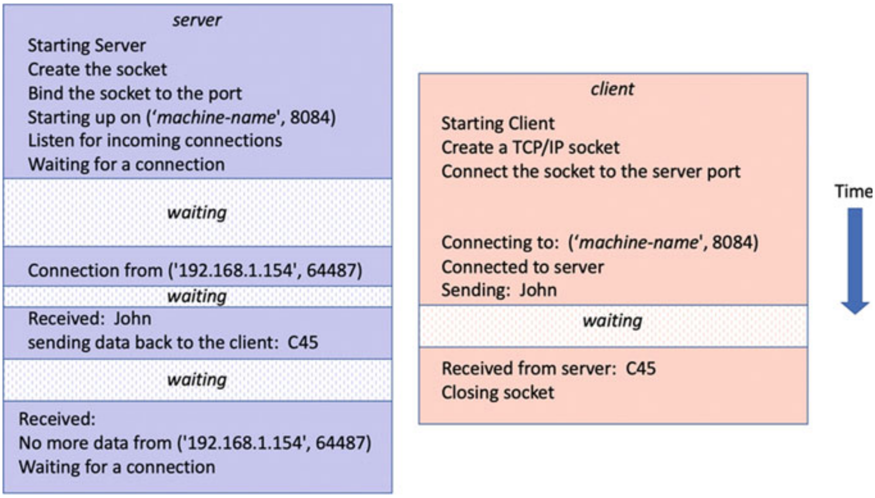
Once a connection has been made the client can, then send the encoded message string to the server. The server will then send back a response which the client must decode. It then closes the connection.

The implementation of the client is given below:

```
import socket

def main():
    print('Starting Client')
    print('Create a TCP/IP socket')
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print('Connect the socket to the server port')
    server_address = ('localhost', 8084)
    print('Connecting to: ', server_address)
    try:
        sock.connect(server_address)
        print('Connected to server')
        print('Send data')
        message = 'John'
        print('Sending: ', message)
        sock.send(message.encode())
        data = sock.recv(1024).decode()
        print('Received from server: ', data)
    except ConnectionRefusedError as err:
        print(f'Error connecting to server: {err}')
    except Exception as exp:
        print(f'Something went wrong: {exp}')
```

```
        finally:
            print('Closing socket')
            sock.close()

if __name__ == '__main__':
    main()
```

The output from the two programs needs to be considered together.



As you can see from this diagram, the server waits for a connection from the client. When the client connects to the server, the server waits to receive data from the client. At this point the client must wait for data to be sent to it from the server. The server then sets up the response data and sends it back to the client. The client receives this and prints it out and closes the connection. In the meantime, the server has been waiting to see if there is any more data from the client; as the client closes the connection the server knows that the client has finished and returns to waiting for the next connection.

49.5 The Socketserver Module

In the above example, the server code is more complex than the client; and this is for a single threaded server; life can become much more complicated if the server is expected to be a multithreaded server (that is a server that can handle multiple requests from different clients at the same time).

However, the `serversocket` module provides a more convenient, object-oriented approach to creating a server. Much of the *boiler plate* code needed in such applications is defined in classes, with the developer only having to provide their own classes or override methods to define the specific functionality required.

There are five different server classes defined in the `socketserver` module.

- `BaseServer` is the root of the server class hierarchy; it is not really intended to be instantiated and used directly. Instead it is extended by `TCPServer` and other classes.
- `TCPServer` uses TCP/IP sockets to communicate and is probably the most commonly used type of socket server.
- `UDPServer` provides access to datagram sockets.
- `UnixStreamServer` and `UnixDatagramServer` use Unix-domain sockets and are only available on Unix platforms.

Responsibility for processing a request is split between a *server* class and a *request handler* class. The server deals with the *communication* issues (listening on a socket and port, accepting connections, etc.), and the request handler deals with the *request* issues (interpreting incoming data, processing it, sending data back to the client).

This division of responsibility means that in many cases you can simply use one of the existing server classes without any modifications and provide a custom request handler class for it to work with.

The following example defines a request handler that is plugged into the `TCPServer` when it is constructed. The request handler defines a method `handle()` that will be expected to handle the request processing.

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The RequestHandler class for the server.
    """

    def __init__(self, request, client_address, server):
        print('Setup names and offices')
        self.addresses = {'JOHN': 'C45',
                          'DENISE': 'C44',
                          'PHOEBE': 'D52',
                          'ADAM': 'B23'}
        super().__init__(request, client_address, server)

    def handle(self):
        print('In Handle')
        # self.request is the TCP socket connected
        # to the client
        data = self.request.recv(1024).decode()
        print('data received:', data)
        key = str(data).upper()
        response = self.addresses[key]
        print('response:', response)
        # Send the result back to the client
        self.request.sendall(response.encode())

def main():
```

```
    print('Starting server')
    server_address = ('localhost', 8084)
    print('Creating server')
    server =
        socketserver.TCPServer(server_address,
                               MyTCPHandler)
    print('Activating server')
    server.serve_forever()

if __name__ == '__main__':
    main()
```

Note that the previous client application does not need to change at all; the server changes are hidden from the client.

However, this is still a single threaded server. We can very simply make it into a multithreaded server (one that can deal with multiple requests concurrently) by mixing the `socketserver.ThreadingMixIn` into the `TCPServer`. This can be done by defining a new class that is nothing more than a class that extends both `ThreadingMixIn` and `TCPServer` and creating an instance of this new class instead of the `TCPServer` directly. For example:

```
class ThreadedEchoServer(
            socketserver.ThreadingMixIn,
            socketserver.TCPServer):
    pass

def main():
    print('Starting')
    address = ('localhost', 8084)
    server = ThreadedEchoServer(address,
                        MyTCPHandler)
    print('Activating server')
    server.serve_forever()
```

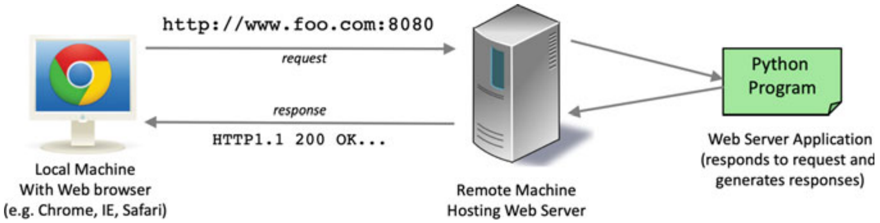
In fact you do not even need to create your own class (such as the `ThreadedEchoServer`) as the `socketserver.ThreadingTCPServer` has been provided as a default mixing of the `TCPServer` and the `ThreadingMixIn` classes. We could therefore just write:

```
def main():
    print('Starting')
    address = ('localhost', 8084)
    server = socketserver.ThreadedEchoServer(address,
                        MyTCPHandler)
    print('Activating server')
    server.serve_forever()
```

49.6 Http Server

In addition to the TCPServer you also have available a `http.server.HTTPServer`; this can be used in a similar manner to the TCPServer, but is used to create servers that respond to the HTTP protocol used by web browsers. In other words it can be used to create a very simple web server (although it should be noted that it is really only suitable for creating test web servers as it only implements very basic security checks).

It is probably worth a short aside to illustrate how a web server and a web browser interact. The following diagram illustrates the basic interactions:



In the above diagram the user is using a browser (such as Chrome, IE or Safari) to access a web server. The browser is running on their local machine (which could be a PC, a Mac, a Linux box, an iPad, a Smart Phone, etc.).

To access the web server, they enter a Universal Resource Locator (URL) address into their browser. In the example this is the URL www.foo.com. It also indicates that they want to connect up to port 8080 (rather than the default port 80 used for HTTP connections). The remote machine (which is the one indicated by the address www.foo.com) receives this request and determines what to do with it. If there is no program monitoring port 8080, it will reject the request. In our case we have a Python program (which is actually the web server program) listening to that port and it is passed the request. It will then handle this request and generate a response message which will be sent back to the browser on the user's local machine. The response will indicate which version of the HTTP protocol it supports, whether everything went ok or not (this is the 200 code in the above diagram—you may have seen the code 404 indicating that a web page was not found, etc.). The browser on the local machine then renders the data as a web page or handles the data as appropriate, etc.

To create a simple Python web server, the `http.server.HTTPServer` can be used directly or can be subclassed along with the `socketserver.ThreadingMixIn` to create a multithreaded web server, for example:

```
class ThreadingHTTPServer(ThreadingMixIn, HTTPServer):
    """Simple multi-threaded HTTP server """
    pass
```

Since Python 3.7 the http.server module now provides exactly this class as a built in facility, and it is thus no longer necessary to define it yourself (see `http.server.ThreadingHTTPServer`).

To handle HTTP requests you must implement one of the HTTP request methods such as `do_GET()`, or `do_POST()`. Each of these maps to a type of HTTP request, for example:

- do_GET() maps to a HTTP Get request that is generated if you type a web address into the URL bar of a web browser or
- do_POST() maps to a HTTP Post request that is used for example, when a form on a web page is used to submit data to a web server.

The `do_GET(self)` or `do_POST(self)` method must then handle any input supplied with the request and generate any appropriate responses back to the browser. This means that it must follow the HTTP protocol.

The following short program creates a simple web server that will generate a welcome message and the current time as a response to a GET request. It does this by using the `datetime` module to create a timestamp of the date and time using the `today()` function. This is converted into a byte array using the UTF-8 character encoding (UTF-8 is the most widely used way to represent text within web pages). We need a byte array as that is what will be executed by the `write()` method later on.

Having done this there are various items of metadata that need to be set up so that the browser knows what data it is about to receive. This metadata is known as *header* data and can include the type of content being sent and the amount of data (content) being transmitted. In our very simple case we need to tell it that we are sending it plain text (rather than the HTML used to describe a typical web page) via the 'Content-type' header information. We also need to tell it how much data we are sending using the content length. We can then indicate that we have finished defining the header information and are now sending the actual data.

The data itself is sent via the `wfile` attribute inherited from the `BaseHTTPRequestHandler`. There are in fact two related attributes `rfile` and `wfile`:

- `rfile` this is an input stream that allows you to read input data (which is not being used in this example).
- `wfile` holds the output stream that can be used to write (send) data to the browser. This object provides a method `write()` that takes a byte-like object that is written out to (eventually) the browser.

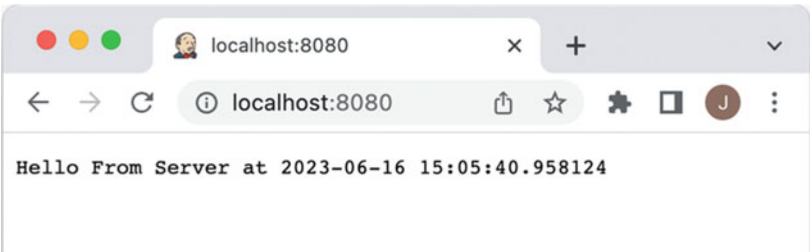
A `main()` method is used to set up the HTTP server which follows the pattern used for the `TCPServer`; however the client of this server will be a web browser.

```
from http.server import BaseHTTPRequestHandler, Thread-
ingHTTPServer
from datetime import datetime

class MyHttpRequestHandler(BaseHTTPRequestHandler):
```

```
        """Very simple request handler. Only supports GET."""

        def do_GET(self):
            print("do_GET() starting to process request")
            welcome_msg = 'Hello From Server at ' +
str(datetime.today())
            byte_msg = bytes(welcome_msg, 'utf-8')
            self.send_response(200)
            self.send_header("Content-type", 'text/plain;
 charset-utf-8')
           self.send_header('Content-length', str(len(byte_msg)))
            self.end_headers()
            print('do_GET() replying with message')
            self.wfile.write(byte_msg)

def main():
    print('Setting up server')
    server_address = ('localhost', 8080)
    httpd = ThreadingHTTPServer(server_address,
MyHttpRequestHandler)
    print('Activating HTTP server')
    httpd.serve_forever()

if __name__ == '__main__':
    main()
```

Once the server is up and running, it is possible to connect to the server using a browser and by entering an appropriate web address into the browsers' URL field. This means that in your browser (assuming it is running on the same machine as the above program) you only need to type into the URL bar `http://localhost:8080` (this indicates you want to use the http protocol to connect up to the local machine at port 8080).

When you do this, you should see the welcome message with the current date and time:



49.7 Online Resources

See the following online resources for information on the topics in this chapter:

- https://docs.python.org/3/howto/sockets.html tutorial on programming sockets in Python.
- https://pymotw.com/3/socket/tcp.html the Python Module of the Week TCP page.
- https://pymotw.com/3/socketserver/index.html The Python Module of the Week page on SocketServer.
- https://docs.python.org/3/library/http.server.html HTTP Servers Python documentation.
- https://pymotw.com/3/http.server/index.html The Python Module of the Week page on the `http.server` module.
- https://www.redbooks.ibm.com/pubs/pdfs/redbooks/gg243376.pdf a PDF tutorial book from IBM on TCP/IP.
- http://flask.pocoo.org/ for more information the Flask micro-framework for web development.
- https://www.djangoproject.com/ provides information on the Django framework for creating web applications.

49.8 Exercises

The aim of this exercise is to explore working with TCP/IP sockets.

You should create a TCP server that will receive a string from a client.

A check should then be made to see what information the string indicates is required, supported inputs are:

- `'Date'` which should result in the current date being returned.
- `'Time'` which should result in the current time being returned.
- `'Date and Time'` which should result in the current date and time being returned.
- Anything else should result in the input string being returned to the client in uppercase with the message 'UNKNOWN OPTION': preceding the string.

The result is then sent back to the client.

You should then create a client program to call the server. The client program can request input from the user in the form of a string and submit that string to the server. The result returned by the server should be displayed in the client before prompting the user for the next input. If the user enters −1 as input, then the program should terminate.

An example of the type of output the client might generate is given below to illustrate the general aim of the exercise:

```
Starting Client
Please provide an input (Date, Time, DataAndTime or -1 to exit): Date
Connected to server
Sending data
Received from server: 2019-02-19
Closing socket
Please provide an input (Date, Time, DataAndTime or -1 to exit): Time
```

```
Connected to server
Sending data
Received from server: 15:50:40
Closing socket
Please provide an input (Date, Time, DataAndTime or -1 to exit):
DateAndTime
Connected to server
Sending data
Received from server: 2019-02-19 15:50:44.720747
Closing socket
Please provide an input (Date, Time, DataAndTime or -1 to exit): -1
```

Chapter 50
Web Services in Python

50.1 Introduction

This chapter looks at RESTful web services.

50.2 RESTful Services

REST stands for Representational State Transfer and was a termed coined by Roy Fielding in his Ph.D. to describe the lightweight, resource-oriented architectural style that underpins the web. Fielding, one of the principle authors of HTTP, was looking for a way of generalising the operation of HTTP and the web. He generalised the supply of web pages as a form of data supplied on demand to a client where the client holds the current state of an exchange. Based on this state information the client requests the next item of relevant data sending all information necessary to identify the information to be supplied with the request. Thus the requests are independent and not part of an ongoing stateful conversation (hence state transfer).

It should be noted that although fielding was aiming to create a way of describing the pattern of behaviour within the web, he also had an eye on producing lighter weight web-based services (than those using either proprietary Enterprise Integration frameworks or SOAP-based services). These lighter weight HTTP-based web services have become very popular and are now widely used in many areas. Systems which follow these principles are termed RESTful services.

A key aspect of a RESTful service is that all interactions between a client (whether some JavaScript running in a browser or a standalone application) are done using simple HTTP-based operations. HTTP supports four operations; these are HTTP Get, HTTP Post, HTTP Put and HTTP Delete. These can be used as verbs to indicate the type of action being requested. Typically these are used as follows:

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_50

- Retrieve information (HTTP Get).
- Create information (HTTP Post).
- Update information (HTTP Put).
- Delete information (HTTP Delete).

It should be noted that REST is not a standard in the way that HTML is a standard. Rather it is a design pattern that can be used to create web applications that can be invoked over HTTP and that give meaning to the use of Get, Post, Put and Delete HTTP operations with respect to a specific resource (or type of data).

The advantage of using RESTful services as a technology, compared to some other approaches (such as SOAP-based services which can also be invoked over HTTP), is that

- The implementations tend to be simpler.
- The maintenance easier.
- They run over standard HTTP and HTTPS protocols.
- Do not require expensive infrastructures and licenses to use.

This means that there is lower server and server side costs. There is little vendor or technology dependency, and clients do not need to know anything about the implementation details or technologies being used to create the services.

50.3 A RESTful API

A RESTful API is one in which you must first determine the key concepts or *resources* being represented or managed.

These might be books, products in a shop, room bookings in hotels, etc. For example a bookstore related service might provide information on resources such as books, CDs and DVDs, etc. Within this service books are just one type of resource. We will ignore the other resources such as DVDs and CDs, etc.

Based on the idea of a book as a resource we will identify suitable URLs for these RESTful services. Note that although URLs are frequently used to describe a web page—that is just one type of resource. For example, we might develop a resource such as

