Introduction to Programming with Python

Chris McCool, Michael Halstead & Patrick Zimmer

October 2025

Contents

Chapter 1

Programs and Variables

In this chapter we briefly introduce the core idea behind what a computer program is and then dive into how to store results and intermediate results in variables. In doing so we talk about how to name the variables with identifiers and then introduce some of the basic types of data (data types) in Python.

1.1 What is a Computer Program?

A computer program is a set of ordered operations. These operations are followed in order and stored somewhere on the computer (machine). Often the aim of these programs is to automate tasks that can be well described by the set of ordered operations.

To write a program you do so in a programming language. There are many different types of programming languages these range from high-level languages such as Python through to low-level languages like assembly code. High level languages often have greater similarity to current spoken languages such as English, an example is given by the Python code below.

```
1 for element in my_list:
2     print(element)
```

If we consider the code above it would be natural to think that we take elements (`element`) from the `my_list` and print these elements. By contrast a low-level language such as assembly is more verbose and difficult to follow.

Assembly code is meant to be human interpretable while still being close to the code that the computer (machine) uses; the machine-level code is usually called machine code. An example of adding two numbers in Assembly is given below.

```
MOV AL, NUM1
ADD AL, NUM2
```

The above code requires more thought even for the simple idea of adding two numbers. This is because we first need to specify which register to move the first number to (`NUM1`) before then calling add on the second number (`NUM2`). Representing more complicated ideas like looping over a list and printing the result leads to many more lines in Assembly and becomes increasing more difficult to read. This level of complexity is a motivating factor for using higher level languages, however, it's important to know that at the end all high level languages are converted to lower level code like Assembly or Machine code because those are the instructions that the computer can understand. We will now shift our attention purely to Python and how to write Python code with variables.

1.2 Variables

Variables are a fundamental concept for many programming languages. This is because they store the results of an operation. A simple example of an operation is the addition of two numbers

```
8+2
```

or a more complicated operation such as

```
2*(8+2)+1
```

The issue with the above Python code is that we never store the result of either operation. Variables let us store this result for use later on.

An example of how to use a variable is to take the above code and store the two results in variables. Let's make this concrete by storing the results of the two lines above in variables called `result1` and `result2`.

```
result1 = 8+2
result2 = 2*result1+1
```

In the above code you can see that we have stored the first result as `result1` and then re-used this in the second operation which is then stored in `result2`. This is a demonstration of how there can be interdependence between operations and how we can build up increasingly complicated relationships by

storing results in variables. But, before we can run off and make use of variables we need understand how we can name or *identify* variables. That is, what is the syntax and generally acceptable ways to name our variables.

1.3 Identifiers

Identifiers provide a way to define the name of a variable and even a function. In Python, identifiers must respect the following rules.

1. Consist of a combination of letters either lower case ([a-z]) and/or uppercase ([A-Z]), digits [0-9] or an underscore "_".

2. Does not start with a digit.

The above are the basic rules that **must** be respected. Examples that follow the above rules are given below.

```
my_variable = 1
my_variable2 = 2
_my_variable3 = 3
```

By contrast, examples that don't follow the above rules are given below.

```
3rd_variable = 3
my_*th_variable = 2
```

In addition, to the above rules there are some general principles for the use of identifiers that should also be followed.

The general principle of an identifier is that it should be meaningful. This means that another programmer, someone other than you, could understand the meaning of the variable through its name (identifier). The following are some examples that follow this principle.

```
result1 = 8+2
result2 = 2*result1+1
```

The following is the same example but with identifiers that don't follow this principle.

```
simpson = 8+2
roundabout = 2*simpson+1
```

In the above example the names give no clue as the use or purpose of the variable. Such situations should be avoided.

1.4 Numerical Data in Python

There are several different types of basic data types available. The common ones are numerical types such as integers (whole numbers) and floating point numbers (numbers with decimal places). For convenience we'll call these *ints* and *floats* respectively.

Some of the operators that can be applied to *ints* and *floats* are given in Table 1.1. These operators cover basic numerical operations such as addition, subtraction, and division. This lets us use Python as a calculator which can store intermediate results. Although this can be useful, it does not let us understand how more complicated operations can be put together.

Table 1.1: Table of operators for Integer and Floating Point data types in Python.

Operator	**Name**
-	subtraction
+	addition
*	multiplication
/	division
%	modulo
**	to the power of

To enable us to better understand how to write more complicated code we first introduce two more variables. The first is the idea of a string as this allows us to understand how to print out information to the screen. The second is the idea of a boolean which is critical to the idea of control statements.

1.5 Strings

Strings allow us to do things like print information to the screen and to read in data. They will be used extensively in our examples and it's important to understand how they work.

A string is a collection of *characters* and often these are ASCII characters. These include the characters in the alphabet, both lower case [a-z] and uppercase [A-Z]. Numbers can also be represented such as [0-9] as well as other characters such as "ˊ", "/", ",", and many more. In short, these are the

characters that are generally available on your keyboard and to see a range of ASCII characters have a look at the Python documentation[1].

You can make a new variable as a string using a set of either single quotes '...' or double quotes "...". Below is an example of each of these

```
line1 = "This is the first line of the string."
line2 = 'This is the second line of the string.'
```

Why would you use single quotes instead of double quotes, or vice versa? An example of where you might use one over the other is if a single or double quote is in the string. In such a case, you would use the other formation of the string.

Let's give a concrete example of using a double quote in a string. If we tried put put quotes around the word first in the variable `line1` we would write the following code

```
line1 = "This is the "first" line of the string."
```

This would lead to an error, why? Give this a try and get used to seeing error messages and thinking about what it's telling you. Your error message should look something like the message below.

```
  File "<ipython-input-12-8228df7d7e94>", line 1
    line1 = "This is the "first" line of the string."
                          ^
SyntaxError: invalid syntax
```

What this is saying is that after the " there is a syntax error and so the program cannot be executed further. In this example you can even see that the ^ symbol on `line3` is indicating where the interpreter thinks the error is.

The error above will occur because the string for the variable on `line1` will stop the second double is reached. This would equate to

```
line1 = "This is the "
```

The rest causes an error because it is invalid syntax for a variable. This could be avoided if instead of using double quotes to define the string we used single quotes such as below.

```
line1 = 'This is the "first" line of the string.'
```

We can build strings together by concatenating them.

Strings can also be added together, called concatenation, using the "+" symbol. We demonstrate this by joining the two lines that we had before.

```
line1 = "This is the first line of the string."
line2 = 'This is the second line of the string.'
all_lines = line1 + line2
print(all_lines)
```

[1] https://python-reference.readthedocs.io/en/latest/docs/str/ASCII.html

The print function is being used to display the string to the screen, this is something that we will regularly use. In the above example the output that we get doesn't seem to be quite what we want as we get the two lines together on the same line.

```
This is the first line of the string.This is the second line
    of the string.
```

How do we split this across multiple lines? We can achieve this by using a special character which is accessed by the escape character "\".

Let's break down the idea of the escape character "\". This character says that the next character that comes should be treated differently. Such an operation is needed as some very important characters are not available on our keyboard as input. The new line character is an example of this and the code to get the new line character is to use the escape character and then n like this "\n". There are several other special characters and if you are interested you should explore the relevant part of the Python documentation[2]. With this special character we can now correctly join the two lines using something like the following code.

```
line1 = "This is the first line of the string."
line2 = 'This is the second line of the string.'
all_lines = line1 + '\n' + line2
print(all_lines)
```

The above code provides the following result.

```
This is the first line of the string.
This is the second line of the string.
```

1.6 Booleans

We will now finish our introduction of variables by talking about booleans. A boolean can be either "True" or "False". This concept is critical to many programming languages and is fundamental to conditional statements.

An example of how to get a boolean value is to compare values. For instance, if we do a set of calculations and then check if it's greater than a pre-defined value this will either be "True" or "False". An example of this is below.

```
threshold = 42
result1 = 8+2
result2 = 2*result1+1
comparison_result = result2 > threshold
print(comparison_result)
```

[2] https://python-reference.readthedocs.io/en/latest/docs/str/ASCII.html

The above code provides the following result.

```
False
```

We can manipulate and combine booleans using operators such as those given in Table 1.2.

Table 1.2: Table of Boolean operators in Python.

Operator	Name	Description
`not`	Invert the boolean value	True becomes False and vice versa
`and`	Logical *and* operation	Is only True if both sides of the operation are True
`or`	Logical *or* operation	Is only False if both sides of the operation are False

Logical statements can be developed using booleans and their operators. These are often used for conditional statements such as `if` statements that we will see in the next chapter. A brief example is how to use the `and` operator which only results in "True" result if both sides of the operator are true. Extending the example from before, we can combine two results using the `and` operator.

```
threshold1 = 42
result1 = 8+2
result2 = 2*result1+1
comparison_result1 = result2 > threshold1
comparison_result2 = result2 > result1
result_of_and = comparison_result1 and comparison_result2
print(comparison_result1, comparison_result2, result_of_and)
```

In the above the `result_of_and` is "False" even though `comparison_result2` is "True". This is because `comparison_result1` is "False". We produce the output of the print statement below.

```
False True False
```

1.7 Practical Work

1.7.1 Print Function

The print function is an important feature in the Python language. It allows you to print to your terminal numerous types of data, this is a common tool for debugging and general outputs.

1. Using the print function for "hello world!" Any function in Python can be considered as a function header like `print(...)` and then a set of inputs to the function within the parenthesis. For example, if we wanted to print "hello world!" to the terminal we would use:

```
print("hello world!")
```

or,

```
print('hello world!')
```

Now you try and print various things to the terminal, try strings, integers, and floating point numbers.

1.7.2 Comments

Commenting in Python is another great tool that helps in numerous ways. Let's start with basic commenting, if I want to write something in my code but I **DO NOT** want it to be complied I use the # operator. Such that,

```
# this is a comment
x0 = 'hello world!'
# now I will print it
print(x0) # anything after that # won't be compiled
```

You should have noticed that you can have an operation (print) and a comment on the same line as long as the # comes after your desired operation. Comments are great tools to help you understand things when you go back through your code, or for someone else to go through your code. If you find something particularly challenging you can write a lot of comments as a reminder to yourself about that exercise.

1.7.3 Variables

After you have created your variables or identifiers don't forget to print them to screen using the above techniques.

1. Perform the following operations and store them in unique variables:

```
x0 = 10+3
(10+2)/3
10*4
10-5
```

2. Now using the unique variable substitute them into the following operations, remember to create new variables:

```
y0 = x0*2 # (10+3)*2
((10+2)/3)/2
10*4+6
(10-5)+7
```

3. In the previous operation we used the variables from exercise 1 to create new variables in exercise 2. Now we will use identifiers to do some mathematical operations. Create the identifiers below then use these to perform different mathematical operations such as addition (+), subtraction (−), division (/), and multiplication (∗).

```
var0 = 10
var1 = 2
var3 = 3
var4 = 12

out0 = (var0+var4)*var1
```

4. Now let's perform some of the special mathematical operations in python: to the power of (∗∗), floor division (//), and modulo (%). Note that floor division performs division and ignores (discards) the remainder, this is equivalent to round the result down to the nearest integer (whole number). By contrast, modulo returns the remainder of the division operation.

```
out0 = var0**var1
out1 = var0//var3
out2 = var0%var1
```

5. Strings as variables. Create some basic strings as variables.

```
str0 = 'hello world!'
print(str0)
```

What about if we want to put double quotes (") or single quotes (') in a string? Create at least two new strings, one that includes " and another that includes ' inside the string. HINT: encapsulate them in the opposite type.

6. Adding strings together. Create any number of strings then use the + sign to add them together to create one whole string. Be careful though not to forget the spaces...

```
str0 = 'hello'
str1 = " "
str2 = "world"
str3 = '!'

out0 = str0 + str1 + str2 + str3
print(out0)
```

7. Using special characters: `\n` (end of line) `\t` (tab). Play with both the end of line and tab escape.

```
str0 = 'this'
str1 = 'that'

out0 = `this'+'\t'+`that'
print(out0)

str0 = "this\n"
str1 = "that"

out0 = str0+str1
print(out0)
```

You should notice the special behavior these characters have when you print them to the terminal.

8. Boolean operations: $<$, $>$, $==$, $\leq$, $\geq$, and $!=$. Use each of these operators on some numerical values and print the output. Don't forget to use identifiers.

```
var0 = 5
var1 = 4
var2 = 6

out0 = var1<var0
out1 = var2<=var0
out2 = var0==var0
out3 = var0!=var2
```

9. Booleans with strings: $==$, and $!=$. Create string variables and check if they are equal, try using capital letters and see if Python is case sensitive.

```
str0 = 'dog'
str1 = "dog"
str2 = 'cat'

out0 = str0 == str1
print(out0)
out1 = str0 != str2
print(out1)
```

10. Further boolean operators: and, or, not. From the previous question use these boolean operators to concatenate different mathematical booleans.

```
out0 = var0>=var1 and var0<=var2
```

Chapter 2

Control Statements

Control statements are a fundamental concept in programming. They enable us to build dependencies within our code and one of the easiest examples of this is the `if` control statement. When we explain the ideas behind control statements we will also briefly explain the idea of code blocks which are defined through indentation. Most of this chapter describes the `if` control statement and indentation levels. At the end of this chapter we also briefly introduce the `for` and `while` loops. A deeper discussion about loops, such as the `for` loop, is left until Chapter 4 when we apply these concepts to other types of variables.

2.1 The "if" statement

The `if` control statement is an example of a conditional statement. This is because the logical flow of the code that is executed depends on whether or not conditional statement is either *True* or *False*; a boolean. In its simplest form, a particular section of code is only executed if a particular condition is met. Below is an example where a set of calculations is conducted and provide the result of this calculation exceeds a threshold then another section of code is executed (line 5).

```
1 threshold1 = 42
2 result1 = 8+2
3 result2 = 2*result1+1
4 if result2 > threshold1:
5   print("The final result has exceeded the threshold")
```

In the above code, the `print` statement is only executed when `result2` is greater than `threshold1`; the output of `result2 > threshold1` is a boolean result. In this particular version of the code `result2` is less than `threshold1` and so the code block (with the `print`) of the `if` statement is never executed.

At this point, you might be wondering, what is meant by the code block. There are two definitions for a code block:

1. the group of code which is at the same indentation level, and
2. are uninterrupted lines of code (without blank lines).

In these notes we concentrate on the first one which is the group of code at the same indentation level. An example of this is the `if` conditional statement and the code associated to this is indented. But, why do we need this indentation level?

2.1.1 Indentation (matters)

There are two important aspects that need to be discussed before going further, these are the use of the ":" and indentation. The symbol ":" denotes that the code assocaited to a statement will follow. This is used in conjunction with control statements (`if`, `elif`, `else`, `for`, and `while`), as well as when we define a function (using `def` see Section 5.2) or when we define a class (using `class` see Section 8.1). The code block that is then associated with that then has to be indented; the amount of indentation has to be the same through your program.

All of the code associated with a particular statement (control statement, function or class) will be indented to the same level (by the same amount). This is why we refer to it as a code block and is the main definition of a code block that we consider in these notes. In other words, indentation defines if a set of code is associated with a certain block of code, and consequently if it belongs to a particular control statement. This provides a clear and easy way to see how code relates to statements (control statements, functions or classes).

Let's give a concrete example.

```
1 if result2 > threshold1:
2   print("The final result has exceeded the threshold")
3   result3 = result2*2+1
```

In the example above the lines 2-3 are a block of code associated with `if` control statement. This means that `result3` is calculated only if the `if` statement is `True`. By contrast, in the example below only line 2 is associated with `if` control statement and so `result3` is always executed.

```
1 if result2 > threshold1:
2   print("The final result has exceeded the threshold")
3 result3 = result2*2+1
```

Therefore, it is very important to pay attention to the indentation level of your code as it can lead to very different behaviour. We will now go back to our discussion regarding the `if` conditional statement.

2.1.2 Levels of Indentation

There can be multiple levels of indentation. This can happen if we have an `if` statement within another `if` statement.

```
1 if result2 > threshold1:
2   print("The final result has exceeded the threshold")
3   result3 = result2*2+1
4
5   if result3 > 2*threshold1:
6     print("The value of result3 is more than 2*threshold1")
7     result4 = result3*result3
```

In the above code, the indentation of the second `if` statement, on line 5, means that it will only ever be executed should the first `if` statement be true, on line 1. Furthermore, lines 6 and 7 will only be executed if both the first and second `if` statements are *True*. The lines 5-7 are sometimes this is referred to as a nested `if` statement as it only accessible from within the first `if` statement (line1).

2.1.3 More on `if`: using `elif` and `else`

The previous example showed just the `if` conditional and we now expand to include more complicated examples with the `elif` and `else` statements. The combination of `if` and `elif` statement provides the possibility that another condition can be met. The conditions are evaluated sequentially and only the first one that is `True` is entered. An example of the `elif` statement is given below.

```
threshold1 = 42
result1 = 8+2
result2 = 2*result1+1
if result2 > threshold1:
  print("The final result has exceeded the threshold")
elif result2 > threshold1/2:
  print("The final result is more than halfway to the
    threshold")
```

The `else` statement allows us to catch and actively deal with the possibility that none of the previous conditionals were `True`. This allows us to have "default" code that is executed if all statements were `False`. In such a case, the code contained in the `else` conditional statement is executed. An example of the `elif` statement is given below.

```
1 threshold1 = 42
2 result1 = 8+2
3 result2 = 2*result1+1
4 if result2 > threshold1:
5   print("The final result has exceeded the threshold")
6 elif result2 > threshold1/2:
7   print("The final result is more than halfway to the
     threshold")
8 else:
9   print("The final result is not more than halfway to the
     threshold")
```

2.2 The "for" loop

The `for` loop is another example of a control statement and is used frequently in Python. The `for` loop performs the same set of operations, defined in the code block, for a pre-defined number of iterations. In its simplest form this loop takes a count, an integer, and counts up until all the possible numbers are finished. This idea can be seen in the code snippet below, where the `range(n)` function provides us with the numbers from 0 to $n-1$. This is because indexing in Python always starts at 0, we'll cover this in more detail in later sections.

```
1 for x in range(10):
2   curr_str = "In iteration" + str(x+1) + "of the for loop"
3   print(curr_str)
```

Note that in the above code we have used the term $x+1$ to ensure that the numbers we print start at 1.

The above code is another example of having code blocks through indentation. In the above example the code block, lines 2-3, all have the same indentation level and will be executed for each and every step of the `for` loop.

2.3 The "while" loop

The `while` is a conditional control statement that leads to a loop, potentially executing the same set of code multiple times. It is conditional, because it has a `boolean` test at the beginning (either `True` or `False`). When the conditional is `True` then code block within the `while` loop is executed once and the conditional is then evaluated again. This leads to the looping structure.

If we rethought the example of the `for` loop it could also be written as a `while` loop as shown below.

```
n = 10
x = 0
while x < n:
  curr_str = "In iteration " + str(x+1) + " of the for loop"
  print(curr_str)
  x = x+1
```

Compared to the `for` loop, the above code is more verbose and it takes longer to understand what the loop is trying to achieve. The above issues leads to the `for` loop being preferred over the `while` loop.

A further issue with the `while` loop is the potential to enter a state where the loop is never exited. This is sometimes called an infinite loop. Such a state can be achieved if the conditional will always be `True`. Below we give a trivial example of this.

```
n = 0
x = 1
while x > n:
  curr_str = "In iteration " + str(x+1) + " of the for loop"
  print(curr_str)
  x = x+1
```

For the above reason `while` loops are used sparingly in Python, and other languages.

2.4 Practical Work

2.4.1 Recap: Boolean Operators

Let's recap on what boolean operators are in python: $<$, $>$, $==$, $\leq$, $\geq$, and $!=$. Create different operators and make sure you know what each one does.

```
var0 = 5
var1 = 4
var2 = 6

out0 = var1<var0
out1 = var2<=var0
out2 = var0==var0
out3 = var0!=var2
```

So *out*0 through *out*3 will supply a boolean value, either $True$ or $False$ depending on the conditions.

2.4.2 If Statement

If statements are a control statement that "enters" the condition based on whether the statement is $True$. We create the "If" statement and end it with a :.

```
if <condition>:
  pass
```

The condition is what needs to be true, and the line always ends with the : operator. You will notice that the line after the if statement is indented (spaced or tabbed) in by a certain amount. This indentation is an important part of python, it says that everything under this statement is contained by the statement. Such that:

```
if <condition>:
  # everything spaced in like this is contained within the if
    statement
  pass
# now this is back at the root space so is not contained
   within the if statement
<another operation>
```

1. Let's create a basic if statement using our boolean control statements. Let $x = 15$, $y = 10$, $z = 5$, and $a = 10.1$. Now use these different values with boolean conditions to activate the if statement.

```
if x >= y:
  # contained within the if statement
  print('inside the if', x, '>=', y)
# outside of the if
print('outside the if')
```

2.4.3 More on Indentation

In the above example, we will always print "outside the if" regardless of what happens with the if statement. This is purely because of the indentation, indentation matters. But what about if we want to have embedded if statements?

```
if <condition1>:
  # indent 1
  if <condition2>:
    # indent 2
    pass
  # indent 1
# root indent
```

In this case the second if statement will only be reached if the first if statement is "True". Let's try this:

```
if x >= y:
  print(x, '>=', y)
  if z < a:
    print(x, '>=', y, 'and', z, '<', a)
```

1. Play around with this, try different values, try different booleans too.

2.4.4 If, Elif, Else

As an extension to the if statement we can include two extra statements, these two need to preceded an if statement. The first, $elif$, translates to "else if", and can be thought of as if all the above statements are false try this one. You can have multiple $elif$ statements after an if statement.

```
if <condition>:
  # first indent of if statement
  pass
# back to the root
elif <condition>:
  # first indent of the elif statement
  pass
elif <condition>:
```

```
  # first indent of the second elif statement
  pass
```

1. You should notice that this is different to having embedded *if* statements or multiple *if* statements one after the other. Create both multiple *if* statements and then create an *if* with *elif* statements and see what the difference is.

```
if <condition>:
  # first if indentation
if <condition>:
  # second if indentation
if <condition>:
  # third if indentation
```

Now we will look at the *else* statement. As with *elif* it needs to occur after an *if* statement, however, unlike *elif* it can only be used once. So it can only be used at the end of the string of statements.

```
if <condition>:
  # first if indent
elif <condition>:
  # first elif indent
else:
  # else indent
```

From the above pseudo code you should notice that it has no condition. This is because it will only be entered if all the proceeding conditions are false, and it will always enter this statement if it reaches it.