```
/bookservice/book
```

from this we could develop a URL based API, such as

```
/bookservice/book/<isbn>
```

where the variable isbn (which stands for international Standard Book Number) indicates a unique number to be used to identify a specific book whose details will be returned using this URL.

We also need to design the representation or formats that the service can supply. These could include plain text, JSON, XML, etc., JSON standards for the JavaScript Object Notation and is a concise way to describe data that is to be transferred from a service running on a server to a client running in a browser. This is the format we will use in the next section. As part of this we might identify a series of operations to be provided by our services based on the type of HTTP method used to invoke our service and the contents of the URL provided. For example, for a simple BookService this might be:

- GET `/book/<isbn>`—used to retrieve a book for a given ISBN.
- GET `/book/list`—used to retrieve all current books in JSON format.
- POST `/book` (JSON in body of the message)—which supports creating a new book.
- PUT `/book` (JSON in body of message)—used to update the data held on an existing book.
- DELETE `/book/<isbn>`—used to indicate that we would like a specific book deleted from the list of books held.

Note that the *parameter* `isbn` in the above URLs actually forms part of the URL path.

50.4 Python Web Frameworks

There are very many frameworks and libraries available in Python that will allow you to create JSON-based web services; and the shear number of options available to you can be overwhelming. For example, you might consider

1. Flask,
2. Django,
3. Web2py and
4. CherryPy to name just a few.

These frameworks and libraries offer different sets of facilities and levels of sophistication. For example Django is a full-stack web framework; that is it is aimed at developing not just web services but full blown web sites.

However, for our purposes this is probably overkill and the Django Rest interface is only part of a much larger infrastructure. That does not mean of course that we could not use Django to create our bookshop services; however, there are simpler options available. The web2py is another full-stack web framework which we will also discount for the same reason.

In contrast Flask and CherryPy are considered non-full-stack frameworks (although you can create a full-stack web application using them). This means that they are lighter weight and quicker to get started with. CherryPy was original rather more focussed on providing a remote function call facility that allowed functions to be invoked over HTTP; however this has been extended to provide more REST like facilities.

In this chapter we will focus on Flask as it is one of the most widely used frameworks for lightweight RESTful services in Python.

50.5 Online Resources

See the following online resources for information on the topics in this chapter:

1. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm Roy Fieldings' Ph.D. Thesis; if you are interesting in the background to REST read this.
2. https://wiki.python.org/moin/WebFrameworks for a very extensive list of web frameworks for Python.
3. https://www.djangoproject.com/ for information on Django.
4. http://www.web2py.com/ Web2py web framework documentation.
5. https://cherrypy.org/ For documentation on the CherryPy web framework.
6. http://flask.pocoo.org/ For information and examples on the Flask web development framework.
7. http://flask.pocoo.org/docs/1.0/foreword/#what-does-micro-mean Flasks explanation of what micro means.
8. https://www.json.org/ Information on JSON.
9. https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface WSGI Web Server Gateway Interface standard.
10. https://curl.haxx.se/ Information on the curl command line tool.
11. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status HTTP Response Status Codes.

Chapter 51
Flask Web Services

51.1 Introduction

This chapter introduces the Flask framework used to create RESTful services.

51.2 Flask

Flask is a web development framework for Python. It describes itself as a micro-framework for Python which is somewhat confusing; to the point where there is a page dedicated to this on their web site that explains what it means and what the implications are of this for Flask. According to Flask, the *micro* in its description relates to its primary aim of keeping the core of Flask simple but extensible. Unlike Django it doesn't include facilities aimed at helping you integrate your application with a database for example. Instead Flask focusses on the core functionality required of a web service framework and allows extension to be used, as and when required, for additional functionality.

Flask is also a convention over configuration framework; that is if you follow the standard conventions, then you will not need to deal with much additional configuration information (although if you wish to follow a different set of conventions, then you can provide configuration information to change the defaults). As most people will (at least initially) follow these conventions it makes it very easy to get something up and running very quickly.

As Flask is not a built-in library you will need to install it into your environment for example using pip, for example:

```
pip install flask
```


J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_51

At the time of writing the version Flask used was 2.3.2, this is the version against which the examples in this chapter and the next have been run.

51.3 Hello World in Flask

As is traditional in all programming languages we will start with a simple 'Hello World' style application. This application will allow us to create a very simple web service that maps a particular URL to a function that will return JSON format data. We will use the JSON data format as it is very widely used within web-based services.

51.3.1 Using JSON

JSON standards for JavaScript Object Notation; it is a lightweight data-interchange format that is also easy for humans to read and write. Although it is derived from a subset of the JavaScript programming language, it is in fact completely language independent and many languages and frameworks now support automatically processing of their own formats into and from JSON. This makes it ideal for RESTful web services.

JSON is actually built on some basic structures:

- A collection of name/value pairs in which the name and value are separated by a colon ':' and each pair can be separated by a comma ','.
- An ordered list of values that are encompassed in square brackets ('[]').

This makes it very easy to build up structures that represent any set of data, for example a book with an ISBN, a title, author and price could be represented by:

```
{
    "author": "Phoebe Cooke",
    "isbn": 2,
    "price": 12.99,
    "title": "Java"
}
```

In turn a list of books can be represented by a comma separated set of books within square brackets. For example:

```
[ {"author": "Gryff Smith","isbn": 1, "price": 10.99, "title":
"XML"},
{"author": "Phoebe Cooke", "isbn": 2, "price": 12.99, "title":
"Java"}
{"author": "Jason Procter", "isbn": 3, "price": 11.55, "title":
"C#"}]
```

51.4 Implementing a Flask Web Service

There are several steps involved in create a Flask web service; these are:

1. Import flask.
2. Initialise the Flask application.
3. Implement one or more functions (or methods) to support the services you wish to publish.
4. Providing routing information o route from the URL to a function (or method).
5. Start the web service running.

We will look at these steps in the rest of this chapter.

51.4.1 A Simple Service

We will now create our *hello world* web service. To do this we must first import the flask module. In this example we will use the Flask class and `jsonify()` function elements of the module.

We then need to create the main application object which is an instance of the Flask class:

```
from flask import Flask, jsonify

app = Flask(__name__)
```

The argument passed into the `Flask()` constructor is the name of the application's module or package. As this is a simple example we will use the `__name__` attribute of the module which in this case will be `'__main__'`. In larger more complex applications, with multiple packages and modules, then you may need to choose an appropriate package name.

The Flask application object implements the Web Server Gateway Interface (WSGI) standard for Python. This was originally specified in PEP-333 in 2003 and was updated for Python 3 in PEP-3333 published in 2010. It provides a simple convention for how web servers should handle requests to applications. The Flask application object is the element that can route a request for a URL to a Python function.

51.4.2 Providing Routing Information

We can now define routing information for the Flask application object. This information will map a URL to a function. When that URL is, for example, entered into a web browsers URL field, then the Flask application object will receive that request and invoke the appropriate function.

To provide route mapping information, we use the `@app.route` decorator on a function or method.

For example, in the following code the `@app.route` decorator maps the URL `/hello` to the function `welcome()` for HTTP Get requests:

```
@app.route('/hello', methods=['GET'])
def welcome():
    return jsonify({'msg': 'Hello Flask World'})
```

There are two things to note about this function definition:

- The `@app.route` decorator is used to declaratively specify the routing information for the function. This means that the URL '`/hello`' will be mapped to the function `welcome()`. The decorator also specifies the HTTP method that is supported; in this case `GET` requests are supported (which is actually the default so it does not need to be included here but is useful from a documentation point of view).
- The second thing is that we are going to return our data using the JSON format; we therefore use the `jsonify()` function and pass it a Python Dictionary structure with a single key/value pair. In this case the key is '`msg`' and the data associated with that key is '`Hello Flask World`'. The `jsonify()` function will convert this Python data structure into an equivalent JSON structure.

51.5 Running the Service

We are now ready to run our application. To do this, we invoke the `run()` method of the Flask application object:

```
app.run(debug=True)
```

Optionally this method has a keyword parameter `debug` that can be set to `True`; if this is done, then when the application is run some debugging information is generated that allows you to see what is happening. This can be useful in development but would not typically be used in production.

The whole program is presented below:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/hello', methods=['GET'])
def welcome():
    return jsonify({'msg': 'Hello Flask World'})

app.run(debug=True)
```

When this program is run, the initial output generated is as shown below:

```
* Serving Flask app "hello_flask_world" (lazy loading)
```

```
* Environment: production
  WARNING: This is a development server. Do not use it in a production
deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 274-630-732
```

Of course we don't see any output from our own program yet. This is because we have not invoked the `welcome()` function via the `/hello` URL.

51.6 Invoking the RESTFul Service

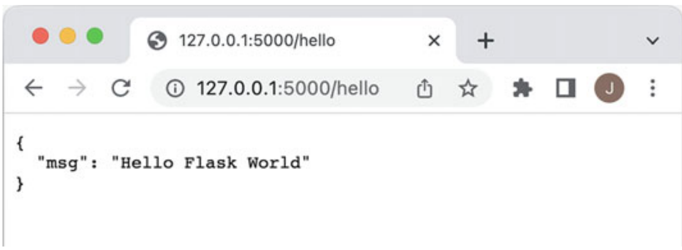
We will use a web browser to access the web service. To do this, we must enter the full URL that will route the request to our running application and to the `welcome()` function.

The URL is actually comprised of two elements: the first part is the machine on which the application is running and the port that it is using to listen for requests. This is actually listed in the above output—look at the line starting 'Running on'. This means that the URL must start with http://127.0.0.1:5000. This indicates that the application is running on the computer with the IP address 127.0.0.1 and listening on port 5000.

We could of course also use localhost instead of 127.0.0.1.

The remainder of the URL must then provide the information that will allow Flask to route from the computer and port to the functions we want to run.

Thus the full URL is http://127.0.0.1:5000/hello and thus is used in the web browser shown below:



As you can see the result returned is the text we supplied to the `jsonify()` function but now in plain JSON format and displayed within the web browser.

You should also be able to see in the console output that a request was received by the Flask framework for the GET request mapped to the `/hello` URL:

```
127.0.0.1 - - [16/Jun/2023 15:10:11] "GET /hello HTTP/1.1" 200 -
```

One useful feature of this approach is that if you make a change to your program, then the Flask framework will notice this change when running in development mode and can restart the web service with the code changes deployed. If you do this, you will see that the output notifies you of the change:

```
* Detected change in 'hello_flask_world.py', reloading
* Restarting with stat
```

This allows changes to be made on the fly, and their effect can be immediately seen.

51.6.1 The Final Solution

We can tidy this example up a little by defining a function that can be used to create the Flask application object and by ensuring that we only run the application if the code is being run as the main module:

```
from flask import Flask, jsonify, url_for

def create_service():
    app = Flask(__name__)

    @app.route('/hello', methods=['GET'])
    def welcome():
         return jsonify({'msg': 'Hello Flask World'})

    with app.test_request_context():
        print(url_for('welcome'))

    return app

if __name__ == '__main__':
    app = create_service()
    app.run(debug=True)
```

One feature we have added to this program is the use of the `test_request_context()`. The test request context object returned implements the context manager protocol and thus can be used via a `with` statement; this is useful for debugging purposes. It can be used to verify the URL used for any functions with routing information specified. In this case the output from the print statement is '/hello' as this is the URL defined by the `@app.route` decorator.

51.7 Online Resources

See the following online resources for information on the topics in this chapter:

1. https://flask.palletsprojects.com/en/2.3.x/ For information and examples on the Flask web development framework.
2. https://flask.palletsprojects.com/en/2.3.x/design/#what-does-micro-mean Flasks explanation of what micro means.
3. https://curl.haxx.se/ Information on the curl command line tool.

Chapter 52
Flask Bookshop Web Service

52.1 Introduction

This chapter explores a larger RESTFul application written using the Flask framework. The chapter discusses the design of the RESTFul application, the implementation of the domain model, converting the domain model into JSON and defining the various services supported by GET, POST, PUT and DELETE.

52.2 Building a Flask Bookshop Service

The previous chapter illustrated the basic structure of a very simple web service application. We are now in a position to explore the creation of a set of web services for something a little more realistic; the bookshop web service application.

In this chapter we will implement the set of web services described earlier in the previous chapter for a very simple bookshop. This means that we will define services to handle not just the GET requests but also PUT, POST and DELETE requests for the RESTful bookshop API.

52.3 The Design

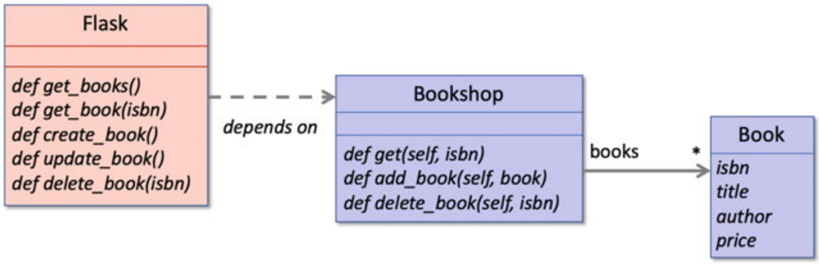
Before we look at the implementation of the Bookshop RESTful API, we will consider what elements we for the services.

One question that often causes some confusion is how web services relate to traditional design approaches such as object-oriented design. The approach adopted here is that the Web Service API provides a way to implement an *interface* to appropriate functions, objects and methods used to implement the application/domain model.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_52

This means that we will still have a set of classes that will represent the *Bookshop* and the *Books* held within the bookshop. In turn the functions implementing the web services will access the bookshop to retrieve, modify, update and delete the books held by the bookshop.

The overall design is shown below:



This shows that a `Book` object will have an ISBN, a title, an author and a price attribute.

In turn the `Bookshop` object will have a `books` attribute that will hold zero or more books. The books attribute will actually hold a `List` as the list of books needs to change dynamically as and when new books are added or old books deleted.

The `Bookshop` will also define three methods that will

- Allow a book to be obtained via its ISBN.
- Allow a book to be added to the list of books.
- Enable a book to be deleted (based on its ISBN).

Routing information will be provided for a set of functions that will invoke appropriate methods on the bookshop object. The functions to be decorated with `@app.route`, and the mappings to be used, are listed below:

- `get_books()` which maps to the `/book/list` URL using the HTTP Get method request.
- `get_book(isbn)` which maps to the `/book/<isbn>` URL where ISBN is a URL parameter that will be passed into the function. This will also use the HTTP GET request.
- `create_book()` which maps to the `/book` URL using the HTTP Post request.
- `update_book()` which maps to the `/book` URL but using the HTTP Put request.
- `delete_book()` which maps to the `/book/<isbn>` URL but using the HTTP Delete request.

52.4 The Domain Model

The domain model comprises the `Book` and `Bookshop` classes. These are presented below.

The `Book` class is a simple *Value* type class (i.e. it is data oriented with no behaviour of its own):

```
class Book:
    def __init__(self, isbn, title, author, price):
        self.isbn = isbn
        self.title = title
        self.author = author
        self.price = price

    def __str__(self):
  return self.title + 'by ' + self.author + '@ ' + str(self.price)
```

The `Bookshop` class holds a list of books and provides a set of methods to access books, update books and delete books:

```
class Bookshop:
    def __init__(self, books):
        self.books = books

    def get(self, isbn):
        if int(isbn) > len(self.books):
            abort(404)
        return list(filter(lambda b: b.isbn == isbn,
self.books))[0]

    def add_book(self, book):
        self.books.append(book)
    def delete_book(self, isbn):
         self.books = list(filter(lambda b: b.isbn != isbn,
self.books))
```

In the above code, the `books` attribute holds the list of books currently available.
The `get()` method returns a book given a specified ISBN.
The `add_book()` method adds a book object to the list of books.
The `delete_book()` method removes a book based on its ISBN.
The `bookshop` global variable holds the `Bookshop` object initialised with a default set of books:

```
bookshop = Bookshop(
    [Book(1, 'XML', 'Gryff Smith', 10.99),
     Book(2, 'Java', 'Phoebe Cooke', 12.99),
     Book(3, 'Scala', 'Adam Davies', 11.99),
     Book(4, 'Python', 'Natalia Nadal', 15.99)])
```

52.5 Encoding Books into JSON

One issue we have is that although the `jsonify()` function knows how to convert built-in types such as strings, integers, lists and dictionaries into an appropriate JSON format, it does not know how to do this for custom types such as a `Book`.

We therefore need to define some way of converting a `Book` into an appropriate JSON format.

One way we could do this would be to define a method that can be called to convert an instance of the `Book` class into a JSON format. We could call this method `to_json()`. For example:

```
class Book:
    """ Represents a book in the bookshop"""
    def __init__(self, isbn, title, author, price):
        self.isbn = isbn
        self.title = title
        self.author = author
        self.price = price
    def __str__(self):
        return self.title + 'by ' + self.author + '@ ' + str(self.price)
    def to_json(self):
        return {
            'isbn': self.isbn,
            'title': self.title,
            'author': self.author,
            'price': self.price
        }
```

We could now use this with the `jsonify()` function to convert a book into the JSON format:

```
jsonify({'book': book.to_json()})
```

This approach certainly works and provides a very lightweight way to convert a book into JSON.

However, the approach presented above does mean that every time we want to *jsonify* a book we must remember to call the `to_json()` method. In some cases this means that we will also have to write some slightly convoluted code. For example if we wish to return a list of books from the `Bookshop` as a JSON list, we might write:

```
jsonify({'books': [b.to_json() for b in bookshop.books]})
```

Here we have used a list comprehension to generate a list containing the JSON versions of the books held in the bookshop.

This is starting to look overly complex, easy to forget about and probably error prone.

Flask itself uses JSON providers to encode types into JSON. Flask provides a way of creating your own JSON Providers that can be used to convert a custom type, such as the `Book` class, into JSON. Such an JSON Provider can automatically be used by the `jsonify()` function.

To do this, we must implement an *Provder* class; the class will extend the flask.`json.provider.DefaultJSONProvider` superclass.

The class must define a method `dumps(self, obj: t.Any, **kwargs: t.Any)` to convert an object to JSON and `loads(self, s: str | bytes, **kwargs: t.Any)` to do the reverse.

The `dumps()` method takes an object and returns the JSON representation of that object. We can therefore write an encoder for our RESTFul service as follows:

```
class BookJSONProvider(DefaultJSONProvider):
    def dumps(self, obj: t.Any, **kwargs: t.Any):
        if isinstance(obj, Book):
            return {'book': obj.to_json()}
        elif isinstance(obj, list):
             return {'books': list(map(lambda b: b.to_json(),
obj))}
        return super().dumps(obj, **kwargs)
```

The dumps() method here checks to see if the value being passed to it is a book; if it is, then it will use the to_jason(0 method to convert the book into JSON and return the result. If the value passed in is a list, then it is assumed that it is a list of books and the same process is performed across all the books in the list. If it is anything else, then the responsibility of handling the conversion to JSON is delegated to the parent class.