2. Play around with this using various conditions and print out some values to know when you are in each of the statements.

2.4.5 For Loop

As another example of a control statement, *for* loops are able to iterate over some input. They follow similar standards to the *if* where they are required to end with a : and everything that is to be contained within them needs to be indented.

```
for <value> in <iterator>:
  # first for loop indentation
  print(x)
```

Range

The range function creates an iterable output,

```
x = range(<lower>, <upper>)
```

where x starts at the value $< lower >$ and iterates integers up to the $< upper > -1$. Such that, $range(0, 10)$ will give you the values: $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$, so 10 values.

```
x = range(10) # returns [0,1,2,3,4,5,6,7,8,9]
```

1. Play around with the range function until you understand what it is outputting.

Iterating Over Range in a For Loop

The key component of the $< iterator >$ in a for loop is that it can be iterated over. Range provides an output that we can iterate over. Let's try this now.

```
for i in range(10):
  # for loop indentation
  print(i)
```

1. Implement this for yourself, play around with different ranges.
Similar to the if statement we can embed for loops.

```
for i in range(10):
  # first for loop indentation
  print('first', i)
  for j in range(i,i+5):
    print('second', j)
```

2. Try embedding a for loop inside another for loop.

2.4.6 While Loop

The *while* loop is similar to the for loop except for the fact it is conditional instead of iterable. The *while* loop will continue operating until its condition is met! Be careful with this type of loop as it is very easy to enter an infinite loop.

```
while <condition>:
  # while loop iteration
  # change to the conditional component
```

1. Let's implement our *while* loop where x starts at 0 and increases by a single integer until it reaches 10. What happens if we forget to increase x?

Chapter 3

More Variables

There are two frequently used data types in Python the *list* and the *dictionary*. A *list* is an ordered set of data of any type which is indexed by using it's entry number (an integer) in the list. A *dictionary* is set of data of any type which is indexed by any immutable data type. This chapter will not cover the details of immutable data types, but rather show the general use cases for lists and dictionaries. the ideas behind mutable and immutable data types are covered in Section 7.1.

3.1 Lists

A *list* is an ordered set of data of any type which is indexed by using it's entry number (an integer) in the list. The index for a list starts at 0. In fact, indexing in Python always starts at 0 and this is the same for several other programming languages such as C, C++, and Java.

In Python, a list is defined by the use of square brackets "[]". We can make an empty list or pre-fill a list with values as shown in the code snippets below.

```
1 empty_list = []
2 prefilled_list = [1,2,3,4]
3 print(prefilled_list[0])
4 print(prefilled_list[3])
```

In the above example we have printed the first and last value in the variable *prefilled_list*. Note that the entries in a list are separate by a comma as you can see with the pre-filled list on line 2.

We can also add to the list by appending to it or even concatenating two lists together. Appending lets us add a single new value to the end of list. If we want to combine two lists then we can concatenate them together. Python

provides a concise way to do this by using the "+" symbol, other methods also exist and you are encouraged to explore the python documentation to find out more about this.

```
list1 = [1, 2, 3, 4]
list2 = [6, 7, 8]

list1.append(5)
print(list1)

list1 = list1 + list2
print(list1)
```

There are two more important concepts to help in understanding lists, knowing how many items are in it and knowing that anything can be an item in your list.

A list can hold any data type. This means it can hold integers, strings, booleans or any other valid data type in Python. This includes another list. Furthermore, it can hold any combination of data types. This means the list doesn't have to hold all integer values, but rather can be any combination of valid data types in Python.

Finally, if you need to know how long your list is you can use the command `len`, such as `len(list1)`. This will provide the number of elements you have in a list. It is often combined with the use of the command `range` which we saw being used with `for` loops in Section 2.2.

3.2 Dictionaries

A *dictionary* holds a set of any type of data but unlike a *list* it is indexed not by its entry number but by a *keyword*. This *keyword* can be any immutable data type, which essentially means something that can't change (see Section 7.1 for details on mutable and immutable types). Common examples of immutable data types are strings and integers. Strings are frequently used as a meaningful (human understandable) keyword for use in dictionaries.

A *dictionary* is defined by the use of curly brackets "{}". We can make an empty dictionary and add to it simply by defining the *keyword* and its associated entry. When adding an entry we reference the dictionary using square brackets "[]" and then give the *keyword*.

```
my_first_dict = {}
my_first_dict['entry1'] = 1
my_first_dict['entry2'] = 2
my_first_dict['entry_list'] = [1,2,3]
my_first_dict['entry2'] = 3
print(my_first_dict)
```

As you can see above it is important to keep an eye on the keywords in the dictionary as you can easily overwrite what was previously there. In the above code snippet, we defined the keyword `'entry2'` twice and this means that it now only has the last value that we assigned which is "3".

The entry in a dictionary can be any valid data type in Python and is commonly referred to as a value. For this reason, you will frequently see in Python a reference to a key, value pair. This concept is often associated to dictionaries and it refers to the keyword (key) that is the index into a dictionary and the entry in the dictionary (value) associated with it. From now on when we discuss dictionaries we will refer to keys (not keywords) and values (not entries).

Finally, we finish this brief introduction with how to pre-fill a dictionary and to find out all of the keys in a dictionary. Just like a list, we can pre-fill a dictionary and this is achieved by giving the key, value pairs. First the key is given followed by a ":" and then the value, as shown below; all of this is surrounded by curly brackets. Each key value pair is separated by a comma, similar to how this was done for a *list*.

```
my_first_dict = {'entry1': 1, 'entry2': 2, 'entry_list':
    [1,2,3]}
print(my_first_dict)
```

If we have a dictionary we can find out all of the keys it has by using the function `keys`. This gives us all the keys in the dictionary. We can see this idea below.

```
my_first_dict = {'entry1': 1, 'entry2': 2, 'entry_list':
    [1,2,3]}
print(my_first_dict)
all_keys = my_first_dict.keys()
print(all_keys)
list_keys = list(all_keys)
print(list_keys)
print(list_keys[0])
```

Although the above code might make you think that the variable `all_keys` is a list it is not. But, we can turn it into a list by doing so explicitly in Python. This is what the last 2 lines of the code example above are doing. We will explore lists and dictionaries in a bit more detail in the next chapter and describe how we can go through each element or key, value pair by using `for` loops.

3.3 Practical Work

3.3.1 Lists

A list is an ordered set of information, you can instantiate an empty list with either:

```
lst0 = list()
lst1 = []
```

1. Print these out and see if there is any difference.

2. Create some lists, you can put anything in a list.

```
lst0 = [12, 2.2, 'dog']
```

3. As a list is ordered we can index into the list using an integer, if I wanted to print 'dog' from the above example I would index using 2. Remember that indexing starts at 0, so

```
print(lst0[0])

>>> 12
```

Index into your created lists using the integer values.

4. We can also index into our lists using negative numbers. These negative indexs are taken from the last element of the list.

```
print(lst0[-1])

>>> 'dog'
```

Index into your created lists using negative numbers.

5. We can also change the element in the list using either positive or negative indexs. If we want to change the first element of our list:

```
lst0[0] = 24
print(lst0[0])

>>> 24
```

Change some of the elements in your list.

6. What about if we want to add two lists together to create a new list that contains both sets?

```
lst0 = [12, 2.2, 'dog']
lst1 = [2023, 'cat']
lst2 = lst0 + lst1
print(lst2)

>>> [12, 2.2, 'dog', 2023, 'cat']
```

Remember that the order is preserved by the new list so the values in $lst1$ will come after $lst0$ in the new list.

7. What about if we want to add a new value to an existing list? For this we will use the *insert* function. In the *insert* function we need to define two values, the index where we want to insert to, and the value we are placing there.

```
lst2.insert(0, False) # we can insert anything so I am
    inserting a boolean here
print(lst2)

>>> [False, 12, 2.2, 'dog', 2023, 'cat']
```

Insert some new values in your list.

8. What happens if you try and insert a value outside of the existing indices?

```
lst2.insert(14, 'high')
```

9. Now what if we know that we want to insert values at the end of a list? We can use the *append* function.

```
lst0 = [12, 2.2, 'dog']
lst0.append('cat')
print(lst0)

>>> [12, 2.2, 'dog', 'cat']
```

Append some values to your current lists.

10. How do we remove values from our lists? The function for this is *pop*. This takes either one or no input values, if you provide a value to pop this is the index that it will pop out. What happens if you don't supply an index?

```
lst0.pop(1)
print(lst0)

>>> [12, 'dog', 'cat']
```

3.3.2 Dictionaries

Dictionaries are similar to lists, however, instead of them being ordered by an index they are ordered by a key. When you use the key as an input to your dictionary you can return a value.

1. First, let's create an empty dictionary.

```
dit0 = {}
dit1 = dict()
```

2. To create a key:value pair you perform it in the following manner:

```
dit0 = {'one':1, 2:'two', 'true':True, 'lst':[12, 2.2, 'dog'
    ]}
```

So, the first key:value pair is 'one' and 1. The second element uses an integer as its key and a string as its value, next we have a string and a boolean, and finally, we have a string and a list. Create your own dictionaries and play around with what is and isn't possible.

3. What about if we want to access one of the values? To do this we use the appropriate key as an index into the dictionary.

```
print(dit0['lst'])

>>> [12, 2.2, 'dog']
```

Try this in your dictionaries. What happens if you try to access a key that doesn't exist?

4. Next, we will insert a new key:value pair to our dictionary. This is as simple as encapsulating the key inside the square brackets.

```
dit0['new'] = 'this is a new value'
print(dit0)

>>> {'one':1, 2:'two', 'true':True, 'lst':[12, 2.2, 'dog'], '
    new':'this is a new value'}
```

Try this in your own dictionaries.

5. Now let's change one of the values in the dictionary:

```
dit0['new'] = 2023
print(dit0)

>>> {'one':1, 2:'two', 'true':True, 'lst':[12, 2.2, 'dog'], '
    new':2023}
```

Change the values in your dictionaries.

6. Finally, how do we remove a key:value pair from our dictionary? Once again the pop function is useful here, however instead of an index we use the key.

```
dit0.pop('new')
print(dit0)

>>> {'one':1, 2:'two', 'true':True, 'lst':[12, 2.2, 'dog']}
```

Try this for yourself.

Chapter 4

Loops on Lists and Dicts

In this chapter we describe how to operate over all the elements of a list or dictionary. From here on we generally refer to a list as a `list` and a dictionary as a `dict`. Previously we showed we could make a `list` and `dict` to store a set of data. We also showed we could extract information from a particular element in the `list` or `dict`. However, there will be times where you want to operate over all the elements in a `list` or `dict`. Python provides an interface to quickly and easily achieve this. First, we present the "brute force" approach to doing this and then present the more convenient interface that Python provides to iterate, or go through, every element of a `list` or `dict`; in the case of a `dict` this is usually the key, value pair.

4.1 Looping Over Lists

The first idea for how to loop over a `list` or `dict` would be to exploit the existing ways to query them. We show this first for a `list` and then for a `dict`.

For a `list` we index based on the entry number, which start at 0. We can find out the length of the `list` using `len(.)`. Using this we could formulate a `for` loop to go through every element and print it. Let's do a practical example of this.

```
my_list = [0,2,4,6,8,10]
for i in range( len(my_list) ):
  print("We have value", my_list[i])
```

Which would output the following to the screen.

```
We have value 0
We have value 2
We have value 4
We have value 6
We have value 8
We have value 10
```

The above code has done exactly what we want and puts together our understanding of both a `list` and `for` loop. Furthermore, we've made use of the function `len(.)` as well as `range(.)`. Although this code works as expected and achieves the desired outcome, in Python there is an even simpler way to iterate through a `list` and we show this below.

4.1.1 Iterating Over a List

We can iterate over a `list` by using an *iterator*. An iterator provides a way to get the *next* item from our data structure. When applied to a `list` this iterator returns the next *element* of the list. Because a `list` is an in-built data type in Python this iterator is very easy to access. In fact all we have to do is formulate the `for` loop correctly. In Python the syntax ends up sounding like an English sentence where we say "**for** each **element in** my **list**". We write this in concrete Python code below.

```
my_list = [0,2,4,6,8,10]
for element in my_list:
  print("We have value", element)
```

The above code exactly the same output as before. This is because an *iterator* provides us with the next element in the data structure and the for loop operates over all of these until it reaches the end. Furthermore, it is ordered because a list is ordered. The story for a `dict` is similar but slightly different.

4.2 Looping Over Dictionaries

Similar to lists, the first idea for how to loop over a `dict` would be to exploit the existing ways to query the `dict`. For a `dict` each index is given by a

unique key. To find out which keys exist in a `dict` we can use `keys()`, we could then iterate over the list of keys of the `dict` using a `for` loop. Let's do a practical example of this.

```
my_dict = {"entry1": 12, "entry2": 13, "entry4": 24}
keys_in_dict = list(my_dict.keys())
for k in keys_in_dict:
  print("For key", k, "we have entry", my_dict[k])
```

Which would output the following to the screen.

```
For key entry1 we have entry 12
For key entry2 we have entry 13
For key entry4 we have entry 24
```

The above code has done exactly what we want and puts together our understanding of iterating over `lists` to then go through all of the keys in a `dict`. Furthermore, we've made use of `keys()` to get all unique keys in the `dict` and converted them to a `list`. As was the case with the example of a `list`, there is actually a much simpler way of achieving this in Python.

4.2.1 Iterating Over a Dictionary

Similar to `lists`, we can also iterate over a `dict` by using an *iterator*. For dictionaries you use the function `items()` which gives you an iterator over the *(key,value)* pairs in a dictionary. This allows us to simplify our previous code by decoding the *(key,value)* pair in the `for` loop statement itself. We write this in concrete Python code below.

```
my_dict = {"entry1": 12, "entry2": 13, "entry4": 24}
for k,v in my_dict.items():
  print("For key", k, "we have entry", v)
```

Which provides exactly the same output as before. This is because an *iterator* provides us with the next element in the data structure and the for loop operates over all of these until it reaches the end. In this case, for a dictionary it directly provides the key and the value pair, we simply need to decode this (e.g. `k,v`) in the for loop.

4.3 Practical Work

4.3.1 Looping Over Lists

In this exercise, we will use the iterator to loop over a list using a for loop. To do this we will need the *range* function covered previously and the *len* function contained within the list class. The *len* function returns the number of elements in the list.

```
lst0 = [10,20,30,40,50]
print(len(lst0))

>>> 5
```

Complete the following exercises using the `for i in range(len(<your list>))` and loop over the list using i as the index to the list.

1. Create a list with 5 integers.

2. Loop through the list and print the value to the screen.

3. Create a new list with an integer, a string, a float, and a boolean.

4. Loop through the list and display the value to the screen.

4.3.2 Directly Iterate Over a List

You also have the ability to iterate directly over a list instead of using an index to obtain the values. If we use `for i in <your list>`, i will be each value within `<your list>` in order.

1. Create a list with N values of any type.

2. Iterate directly over the list and display values to the screen.

4.3.3 Looping Over Dictionaries

1. Create a dictionary with at least 4 key:value pairs.

2. Obtain the keys from the dictionary (`dit0.keys()`).

3. Loop over the dictionary using the keys and display both the key and the value.

4.3.4 Looping Over the Values in a Dictionary

In the previous exercise, we used the keys to iterate over the values. We can also iterate over the dictionary and obtain the values directly: `for i in dit0.values()`. Unfortunately, we don't know what the key is that is associated with that value but sometimes we don't necessarily need it.

1. Create a dictionary with at least 4 key:value pairs

2. Loop through the values and display them to the screen.

4.3.5 Iterating Over Key:Value Pairs in a Dictionary

Dictionaries have a special ability to iterate over the key:value pairs directly: `for k, v in dit0.items()`. In this case k is the keys, and v is the associated value with that key. The order of k and v must be preserved, i.e. you can't have the v before the k.

1. Create a dictionary with at least 4 key:value pairs.

2. Iterate over the dictionary using the `.items()` function and display the key value pair to the screen.

Chapter 5

Functions and Comments

Being able to re-use code is a key concept for programming. As you write more and more code you will notice that there will be similarities with what you previously wrote. You may even simply copy and paste code from one place to another and be happy that this has saved you time and effort. At this point, you should be thinking about how you can group those bits and pieces of code together so that you can quickly use them again, but without copying and pasting them. A general way of grouping code together is to make a function and then for keeping track of several functions to put them into modules; we cover modules in Chapter 6.

In addition to keep code grouped together you will also want to comment your code. Commenting code is important because it leaves statements about what you think a group of code, function or code block, should be achieving. This makes it easier for someone else to look at your code and understand its intent. It is particularly important if you have a bug in your code that needs to be fixed. Finally, it helps you as well because you have left your own pointers for when you revisit code that you wrote a month ago.

5.1 Comments

Comments are critically important for almost all programming languages. The code provides the implementation of your idea. The comments provide you a chance to state the intent of the code. This is particularly important for quickly imparting why your code is written in a particular way and helps greatly if your code has any sort of bug. You should **always** comment your code no matter what programming language you use. In Python there are two ways to comment your code using the hash symbol (`#`) or triple quotation marks (`"""`).

The hash symbol (#) is used to comment a single line or part of a line. Everything after the # on a line is then ignored by the interpreter. This means it's not considered to be the code and instead is a comment. Below we give three examples of a single line comment.

```
1 my_list = [1,2,3,4] # This is my list
2 # Go through and print everything in the list
3 for e in my_list:
4     print(e)
5 print("Program finished.")
6 #my_list.append(5)
```

On line 2 we have commented the whole line and everything is ignored by the interpreter. One line 1, everything after the # is considered to be commented and is ignored by the interpreter. However, everything before the # is considered to be Python code and will be interpreted and executed as per normal. Finally, on line 6 the whole line has been commented and ignored even though it could be valid Python code.

The triple quotes (""") are used to incdicate that an entire line or multiple lines are commented. The three double quotes in a row opens the comment and the end of the comment is indicated by three more double quotes in a row. This can be done on a single line or over multiple lines. This is useful for when a lot of information is needed to describe what is happening the code that you are commenting. A great example of this is that comments for a function are often given using this approach at the very beginning of the function. In this way anyone reading the code can first see what the intended outcome of the function should be before delving into the code itself. Everything contained within the three double quotes (start and end) are considered to be part of the comment and are not executed or interpreted as being Python code.

```
1  """ This is my list """
2  my_list = [1,2,3,4]   """ This is my list """
3
4  """
5  Below we go through and print everything that
6  we find in the list
7  my_list.append(5)
8  """
9  for e in my_list:
10     print(e)
11 print("Program finished.")
```

Between lines 4 and 8 we have a multi-line comment and everything is treated as being a comment and not Python code. Even line 7, which is valid Python code is not executed and treated as a comment. On line 1 we have used the

triple quotes to provide a single line comment. However, on line 2 we have done the same thing but this is **not** valid. This is because when using the triple quotes Python expects the whole line to be commented and that has not been the case. Line 2 actually leads to an error for the interpreter. But, if we changed line 2 to make use of the # comment it would be valid and this is the case for the code below.

```
1 """ This is my list """
2 my_list = [1,2,3,4]  # """ This is my list """
3
4 """
5 Below we go through and print everything that
6 we find in the list
7 my_list.append(5)
8 """
9 for e in my_list:
10     print(e)
11 print("Program finished.")
```

5.2 Functions

Functions enable us to group code together so that it can be used in multiple places with minimal effort. An advantage of being able to use the same code in multiple places is that maintaining the code becomes easier. If we have an error in a function we only need to fix it in one place if we copied and pasted the same code we would need to fix everywhere. This is because any errors in the code can be fixed in the single function rather than in the numerous places where the code has been "copied and pasted" to.

In Python, functions are denoted by the `def` command. The format is `def` followed by the input arguments and a simple case is shown below where there are no input arguments and it returns a value, in this case for the value *pi*.

```
1 def my_pi_function():
2     return 3.14
```

The name of the function follows the rules of any identifier and we discussed this earlier for variables in Section 1.3. Increasing complicated functions can be derived by passing more input arguments and even returning more results.

You can add comments to a function by using the triple quotation marks. Generally you should do this immediately after the `def` statement. This is because Python has ways to automatically extract documentation and if you provide the triple quotation mark comments at the start of your function it will be extracted automatically. The triple quotation marks should be

indented so that it is clear the comment is for the code block associated with the function. An example of providing such comments in Python for a function is given below.

```
def my_pi_function():
  """
  This function returns the value of pi to two decimal places
   .
  """
  return 3.14
```

The above function also returns a value to us, in this case the floating point value `3.14` by using `return`.

To provide input arguments to a function the input arguments need to have an identifier. This is so that they can referred to within the code of the function. We provide an example of this below.

```
import math

def my_func2(value_to_square, value_to_sqrt):
    ret_val1 = math.pow(value_to_square, 2)
    ret_val2 = math.sqrt(value_to_sqrt)
    return ret_val1, ret_val2
```

The above code allows us to pass two input arguments `value_to_square` and `value_to_sqrt`. Some processing is then performed and then it returns two values `ret_val1` and `ret_val2`. When we return more than one variable, they are separated by a comma. What happens in Python is that the information being returned from the function is actually being put together into a single variable whose data type is a `tuple`. A key idea behind a `tuple` is that it can hold several elements, but unlike a `list` it is immutable. We will explain `tuples` in more detail in Section 7.2.