This new BookJSONProvider is registered with the Flask app when it is created, for example:

```
app = Flask(__name__)
app.json=BookJSONProvider(app)
```

Now if we wish to encode a single book or a list of books, the above encoder will be used automatically and thus we do not need to do anything else. Thus our earlier examples can be written to simply by referencing the `book` or `bookshop.books` attribute:

```
jsonify(book)
jsonify(bookshop.books)
```

Notice that this is simpler than the first example as we don't include the dictionary format of `{'book': book.to_json()}` etc.

One final point that should be noted about `jsonify()` is that it doesn't just convert an object into JSON format. It also returns a flask.Response object with the mine type set to application/json. This means that the object returned is suitable to be used as the response for a RESTFul service.

52.6 Setting Up the GET Services

We can now set up the two services that will support GET requests, these are the

- /book/list and /book<isbn> services.

  The functions that these URLs map to are given below:

```
@app.route('/book/list', methods=['GET'])
def get_books():
    response= jsonify(bookshop.books)
    return response

@app.route('/book/<int:isbn>', methods=['GET'])
def get_book(isbn):
    book = bookshop.get(isbn)
    return jsonify(book)
```

The first function merely returns the current list of books held by the bookshop in a JSON structure using the key books.

The second function takes an ISBN number as parameter. This is a URL parameter; in other words part of the URL used to invoke this function is actually dynamic and will be passed into the function. This means that a user can request details of books with different ISBNs just by changing the ISBN element of the URL, for example:

- /book/1 will indicate that we want information on the book with the ISBN 1.
- /book/2 will indicate we want information on the book with ISBN 2.

In Flask to indicate that something is a URL parameter rather than a hard coded element of the URL, we use angle brackets (<>). These surround the URL parameter name and allow the parameter to be passed into the function (using the same name).

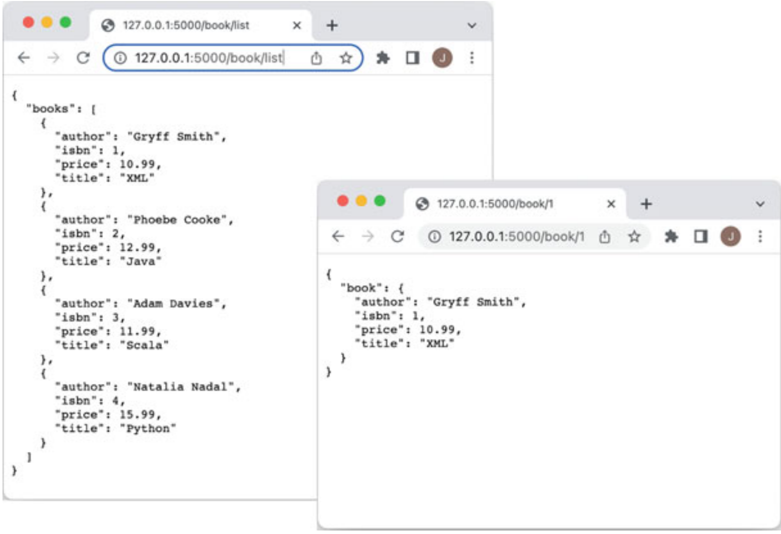
In the above example we have also (optionally) indicated the type of the parameter. By default the type will be a string; however we know that the ISBN is in fact an integer and so we have indicated that by pre-fixing the parameter name with the type int (and separated the type information from the parameter name by a colon ':'). There are actually several options available including

- string (the default).
- int (as used above).
- float for positive floating point values.
- uuid for uuid strings.
- path which dislike string but accepts slashes.

We can again use a browser to view the results of calling these services; this time the URLs will be

- http://127.0.0.1:5000/book/list
- http:/127.0.0.1:5000/book/1

for example:



As you can see from this the book information is returned as a set of *key / value* pairs in JSON format.

52.7 Deleting a Book

The *delete a book* web service is very similar to the *get a book* service in that it takes an ISBN as a URL path parameter. However, in this case it merely returns an acknowledgement that the book was deleted successfully:

```
@app.route('/book/<int:isbn>', methods=['DELETE'])
def delete_book(isbn):
    bookshop.delete_book(isbn)
    return jsonify({'result': True})
```

However, we can no longer test this just by using a web browser. This is because the web browser uses the HTTP Get request method for all URLs entered into the URL field. However, the delete web service is associated with the HTTP Delete request method.

To invoke the `delete_book()` function, we therefore need to ensure that the request that is sent uses the DELETE request method. This can be done from a client that can indicate the type of request method being used. Examples might include another Python program, a JavaScript web site, etc.

For testing purposes, we will however use the `curl` program. This program is available on most Linux and Mac systems and can be easily installed, if it is not already available, on other operating systems.

The `curl` is a command line tool and library that can be used to send and receive data over the internet. It supports a wide range of protocols and standards and in particular supports HTTP and HTTPS protocols and can be used to send and receive data over HTTP/S using different request methods.

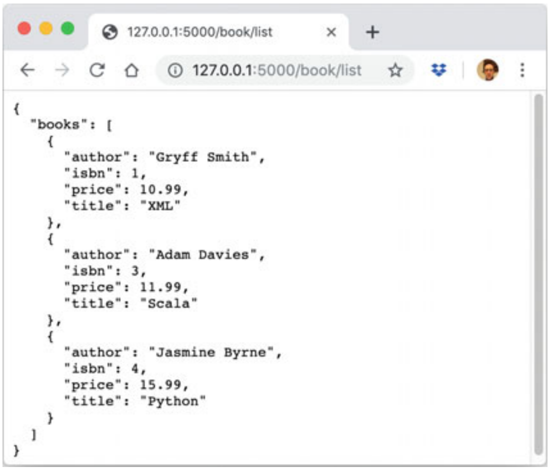
For example, to invoke the `delete_book()` function using the `/book/2` URL and the HTTP Delete method we can use `curl` as follows:

```
curl http://localhost:5000/book/2 -X DELETE
```

This indicates that we want to invoke the URL (http://localhost:5000/book/2) and that we wish to use a custom request method (i.e. Not the default GET) which is in the case DELETE (as indicated by the -X option). The result returned by the command is given below indicating that the book was successfully deleted.

```
{
  "result": true
}
```

We can verify this by checking the output from the `/book/list` URL in the web browser:



This confirms that book 2 has been deleted.

52.8 Adding a New Book

We also want to support adding a new book to the bookshop. The details of a new book could just be added to the URL as URL path parameters; however as the amount of data to be added grows this would become increasingly difficult to maintain and verify. Indeed although historically there was a limit of 2083 characters in Microsoft's Internet Explore (IE) which has theoretically be removed since IE8, in practice there are typically still limits on the size of the URL. Most web servers have a limit of 8 KB (or 8192 bytes) although this is typically configurable. There may also be client side limits (such as those imposed by IE or Apple's Safari [which usually have a 2 KB limit)]. If the limit is exceeded in either a browser or on the server, then most systems will just truncate the characters outside the limit (in some cases without any warning).

Typically such data is therefore sent in the body of the HTTP request as part of a HTTP Post request. This limit on the same of a Post requests message body is much higher (usually up to 2 GB). This means that it is a much more reliable and safer way to transfer data to a web service.

However, it should be noted that this does not mean that the data is any more *secure* than if it is part of the URL; just that it is sent in a different way.

From the point of view of the Python functions that are invoked as the result of a HTTP Post method request it means that the data is not available as a parameter to the URL and thus to the function. Instead, within the function it is necessary to obtain the request object and then to use that to obtain the information held within the body of the request.

A key attribute on the request object, available when a HTTP request contains JSON data, is the `request.json` attribute. This attribute contains a dictionary like structure holding the values associated with the keys in the JSON data structure.

This is shown below for the `create_book()` function.

```
from flask import request, abort

@app.route('/book', methods=['POST'])
def create_book():
    print('create book')
    if not request.json or not 'isbn'in request.json:
        abort(400)
    book = Book(request.json['isbn'],
                request.json['title'],
                request.json.get('author', ""),
                float(request.json['price']))
    bookshop.add_book(book)
    return jsonify(book), 201
```

The above function accesses the `flask.request` object that represents the current HTTP request. The function first checks to see that it contains JSON data and that the *ISBN* of the book to add, is part of that JSON structure. If the ISBN is not, then the `flask.abort()` function is called passing in a suitable HTTP response status code. In this case the error code indicates that this was a Bad Request (HTTP Error Code 400).

If however the JSON data is present and does contain an ISBN number, then the values for the keys `isbn`, `title`, `author` and `price` are obtained. Remember that JSON is a dictionary like structure of keys and values; thus treating it in this way makes it easy to extract the data that a JSON structure holds. It also means that we can use both method and key-oriented access styles. This is shown above where we use the `get()` method along with a default value to use, if an author is not specified.

Finally, as we want to treat the price as a floating point number we must use the `float()` function to convert the string format supplied by JSON into a float.

Using the data extracted we can instantiate a new `Book` instance that can be added to the bookshop. As is common in web services we are returning the newly created book object as the result of creating the book along with the HTTP response status code 201, which indicates the successful creation of a resource.

We can now test this service using the `curl` command line program:

```
curl -H "Content-Type: application/json" -X POST -d '{"title":"Read
a book", "author":"Bob","isbn":"5", "price":"3.44"}'http://
localhost:5000/book
```

The options used with this command indicate the type of data being sent in the body of the request (`-H`) along with the data to include in the body of the request (`-d`). The result of running this command is:

```
{
 "book": {
  "author": "Bob",
  "isbn": "5",
  "price": 3.44,
  "title": "Read a book"
 }
}
```

Illustrating that the new book by Bob has been added.

52.9 Updating a Book

Updating a book that is already held by the bookshop object is very similar to adding a book except that the HTTP Put request method is used.

Again the function implementing the required behaviour must use the `flask.request` object to access the data submitted along with the PUT request. However, in this case the ISBN number specified is used to find the book to be updated, rather than the specifying a completely new book.

The `update_book()` function is given below:

```
@app.route('/book', methods=['PUT'])
def update_book():
    if not request.json or not 'isbn'in request.json:
         abort(400)
    isbn = request.json['isbn']
    book = bookshop.get(isbn)
    book.title = request.json['title']
    book.author = request.json['author']
    book.price = request.json['price']
```

```
    return jsonify(book), 201
```

This function resets the `title`, `author` and `price` of the book retrieved from the bookshop. It again returns the updated book as the result of running the function.

The `curl` program can again be used to invoke this function, although this time the HTTP Put method must be specified:

```
curl -H "Content-Type: application/json" -X PUT -d '{"title":"Read
a book", "author":"Bob Jones","isbn":"5", "price":"3.44"}'http://
localhost:5000/book
```

The output from this command is:

```
{
 "book": {
  "author": "Bob Jones",
  "isbn": "5",
  "price": "3.44",
  "title": "Read a book"
 }
}
```

This shows that book 5 has been updated with the new information.

52.10 What Happens if We Get It Wrong?

The code presented for the bookshop web services is not particularly defensive, as it is possible to try to add a new book with the same ISBN as an existing one. However, it does check to see that an ISBN number has been supplied with both the `create_book()` and `update_book()` functions.

However, what happens if an ISBN number is not supplied? In both functions we call the `flask.abort()` function. By default if this happens an error message will be sent back to the client.

For example, in the following command we have forgotten to include the ISBN number:

```
curl -H "Content-Type: application/json" -X POST -d '{"title":"Read
a book", "author":"Tom Andrews", "price":"13.24"}'http://
localhost:5000/book
```

This generates the following error output:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>400 Bad Request</title>
<h1>Bad Request</h1>
<p>The browser (or proxy) sent a request that this server could not
understand.</p>
```

The odd thing here is that the error output is in HTML format, which is not what we might have expected since we are creating a web service and working with JSON. The problem is that Flask has default to generating an error HTML web page that it expects to be rendered in a web browser.

We can overcome this by defining our own custom error handler function. This is a function that is decorated with an `@app.errorhandler()` decorator which provides the response status code that it handles. For example:

```
@app.errorhandler(400)
def not_found(error):
   return make_response(jsonify({'error': 'Book Not Found'}), 400)
```

Now when a 400 code is generated via the `flask.abort()` function, the `not_found()` function will be invoked and a JSON response will be generated with the information provided by the `flask.make_response()` function. For example:

```
curl -H "Content-Type: application/json" -X POST -d '{"title":"Read
a book", "author":"Tom Andrews", "price":"13.24"}'http://
localhost:5000/book
```

The output from this command is:

```
{
 "Error": "Book Not Found"
}
```

52.11 Bookshop Services Listing

The complete listing for the bookshop web services application is given below:

```
import typing as t
from flask import Flask, jsonify, request, abort, make_response
from flask.json.provider import DefaultJSONProvider

class Book:
    def __init__(self, isbn, title, author, price):
        self.isbn = isbn
        self.title = title
        self.author = author
        self.price = price

    def __str__(self):
```

```
        return self.title + 'by '+ self.author + '@ '+
str(self.price)

    def to_json(self):
        return {
            'isbn': self.isbn,
            'title': self.title,
            'author': self.author,
            'price': self.price
        }

class BookJSONProvider(DefaultJSONProvider):

    def dumps(self, obj: t.Any, **kwargs: t.Any):
        if isinstance(obj, Book):
            return {'book': obj.to_json()}
        elif isinstance(obj, list):
            return {'books': list(map(lambda b: b.to_json(), obj))}
        return super().dumps(obj, **kwargs)

class Bookshop:
    def __init__(self, books):
        self.books = books

    def get(self, isbn):
        if int(isbn) > len(self.books):
            abort(404)
        return list(filter(lambda b: b.isbn == isbn,
self.books))[0]

    def add_book(self, book):
        self.books.append(book)

    def delete_book(self, isbn):
        self.books = list(filter(lambda b: b.isbn != isbn,
self.books))

bookshop = Bookshop([Book(1, 'XML', 'Gryff Smith', 10.99),
                     Book(2, 'Java', 'Phoebe Cooke', 12.99),
                     Book(3, 'Scala', 'Adam Davies', 11.99),
                     Book(4, 'Python', 'Natalia Nadal', 15.99)])

def create_bookshop_service():
    app = Flask(__name__)
    app.json=BookJSONProvider(app)

    @app.route('/book/list', methods=['GET'])
    def get_books():
        return jsonify(bookshop.books)

    @app.route('/book/<int:isbn>', methods=['GET'])
    def get_book(isbn):
        book = bookshop.get(isbn)
        return jsonify(book)

    @app.route('/book', methods=['POST'])
    def create_book():
```

```
        print('create book')
        if not request.json or not 'isbn'in request.json:
            abort(400)
        book = Book(request.json['isbn'],
                    request.json['title'],
                    request.json.get('author', ""),
                    float(request.json['price']))
        bookshop.add_book(book)
        return jsonify(book), 201

    @app.route('/book', methods=['PUT'])
    def update_book():
        if not request.json or not 'isbn'in request.json:
            abort(400)
        isbn = request.json['isbn']
        book = bookshop.get(isbn)
        book.title = request.json['title']
        book.author = request.json['author']
        book.price = request.json['price']
        return jsonify(book), 201

    @app.route('/book/<int:isbn>', methods=['DELETE'])
    def delete_book(isbn):
        bookshop.delete_book(isbn)
        return jsonify({'result': True})

    @app.errorhandler(400)
    def not_found(error):
       return make_response(jsonify({'Error': 'Book Not Found'}),
400)