You have now seen examples of functions with multiple input arguments and multiple return values. An example of how to capture these return values is also given below in the code snippet which calls the two previously defined functions.

```
pi_variable = my_pi_function()

square_pi, sqrt_pi = my_func2(pi_variable, pi_variable)

print(pi_variable, square_pi, sqrt_pi)
```

There are few things to note here. When calling a function we use the curved brackets "()" even if we pass no arguments. Also, when capturing multiple return values we can treat each value separately by defining new identifiers and separating them with a comma. We can also put them a single variable and index into this as shown below.

```
ret_val = my_func2(pi_variable, pi_variable)

print(pi_variable, ret_val[0], ret_val[1])
```

If you want to use a function in your code you **must** define the function before you use it. This makes sense because otherwise when your code attempts to be executed, particularly sequentially, there is no chance to know what the function being called is unless you have already defined it. We make this clear by putting the example above together.

```
def my_pi_function():
  """
  This function returns the value of pi to two decimal places
   .
  """
  return 3.14

import math

def my_func2(value_to_square, value_to_sqrt):
    ret_val1 = math.pow(value_to_square, 2)
    ret_val2 = math.sqrt(value_to_sqrt)
    return ret_val1, ret_val2

pi_variable = my_pi_function()
ret_val = my_func2(pi_variable, pi_variable)

print(pi_variable, ret_val[0], ret_val[1])
```

If we move the code from lines 14 & 15, where we make use of `my_func2`, and put it on line 8 there will be an error because the function has yet to be defined.

5.2.1 Using `help`

When trying to understand what a function does or what arguments it accepts you can, in interactive environments like the Python interpreter or Jupyter notebook, use the `help` command. The `help` command can be executed as follows `help(<elem>)` where `<elem>` is what you wish to query to find more information about,

```
help(my_pi_function)
```

If the developer has provided comments in the standard manner for a function, using triple quotes at the start, then `help` will output this documentation as well as the names given to any variables expected. In this way you, and other developers, provide a way to quickly understand the requirements

of the code through the expected input values and describe the expected output of the function or code.

5.3 Practical Work

5.3.1 Comments Recap

At the start of your code it is always good to put a brief description of what is contained within the file. For this weeks practical you could do something as follows.

```
"""
Practical 5: Comments and Functions

Created MAH 20230106
"""
```

Notice I used three “ marks to indicate that everything contained within these is a comment and is not to be compiled.

The # key is the other way to use commenting. Everything after the # is considered a comment and will not be compiled.

```
# creating lists
lst0 = [1,21,12,'dog'] # my first list
# lst1 = ['this', 'that', 'then']
# printing the lists
print(lst0)
print(lst1)
```

What is wrong with the above snippet of code?

5.3.2 Functions

The basic function begins with the `def` command, then you need to give it a name following the standard variable rules (i.e. must start with a letter). The code snippet below gives a brief overview, the `*inputs` outlines the fact that we can have any number of inputs to the function.

```
def <function_name>(*inputs):
  <operations>
  return *outputs
```

1. Create a function called `my_pi_function` that takes nothing as input and returns the value 3.14159265359.

2. Create a new function `my_pi_round_function` that takes as input a decimal place to round the output to.

For this function, you will want to use the `round(<variable>, <value to round to>)` function inside your function.

3. Create a circle function that takes as input a radius. This function should use your pi function from previously to calculate the diameter, circumference, and area. All three of these values should be returned.

```
def <function_name>(R):
  <operations>
  return d, c, a
```

Keep in mind that the pi function you created earlier needs to be created before it is used.

Chapter 6

Modules

Once we end up with several functions we will want a way to store them in their own area and then we use them with multiple programs we write. If we just group our code with functions we will not be able to use the functions with multiple programs. A way to achieve this is to make our own module. In other programming languages these are considered to be a library. In fact, the idea of a module isn't new by now as we have already seen this idea when we used the `import math` command to make use of special mathematical operations like power (`pow`) and square root (`sqrt`). In this chapter we will discuss more details on how modules can be used and also how to make your own modules.

6.1 Importing From a Module

There are several ways to import information from a module. We have been using a fairly common approach when we have made use of the math module. We have been importing everything to do with the math module by writing `import math`. Then when we have made use of something from this module we have written `math.`. This is a pre-cursor that let's Python know that we are using something from that particular module called `math`. The "." is then referencing inside and then we specifiy what we want to use, for instance a particular function.

We will now make this description more general by saying if we import any module such as `my_module`. When we make use of something from that module we always use the specify this by using the module name followed by a dot (".") such as `my_module.`. An example of this is below.

```
1 import my_module
2 print(my_module.my_function())
3 print(my_module.a_value)
```

The above approach make use of the explicit reference to the module by using the module name. On line 3 we've shown that we can make use of not only functions in the module (line 2) but there can also be variables as well.

There are other methods to achieve the above. For instance, rather than referencing to the module by its name we can make a convenient substitution for the name. In Python we achieve this by using

```
1 import <module> as <label>
```

An example of this could be for numpy by doing `import numpy as np`. After this whenever we want to reference or using something from the `numpy` module we can just write `np.` which leads to fewer characters to write. However, the cost is that because the module name is no longer being used there can be confusion about what the short name stands for. For this reason the use of `import <module> as <label>` should be use sparingly. Also, this approach still gives us access to all of functions, classes and variables of the `numpy` module. However, sometimes we might only want one or two things from a particular module.

To access only a limited set of information from a module we can specific exactly what should be imported from that module. For this we make use of the `from` statement with the pattern being

```
1 from <module> import something, something_else
```

Then whenever we want to use `something` or `something_else` we don't need to specify the module and do not need to reference the library that they came from.

```
1 from <module> import something, something_else
2 print(something())
3 print(something_else())
```

The above approach should also be used sparingly because it is not always explicit what function is being called because different modules can have the same function names.

6.2 Making a Module

The purpose of a module is to bring together a group of code that provides a set of related operations. The `math` module is an example of this. But, how do you make your own module?

To make your own module all you need to do is to define a Python file (.py). You can then add your functions to this Python file and import it in other scripts that you write. For the easy of use, and discussion, we will consider that you do this by making the module file (python file) in the same directory as your script.

Let's make an example module called *my_module.py*. We'll put code into it for example purposes only and so the file looks like the code snippets below which has the functions `square_function` and `add_newline`

```
def square_function(val):
  return val*val


def add_newline(curr_str):
  return curr_str + '\n'

PI=3.14
```

We can now import the module `my_module` into our script or code as per normal.

```
import my_module

my_string = "Hello"
print(my_string)
print(my_module.add_newline(my_string))
print(my_module.PI)
```

Note that because we used the standard `import <module>` pattern we need to state which module we want to use to get access to `add_newline`. We've also shown in this example that you can access things like variables, in this case `PI`, that have also been defined in your module. You now have all the tools needed to make your own modules and import them into your current scripts to use all the code that you have previously generated.

6.3 Practical Work

6.3.1 Importing Libraries

Here we will import numpy and display pi in different manners. This all results in the same value but just uses different importing methods.

1. `import numpy`
Once imported print out `numpy.pi`.

2. `import numpy as np`
Once imported print out `np.pi`.

3. `from numpy import pi`
Print `pi`.

4. Import both the `sqrt` (square root) and `pow` (power) function from numpy and use them. You may need to look at google to work out how to use them, particularly `pow`.

6.3.2 Creating Modules

1. Create a new file in your working directory called `mymodule.py`.

2. Inside `mymodule.py` create a function that can calculate the circumference and area of a rectangle. It should take as inputs the height and width of the rectangle, plus some rounding factor.

3. Import your rectangle function and use it with different values of floating point numbers.

4. Create a function that takes as input the height and width of a rectangle. If the height and width are the same this function should print out that the object is a square, otherwise it should print out it is a rectangle. The final output should be the circumference and area of the rectangle. Use your previously created rectangle function inside this new function!

5. Import this function and check that it works for squares and rectangles.

Chapter 7

More on Functions and Variables

In this chapter we sketch out some more of the details about functions and variables. We first consider functions including default initial values and variable length arguments. The idea behind mutable and immutable types is then described and we make this concrete by showing that some variables passed to functions can be altered within the function because they are mutable.

7.1 Mutable vs Immutable

A mutable type is something that can change whereas an immutable type is something that cannot change. Two examples of mutable data types are *lists* and *dictionaries*, this is because the values inside them can change. A concrete example of this is given below where we change the data contained in the list or dictionary.

```
1 my_first_dict = {'entry1': 1, 'entry2': 2, 'entry_list':
      [1,2,3]}
2 print(my_first_dict)
3 my_first_dict['entry_list'] = "empty"
4 print(my_first_dict)
```

At this point you would ask well, isn't this also true for the following code for the variable *my_var1*.

```
1 my_var1 = 2
2 print(my_var1)
3 my_var1 = 3
4 print(my_var1)
```

The variable *my_var* has the same identifier but the contents appear to have changed. To better understand what is going on we have to dive deeper.

In the above code for *my_var1* what is happening is that from line1 to line3 there is a completely new variable being made. We can see this if we use the *id* function within Python. What *id* returns is the memory address of the variable. In Python, if a new variable is made a new area of memory is given to this variable. So, in our code if we print the id of *my_var1* we see that for the new value of *my_var1* an entirely new variable is made.

```
1 my_var1 = 2
2 print(id(my_var1), my_var1)
3 my_var1 = 3
4 print(id(my_var1), my_var1)
```

By contrast, for a list or dictionary, the same memory address can be used but the contents of this address can change (e.g. it is mutable). You can see this concretely by printing the memory address (id) of the dictionary as shown below.

```
1 my_first_dict = {'entry1': 1, 'entry2': 2, 'entry_list':
    [1,2,3]}
2 print(id(my_first_dict), my_first_dict)
3 my_first_dict['entry_list'] = "empty"
4 print(id(my_first_dict), my_first_dict)
```

For the purposes of an introduction to Python we do not delve any deeper into the idea of mutable and immutable types. Other chapters will delve into this topic more deeply, for instance the implication this can have for functions in Python.

7.2 More on Data Types

There are several different types of basic data types available. The common ones are numerical types such as integers (whole numbers) and floating point numbers. These have varying degrees of accuracy or range of values for their representation from using 8-bits through to 64-bits; a discussion on the implications of these is beyond the scope of this material. In Table 7.1, we've presented the basics of integers and floating point numbers. The value at the end represents the number of bits B used to represent the number, being one from the list of $[8, 16, 32, 64]$. Unsigned integers can be used for only positive whole numbers from 0 to $2^8 - 1$ while signed integers represent negative and positive whole numbers from $-(2^7$ to $2^7 - 1)$. The operators that can be applied to *ints* and *floats* are given in Table 1.1.

Table 7.1: Table of Integer and Floating Point data types in Python.

Signed Integers	Unsigned Integers	Floating Point
int8	uint8	float8
int16	uint16	float16
int32	uint32	float32
int64	uint64	float64

7.3 More on Functions

7.3.1 Default Values

Sometimes we want the arguments in a function to have default values. This can be easily achieved once we know the rules for doing so. When we define the function, using `def`, we need to provide the required arguments and we can then give any number of arguments with default values. This pattern is given in pseudo-code below.

```
def my_function(required_arg_1, ...., required_arg_n,
    arg_with_default_1, ..., arg_with_default_n):
```

The default values can be any valid type such as integers, floats, lists, and dictionaries to name just a few.

There are several times when having a default is useful and many functions make use of this. Let's provide a concrete example for doing a calculation, in particular for estimating the variance of some values in a list. If we have a list of floating point values and we implement our own variance function a possible issue is to divide by very small numbers. To ensure numerical stability, we will often introduce that the denominator has at least a small value. An example of this is shown below.

```
import math

def mean_std_norm_data(data, mean_value, variance_value,
    epsilon=1e-6):
  m = len(data) # the number of elements we estimate the
    variance over

  # Normalise the data
  norm_data = []
  for x in range(m):
    norm_value = (data[x] - mean_value)/ (variance_value+
    epsilon)
    norm_data.append(norm_value)
  return norm_data
```

7.3.2 Variable Number of Arguments

Sometimes we want to have a variable number of arguments to a function. This can be easily achieved once we know the rules for doing so. There are two patterns one for just a list of arguments and one for a list of arguments with keywords (e.g. a name for each argument). The following is the pattern for a list of arguments referred to as `*args`.

```
def var_num_args(*args):
  for curr_arg in args:
    print("var_num_args:", curr_arg)
var_num_args('a','hello','finished')
```

The following is the pattern for a list of keyword arguments referred to as `**kwargs`.

```
def var_num_args(**kwargs):
  for key, value in kwargs.items():
    print("var_num_args:", key, "with value", value)
var_num_args(first='a',second='hello',third='finished')
```

7.4 Decorators

In essence a decorator works through the idea of defining a function within a function. What do we mean by a function within a function? Below we give a concrete example.

```
def my_hello_world():
  def say_gday():
    return "G'day"
  return say_gday()

hi = my_hello_world()
print(hi)
```

The output of the above code is.

```
G'day
```

This is because the nested function, in this case `say_gday`, can access the variables of the preceding function `my_hello_world`. This idea can be very useful and we demonstrate this by introducing the idea of a *decorator*.

First, the pattern of a decorator is to enable us to:

1. do something before calling a function,
2. then call the function, and
3. then do something after the function has been called.

In code this is given below with an example where we pass a function, called `my_func` to the decorator function.

```
def my_decorator(func):
  def wrapper():
    print("Before the function is called.")
    func()
    print("After the function is called.")
  return wrapper

def my_function():
  print("Do something")

func_to_run = my_decorator(my_function)
func_to_run()
```

The output of the above code is.

```
Before the function is called.
Do something
After the function is called.
```

What happens in the above code is that we give to `my_decorator` the function (`my_function`) which is then executed in the nested function called `wrapper`. The above code snippet is the long way of employing in decorator, however, Python provides a simpler way to do this.

Python gives a shorthand way of providing decorator to "wrap" around a function using the `@` symbol. We can apply a decorator to a function by

write `@func_name` just before the definition of the function. A code snippet to do this is given below.

```
def my_decorator(func):
  def wrapper():
    print("Before the function is called.")
    func()
    print("After the function is called.")
  return wrapper

@my_decorator
def my_function():
  print("Do something")

my_function()
```

7.4.1 Example: timing decorator

We now describe a useful example for a decorator which is to use it work out how long a function takes to execute. The `time` module is used in the code snippet below.

```
import time

def timing_decorator(func):
  def wrapper():
    start_time = time.time()
    func()
    stop_time = time.time()
    total_time = stop_time - start_time
    print("The function took", total_time,"seconds")
  return wrapper

@timing_decorator
def my_function():
  print("Increment something for 1000 iterations")
  y = 0
  for x in range(1000):
    y += 1

my_function()
```

The output of the above code is.

```
Increment something for 1000 iterations
The function took 4.3392181396484375e-05
```

7.5 Practical Work

7.5.1 Mutable versus Immutable

Now we will take a look at the difference between mutable variables and immutable variables. Mutable variables are variables that we can change the internals of without changing their memory address. Immutable variables are the opposite, if we change them (if they can be changed) we are storing them in a new location.

1. Create a list with at least 3 items inside. Use the `id()` function and print the location of the list.

Change one of the variables within the list. Now check the `id` again, what do you see?

2. Now, what happens if we change a mutable object within a function? Create a function that takes as input: a list, an index, and what to change the value to. Inside the function make sure that the index is within the list, i.e. it isn't greater than the length of the list. Change the list at the specified index.

```
def <function_name>(lst, index):
  lst[index] = <something>
```

Now create a new list with at least 4 items. print out both the list id and the internal values within the list. Now use the function you created on that list.

Once again, print out the id of the list and the list contents. What do you see?

3. Do the same with dictionaries, can you use the same function that you created for the lists?

4. Strings are immutable objects, you can not change them without moving them to a new location in memory. Try changing an index into a string.

```
str0 = 'hello world!'
str0[3] = 'k'
```

What happens?

5. Tuples are another container class in Python. However, they are immutable objects i.e. they cannot be changed without changing their memory location. This is a benefit and a negative depending on what you want to do. To create a tuple we use the `()` brackets, like lists they can contain anything we want. Create a tuple that stores at least 4 items.

```
tup = (1,2,'this',1212)
```

Using the index changing function from exercise 2 see what happens when we try to change the created tuple? Can you think of when this would be beneficial? Compare this to a list or dictionary which is mutable?

7.5.2 More on Functions

In Python we have two types of inputs to a function, with and without default values. Previously, we have used different names for each input specifically, like the following:

```
def <function name> (input0, input1, input2, ..., default0=1,
    default2=2 ...):
```

We then use these specific variable names to do things within the function. However, under certain circumstances, we can use `*args` (arguments) and `**kwargs` (keyword arguments) instead. In our previous examples we have calculated the diameter, circumference, and area of a circle using a single radius input.

```
def circle(r):
  d = r*2
  c = 2*3.14*r
  a = 3.14*(r**2)
  return d, c, a
```

What about if we wanted to calculate multiple radius values? In this case we could set r to *args (it doesn't have to be args but this is the standard notation).

```
def circle(*args):
  output = {}
  for r in args:
    d = r*2
    c = 2*3.14*r
    a = 3.14*(r**2)
    output[r] = {'d':d, 'c':c, 'a':a}
  return output
```

What are we returning here? How do we interpret the output? Can you see a problem with this code?

In this case, we haven't supplied a pi value to the function, we have used a hard-coded value. We can use **kwargs to supply default values. Keep in mind that standard inputs (*args) need to be before default arguments (**kwargs), always! Let's store the value for pi using the kwargs arguments.

```
for circle(*args, **kwargs):
  if 'pi' in kwargs:
    pi = kwargs['pi']
  else:
    pi = 3.14
  print( pi )
```

How do you think this should work?

1. Create a rectangle function that takes either 1 or two arguments in the *args. If there is only one argument we calculate it as a square, if there are two we calculate things as a rectangle. Next, in the **kwargs tell us the rounding value we will use? As a default it should be 2 decimal places.

7.5.3 Decorators

Decorators are considered functions within functions. They can be useful for timing how long a function takes to run.

1. Create a timing function within a function that times how long it takes to run a for loop over `N` iterations, where `N` is an input to the function. To do the timing we will import the time library. To get the run time at any point in the code you can use `time.time()`.

The problem with this function is that outside of `myfunc` the `internal` function does not exist.

2. Update the `myfunc` function to take as input another function and `N`. In this new function we will put the for loop, all the other timing information will remain in `myfunc`.

3. Now extract that even further and use the `@myfunc` above the function you created to do the for loop. Remember, in `myfunc` you need to return the `internal` function as this is the decorator function we will be using.

```
def myfunc(func):
  def internal(N): # the function I will be using to decorate
    func(N) # the actual function that we will be using
  return internal # returning the decorator

```

```
6 @myfunc # the important stuff to note here is internal
     function from above
7 def foo(N):
8   <internal stuff>
9
10 foo()
```

Chapter 8

Basics of Classes

Classes are an invaluable programming tool. They allow us to *encapsulate* information (data/variables) and functionality (functions). Furthermore, through *inheritance* and *polymorphism* we can quickly generate new classes by only overriding particular behaviour (e.g. functions) of the parent, or super, class. In fact, every variable in Python relies on a class. In this chapter we will give a brief overview of classes and in particular how they are defined in Python.

8.1 Making a Class

Let's start our discussion about classes with a very simple example which contains data and then provides us ways to apply some operations to the data. This example gives us the insight that a class can hold variables as well as functions. In object-orientied programming the data or a variable in a class is often referred to as an *attribute* of the class and a function in a class is referred to as a *method*. Let's now give an example of how to make a class in Python.

In Python you can make your own class by using the `class` definition. This is similar to the use of `def` for functions except now everything related to the class is intended. Another concept for a class is that we need a way to create or initialise the class upon creation. In Python this is achieved by defining a way to initialise a class by defining a special method (function of a class) called init and referred to in code as `__init__`. An example of this is below.

```
class MyFirstClass:
  def __init__(self):
    self.iteration = 0
    self.value = 10

  def add_to_value(self, value_to_add):
    self.value = self.value + value_to_add
    self.iteration = self.iteration + 1
```

There are several aspects that we'll now discuss about the above code.

1. The class `MyFirstClass` is declared as being a class by using `class`. The identifier for the class follows the rules for an identifier as we discussed in Section 1.3. Notice that all the lines after the declaration of the class are now indented. This indicates that all the code below is associated to this class as it is defined within the indented code block after the `:`.

2. The `__init__` method is declared as being a function (method of this class) using `def`, all the code for this is then also indented after the `:`. This is also the case for the method `add_to_value`.

3. There is something called `self` which is an argument to each method. This is because `self` allows us to refer to the current state of this particular *instance* of the class to access its methods (functions) and attributes (data). This is how you can access the attributes of this class which are `iteration` and `value`. This idea of an instance of class is something we will now discuss further.

In the above we just made a statement about an *instance* of a class, what is an instance of a class? An instance of a class is an object which now exists. The object (instance of a class) will now have the relevant methods and attributes, but now we can alter the state of this object. The state of the object is usually recorded through its attributes (variables). In the above example we have two attributes `iteration` and `value`. Just in case this explanation didn't work we give you a second explanation and then some associated code to clarify.

A class describes the template of what data and operations can be performed. When you *make* a particular *instance* of a class, this means you have a new version of that template. This is an object. The values in each *instance* (object) can change over time. Let's describe this concretely in code below.

```
object1 = MyFirstClass()
object2 = MyFirstClass()
print(object1.iteration, object1.value)
print(object2.iteration, object2.value)

object1.add_to_value(3)
object1.add_to_value(300)
print(object1.iteration, object1.value)
print(object2.iteration, object2.value)
```

In the above code we made two objects which are instances of `MyFirstClass` called `object1` and `object2`. As we go through our program we make some changes to `object1` and these changes only occur in that *object*. Try the above code and see what happens!

8.2 Inheritance

Classes are useful not just because they can encapsulate data and associated operations, they also provide a property called *inheritance*. Inheritance allows us to take the methods and attributes of a parent (previous) class and use them in our new (child) class. Why would this be useful? Well, let's say we thought of new ways to manipulate our data and we wanted a separate class to do this but still also have the old ways of storing and manipulation the data? In that case we could inherit from the parent (previous) class and then define some new methods, or even attributes.