     return app

if __name__ == '__main__':
    app = create_bookshop_service()
    app.run(debug=True)
```

52.12 Exercises

The exercises for this chapter involve creating a web service that will provide information on stock market prices.

The services to be implemented are

Get method:

- `/stock/list` this will return a list of the stocks that can be queried for their price.
- `/stock/ticker` this will return the current price of the stock indicated by ticker, for example `/stock/APPL` or `/stock/MSFT`.

Post method:

- `/stock` with the request body containing JSON for a new stock ticker and price, for example {'IBM': 12.55}.

  Put method:

- `/stock` with the request body containing JSON for an existing stock ticker and price.

  DELETE method:

- `/stock/<ticker>` which will result in the stock indicated by the ticker being deleted from the service.

You could initialise the service with a default set of stocks and prices such as `[('IBM', 12.55), ('APPL', 15.66), ('GOOG', 5.22)]`.

You can test these services using the `curl` command line tool.

Part XI
Data Science: Data Analytics and Machine Learning

Chapter 53
Introduction to Data Science

53.1 Introduction

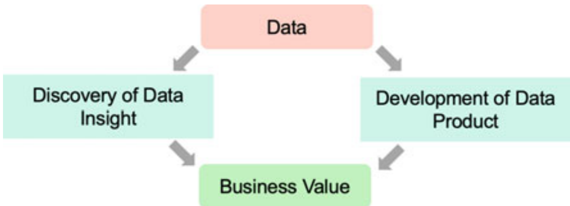
In this chapter we will introduce Data Science and its related topic of data analytics, what it is and how it is used. We will also explore how Python can be used to support Data Science type applications.

53.2 Data Science

Data Science is an interdisciplinary field that uses techniques from a variety of different disciplines and subjects to extract knowledge from data in various forms. The intention is to gain new understanding from that data either to help decision-making or otherwise provide benefit to an organisation such as to generate additional value.

A simple example might be to note that when a supermarket store card member suddenly starts buying nappies, they might well also start to buy ready meals, etc. Therefore, it might be useful to put adverts for ready meals in the baby isle.

The following diagram aims to provide an overview of how Data Science can be considered to add value.



J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_53

On the *left-hand side* of the diagram we can see that Data Science can be used to provide insight into the data, for example to further the understanding of any latent or hidden patterns within the data which might be of use to the organisation. This might, for example, relate to customer or individual patterns of behaviour that might not be obvious from the raw data. The information (or knowledge) obtained from this might then be used to influence decisions made by the organisation such as what promotions to push or what process or procedural changes to make.

On the *right-hand side* of the diagram, we have the 'Development of Data Product' which relates to the creation of products (typically software products such as data classifiers or predictors). These tools might be developed using existing data and then used to analyse future data. An example might be a machine learning classifier-based system that is trained to identify fraudulent bank loan applications. Once it has been trained it can be used to analyse new loan applications to help identify future fraud.

53.3 Data Science Tools and Techniques

Data Science covers a wide range of topics from the business domain, through IT to statistics and mathematics. Some of the topics that relate to Data Science include:

- **Data identification**. Identifying the data to be used for a particular project is an important aspect of any Data Science project.
- **Data acquisition**. Once the data has been identified it needs to be acquired, and as discussed later in this chapter this can be a significant task in its own right.
- **Data integrity**. Data integrity relates to the accuracy, completeness, consistency and validity of the data to be analysed. Typically this involves examining the data to ensure that the data has integrity; for example, if you consider the population at large and found that your data set suggested that everybody in your organisation had blue eyes, there is probably an issue with that data, as statistically it is highly unlikely that in an organisation of any significant size everyone would have blue eyes. Further investigation might for example uncover a bug in the data collection system in which all eye colours have been defaulted to blue, and thus in the data set being used, eye colours cannot be relied upon.
- **Data pre-processing**. Data pre-processing/cleansing often referred to as data munging within the data analytics world involves carrying out initial work to convert the data into formats and layouts that can be processed. For example, this might involve converting a PDF file into a plain text format or extracting data from a custom data file format into a CSV or Excel format.
- **Data exploration**. This involves exploring, examining and generally coming to an understanding of the data available. This process usually involves some combination of data visualisation tools, programming language libraries, database queries, etc. The aim is not yet to identify patterns in the data or to use the data to create some analysis tool instead it is just to get a feel for the data itself in its raw state, for example, how much data is missing from that expected, what types of data are

present, how much is in formal structures and how much unstructured. What types of data are present, for example just because a value is a string does not mean that it does not contain data which might be treated as a timestamp, an integer or a floating point number, etc. Exploratory data analysis is a similar but wider topic in that it usually involves some form of statistical analysis to determine very general trends in the data. Data exploration in general is an important step in the whole Data Science process as it allows an understanding of the data available. This in turn helps the data scientist to make better decisions when deciding how to process the data. It is important to understand that successful data exploration starts with an open mind, any pre-conceptions about the data should be left behind as it can bias any and all further analysis (i.e., if you are going looking for some particular trend in the data, you will probably find some way of identifying it even if it was not there to start with).

- **Data visualisation**. Data visualisation is simply the graphical representation of information and data using graphs, charts, maps, gauges etc. Such tools provide an accessible method for viewing and understanding data, outliers and general trends. Specialist data visualisation tools may be used alongside programming libraries that help to present the data within a program to an end-user.
- **Databases**. Most data is held within some form of database and Python provides interfaces that allow a programmer to access that data. These interfaces make it easy to query a database for information and pull that information back into the Python program for further progressing. For example, a database of client information may be queried to retrieve details held about clients.
- **Distributed processing**. The data sets being processed in a Data Science project can be very large indeed, and it can take a significant amount of time to apply some algorithm to the analysis of that data. Distributing that processing amongst a number of processors or indeed across a server cluster can be very beneficial. Having an understanding of distributed processing techniques and of the libraries that support such processing within a Data Science context can be very useful.
- **Statistical and mathematical modelling**. Although this can be a surprise for the aspiring data scientist, who thinks they will be using advanced analytical libraries or machine learning algorithms on a regular basis, in many cases the application of well-established statistical or mathematical modelling techniques is often enough. Therefore, having a strong background in statistical and mathematical modelling can be essential.
- **Business intelligence and BI dashboards**. Business Intelligence (or BI) and BI Analytics refer to the collective tools, techniques, applications, and infrastructure that can be used by (typically) non-programmers to help businesses make decisions and evaluate performance. These systems often benefit from, and integrate with, the results obtained from and insights generated by, Data Science projects. For example, it is not uncommon to integrate the results generated by a Data Science project into a BI dashboard such that the end-user obtains all the information they need in one place and in a format that they are used to.

- **Machine learning**. Machine learning is the application of algorithms and techniques often derived from the AI community that can be used to generate further understanding of the data or classifiers and predictors based purely on analysis of that data. In many cases these algorithms make extensive use of statistical techniques under the umbrella of AI.

53.4 Data Analytics Process

Within the area of Data Science, data analytics can be viewed as being the part that seeks to convert operational observations into issues that we either know we know or know we don't know. Thus data analytics is the process of examining data sets in order to draw conclusions about the information they contain, increasingly with the aid of specialised systems and software.

The general lifecycle used with a data analytics project is shown below.

There are several steps that typically comprise the data analytics lifecycle. These steps will be discussed below:

1. **Discovery**. In this phase of the data analytics lifecycle, the aim is to find/obtain the data that will be used for the rest of the lifecycle. This may not always be as easy as it sounds, as data may be stored in multiple different ways in different parts of the organisation. The data may be in structured formats such as relational databases or CSV files or it may be in semi-structured formats such as emails or indeed in unstructured formats such as PDF files or images. Additionally, the

data scientist may not by default have access to all this information and may need to work with the organisation (and in some cases with external organisations) to obtain access to the data or obtain appropriate redacted data sets, etc. One intriguing aspect of this process may be actually finding out whether the data exists or not, in some organisations you might be told explicitly that the data does or does not exist, but further investigation may contradict what you have been told. Very large organisations, who may benefit the most from Data Science type activities, often have large and complex structures in which knowledge of what data is held and where is often obfuscated (intentionally or unintentionally). The data scientist must access this data in order to be able to carry out any further work on it.

2. **Data preparation**. Having gained access to the data, the data scientist must now prepare the data so that it can be processing using the tools, programming languages and techniques available to them. This may not be as easy as it sounds; for example, it may be necessary to convert PDF files to plain text documents, or to analyse images for contained textual data. In many cases data may be missing, either because of problems with the initial data capture process or due to different parts of an organisation using different processes. In such cases a decision needs to be made about how to handle missing data. This is often not trivial. For example, as a simple case consider a set of temperatures for 12 pm each day through August. What should you do if some of those temperature readings are missing (maybe there was a technical fault with the monitoring hardware and one in five readings was not taken). How should you represent those missing data points—with a blank, with -1, or with 0? If you use a blank, then when you try to calculate an average temperature, you might find that you add together 25 values but divide them by 31 to get a daily average. Which would give a misleading result! If you use -1, then you will have a full 31 readings, but all the -1 readings might imply that it was freezing on those particular days. If you leave the days without readings out completely and then graph them, the graph will have a potentially odd look to it, etc. In general, an appropriate strategy needs to be adopted for each data set given whatever models and tasks will be applied to it later in the lifecycle.
3. **Model planning**. It is important to consider which data analytics and machine learning models might be appropriate for the tasks to be performed. This may involve some research and exploration into the data itself. From this, decisions may be made about which analytical techniques to apply or which machine learning algorithms to use. In many cases this involves experiments with some (or all) of the data available to explore the options available. For example, there are several classifier style machine learning algorithms available such as *nearest neighbour* and *Random Forest* Decision Trees. However, which algorithm should be used relates both to the data involved and to the actual purpose of the final system.
4. **Model building**. Once the data scientist (or the team) has selected which models to use they can start building the models and applying the data to them. Typically, this is an iterative process. This may be because only a sample of the data was

used for the model planning phases and additional concerns or issues may be raised when all the data is applied to a specific model.

5. **Communication of results**. The way in which the results are communicated to the stakeholders varies widely depending upon the organisation, the intent behind the project and the tools available. In some cases, the data scientist may be expected to present the results to management with the implications or insights gained from the data outlined. In other situations, a graphical BI dashboard system may be created so that stakeholders (e.g. senior management) can see the results for themselves and take appropriate actions. Other systems might generate some form of automated analysis tool that might identify particular trends in the data and subsequently mark these for further investigation by human actors, etc.
6. **Operationalisation**. The final step might be to operationalise the systems developed. This might be to embed the BI dashboard into the day-to-day tools used by the final recipients of the data. Alternatively, it might be to embed a piece of software into the current workflow of existing systems such as in the potential identification of fraudulent loan applications.

Much debate is given to how automated such systems should be, for example should a loan be rejected automatically by a fraud identification system generated from a machine learning algorithm or should it be reviewed by a human before final acceptance or rejection is made. That is beyond the topic of this book and may well be a decision that is made on a project-by-project or organisation by organisation basis.

53.5 Python and Data Science

There are many tools available to the Data Scientist to help them in their day-to-day work. Some of the these tools are large complex pieces of pre-built software such as the SAS suite of tools; others are libraries available for a particular language or languages such as Apache Spark for Java, Kotlin and Scala. There are also specialist programming languages, originally intended for a specific purpose (such as statistical analysis) which are increasing used for Data Science type projects such as R.

In contrast, Python is a general-purpose programming language which is very well suited to Data Science type projects and for which numerous third-party libraries have been developed to make it easier to create a Data Science project.

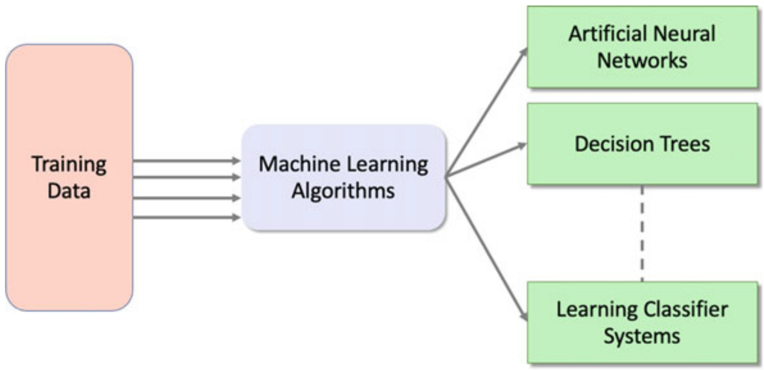
There are a range of commonly used Data Science libraries, we will look at several in the reminder of this book in particular we will consider:

- **NumPy** which provides sophisticated facilities for handling numbers.
- **SciPy** which stands for Scientific Python.
- **Pandas** is a data manipulation and analysis suite of modules. It provides facilities for reading data from a wide range of different formats, for managing and handling missing data, for reshaping data, etc.

- **Matplotlib** is a sophisticated graphing library already discussed earlier in this book.
- **Seaborn** is a statistical data visualisation library that is based on builds on Matplotlib. It provides a higher-level interface for creating and presenting statistical graphics.
- **SciKitLearn** provides a set of machine learning implementations that can be used with Data Science projects to create classifiers, regression predictors, clustering algorithms, dimensionality reduction libraries, etc.

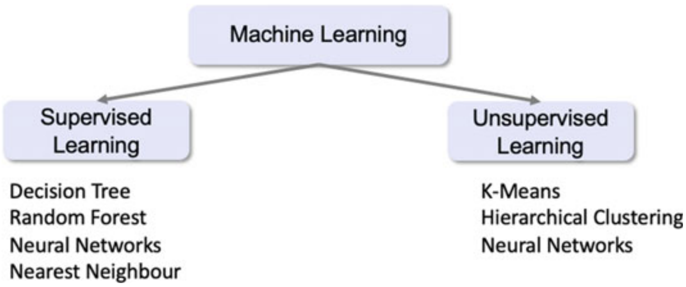
53.6 Machine Learning for Data Science

Following on from the previous section on data analysts it is not uncommon to apply some machine learning techniques when analysing data, for example to perform some classification of new data using previously seen data. That is, machine learning is a method of data analysis that automates analytical model building. Using algorithms that iteratively learn from data, machine learning allows computers to find hidden insights without being explicitly programmed. The general model is shown below:



This illustrates that training data (which may be what is called supervised or unsupervised) is fed into a machine learning algorithm and the output from this is a system that might be used to classify new data, predict behaviour based on new data or make some decision based on new data.

There are a range of machine learning algorithms (often referred to as models) provided by SciKitLearn that can be used to predict future patterns based on past data. These can be divided up into supervised and unsupervised learning systems.



A supervised learning system involves teaching a system, using data that is tagged or marked with known results. For example, loan data with applications marked as being either fraudulent or not, etc. In contrast an unsupervised learning system involves the system itself identifying patterns or clusters within the data that it is presented with. The term unsupervised is intended to indicate that there are no known (or at least provided) correct answers. This allows users to understand or learn more about the data. The term unsupervised is intended to indicate that there is no known (or at least provided) correct answers (and hence there is no teacher). In some situations there is a large amount of data, and only some of that data can be labelled with appropriate results or conclusions. Such problems are sometimes referred to as semi-structured machine learning. In many cases a combination of supervised and unsupervised techniques are required to analyse such data.

Each of the machine learning algorithms commonly used with Python has pros and cons and situations in which they are best applied. It is therefore useful to have a toolbox of such algorithms that can be applied as and when appropriate.

There are several commonly used libraries available in Python for this including SciKitLearn (aka SKLearn), TensorFlow and PyTorch. In general SciKitLearn and TensorFlow have about the same share of the Python market place with PyTorch being just behind, although all three are close to 30% in terms of Python developers usage.

In the remainder of this book we will be focussing on the use of supervised learning with *K*-Nearest Neighbour, Decision Trees and Random Forest machine learning algorithms.

53.7 Online Resources

For further information machine learning see:

- Harvard Business Review—"What every manager should know about Machine learning" http://hbr.org/2015/07/what-every-manager-should-know-about-machine-learning.

- Quick Reference cheat sheet on ML Algorithms/uses http://www.lauradhamilton.com/machine-learning-algorithm-cheat-sheet.
- 10 Machine Learning Methods that Every Data Scientist Should Know https://towardsdatascience.com/10-machine-learning-methods-that-every-datascientist-should-know-3cc96e0eeee9.
- https://numpy.org/ Numpy.
- https://scipy.org/ SciPy.
- https://pandas.pydata.org/ Pandas.
- https://matplotlib.org/ MatplotLib.
- https://seaborn.pydata.org/ Seaborn.
- https://scikit-learn.org/ SciKitlearn.

Chapter 54
Pandas and Data Analytics

54.1 Introduction

The COVID-19 pandemic has been the defining feature of the first years of the 2020s, and the related data has been the focus of much research. Many organisations, including Google and the UK Government, have made this data publicly available, so that it can be analysed by academics, students and other interested parties. Such analysis can be achieved using a variety of tools from MATLAB, through R to Python and its data analytics stack that includes the Pandas data analytics library and Matplotlib graphing library. In this chapter we will look at two data sets that can be obtained free from public sources and how the Python Pandas library can be used to perform some simple data analytics.

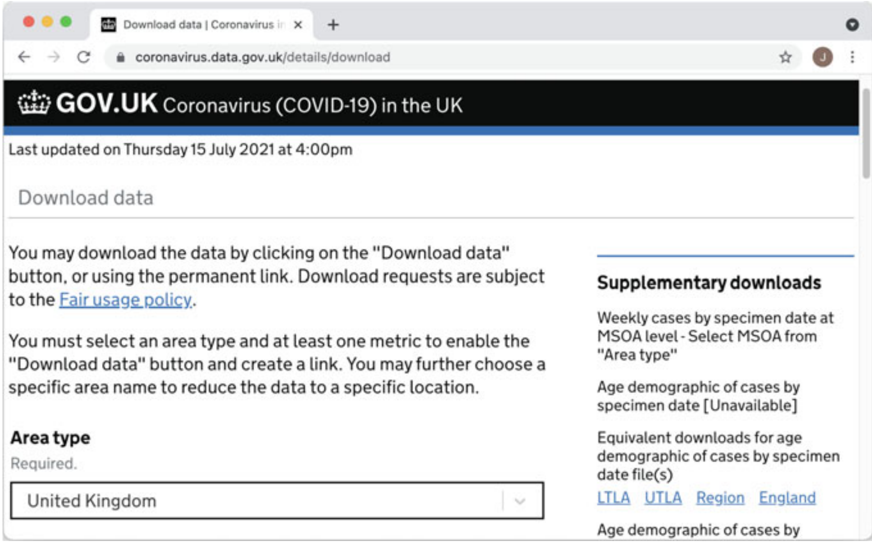
54.2 The Data

We shall be using two data sets to explore data around the COVID pandemic through the Python Pandas library. These data sets are described below. Both data sets are available within the GitHub repository for this chapter; see https://github.com/johnehunt/advancedpython3_2nd/tree/main/chapter54_pandas.

54.2.1 The UK Government COVID Data Set

The first data set is provided by the UK Government as part of its coronavirus (COVID-19) data provision. The web site is illustrated below:

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_54



On this site the user can select from various criteria to identify the region, the metrics, the date range of interest and the data format they require. For this chapter we selected the UK as the area, the latest data available for 2020, the CSV data format (as that is easy to work with) and a set of metrics to download. The metrics selected were daily hospital cases, daily new admissions, new COVID cases, as well as the number of people who had their first and second vaccinations, as shown below:

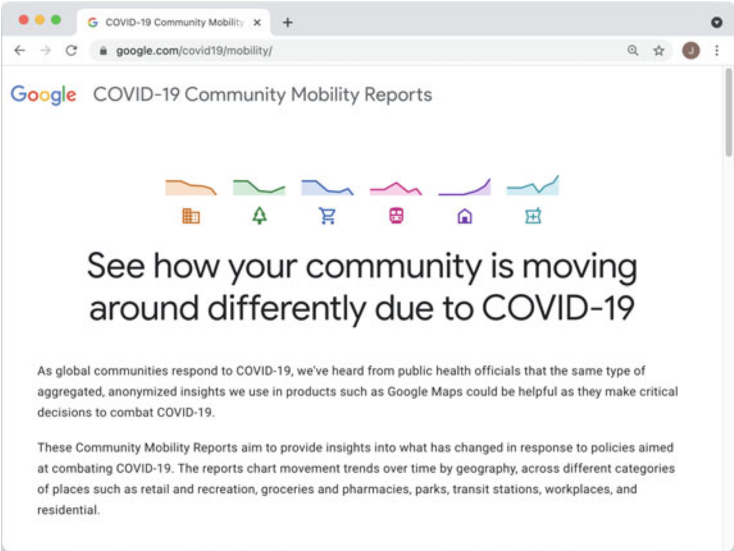
To make it possible to repeat such experiments, the web site also provides a permanent link for data request, which is given below:

https://api.coronavirus.data.gov.uk/v2/data?areaType=overview&metric=hospitalCases&metric=newAdmissions&metric=newCasesByPublishDate&metric=newPeopleVaccinatedFirstDoseByPublishDate&metric=newPeopleVaccinatedSecondDoseByPublishDate&format=csv.

We are using this data set from the middle of 2021 as the data at the top of the set is quite well populated and interesting. Although the latest data sets are available for 2023 in the UK, little data is now being recording regarding things such as new cases and numbers vaccinated.

54.2.2 The Google Mobility Data Set

The second data set to be used is provided by Google. It is accessible from the Google COVID-19 Community Mobility Reports site illustrated below:



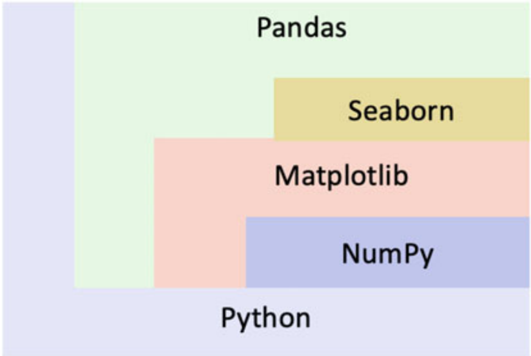
It makes available two files that can be downloaded; the first provides a global view of mobility, while the second provides separate data files for all the regions covered by Google. This second data set is the one which was downloaded for this chapter. This download is provided as a ZIP file. When unzipped it contains many different data files for different countries. For this chapter the GB data file was selected.

The data is quite wide ranging and includes information on the percentage change of Google users for visits and length of stay at different places such as retail locations, parks and transport hubs.

Both data sets will now be processed using the Python Pandas library.

54.3 Python Pandas

The Python Pandas library provides facilities that simplify accessing, processing and visualising data sets. Pandas itself builds on other libraries such the NumPy library. It is also commonly used with graphing libraries such as Matplotlib and Seaborn. The relationship between all of these and Python is illustrated below.



The intention of the above diagram is to illustrate how one library often builds on another but can be used independently or in combination.

Pandas is not provided as a default part of the Python environment and instead must be installed along with any other third-party libraries being used. This can be done using `pip` or `conda` (depending on how you are managing your Python environments). Pip is a tool used to install Python packages and is provided as part of Python. Conda, another package management tool, is part of Anaconda and is a popular choice for data scientists using Python (this is discussed in a later chapter). Both can be used to install Pandas as shown below:

```
pip install pandas
```

Or if you are using conda:

```
conda install pandas
```

Note that there are two major versions of Pandas available at the time of writing; there is the original 1.x series and there is the 2.x series. From a developers point of view both look very similar; however, Pandas 2.0 has had a significant re-write under the hood which has significantly improved its performance on very large data sets. We will be using Pandas 2.0.2 in these examples, and thus we will benefit from the performance benefits available.

To ensure that you are using the latest version of Pandas, you can explicitly specify the version to use, for example by specifying the version number when you install it, such as

```
pip install pandas=2.0.2
```

If however you wish to use an older version of Pandas, then you can also specify that—for example:

```
pip install pandas==1.5.3
```

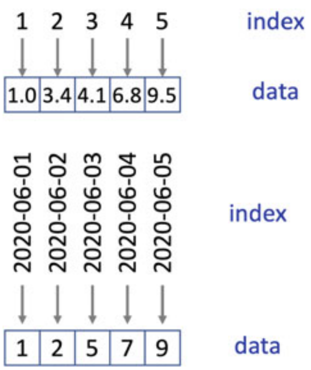
As we will also be using Matplotlib and Seaborn, we will need to install both these third-party libraries as well:

```
pip install matplotlib
pip install seaborn
```

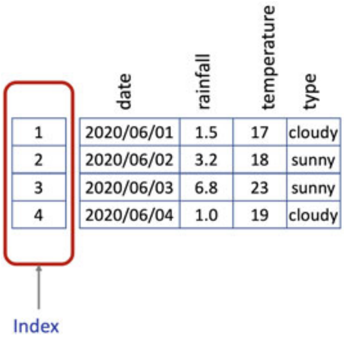
```
conda install matplotlib
conda install seaborn
```

54.3.1 Pandas Series and DataFrames

The key concepts within Pandas are the Series and the DataFrame. A series is a one-dimensional array like object that holds data of a specific type (such as an integer, a string or a float). Each value is accessible via an index value which can be a numeric integer such as 0 or 1 or a label such as a string or a timestamp. The following diagram illustrates two such series; one is indexed numerically from zero and holds a sequence of floats, and the second is indexed by a set of timestamps and holds a sequence of integers.



A DataFrame is a tabular structure, a bit like a spreadsheet. It has columns and rows. Each column has a type and a label. The columns are represented by series. Each row has an index. The index could be numeric or some other value such as a sequence of dates. A DataFrame can be visualised as shown below:



We will use DataFrames to load, process and analyse the COVID-related data sets.

54.4 Loading and Analysing UK COVID Data Set

The first thing we will do is load the CSV file, obtained from the UK Government COVID data portal, into a Pandas DataFrame. This is very easy to do with Panadas as it provides a function called `read_csv()` that can be used for this exact purpose (there are also other forms of the read function such as `read_excel` and `read_sql`).

All we need to do is import the Pandas library and then call the function. Note that it is a common convention within the Python world to alias Pandas to pd as a shorthand reference to the library.

The following code loads the data set into a DataFrame:

```
import pandas as pd
import seaborn as sns

sns.set()

df1 = pd.read_csv('overview_2021-07-15.csv')
```

The variable df1 now contains a reference to a Pandas `DataFrame`. There can be a lot of data held in a DataFrame which can make it hard to visualise; however the functions `head()` and `tail()` can be used to see the first few, and the last few, rows in the `DataFrame`:

```
print(df1.head().to_string())
print(df1.tail().to_string())
```

The above code obtains the head and the tail of the `DataFrame` and then converts them into a string format that is suitable for printing. The output from these is given below:

```
     areaCode        areaName  areaType        date  hospitalCases  newAdmissions
newCasesByPublishDate  newPeopleVaccinatedFirstDoseByPublishDate
newPeopleVaccinatedSecondDoseByPublishDate
0  K02000001  United Kingdom  overview  2021-07-15            NaN            NaN
48553                                       NaN
NaN
1  K02000001  United Kingdom  overview  2021-07-14         3786.0            NaN
42302                                   60374.0
185661.0
2  K02000001  United Kingdom  overview  2021-07-13         3623.0            NaN
36660                                   59073.0
158276.0
3  K02000001  United Kingdom  overview  2021-07-12         3415.0            NaN
34471                                   54296.0
125360.0
4  K02000001  United Kingdom  overview  2021-07-11         3144.0          582.0
31772                                   42000.0
107620.0
```

```
      areaCode        areaName  areaType        date  hospitalCases  newAdmissions
newCasesByPublishDate  newPeopleVaccinatedFirstDoseByPublishDate
newPeopleVaccinatedSecondDoseByPublishDate
527  K02000001  United Kingdom  overview  2020-02-04            NaN            NaN
0                                       NaN
NaN
528  K02000001  United Kingdom  overview  2020-02-03            NaN            NaN
0                                       NaN
NaN
529  K02000001  United Kingdom  overview  2020-02-02            NaN            NaN
0                                       NaN
NaN
530  K02000001  United Kingdom  overview  2020-02-01            NaN            NaN
0                                       NaN
NaN
531  K02000001  United Kingdom  overview  2020-01-31            NaN            NaN
2                                       NaN
NaN
```

This is still quite hard to read, not least as there are numerous columns in the data but also because of the presence of a lot of NaN values which indicate missing values.

Despite this we can already see a few things about this data; for example, there are *531* rows and that there are several columns which don't provide much in the way of meaningful information for us. For example, we know that this data is about the UK; thus areaCode, areaName and areaType which all indicate that this is data about the UK are not particularly useful and all have the same values. We can therefore choose to drop these columns from the DataFrame:

```
df1.drop(['areaCode', 'areaName', 'areaType'],
         axis='columns',
         inplace=True)
print(df1.head().to_string())
```

The `drop()` method of the `DataFrame` is given a list of the columns to remove, followed by the axis option which indicates its columns we are dropping not rows. The final parameter `inplace` indicates that it should change the DataFrame itself rather than generate a new DataFrame for the result.

Now when we look at the output from the DataFrame, we have less columns and less meaningless data:

```
         date  hospitalCases  newAdmissions  newCasesByPublishDate
newPeopleVaccinatedFirstDoseByPublishDate  newPeopleVaccinatedSecondDoseByPublishDate
0  2021-07-15            NaN            NaN                  48553
NaN                                           NaN
1  2021-07-14         3786.0            NaN                  42302
60374.0                                  185661.0
2  2021-07-13         3623.0            NaN                  36660
59073.0                                  158276.0
3  2021-07-12         3415.0            NaN                  34471
54296.0                                  125360.0
4  2021-07-11         3144.0          582.0                  31772
42000.0                                  107620.0
```

Of course, we can go further, we can now easily see that there is a date column with individual dates starting from July 15, 2021. However, at the moment all that the DataFrame knows about this column is that it contains objects (specifically strings). However, if we tell the DataFrame that this column actually represents dates, or more specifically datetime information, then we can perform date-related comparisons on it, such as selecting all rows between a start and end date.

We can tell the DataFrame that the date column should be treated as a set of `datetime` objects using the Pandas `to_datetime()` function:

```
df1['date'] = pd.to_datetime(df1['date'])
```

This code tells Pandas to convert the date column in the `df1` DataFrame to a set of `datetime` instances and then to overwrite the original column with the new version. Note how columns can be accessed using the square bracket (index) notation.

We can now perform several operations on the DataFrame using the date column:

```
# Sort the rows into ascending date order
df1.sort_values(by=["date"],
                ignore_index=True,
                inplace=True)
# Want to select 2020-02-15 to 2020-12-31 dates
# Set up a mask to indicate the date election
date_mask = (df1['date'] > '2020-02-14') & (df1['date'] <=
'2020-12-31')
# Select all the rows that meet the mask search criteria
df1 = df1.loc[date_mask]
```

This code snippet does three things:

1. It resorts the rows in the DataFrame into ascending (rather than descending) date order. This is done using the `sort_values()` method available from the DataFrame object. This function takes three parameters:

   a. `by`—which is used to indicate the column to use for sorting.
   b. `ignore_index` which indicates that the current index should be ignored.
   c. `inplace` option as used previously.

2. We then want to select just those rows associated with dates between February 14, 2020, and December 31, 2020. This is because we are interested only in the 2020 data, and the Google data only covers from February 14, 2020, onwards. To do this, we set up a mask. This returns a series of `True` and `False` values for each row in the DataFrame. If a row meets the condition specified in the test, then the value True is generated; otherwise it is False.
3. The DataFrame `loc()` method is then used to select the rows which match the condition in the mask. The `loc()` method will return each row where there is a corresponding True in the `date_mask`.

If we now look at the head and tail of the `DataFrame,` we can see:

```
         date  hospitalCases  newAdmissions  newCasesByPublishDate
newPeopleVaccinatedFirstDoseByPublishDate  newPeopleVaccinatedSecondDoseByPublishDate
15 2020-02-15            NaN            NaN                      0
NaN                                          NaN
16 2020-02-16            NaN            NaN                      0
NaN                                          NaN
17 2020-02-17            NaN            NaN                      0
NaN                                          NaN
18 2020-02-18            NaN            NaN                      0
NaN                                          NaN
19 2020-02-19            NaN            NaN                      0
NaN                                          NaN
```

```
          date  hospitalCases  newAdmissions  newCasesByPublishDate
newPeopleVaccinatedFirstDoseByPublishDate  newPeopleVaccinatedSecondDoseByPublishDate
331 2020-12-27        22764.0         2870.0                  30501
NaN                                          NaN
332 2020-12-28        24049.0         3133.0                  41385
NaN                                          NaN
333 2020-12-29        25549.0         3249.0                  53135
NaN                                          NaN
334 2020-12-30        26551.0         3288.0                  50023
NaN                                          NaN
335 2020-12-31        26578.0         2915.0                  55892
NaN                                          NaN
```

The above output shows that the *head of the data* starts on February 14 and the *tail* ends on December 31, 2020, and that there are now 335 rows of data.

We can also see that there are still a large number of `NaN` values in the printout. It can be useful to see how many such values there are for each column. We can do this by testing each value in the DataFrame for `NaN` (or nothing or null) and then counting the results for each column. As the DataFrame allows operations to be performed in batch (as efficiently as possible) we can do this using two batch functions `isnull()` and `sum()`:

```
is_null_count = df1.isnull().sum()
print(is_null_count)
```

From this we get:

```
date                                          0
hospitalCases                                41
newAdmissions                                37
newCasesByPublishDate                         0
newPeopleVaccinatedFirstDoseByPublishDate   321
newPeopleVaccinatedSecondDoseByPublishDate 321
dtype: int64
```

From this we can see that the last two rows in the above list indicate that the associated columns provide virtually no significant information as they contain no significant data (this is probably because the data is from 2020 and prior to the main vaccine roll out in the UK). We can therefore also drop these columns from our DataFrame:

```
df1.drop(['newPeopleVaccinatedFirstDoseByPublishDate',
          'newPeopleVaccinatedSecondDoseByPublishDate'],
         axis='columns',
         inplace=True)
```

Now if we print a random sample of the DataFrame, we can see that we have a simpler set of data with mostly meaningful information:

```
# Select a random sample of 10 rows from the DataFrame
print(df1.sample(10).to_string())
```

Some of the data generated by the above statement is given below, note as we used the sample() method the rows are selected randomly and are not in order:

```
          date  hospitalCases  newAdmissions  newCasesByPublishDate
198 2020-08-16          979.0           80.0                   1040
208 2020-08-26          829.0          132.0                   1048
145 2020-06-24         4030.0          368.0                    653
278 2020-11-04        13817.0         1677.0                  25177
71  2020-04-11        21254.0         2162.0                   5234
315 2020-12-11        16885.0         1686.0                  21672
113 2020-05-23         8814.0          643.0                   2959
304 2020-11-30        16564.0         1404.0                  12330
46  2020-03-17            NaN            NaN                    407
182 2020-07-31         1266.0          114.0                    880
```

54.5 Loading the Google Mobility Data Set

If we now wished to see how movement changed during the 2020 year, we can take the Google Mobility Data Set for the UK and perform similar analysis. The following code does exactly this:

```
# Load the google Mobility data for the UK
df2 = pd.read_csv('2020_GB_Region_Mobility_Report.csv',
low_memory=False)

# Drop columns that do not provide any additional data
df2.drop(['country_region_code',
          'country_region',
          'sub_region_1',
          'sub_region_2',
          'metro_area',
          'iso_3166_2_code',
          'census_fips_code',
          'place_id'],
         axis='columns',
         inplace=True)
df2['date'] = pd.to_datetime(df2['date'])
df2.rename(columns={'retail_and_recreation_percent_change_
from_baseline': 'retail_and_recreation_change'}, inplace=True)

# Pick up the first 322 rows
df2 = df2.head(321)

print(df2.sample(8).to_string())
```

The only additional to the previous code is that we have renamed the `retail_and_recreation_percent_change_from_baseline` column to `retail_and_recreation_change` as it is shorter to reference. The output from the above code is

```
          date  retail_and_recreation_change
grocery_and_pharmacy_percent_change_from_baseline  parks_percent_change_from_baseline
transit_stations_percent_change_from_baseline  workplaces_percent_change_from_baseline
residential_percent_change_from_baseline
44  2020-03-30                           -74.0
-32.0                                    -33.0
-73.0                                    -69.0                                       27.0
96  2020-05-21                           -64.0
-16.0                                     51.0
58.0                                     -59.0                                       23.0
90  2020-05-15                           -69.0
-21.0                                     21.0
60.0                                     -60.0                                       24.0
37  2020-03-23                           -45.0
-8.0                                       1.0                                      -53.0
-45.0                                     17.0
29  2020-03-15                           -11.0
7.0                                       -1.0                                      -16.0
-4.0                                       2.0
202 2020-09-04                           -21.0
-10.0                                     61.0
35.0                                     -40.0                                        9.0
309 2020-12-20                           -43.0
-7.0                                       5.0                                      -45.0
-13.0                                      7.0
142 2020-07-06                           -36.0
-11.0                                     76.0
43.0                                     -48.0
14.0
```

54.6 Merging Two DataFrames

We now have two DataFrames both containing data where there are associated dates and COVID-related information. As we would like to work with these two DataFrames together, we will merge the two DataFrames we have previously obtained, using the Pandas merge() function. This will join the two DataFrames based on the date with each row holding data from both DataFrames:

```
df3 = pd.merge(df1, df2, on='date')
print(df3.sample(8).to_string())
```

The DataFrame held in the variable `df3` now holds all the columns from the previous two DataFrames:

```
         date  hospitalCases  newAdmissions  newCasesByPublishDate
retail_and_recreation_change  grocery_and_pharmacy_percent_change_from_baseline
parks_percent_change_from_baseline  transit_stations_percent_change_from_baseline
workplaces_percent_change_from_baseline  residential_percent_change_from_baseline
107 2020-06-01         7248.0         604.0                   1570
-56.0                                                  -13.0
74.0                                              -51.0
-53.0                                          19.0
152 2020-07-16         1829.0         166.0                    642
-31.0                                                  -10.0
85.0                                              -45.0
-45.0                                          14.0
128 2020-06-22         4281.0         394.0                    958
-45.0                                                  -13.0
72.0                                              -49.0
-51.0                                          17.0
188 2020-08-21          888.0         109.0                   1033
-26.0                                                  -13.0
56.0                                              -40.0
-45.0                                          12.0
22  2020-03-08            NaN           NaN                     65
4.0                                                     2.0
15.0                                                2.0
1.0                                          -1.0
112 2020-06-06         6149.0         406.0                   1557
-70.0                                                  -22.0
-17.0                                               -57.0
-33.0                                          14.0
273 2020-11-14        16299.0        1613.0                  26860
-59.0                                                  -16.0
-29.0                                               -59.0
-28.0                                          13.0
30  2020-03-16            NaN           NaN                    152
-3.0                                                   20.0
29.0                                              -16.0
-9.0                                           4.0
```

54.7 Analysing the Combined Data

We can now look at the relationships between data from the UK government and from Google COVID mobility portals. For example, we can look to see if there are any positive or negative correlations between this data:

```
print(df3.corr().to_string())
```

This produces the following grid:

```
                                                        date  hospitalCases
newAdmissions  newCasesByPublishDate  retail_and_recreation_change
grocery_and_pharmacy_percent_change_from_baseline  parks_percent_change_from_baseline
transit_stations_percent_change_from_baseline  workplaces_percent_change_from_baseline
residential_percent_change_from_baseline
date                                               1.000000       0.130890
0.081505               0.733346                      0.128420
0.142496                           0.129175
 -0.123013                              0.048206
 -0.090359
hospitalCases                                      0.130890       1.000000
0.900970               0.637143                     -0.551667
-0.251541                          -0.733562
-0.689504                              -0.359466
0.530903
newAdmissions                                      0.081505       0.900970
1.000000               0.618497                     -0.490277
-0.278846                          -0.736805
-0.672790                              -0.374371
0.521693
newCasesByPublishDate                              0.733346       0.637143
0.618497               1.000000                     -0.074673
0.081381                          -0.335174
-0.251770                              -0.069931
0.127715
retail_and_recreation_change                       0.128420      -0.551667
-0.490277             -0.074673                      1.000000
0.765417                           0.375138
0.890613                               0.647121
-0.766577
grocery_and_pharmacy_percent_change_from_baseline  0.142496      -0.251541
-0.278846              0.081381                      0.765417
1.000000                           0.212263
0.698326                               0.549779
-0.539766
parks_percent_change_from_baseline                 0.129175      -0.733562
-0.736805             -0.335174                      0.375138
0.212263                           1.000000
0.267764                               0.036072
-0.249077
transit_stations_percent_change_from_baseline     -0.123013      -0.689504
-0.672790             -0.251770                      0.890613
0.698326                           0.267764
1.000000                               0.839920
-0.882870
workplaces_percent_change_from_baseline            0.048206      -0.359466
-0.374371             -0.069931                      0.647121
0.549779                           0.036072
0.839920                               1.000000
-0.939791
residential_percent_change_from_baseline          -0.090359       0.530903
0.521693               0.127715                     -0.766577
-0.539766                          -0.249077
-0.882870                              -0.939791
1.000000
```

There is a lot of data here, but one of the things we can see is that there appears to be a negative correlation between `HospitalCases` and `retail_and_recreation_change`.

This might be useful to visualise on a graph. We can therefore generate a new DataFrame containing just the date and the `HospitalCases` and `retail_and_recreation_change` data:

```
df4 = pd.concat([df3['date'],
                 df3['hospitalCases'],
                 df3['retail_and_recreation_change']], axis=1)
```

We can now choose to plot this data in different ways. For example, we might decide to generate a simple line graph that illustrates the relationship between these two data columns. To do this, we are using Matplotlib and Seaborn; this means we need to import those libraries so that they are accessible to our program and initialise the graphing configuration values using `sns.set()`:

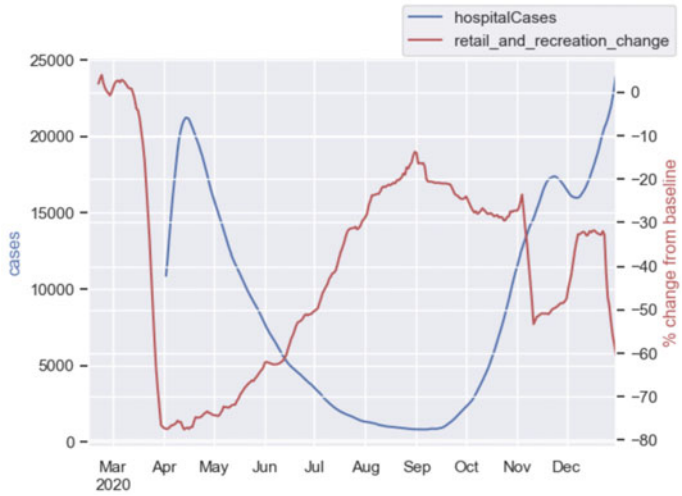
```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set()

# Lets compare the hospital cases against retail and recreation
change
axis1 = df4.plot(x="date", y="hospitalCases", legend=False)
axis2 = axis1.twinx()
df4.plot(x="date",
         y="retail_and_recreation_change",
         ax=axis2,
         legend=False,
         color="r")
axis1.figure.legend()
plt.show()
```

Note that because the two columns have different value ranges, we need to plot them on two separate axes to see their respective changes. To do this we create two axis objects one for the `hospitalCases` and one for the `retail_and_recreation_change` data. We then plot the graph. The resulting graph is shown below:

This graph does highlight the trends in the data, but there is certainly quite a lot of noise in the red line. We might therefore decide to smooth out the date as bit to see the general trends. We can do this by smoothing out each data point by taking an average across the seven preceding days. This can be done using the `rolling()` function and then taking the mean of the rolling values.

```
df5 = pd.concat([df3['date'],
                df3['hospitalCases'].rolling(7).mean(),

df3['retail_and_recreation_change'].rolling(7).mean()], axis=1)
ax1 = df5.plot(x="date", y="hospitalCases", legend=False,
color="b")
ax2 = ax1.twinx()
df5.plot(x="date",
         y="retail_and_recreation_change",
         ax=ax2,
         legend=False,
         color="r")
ax1.set_ylabel('cases', color='b')
ax2.set_ylabel('% change from baseline', color='r')

ax1.figure.legend()
plt.show()
```

These new values can then also be graphed and appropriate labels added:



54.8 Summary

We can quickly build, manipulate and visualise data thanks to Python and free open-source tools such as Pandas, Matplotlib and Seaborn.

In this relatively simple instance we've brought together two sets of freely downloadable UK data to display how the number of COVID-related hospital cases correlated to the movement of shoppers in 2020—the relationship between the two factors is rather striking and a good demonstration of the power of data visualisation.

You can apply these data analysis techniques to a huge variety of data types, at scale. It is possible to extract and refine valuable information from virtually any source.

Chapter 55
Alternatives to Pandas

55.1 Introduction

This chapter considers alternatives to Pandas 2.x including Pandas 1.x, Polars and R.

55.2 Comparing Pandas 2.0.0

If you search on the web, you will find many articles comparing Pandas with a range of tools, from R and PySpark to tools such as Excel, SAS or SQL databases. Some of these are relevant (such as PySpark); others are less appropriate such as SQL. In this section we will quickly compare and contrast several additional tools.

55.3 Pandas 1.x v 2.x

It is perhaps worth starting off by considering how Pandas 2.0 differs from Pandas 1.x.

Various performance enhancements have been implemented in Pandas 2.0 such as copy-on-write enhancements. As an example, a new lazy copy mechanism that defers the copy until the object in question is modified.

One criticism that was levelled at Pandas 1.x related to memory usage on very large data sets. Originally, Pandas was built on top of NumPy data structures.

To explain why this is important, it is useful to understand how Pandas is usually used. In most cases, some data set is loaded from some source such as a file, a CSV file, an Excel file and a database. This can be done using one of the read functions such as `read_csv`, `read_excel` or `read_sql`. This data is loaded into the

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_55

Python/Pandas program, and Pandas determines how to represent that data. This representation is quiet straightforward for integers and floating point numbers but for strings, dates, times, etc., dome processing/decision-making is required as to the best way to hold that data. The fundamental types in Python such as list, tuple and dictionary are not designed for holding large amounts of data and can become very slow if they are used. Pandas therefore chooses to use another representation for arrays and that representation was to use NumPy.

At the time, this was a very good idea at the time as NumPy is very well established and provided significant out of the box support that could be built upon to provide data processing and data analytics facilities.

However, since then various rival libraries or add-ons to Pandas have attempted to improve on the performance of NumPy and NumPy arrays as they were perceived to be too 'slow' for many modern applications and its lack of support for strings and missing values. Indeed Wes McKinney has written an article outlining why NumPy was no longer the best choice now for Pandas in "Apache Arrow" and the "10 Things I Hate About pandas".

In fact, Pandas has been working towards decoupling itself from NumPy for several years (at least since 2018), and with Pandas 2.0.0 PyArrow is now used to support all data types.

55.4 Pandas Versus Other Libraries and Tools

Pandas Versus NumPy

This is not an uncommon analysis to find if you go searching. However, as has been indicated above it is not really an appropriate one as Pandas 1.x was directly built on top of NumPy and Pandas 2.0 can still use NumPy or provide PyArrow as an alternative representation. Perhaps what is worth say is that if all you need is the facilities that NumPy provides, then Pandas is probably overkill for you!

Pandas Versus SciPy

Another comparison you will find is with SciPy (or Scientific Python). Interestingly SciPy is not strictly required for Pandas but is listed as an *optional dependency*. SciPy is another open-source Python library this time oriented around mathematics, science and engineering tasks as it contains modules for linear algebra, integration, interpolation, FFT, image processing, etc. As such SciPy is complementary to Pandas rather than an alternative to it.

Pandas Versus PySpark

PySpark is a Python library for the Scala (and Java)-based Spark ecosystem of tools. Spark is a framework for working with large data sets in a distributed computing environment. In contrast Pandas fundamentally runs operations on a single computer. Thus if you want to exploit the benefits of a set of networked computers for your data analytics tasks, then PySpark may well be a preferable choice. However, PySpark is more complex than Pandas and obviously involves a distributed computing environment and thus requires more set-up than Pandas. In this comparison these are markedly different tools aimed at significantly different sized tasks.

Pandas Versus Dask

The last section indicated that Pandas is primarily a single computer-based solution, whereas PySpark was a multicomputer-based solution. However Dask changes things a little. Dask is library that allows Pandas (and in fact other libraries such as NumPy itself and SciKitLearn) to scale across a distributed computing environment. It is thus possible to use Dask to make Pandas operate in a multiple computer environment and take advantage of multiple computer systems for a data analytics task. This may well be a preferable approach to PySpark if you are already familiar with Pandas. Thus, Dask is not really a competitor of Pandas but an enabler!

Pandas Versus Polars

This is perhaps the most appropriate comparison in this list. Polars is a direct rival to Pandas in terms of its aims and objectives with the added focus of aiming to be faster and more efficient than Pandas. Its main claim to performance is that it uses an arrow-based representation and natively parallelises processing of the data. In actual fact Polars has two APIs an eager API and a lazy API. The eager API is similar in its execution to Pandas. The lazy API only runs code when it needs to which can, in some cases, further improve performance.

Compared to Pandas 1.x Polars has significant performance improvements. However, compared to Pandas 2.0 the improvements are either less marked, not really noticeable or indeed not present as in some cases Pandas 2.0 may be faster. Polars is still an alternative to Pandas 2.0, but the results are less clear cut and any developer who wishes to select one over the other should perform their own performance benchmark tests to determine which suits them best.

One point to note is that there is far more documentation and examples available for Pandas than there are for Polars.

Pandas Versus R

This is an interesting comparison as Pandas is a data analytics library built on top of the general-purpose programming language Python. In contrast R is a programming language and environment designed for statistical computing and graphics. As such there is a fundamental difference in philosophy behind these two approaches. For many developers coming from a traditional programming background Python is a simple to learn programming language which benefits from the huge ecosystem of open-source and commercial libraries available for it. This of course includes Pandas but also all the GUI, graphing, database, restful, etc., libraries available in the market, whereas R is a slightly esoteric programming language with its own set of libraries and ecosystem of add-ons which are not as extensive as Pythons.

55.5 Online Resources

See the following online resources for information on the topics in this chapter:

- NumPy https://numpy.org.
- SciPy https://scipy.org/.
- PySpark https://pypi.org/project/pyspark/.
- Dask https://docs.dask.org/en/stable/.
- Polars https://www.pola.rs/.
- R Programming Language https://www.r-project.org/.
- PyArrow https://arrow.apache.org/docs/python/index.html.
- Apache Arrow and the "10 Things I Hate About pandas" https://wesmckinney.com/blog/apache-arrow-pandas-internals/.
- See 'Enhancing Performance' in Pandas 2.0 for more information. https://pandas.pydata.org/docs/user_guide/enhancingperf.html.
- SAS https://www.sas.com/en_gb/home.html/.

Chapter 56
Machine Learning in Python

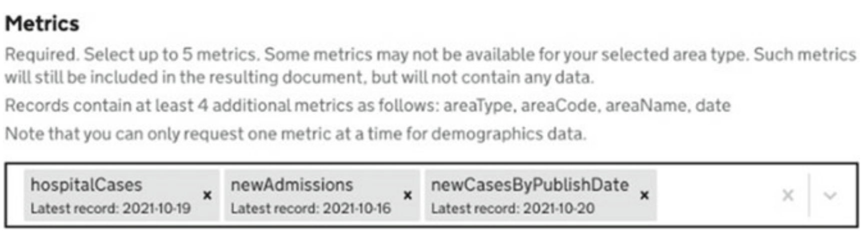
56.1 Introduction

This chapter explores the use of machine learning algorithms to perform further Data Science types takes. It does this by taking the data sets used in the Pandas and data analytics chapter and applies several machine learning systems to create predictors for people's behaviour based on the 2020 COVID data.

56.2 The Data

As with the previous chapter, we shall be using two data sets that provide information on the COVID pandemic. The data sets used for this chapter are available within the GitHub repository for this chapter; see https://github.com/johnehunt/advancedpython3_2nd/tree/main/chapter55_machine_learning.

The first data set is provided by the UK Government as part of its coronavirus (COVID-19) data provision. On their web site it is possible to download data relating to the COVID pandemic using a web from. The user can select the metrics of interest, for example:



J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_56

In this case the metrics represent the number of hospital cases, the number of new admissions and the number of new cases by published date.

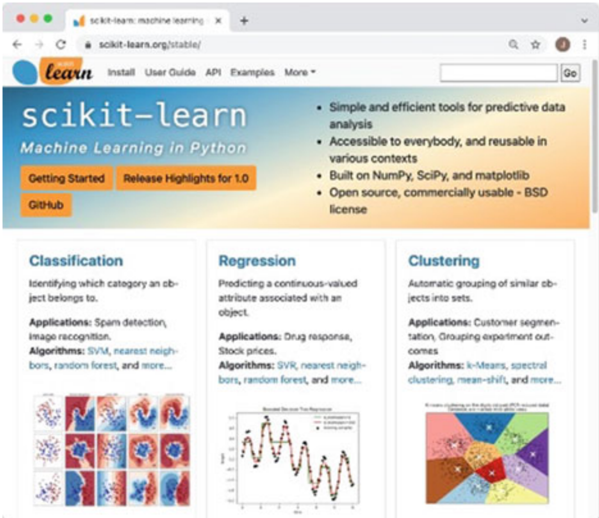
To make it easy to repeat such downloads, they provide a permanent link that allows the same data to be downloaded. The link used to download the data for this chapter is:

https://api.coronavirus.data.gov.uk/v2/data?areaType=overview&metric=hospitalCases&metric=newAdmissions&metric=newCasesByPublishDate&format=csv.

The second data set is provided by Google and is again the Google COVID-19 Community Mobility Reports site. This download is provided as a ZIP file. When unzipped, it contains many different data files for different countries. For this blog the GB data file was selected.

56.3 SciKitLearn

To handle creating various machine learning algorithms/models, we shall be using the SciKitLearn library.



This is a Python library that is built on top of NumPy, SciPy and Matplotlib and provides implementations for a wide range of machine learning approaches including those used for classification, regression, clustering, etc. SciKitLearn is very widely used, is open source and can be used for commercial applications. It is of course not the only library available, and other widely used Python machine learning libraries are TensorFlow and PyTorch.

To install SciKitLearn, you need to add it to your Python interpreter. This can be done using either `pip` or `conda`:

```
pip install scikit-learn
conda install -c condo-forge scikit-learn
```

The version of SciKitLearn used at the time of writing was version 1.2.

In this chapter we will be focussing on the use of supervised learning with K-Nearest Neighbour, Decision Trees and Random Forest machine learning algorithms.

56.4 The Problem

The aim of this work is to generate a system that can predict the percentage change in *retail and recreational mobility* based on the number of hospital cases, new hospital admissions and COVID numbers.

To do this, we need to take the data provided by the UK governments COVID site and the Google Mobility Data and merge it together into a single data set. This process was described in the previous chapter and is therefore not repeated here.

Having obtained the relevant data we now need to split that data into a set of data to be used for *training* the learning system and a set of data that we can use to *test* its performance. To do this, we use the `train_test_split()` function provided by the SciKitLearn library. This function randomly splits the data set into train and test subsets. Using the `test_size` parameter it is possible to indicate the percentage split between training and testing. For example, using `test_size=0.2` indicates that the train set will hold 80% of the data and the test set will hold 20% of the data.

This is illustrated below. Note that `df` holds a Pandas `DataFrame` and that `train_test_split()` returns two DataFrames held in train and test:

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv(merged_covid_data.csv')
train, test = train_test_split(df, test_size=0.2)
```

Based on this split we can now extract the columns in the DataFrame that will be used as the input values (the feature set) and the output values (the target variable). To do this we can indicate to the train and test DataFrame:

```
FEATURES_LIST = ['hospitalCases',
                 'newAdmissions',
                 'NewCasesByPublishDate']
TARGET_VARIABLE = 'retail_and_recreation_change'

training_features = train[FEATURES_LIST].values
training_target = train[TARGET_VARIABLE].values

test_features = test[FEATURES_LIST].values
test_target = test[TARGET_VARIABLE].values
```

A `DataFrame` allows one or more columns to be specified using the square brackets accessor operator (e.g. `[TARGET_VARIABLE]`). These can then be converted into a simple NumPy `ndarray` using the values attribute. These ndarrays can then be used with the various learning systems for training and testing purposes.