In Python we make clear that we are performing inheritance when we define our class. We use the same pattern of `class <identifier>` and now add brackets "()" and between these brackets state which class we are inheriting from. Let's make this clear in some Python code where we make a child class `MySecondClass` that inherits from the parent class `MyFirstClass`.

```
class MySecondClass(MyFirstClass):
  def __init__(self):
    super().__init__() # Call the super class version of
   __init__

  def subtract_from_value(self, value_to_subtract):
    self.value = self.value - value_to_subtract
    self.iteration = self.iteration + 1

object1 = MySecondClass()
object1.add_to_value(3)
print(object1.iteration, object1.value)
object1.add_to_value(300)
print(object1.iteration, object1.value)
object1.subtract_from_value(32)
print(object1.iteration, object1.value)
```

Our new class relies on the old `__init__` of the parent class and has the same attributes and methods. We can explicitly call the parent classes init function by writing `super()`. This is true, for any method of the super class, that we want to make use of directly in the sub-class.

The new class, or child class, `MySecondClass` has a new method called `subtract_from_value` as well as all the attributes and methods of the parent class. We can see this in the instance of `MySecondClass` referred to as `object1`. This instance is able to call the parent's `add_to_value` as well as the new `subtract_from_value`. Also, we didn't define the attributes in the new class and relied on the parent class for this.

All of the above is possible by exploiting *inheritance*, that is we are inheriting behaviour from a parent (previous) class. An advantage of this is that we don't need to rewrite any code and so code maintenance becomes easier. Any update provided in the parent class is automatically being used in the child classes because they are inhereting the behaviour of the parent class. But, what if we wanted the child class to be able to **change** the behaviour of methods previously defined by the parent class? This cannot be achieved using inheritance, instead we rely on another concept in object oriented programming class polymorphism.

8.3 Polymorphism

The idea of polymorphism is that in addition to inheriting things like behaviour the child class can also **change** the behaviour. An example of *polymorphism* would be to *inherit* from `MyFirstClass` and simply change the way that the method `add_to_value` works. We will use an example where we ensure that only positive values can be added. Additionally, we will alter the *init* function and set the initial value to something else. Both of these are examples of polymorphism.

```
1  class MySecondClass(MyFirstClass):
2    def __init__(self):
3      super().__init__() # Call the super class version of
       __init__
4      self.value = 2
5
6    def add_to_value(self, value_to_add):
7      if value_to_add > 0:
8        self.value = self.value + value_to_add
9        self.iteration = self.iteration + 1
10     else:
11       print("Not adding the negative value", value_to_add)
```

We have now introduced the main concepts of classes which is how to define them. That classes can encapsulate data and behaviour. Furthermore, other child classes can inherit behaviour from a previous or parent class and finally that the child classes can be polymorphic. That is, they can change the behaviour of the methods of the parent class.

8.4 Practical Work

8.4.1 Classes

Classes are a powerful yet often confusing tool in Python. In the previous practicals, we created functions to calculate values such as the diameter, area, and circumference of a circle. However, once we have used that function all internal values are removed from python memory and we only have the returned values. Classes have the ability to store both members/attributes (variables) and methods (functions) for us to use. The basic class we will be using is as follows:

```
class <class_name>():
  def __init__(self, *inputs):
    pass
```

There are a few things to unpack here. First, we use `class` to declare that we are making a new class. Then, we declare the class name, you need to follow the same rules as with variables or functions. The methods (internal functions) are declared using `def`. In this example, we are creating a method called `__init__`, this is a built-in method that acts in a very specific way (dunder methods). The `__init__` method is called when we create the class object, so it initalises the class. Anything you need to happen at instantiation you put in this method, such as member initialisation.

1. Create a class called `shape` and in the initialisation store a string of what type of shape it is. You'll need to supply this shape type at initialisation.

```
obj = shape('square')
```

In the class also create a method that outputs a description of the shape using the print function.

```
obj.print_description()

>>> This shape is a square.
```

2. Now let's create a circle class that inherits from shape. Don't forget to initialise the parent class: `super().__init__(<shape type>)`. In this classes `__init__` function we are going to pass the radius and store it as a member.

Next we are going to create three further methods:

1. a method to calculate the diameter; and
2. a method to calculate the circumference; and finally,

3. a method to calculate the area.

Each of these methods will store the individual members/attributes with their desired value.

Finally, we will create a method that calculates all three values and returns them. What happens when you use the print function even though we haven't coded it in the child class?

3. Now overwrite the print method from exercise 1. In this new method you will print the shape type, the radius, the diameter, the circumference, and the area. This is an example of polymorphism.

4. Do the same exercise but with a rectangle class.

Chapter 9

Input and Output

The programs that you write will often need to interface with the world (externally). It is very rare for your programs to require no input and to also provide no output. So far we have seen the output being obtained by using the `print` function, however, saving and loading to files is also important. In this chapter we cover the basics of how to better use the `print` function as well as interacting with files in terms of input and output. We then finish by briefly describing the `pickle` format that is particular to Python.

9.1 The `print` function

The print function is a general function for writing to a stream (file). We have been making extensive use of the `print` function to demonstrate the state of our variables and code. To do this we have been exploiting the fact that the `print` function will provide a string-based output. To date, the stream that we've been using is the default stream which is the screen.

To better understand how the print function works we present the definition of it from the documentation below.

```
print(*objects, sep=' ', end='\n', file=None, flush=False)
```

There is one mandatory argument `*objects` and because of the `*` symbol there can be one or many values for this input argument. In the print function itself each object is converted to a string by calling its associated `str` method. This is possible because every object in Python has some form `str` method associated to it; we discuss this in more detail in Section 11.2. We then have named default arguments.

1. The argument `sep` is the separator to put between the string made from every object (from `*objects`). By putting a space there will be

a single space but you could define this to be anything. It's interesting to put different values for example a semi-colon (`;`) so you can clearly see the impact of this argument.

2. The argument `end` is what to add in addition to the very end of the string. By default this is a newline character ("\n") but you can replace it to be anything.

3. The argument `file` defines where the print statement is applied to. When `None` is supplied this defaults to the screen. Otherwise this needs to be a file which has a valid `write` method, you will see this later in this chapter.

4. The argument `flush` is used to describe when the data is actually written. This is sometimes useful because when you write to a file sometimes it is buffered, so that the data writing process can efficiently share resources. However, in certain cases it can be important to forcibly write (flush) the data buffer. This can be achieved by setting `flush=True` and by default it is `False`.

Armed with the above information we can now use the `print` function in far more creative ways. Below we show one approach where we change the separator from being a space to being a semi-colon. You are encouraged to try different ways to use this function along with its arguments.

```
string1 = "I am"
string2 = "very happy"
string3 = "to meet with you"
print(string1, string2, string3, sep=';', end="!!!!!!\n")
```

The above code has 3 objects as the input arguments (`string1`, `string2`, and `string3`), the separator between them has been replaced by a semi-colon and the last part of the output string (`end`) now consists of exclamation marks (`!`) and the newline character. The output for this is given below.

```
I am;very happy;to meet with you!!!!!!
```

9.2 Files: data in and out

A more general way of outputting and even reading in data is to manipulate files. We'll introduce this concept first with the idea of files with text and how we can load and save such data. For this we'll be relying on our knowledge of strings.

Let's take the example of opening a file for reading in some data. We get access to a file by *opening* it and to do this we make use of the `open` function. This function returns us an object (an instance of a class) which gives us a way to access the data stored in the file which has been opened. It also provides us with a way to close the file so we don't just leave it open. Python provides convenient ways to read one full line by enacting the method `readline`. In essence this function is reading and storing the information until it finds a newline character (`\n`) or similar. There is an additional shortcut in Python which is the function called `readlines`. Rather than reading a single line the `readlines` function reads all of the lines in a file and stores it in a list. We use the shortcut of using `readlines` in the code snippet below to show how to open a file for reading, reading its contents, closing it and then printing out the contents that we read in.

```
1 fp = open('blah.txt','r')
2 data_buffer = fp.readlines()
3 fp.close()
4
5 for curr_line in data_buffer:
6   print(curr_line)
```

In the above set of code, you can see that every time we access a file we need to open it, then perform our operations and finally close it.

When opening the file the function `open` accepts two arguments, both of which are strings. The first argument is the name of the file to access and the second is the mode in which we access the file. The two normal modes that we use are to read the file ('r') or write to the file ('w'). We can also choose to access a file in binary mode ('b'). All of the modes can be stacked into a single string so that we can actually open a file for reading and writing by using 'rw' as the second argument to `open`. More information about reading to and writing to files can be found in the Python documentation.

The above code snippet consists of a set of actions that we will perform whenever we do processing on a stream (e.g. file).

1. Get access or initialize something.
2. Perform operations on what we just got access to.
3. Close what we had access to.

For the file opening example we will, each time, have to write `x=open(...)`, `<do something>` and then `x.close()`. Because this type of pattern occurs frequently Python provides us with a convenient way to do this without having to type all of this code each time. This is provided by the `with` statement.

9.3 The `with` statement

The `with` statement provides a quick and concise way to perform opening and closing of things like files. It automatically handles the creation and deletion of the references to the file. This means that we don't have to call things like `.close()`. The `with` statement follows the pattern as shown below in pseudocode.

```
with <open file statement> as <variable pointing to file>:
  <do stuff>
```

In the pseudo code we don't explicitly close the variable pointing to the file as this is handled by the `with` statement. An example of opening and closing a file using the `with` statement is given below.

```
with open('blah.txt','r') as fp:
  data_buffer = fp.readlines()

for curr_line in data_buffer:
  print(curr_line)
```

The `with` statement can be used for more than just opening files. Other examples are provided in Section 11.3 of this text along with an insight into how this is enabled in Python.

9.4 Pickle Files

Previously we showed how to open text files and to read data in this way. However, data that we want to load and use is often not stored as simple text but rather stored as variables or another intermediate representation. To make it easier to load such data in Python we can make use of pickle files.

Pickle files are a file format for Python that lets us load and save variables quickly and easily. When writing data to a pickle file we need to pay attention to the following:

- the file is open to binary writing the 'b' option, and
- we use the function `dump` from the `pickle` module.

The `dump` function takes two arguments which are the object to write to the pickle file and the file pointer which is where to write the data to. Let's take an example of saving a dictionary called `my_dict`.

```
import pickle
my_dict = {'item1': [1,2,3], 'item2': 458}
fname = "my_pickle.pkl"
with open(fname,'wb') as fp:
  pickle.dump(my_dict, fp)
```

When loading data from a pickle file we need to pay attention to the following:

- the file is open to binary reading the 'b' option, and
- we use the function `load` from the `pickle` module.

The `load` function takes one argument which is the file pointer where the data is to be read from. Let's take an example of loading a dictionary called `my_dict` from a pickle file.

```
import pickle
fname = "my_pickle.pkl"
with open(fname,'rb') as fp:
  my_dict = pickle.load(fp)
print(my_dict)
```

Which gives us the expected output of

```
{'item1': [1, 2, 3], 'item2': 458}
```

When loading from a pickle we can use a different name for the variable because we are defining it in the code. For instance, instead of naming the variable `my_dict` we could have called it `this_dict`. The same approach can be used for saving and loading any other variable whether that be a class, list, dictionary or integer. However, how do we save and load multiple variables?

There are two main approaches saving and loading multiple variables. The first is to save them in a particular order and then load them in a particular order. This sequential approach is achieved by calling `dump` multiple times and then knowing the order that `dump` was called and then loading the same number of times to get the desired objects through multiple calls to `load`. In this course, we do not recommend this approach rather we suggest that you instead store all the variables that you want to save and load in another variable such as a `list` or a `dict`.