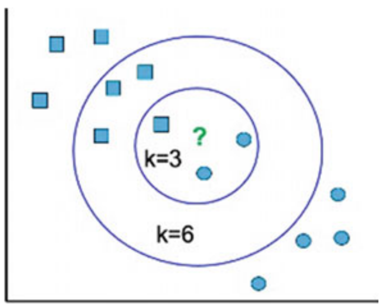
56.5 Using Regression Supervised Learning Systems

In this chapter we will be using three *regression* supervised learning systems. We are using regression models as they can be used to predict a continuous quantity output, whereas machine learning classification algorithms predict a set of discreet values.

We will be using the *K*-Nearest Neighbour Regressor, the Decision Tree Regressor and the Random Forest Regressor from the SciKitLearn library.

56.6 *K*-Nearest Neighbour Regressor

K-Nearest Neighbour (or *K*NN) is one of the simplest and most widely used algorithms within the machine learning world. *K*NN uses past data to predict the classification of new data based on a similarity measure (the nearest neighbour aspect). The closer a new data point is to a group of related data points then the more likely it is to be classified as being of that type of data. The *k* value in the nearest neighbour algorithm indicates how may nearby data points should be considered, to determine the classification of the new value. For example:



In the above diagram, if *k* (the number of nearest neighbours to be considered) is 3, then the ? data point will be classed as a circle. However, if *k* is set to 6, then the ? data point will be classed as a square.

SciKitLearn provides the `KNeighborsRegressor` class as an implementation of a *K*NN regressor. This class can be instantiated with the value of *k* as a parameter (called *n*_neighbours). For example:

```
knn_model = KNeighborsRegressor(n_neighbors=3)
```

The above line of code creates a new *K*NN model instance initialised with the value of *k* (*n*-neighbours) set to 3.

We can now train the *K*NN model object using the training feature set and the training target output, for example:

```
model.fit(training_features, training_target)
```

Once this has run the *K*NN model object has been *taught* based on the training data.

The question now is how well has this worked?

There are two metrics commonly used to assess how *good* a learning system such as the *K*NN regressor is; these are Root Mean Square Error (RMSE) and *R*-squared:

- **RMSE** is a standard way to measure the error of a model in predicting data. It is used in machine learning as a way of evaluating the accuracy (or usefulness) of a trained model. In general, the smaller the RMSE value the better the result. Of course, *small* is a relative term, but when an RMSE value is compared with the RMSE values obtained for alternative models on comparative data sets, it can be used as a relative guide to the utility of one model against another.
- **R-squared** is a statistical measure of how close the data is to the fitted regression line. In other words, it indicates how well the model fits the data as a percentage, with zero indicating a very poor fit and 100% indicating a perfect fit. Thus, the higher the *R*-squared value the better.

We can thus use these metrics to assess how well our *K*NN model fits the data it was trained with as well as how successful it is with the test data (it has never seen before). The following code does exactly that using functions provided by NumPy and SciKitLearn:

```
print(f'KNeighborsRegressor(n_neighbors=3)')

# Determine the metrics - against the training data
pred_train_rf = model.predict(training_feature_set)
trainingRMSE    =    np.sqrt(mean_squared_error(training_target_
attribute, pred_train_rf))
trainingRSquared = r2_score(training_target_attribute,
                            pred_train_rf)
trainingRSquared *= 100

# Determine the metrics based on the test dataset
pred_test = model.predict(test_feature_set)
testingRMSE = np.sqrt(mean_squared_error(test_target_attribute,
                                         pred_test))
testingRSquared = r2_score(test_target_attribute, pred_test)
testingRSquared *= 100
```

The values obtained for the training and testing *R*-squared values are multiplied by 100 to convert them into a normal percentage for printing purposes.

The results obtained from running the metrics against the trained *K*NN model are presented below:

```
KNeighborsRegressor(n_neighbors=3)
Testing KNN against Training data
Training RMSE - 6.8
Training R-squared - 91.5%
Testing KNN against test data
Testing RMSE - 10.3
Testing R-squared - 84.3%
```

In the above metrics we can see that the RMSE values are quite low, and the *R*-squared values are both over 80%. However, can we do better? In the next section we will explore the Decision Tree Regressor model.

56.7 Decision Tree Regressor

A Decision Tree learning system is one that builds a single rooted Decision Tree, typically used to classify future values. The tree is built in general, by splitting the data at each point in the tree based on an algorithm used to determine the best way to differentiate between that data. There are several algorithms used to create a Decision Tree such as ID3, C4.5 and CART (Classification and Regression Tree).

The SciKitLearn library provides the `DecisionTreeRegressor` class. This class follows a similar pattern to the *K*NN regressor. This means that we must instantiate the class and configure the regressor as appropriate. For example:

```
dt_model = DecisionTreeRegressor(max_depth=4,
                                 min_samples_leaf=0.13,
                                 random_state=3)
```

In this case we have configured the `max_depth` of the tree to be `4`. We have also set the `min_samples_leaf` parameter to `0.13`. This parameter is used to indicate the number of samples required before a leaf node is split into a branch (where 0.13 represents 13%). Finally, `random_state` is set to 3 to ensure that the behaviour of the tree is deterministic.

Next, we can train the Decision Tree model instance. Note that we do this in the same way as we did for the *K*NN model instance:

```
dt_model.fit(training_features, training_target)
```

We can now obtain the RMSE and *R*-squared metrics for the `dt_model`. Again, this is done in the same way as for the *K*NN model. This illustrates the common pattern being provided by SciKitLearn for each of the learning systems we are working with.

The results obtained for the Decision Tree Regressor are

```
DecisionTreeRegressor(max_depth=4, min_samples_leaf=0.13,
random_state=3)
Testing Decision Tree against Training data
Training RMSE - 11.1
```

```
Training R-squared - 81.4%
Testing Decision Tree against test data
Testing RMSE - 10.7
Testing R-squared - 80.0%
```

Interestingly these results are not generally as good as for the *K*NN example on the training data. The RMSE value for the *K*NN was 7.2, but here it is 11.1 and the *R*-squared value was 91.3%, whereas here it is 81.4%. However, the test data results are slightly better. For the *K*NN the RMSE value was 11.1, but here it is 10.7 and the *R*-squared results for *K*NN were 77.9, but here they are 80%. However, overall the results are very close and given that the training and testing data is generated randomly, if we ran both experiments again we might get slightly different results. Thus overall, there is not much to separate the two machine learning approaches.

The question is can we do better? One option would be to explore different settings for the DecisionTreeRegressor such as modifying the maximum depth of the tree or changing the percentage required before a leaf node is split into a branch. Alternatively, we could choose to try the Random Forest model (which is essentially a forest of Decision Trees).

56.8 Random Forest Regressor

Decision Trees are useful, but they can overfit the training data, Random Forest overcomes this by essentially creating a forest of Decision Trees. The SciKitLearn library provides the RandomForestRegressor class that can be used to create an implementation of a Random Forest regressor that can be trained in a similar manner to the Decision Tree and *K*NN models.

To create a Random Forest regressor object, we use the `RandomForestRegressor` class and set the parameters used with this class as appropriate, for example:

```
rf_model = RandomForestRegressor(max_depth=4,
                                 n_estimators=500)
```

In this case we have set the maximum depth of the tree and the `n_estimators` parameter (which indicates the size of the forest).

Next, we train the Random Forest object as before:

```
rf_model.fit(training_features, training_target)
```

Finally, we obtain the metrics for the Random Forest model:

```
RandomForestRegressor(max_depth=4, n_estimators=500)
Testing Random Forest against Training data
Training RMSE - 7.4
Training R-squared - 92.4%
Testing Random Forest against test data
Testing RMSE - 9.6
```

```
Testing R-squared - 91.8%
```

Again, the RMSE values are not that dissimilar to those obtained for both the *K*NN and Decision Tree models on the training data. However, the results obtained for the test data are significantly better.

56.9 Summary of Metrics Obtained

The following table summarises the metrics obtained for the three regression models we have created.

Model	Training		Testing	
	RMSE	*R*-squared (%)	RMSE	*R*-squared (%)
*K*NN	6.8	91.5	10.3	84.3
Decision tree	11.1	81.4	10.7	80
Random forest	7.4	92.4	9.6	91.8

Overall, there is not that much difference between the *K*NN and Decision Tree models on the testing data, but the Random Forest approach represents a significant improvement on both for the test data. We shall therefore create a Random Forest Regression-based predictor for the new data obtained for 2021.

56.10 Creating the Regressor Object

We can same our predictor object (implemented using the `RandomForestRegressor` class) to a binary format file using the pickle package provided with Python:

```
import pickle

file = open('random_forest_regressor.pkl', "bw")
pickle.dump(rf_model, file)
```

Once we have done this we can reload and use this regressor to process new data whenever required. This is done again using the Python pickle page:

```
file = open('random_forest_regressor.pkl', "br")
regressor = pickle.load(file)
```

For example, we can use it to analyse COVID data for 2021 to predict the percentage change in retail and recreational mobility.

Using the Pandas library, we can load and configure the CSV file containing the data for 2021. This can involve ordering the data by date, removing columns not used by the regressor object as shown below:

```
df = pd.read_csv('covid_data_2021_only.csv')
df.sort_values(by=["date"], ignore_index=True, inplace=True)
# Store date column for use with output
dates = df['date']

# Drop date as now used in classifier
df.drop(['date'], axis='columns', inplace=True)

# Make sure all columns have a value even if its Zero
df['hospitalCases'] = df['hospitalCases'].fillna(0)
df['newAdmissions'] = df['newAdmissions'].fillna(0)
df['newCasesByPublishDate'] =
df['newCasesByPublishDate'].fillna(0)
```

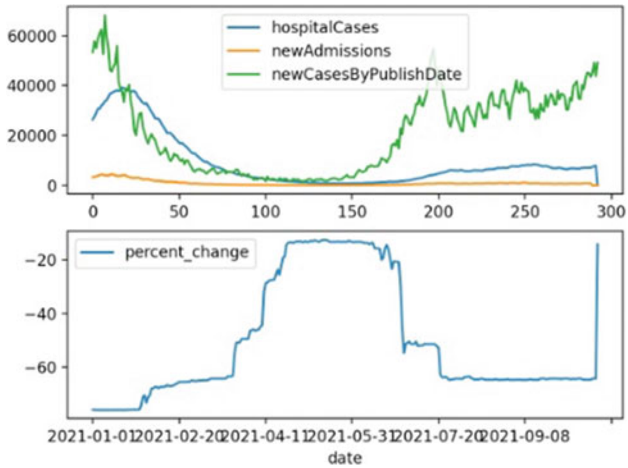
We can use the regressor object to predict the retail and recreational percentage change for the COVID 2021 data using the `predict()` method:

```
predictions = regressor.predict(df)
```

The results variable now holds the predictions for each row in the DataFrame df. However, this is not particularly useful on its own as we now need to relate the predictions back to the 2021 COVID data. We can do this by merging with the COVID data and the dates column into a new DataFrame and graphing both the original data and the predicted percentage change. As the scales are so different, we are graphing them one above the other as shown below.

The top graph illustrates the number of new COVID cases, new hospital admissions and total hospital cases as published by the UK government across the first 10 months of 2021.

The lower graph shows the predicted percentage change in retail and recreational mobility. Note that the lower graphs X scale is negative; the lower down the graph the greater the percentage change is:



These graphs show that as the number of new COVID cases decreases the Random Forest regressor predicts that the percentage change in retail and recreational mobility will decrease (notice the bottom graph has a negative scale on the left). However, as new COVID cases rise then the change in mobility negatively increases.

Note that this ignores vaccinations in 2021.

56.11 Online Resources

- https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm *K*-Nearest Neighbour.
- https://en.wikipedia.org/wiki/Root-mean-square_deviation RMSE (Root Mean Square Error).
- https://en.wikipedia.org/wiki/Coefficient_of_determination *R*-squared
- https://en.wikipedia.org/wiki/Decision_tree_learning Decision Tree Algorithms.
- https://en.wikipedia.org/wiki/ID3_algorithm ID3 algorithm.
- https://en.wikipedia.org/wiki/C4.5_algorithm C4.5 Algorithm.
- https://en.wikipedia.org/wiki/Predictive_analytics#Classification_and_regression_trees_(CART) Classification And Regression Tree (CART).
- https://en.wikipedia.org/wiki/Random_forest Random Forest.
- https://pandas.pydata.org/ Pandas.
- https://scikit-learn.org/ SciKitlearn.

Chapter 57
Pip and Conda Virtual Environments

57.1 Introduction

Many Python developers use a virtual environment to handle the modules installed for a particular project. For many developers pip virtual environments are the go-to way of handling such configurations. For the majority of this book this is the approach we have assumed. However, within the Data Science world just as many developers use `conda` and the associated Anaconda environment. This chapter introduces conda, Anaconda and why they are so popular with data scientist.

57.2 Virtual Environments

As a quick refresher we will consider why a virtual environment is an essential element of any Python developers' workspace.

Python provides pip as its default module installation tool. However, by default pip installs modules into the central Python runtime. This is efficient but only suits a single user on a single machine working on single project scenario. This is because it installs modules into the central Python runtime that module is installed for everyone using that installation. This may not seem like much of an issue but consider that a single user has multiple projects active on their computer. Each project has its own requirements for the modules being used. For example, one project is using Pandas 1.5 and another project is using Pandas 2.0. These are different versions of the same library! If we install one version of Pandas into the central Python runtime, everyone will get that version; if we now install a newer version, it will overwrite the previous version! In general, this is a bad idea. Typically, we want each project to have its own set of third-party libraries and the versions of those libraries.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-40336-1_57

To do this, pip provides a facility known as a *pip virtual environment*. A virtual environment is a configuration of Python modules that can be set up on a per project basis.

That is each individual Python project (or program) can have its own virtual environment and that virtual environment can be activated as and when required. The virtual environment is then used to configure all the modules used, isolating one virtual environment's configuration from another. Although it should be noted that there is no reason why several Python programs cannot share the same virtual environment, it is a very common practice to have a virtual environment per program.

57.3 Working with Pip

To use a virtual environment, it is necessary to first create this virtual environment. This is done using the Python command itself (rather than the pip command), for example:

```
python -m venv venv
```

In the above the Python command is being passed the -m option followed by `venv`. This indicates that you want Python to run the `venv` module. The name `venv` stands for virtual environment. This is followed by the name of the virtual environment you wish to create. A common pattern is to call this venv—this is then created in the current directory, for example:

```
% mkdir proj
% cd proj
% python -m venv venv
% ls -la
total 0
drwxr-xr-x  3 jeh staff  96 19 Jun 17:04 .
drwxr-xr-x 18 jeh staff 576 19 Jun 17:04 ..
drwxr-xr-x  6 jeh staff 192 19 Jun 17:04 venv
```

57.3.1 Activating a Pip Environment

We can now activate the virtual environment using the <virtual_env_name>/bin/activate (or `activate.bin` on Windows) command. On a Mac we can run this command using the source Mac command, for example:

```
% source venv/bin/activate
(venv) %
```

Note that on this Mac you can see the virtual environment that has been activated as it is shown before the prompt in round brackets.

On a Windows machine this file will be a bat file, and therefore you can activate it by directly running the activate.bat file, for example:

```
> venv\Scripts\activate.bat
```

57.3.2 Installing Modules Using Pip

We can now install specific modules into this virtual environment and those will be hidden from any other virtual environment. For example, if we first check to see which is installed in our virtual environment using pip list, we can see:

```
% pip list
Package    Version
---------- -------
pip        23.1.2
setuptools 67.6.1
```

From this we can see that there are two modules installed into the virtual environment. We can now install modules into this environment. To do this, we use the pip install command which has the following syntax:

```
pip install module-name
```

The following terminal session illustrates what happens when we do this for Pandas:

```
% pip install pandas
Collecting pandas
  Using cached pandas-2.0.2-cp311-cp311-macosx_10_9_x86_64.whl
(11.6 MB)
Collecting python-dateutil>=2.8.2 (from pandas)
   Using cached python_dateutil-2.8.2-py2.py3-none-any.whl (247
kB)
Collecting pytz>=2020.1 (from pandas)
  Using cached pytz-2023.3-py2.py3-none-any.whl (502 kB)
Collecting tzdata>=2022.1 (from pandas)
  Using cached tzdata-2023.3-py2.py3-none-any.whl (341 kB)
Collecting numpy>=1.21.0 (from pandas)
  Downloading  numpy-1.25.0-cp311-cp311-macosx_10_9_x86_64.whl
(20.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 20.0/20.0 MB 8.5 MB/
s eta 0:00:00
Collecting six>=1.5 (from python-dateutil>=2.8.2->pandas)
  Using cached six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: pytz, tzdata, six, numpy, python-
dateutil, pandas
Successfully installed numpy-1.25.0 pandas-2.0.2 python-dateutil-
2.8.2 pytz-2023.3 six-1.16.0 tzdata-2023.3
```

If we now ask pip to list what has been installed, we can now see these modules listed:

```
% pip list
Package          Version
---------------- -------
numpy            1.25.0
pandas           2.0.2
pip              23.1.2
python-dateutil 2.8.2
pytz             2023.3
setuptools       67.6.1
six              1.16.0
tzdata           2023.3
```

We can now see that pip has installed Pandas (version 2.0.2) but also the modules that Pandas depends upon such as NumPy 1.25.0 and size (1.16.0).

57.3.3 Deactivating a Pip Environment

We can deactivate the virtual environment and activate other virtual environments as required. For example to deactivate it all, we have to do is issue the command deactivate (or deactivate.bat on a Windows machine).

```
[(venv) % deactivate
jeh@Johns-iMac proj %
```

In many cases this is handled for the developer by tools such as PyCharm.

57.3.4 Check Version of Pip

You can check the version of pip using.

```
pip --version
```

57.3.5 Installing Modules into a Pip Environment

New modules can be installed using pip install <module name>

For example:

```
pip install pandas
```

or if a specific version of a module is required then you can specify the version number after the module name using two '==', for example:

```
pip install pandas==1.5.2
```

To update `pip` itself use:

```
pip install --upgrade pip
```

Note that this is the `pip` command being passed the request to install something, in this case the upgrade for pip. On Windows you might want to use a variation on this command as shown below:

```
python -m pip install --upgrade pip
```

57.3.6 Freezing Modules

The `pip freeze` command can be used to create a *requirements* format file. This file contains information on the installed modules that can be used by the `pip` command to configure another virtual environment to include the same modules (and version of those modules). This allows two users to share their virtual environment configurations.

For example:

```
(genesis_venv)% pip freeze
numpy==1.24.3
pandas==2.0.1
python-dateutil==2.8.2
pytz==2023.3
six==1.16.0
tzdata==2023.3
(genesis_venv) %
```

This prints to the output each of the modules installed. If we wish this information to be stored into a file, we can use the redirection arrow, for example:

```
% pip freeze > johnsenv.txt
```

We can now use this in another virtual environment to set that environment up to use the same set of modules:

```
% pip install -r johnsenv.txt
```

57.4 Conda

So given there is pip and pip virtual environments why do we have conda? Conda is an open-source package management system and environment management system for installing, configuring and managing software packages and dependencies. It is commonly used in Python programming but can also support other programming languages.

There are three main reasons why conda is used instead of pip within many Data Science projects; these are:

1. **Conda manages Python** as well as the modules that are installed for a specific project. That is when a conda environment is set up, you can specify which version of Python should be used with it, for example Python 3.10, or 3.9 or 3.12, etc., whereas with pip Python must already be installed, and it is up to the developer themselves to handle the situation where different versions of Python are required on their machine. A side benefit of this is that the same way of installing Python is used within conda on Macs, Windows and Linux boxes, whereas using pip you need to understand how to install Python on each of these platforms and how manage these different implementations.
2. **Conda can manage and install tool binaries**, whereas pip takes Python source code and compiles it on demand for use by developers.
3. **Conda uses channels** which are hosted on Anaconda. In contrast Pip uses the Python package index repository (PyPi) module repository. The default conda channels are maintained by Anaconda itself (the company tat created conda). It is a curated repository which aims for additional stability and integrity between libraries and environment support. In addition there is the conda-forge community channel, which tends to have additional third-party modules on it and may be required for additional packages.

The benefit with both Anaconda and conda-forge is that a lot of work is done/automated to provide standardised builds of packages.

Technically there is a fourth difference between `pip` and `conda`, pip only works with Python, but conda can be used with any language. However, as we are focussed on Python we will ignore that difference here.

Here are some key features and concepts associated with conda:

- **Package management**: Conda provides facilities for installing, updating and removing software (Python) packages. It provides access to a vast repository of pre-built packages, including libraries, tools and applications. Conda handles dependencies between packages, ensuring compatibility and resolving conflicts.
- **Environment management**: Conda can be used to create its version of virtual environments known as *conda environments*. Environments provide a controlled and reproducible environment for managing the version of Python and the libraries you want to work with on one or more projects.
- **Cross-platform support**: Conda is designed to work across different operating systems, including Windows, macOS and Linux.

- **Conda channels**: Conda uses channels to host and distribute packages. Channels are repositories where packages are hosted and made available for installation. The default channel is the Anaconda repository, but additional channels can be added to access packages from different sources or organisations.
- **Conda-forge**: Conda-forge is a community-driven channel that provides a large collection of open-source packages that are not part of the default Anaconda distribution. It offers an extensive range of packages maintained by contributors from various backgrounds.
- **Anaconda**. Conda is often used in combination with Anaconda, which is a distribution that includes conda along with a comprehensive collection of Python packages and tools for scientific computing and Data Science. It is thus widely used within the Data Science community to manage the Python environments for Data Science projects.

So why use pip at all—the short answer is because it comes with the Python runtime itself and is therefore readily available to anyone with a Python installation. In contrast it is necessary to install `conda` separately to Python. A common way in which people do this is to install Anaconda. A second issue with Anaconda is that it has some restrictions which may make it less suitable for commercial use (unless a commercial license is paid), whereas pip has no such restrictions.

57.5 Anaconda

Anaconda is a popular distribution of the Python programming language, primarily used for Data Science and scientific computing. It aims to a default distribution of tools and libraries for data analysis, machine learning and scientific research.

Anaconda supports a range of IDEs such as Jupyter Notebook, Spyder and Visual Studio Code. These IDEs are pre-configured and bundled with Anaconda, providing a seamless development environment for Data Science and scientific computing.

57.5.1 Installing Anaconda

You can download Anaconda from the Anaconda web site using the URL https://www.anaconda.com/download/. Alternatively, if you are on a Mac, you can use home-brew to install Anaconda via brew cask using:

```
brew install --cask anaconda
.
.
==> Installing Cask anaconda
==> Running installer script 'Anaconda3-2023.03-1-MacOSX-
x86_64.sh'
.
.
🍺  anaconda was successfully installed!
```

You will also need to add the Anaconda bin directory to your PATH variable, for example:

```
export PATH="/usr/local/anaconda3/bin:$PATH"
```

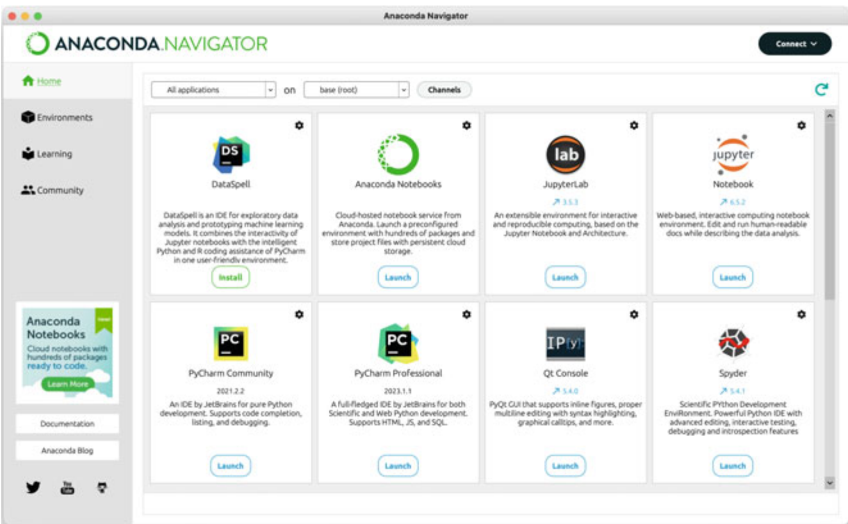
This can be done in your ~/.zshrc file.

You can verify that the Anaconda environment (and conda) has been installed successfully by issuing the conda command at the prompt, for example:

```
conda
```

This should display the conda usage information.

You could also start the Anaconda navigator tool. This can be used to see the modules available, manage your conda environments as well as the IDEs you use:



57.6 Working with Anaconda

When using conda to manage your version of Python and the dependencies that you have in a project you can either use the command line, the Anaconda web interface or facilities embedded in tools such as PyCharm. In this section we will look at the command line tools as this makes what is going on most explicit.

57.6.1 Checking the Conda Version

You can check which version conda you have installed sing he conda info command:

```
% conda info

    active environment : None
      user config file : /Users/jeh/.condarc
populated config files : /Users/jeh/.condarc
         conda version : 23.3.1
   conda-build version : 3.24.0
        python version : 3.10.9.final.0
      virtual packages : __archspec=1=x86_64
                         __osx=10.16=0
                         __unix=0=0
.
.
```

This shows that I have conda version 23.3.1 installed at the moment; it also shows where Anaconda is installed, etc.

57.6.2 Updating Conda

To update conda, you can use conda itself; this can be done using the command:

```
conda update conda
```

Issuing this command will cause conda to update itself and all the packages it uses, for example:

```
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##
  environment location: /usr/local/anaconda3
  added / updated specs:
    - conda
The following packages will be downloaded:

    package                            |    build
```

```
     --------------------------|-----------------
     .
      .
     .

Proceed ([y]/n)? y

Downloading and Extracting Packages

Preparing transaction: done
Executing transaction: done
```

If we now check the conda info, we should see that the conda version has been updated. In this case the conda version is now:

```
conda version : 23.5.0
```

57.6.3 Creating a Conda Environment

The first thing we need to do to use a conda configuration with a project is to create the conda environment.

To create a new conda environment, you use the conda create command along with a name for the environment and optionally the version of Python to install; this is done using the following format.

```
conda create —name <nameofenv> python=<python version>
```

As an example,

```
% conda create --name data-proj-311 python=3.11
Collecting package metadata (current_repodata.json): done
Solving environment: done
## Package Plan ##

  environment location: /usr/local/anaconda3/envs/data-proj-311

  added / updated specs:
    - python=3.11

The following NEW packages will be INSTALLED:

 bzip2            pkgs/main/osx-64::bzip2-1.0.8-h1de35cc_0
 ca-certificates  pkgs/main/osx-64::ca-certificates-2023.05.30-
hecd8cb5_0
  .
  .
Proceed ([y]/n)? Y
.
.
done
#
```

```
# To activate this environment, use
#
#     $ conda activate data-proj-311
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

57.6.4 Listing Available Conda Environments

You can list what conda environments are available using the `conda env list` command, for example:

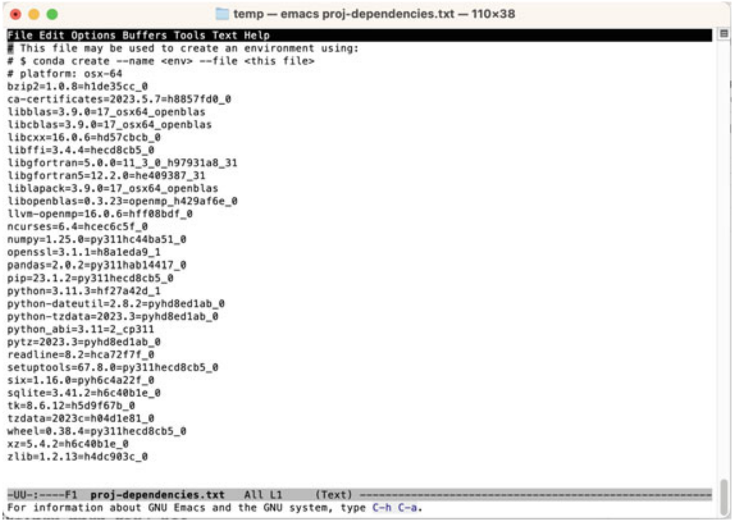
```
% conda env list
# conda environments:
#
base              * /usr/local/anaconda3
bio-proj-310        /usr/local/anaconda3/envs/bio-proj-310
data-proj-311       /usr/local/anaconda3/envs/data-proj-311
test-proj-309       /usr/local/anaconda3/envs/test-proj-309
```

In this example, there are four conda environments available, base, bio-pros-310, data-pro-311 and test-pro-309. Each project has its own version of Python and module dependencies.

You can also use the `--export` option to export the set of dependencies to a text file.

```
conda list --export > proj-dependencies.txt
```

This will create a file in the current directory which describes the modules loaded in the environment their versions and a checksum value to verify that the versions used elsewhere are the same.



Such a file can then be used by others to create a new environment with the same dependencies using the—file option to the conda create command:

```
conda create --name <envname> --file <dependency-file.txt>
```

For example:

```
conda create --name new-proj-env --file proj-dependencies.txt
```

57.6.5 Activating a Conda Environment

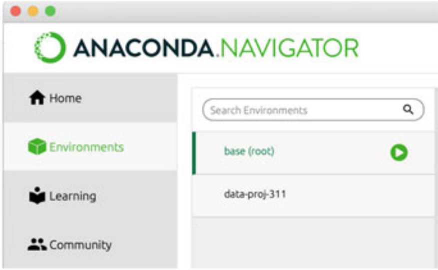
You can activate a conda environment using conda activate <nameofenv>, for example:

```
% conda activate data-proj-311
(data-proj-311) jeh@Johns-iMac ~ %
```

Note that on the Mac machine being used here the current conda environment is now part of the prompt so that you always know which conda environment is the current one.

The name used for the conda environment can be anything, personally I tend to use the name of the project I am working on and the version of Python it has been set up with. This is useful as a quick reference but could become an issue as you can change the version of Python after an environment has been created.

You can also see the environment in the Anaconda navigator as the Environments tab lists current environments:



57.6.6 Deactivating a Conda Environment

The current conda environment can be deactivated using the conda deactivate command, for example:

```
(data-proj-311) jeh@Johns-iMac ~ % conda deactivate
(base) jeh@Johns-iMac ~ %
```

In the above you can see that we are no longer 'in' the data-proj-311 conda environment after deactivating, as the prompt now includes the (base) environment which is the default conda Python environment.

57.6.7 Listing the Modules Loaded into a Conda Environment

To list the available modules (and their versions) in a conda environment use the `conda list` command, for example:

```
% conda list
# packages in environment at /usr/local/anaconda3/envs/data-proj-
311:
#
# Name               Version                Build  Channel
bzip2                 1.0.8              h1de35cc_0
ca-certificates      2023.05.30          hecd8cb5_0
libffi               3.4.4               hecd8cb5_0
ncurses              6.4                 hcec6c5f_0
openssl              3.0.8               hca72f7f_0
pip                  23.1.2         py311hecd8cb5_0
python               3.11.3              hf27a42d_1
readline             8.2                 hca72f7f_0
setuptools           67.8.0         py311hecd8cb5_0
sqlite               3.41.2              h6c40b1e_0
```

```
tk                 8.6.12             h5d9f67b_0
tzdata              2023c             h04d1e81_0
wheel              0.38.4       py311hecd8cb5_0
xz                 5.4.2              h6c40b1e_0
zlib               1.2.13             h4dc903c_0
```

This shows the default set of modules loaded for any new conda environment. Note python is itself listed as a module within a conda environment.

You can also use the Anaconda navigator tool to see what has been installed in a particular environment.

57.6.8 Removing an Anaconda Environment

To remove a conda environment, use the conda env remove —name <envname> command, for example:

```
conda env remove --name data-proj-311
```

Alternatively use the Anaconda navigator tool and select remove for a given environment.

57.6.9 Installing a Module into a Conda Environment

Of course one of the reasons for using conda in the first place is to be able to install and manage additional module. To install a module into a conda environment, use the conda install <module-name> command, for example:

```
(data-proj-311) jeh@Johns-iMac ~ % conda install pandas
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /usr/local/anaconda3/envs/data-proj-311

  added / updated specs:
    - pandas
The following packages will be downloaded:

    package                        |    build
   ---------------------------|-----------------

    .
    .
    numpy-1.24.3                   | py311h728a8a3_1    11 KB
    numpy-base-1.24.3              | py311h53bf9ac_1   6.9 MB
    pandas-1.5.3                   | py311hc5848a5_0  12.5 MB
```

```
    ---------------------------------------------------------------
                                        Total:   202.1 MB
The following NEW packages will be INSTALLED:

  .
  .
  numpy      pkgs/main/osx-64::numpy-1.24.3-py311h728a8a3_1
  numpy-base pkgs/main/osx-64::numpy-base-1.24.3-py311h53bf9ac_1
  pandas     pkgs/main/osx-64::pandas-1.5.3-py311hc5848a5_0

Proceed ([y]/n)? y

Downloading and Extracting Packages
(data-proj-311) jeh@Johns-iMac ~ %
```

which has installed Pandas and associated supporting modules. The only issue with this is that, at the time of writing, conda defaulted to installing Pandas 1.5.3, whereas the current release is 2.0.2. To ensure that the current version of a module is installed, we can use:

```
conda install <library-name>=<library-version>
```

For example:

```
% conda install pandas=2.0.2
```

Unfortunately in this case, at the time of writing, the main Anaconda repository doesn't have the latest version of Pandas available so we have to use the condo-forge channel, as shown below:

```
% conda install -c conda-forge pandas=2.0.2
```

This will install Pandas 2.0.2 into the current conda environment.

57.6.9.1 Removing Anaconda

You can remove Anaconda from your system using the Anaconda-clean program. This can be installed using conda and then executed directly, for example:

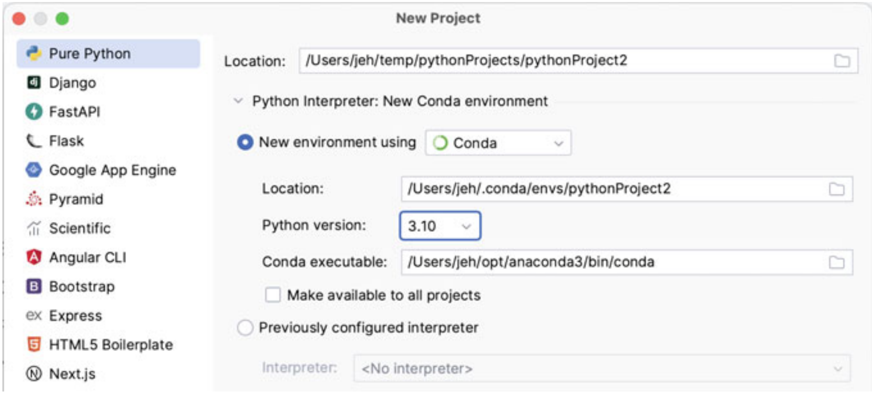
```
conda install anaconda-clean
```

then

```
anaconda-clean --yes
```

57.7 Anaconda in PyCharm

It is also possible to work with conda in PyCharm. For example when creating a new project, you can select to use conda rather than a pip virtual environment when creating a project:



In this case the new environment will be managed by conda and stored in the `.conda` directory rather than in the current project.

57.8 Online Resources

- https://www.anaconda.com/ Anaconda Home page.
- https://www.anaconda.com/download/ Anaconda download page.
- https://medium.com/ayuth/install-anaconda-on-macos-with-homebrew-c94437d63a37 guide to installing Anaconda on a Mac using HomeBrew.