The second approach to saving and loading multiple variables is to place them in either a `list` or `dict`. In fact, we saw that for the previous example because `item1` was in fact a list. But let's do this explicitly for 3 variables in the code below.

```
import pickle
fname = "my_pickle.pkl"
var1 = [1,2,3,4]
var2 = "This is my second variable"
var3 = {'entry1': [22,15], 'entry2': 'Some stuff'}
my_pickle_data = {'item1': var1, 'item2': var2, 'item3': var3
    }
with open(fname,'wb') as fp:
  pickle.dump(my_pickle_data, fp)
```

We could then load the data by loading the data dictionary similar to what we did before. The data could then be accessed by using the keys 'item1', 'item2', and 'item3' as shown below.

```
import pickle
fname = "my_pickle.pkl"
with open(fname,'rb') as fp:
  my_pickle_data = pickle.load(fp)
print(my_pickle_data['item1'])
print(my_pickle_data['item2'])
print(my_pickle_data['item3'])
```

This gives us the expected output below.

```
[1, 2, 3, 4]
This is my second variable
{'entry1': [22, 15], 'entry2': 'Some stuff'}
```

9.5 Practical Work

9.5.1 Deeper into the Print Function.

To date, we have only used the basic functionality of the `print` function. Generally, the print function uses the following inputs.

```
print(*args, sep=' ', end='\n', file=None, flush=False)
```

We have previously played with `*args` and know that we can supply a large number of inputs to the function using this. Then `sep` is used to say how we split up the different inputs, and `end` informs the function of the final output in the string. Remember that `'\n'` is the end-of-line indicator.

1. Play around with this function using these two inputs and check how it functions. Other `sep` or `end` strings could be spaces, tabs (`'\t'`), or any other string type (even letters of other strings).

9.5.2 Data input and output.

In this part of the practical, we will learn how to write information to a file and then read it back. We will see that you can use a number of techniques to do this.

First, let's see how we can write to a file using `write`:

```
fid = open('file.txt', 'w')
fid.write( <text> )
fid.close()
```

We can also do this with the `print` function:

```
fid = open('file.txt', 'w')
print(<text>, file=fid)
fid.close()
```

1. Play with both of these types of file creation functions. What is the difference between the two techniques? What happens with `write` if you don't include an `'\n'` character?

2. Currently, when we opened the file we completely overwrite it by using `'w'` in the `open` function call. What about if we want to add to an existing file? Try creating the file again but instead use the append `'a'` variable instead of `'w'`. Don't forget to close the file!

3. Now we will read in from the same file. Open this file using the same command as previously but we will now use the read `'r'` variable instead of the write `'w'` variable. Read in all the contents of the file using `fid.readlines()`. Close the file. Then perform a for loop over the readlines variable and print out what was in the file. What is the type (`type()`) of each of these lines, even if you wrote out numbers to the file?

9.5.3 Data input and output using 'with'.

In the previous section we opened and closed the file manually. If you forgot to close the files strange things can happen in your code. There is a convenient way to do this where it closes the files automatically, we call this the `with()` statement.

```
with open(<filename>, 'w') as fid:
  <do something with the file>
# now it will automatically close the file.
```

1. Use the `with` statement to open a file and write something to it. Then use the `with` statement to read in the file contents.

9.5.4 Pickle library.

In the previous section we have written to a text file, however, when we read it back in it was a string. In this section we will learn how to write out storage classes and how to read them back in. For this, we will use the pickle library.

To write:

```
x = <something>
with open(<filename>.pkl, 'wb') as fid:
  pickle.dump(x, fid)
```

To read back in:

```
with open(<filename>.pkl, 'rb') as fid:
  x = pickle.load(fid)
print(x)
```

In both of these examples we have used `'wb'` and `'rb'` instead of `'w'` and `'r'`, this is because we are writing as *binary* to store bytes instead of strings.

1. Write a dictionary, list, and tuple to a pickle file individually. Then read it back in and print it to the screen.

2. What about if we want to store multiple containers in the same pickle file? Try to store an integer, list, and dictionary in the same pickle file, then read it back in and print the outputs.

Chapter 10

Numerical Python

A common use case for programming is to perform calculations, or numerical operations. This is particularly true for data analysis, signal processing and machine learning which often require operations to be performed on vectors and matrices. In this chapter we introduce some Python libraries that support such operations as well as how to plot, or visualise, results. We introduce the NumPy numerical library which has many similarities to other toolkits such as Matlab; an overview of the operations in NumPy and Matlab can be found online[1] and a general guide for NumPy is also available[2]. Finally, it is assumed that the reader understands basic vector and matrix operations such as matrix addition, matrix multiplication, transpose, and element-wise multiplication.

10.1 Plotting Stuff

If we're looking at data it's often interesting to consider how to visualise it. A common method for visualisation is to plot the data and usually there are toolkits to do this for you. For Python a commonly used toolkit for this purpose is the `matplotlib` library. This library has a rich set of functionality and we briefly introduce the ideas it has for line plotting. As with all the other material, we encourage the interested readers to pursue this further through other forums including the documentation and online discussions.

To make use of `matplotlib` you need to import it first using something like the following.

```
1 import matplotlib.pyplot as plt
```

[1] `http://mathesaurus.sourceforge.net/matlab-numpy.html`
[2] `https://numpy.org/doc/stable/index.html`

Here, we have used the `import <x> as <y>` so that when use the module `matplotlib.pyplot` we can just type `plt` instead. Now that we've imported the module we will use the line plotting capabilities, these can be accessed by calling the `plot` function. The `plot` function takes as arguments the values for the x-axis and the y-axis respectively. There are multiple potential extra arguments that we can also use which could define the width of the line, the type of the line (dashed, dots, dash-dots, etc.) as well as its colour; we enoucrage the interested reader to read the associated documentation for this function. Below is a code snippet showing how to call `plot` for a list of x and y values.

```
1 import matplotlib.pyplot as plt
2
3 y = [0.4,0.5,0.8,0.2,0.8,1.2]
4 x = [1,2,3,4,5,6]
5 plt.plot(x, y, 'b--', linewidth=2.)
6 plt.show()
```

At the very end of the code we have made use of `plt.show()` which says that what we've been *writing* to the plot should now be displayed to the screen. This would result in us seeing something like the plot given in Figure 10.1.

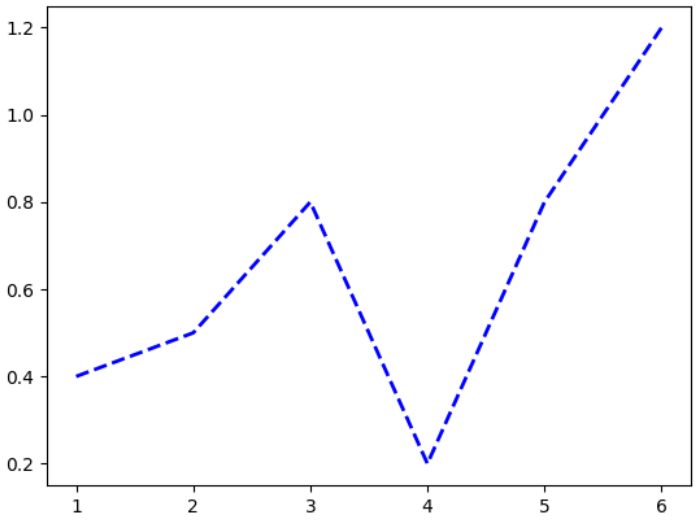

Figure 10.1: A line plot based on the code snippet. This shows an example of a line plot with dashed lines.

More complicated interactions with plotting can also be achieved. Another example is to include adding a marker to each data point and a code snippet for this is also included below.

```
import matplotlib.pyplot as plt

y = [0.4,0.5,0.8,0.2,0.8,1.2]
x = [1,2,3,4,5,6]
plt.plot(x, y, 'b-x', linewidth=2., markersize=12)
plt.show()
```

The output of the code snippet below is provided in Figure 10.2.

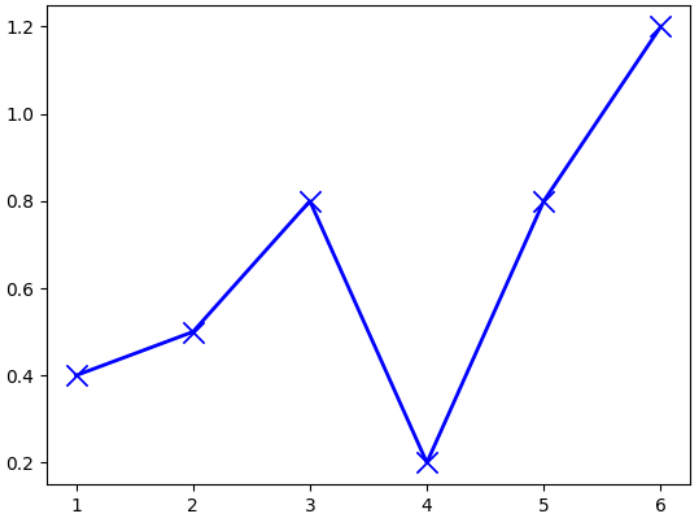

Figure 10.2: A line plot based on the code snippet. This shows an example of a line plot with a solid line and each point is indicated by a marker using X.

10.2 Matrices

The NumPy library enables you to perform almost any valid matrix operation. Simple examples are addition, subtraction and multiplication. Let's show this for relatively simple cases, if we have a matrix with 3 rows and 3 columns which has the shape $(3, 3)$.

```
import numpy as np
A = np.random.randn(3,3)
B = np.random.randn(3,3)

C = A+B
D = A-B
print(A, B, C, D)
```

For multiplication the question is whether we intend to do matrix multiplication or element-wise multiplication. Matrix multiplication is achieved using the `@` symbol while matrix element-wise multiplication is achieved using the `*` symbol.

```
import numpy as np
A = np.random.randn(3,3)
B = np.random.randn(3,3)

C = A*B
D = A@B
print(A, B, C, D)
```

Some other operations of note include:

- taking a row or column slice from a matrix
- `min`, `max`, `argmin`, `argmax`

There are many others but we leave it to the interested reader to explore the documentation further on this subject.

A common operation that has to be performed are operations of vectors on a matrix. Unfortunately, this can introduce some complications as described below.

10.3 Operations on Vectors and Matrices

Vectors in NumPy are retained as having just a single dimension (1D) and this is problematic. As such the general suggestion is to be extremely careful with explicitly using a vector in NumPy. The issue is that by having only a single dimension various subsequent operations can lead to erroneous output. For instance, performing the transpose on a 1D variable leads to the identical 1D variable which is not the purpose of a transposition. Let's make this concrete by first describing some basic operations for vectors and matrices.

Normally, we will have either a row vector or a column vector. The concept of a row or column vector already assumes there are two-dimensions. We can see this by writing down a row vector

$$\mathbf{x} = \begin{bmatrix} x_0, x_1, \ldots, x_{n-1} \end{bmatrix} \tag{10.1}$$

which has 1 row of n columns with an entry per column, such a shape is commonly written as being $(1, n)$. The transpose of this is the column vector

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} \tag{10.2}$$

which has n rows of 1 column with an entry per row, such a shape is commonly written as being $(n, 1)$. For our example we can make a column-vector with randomised values as shown below.

```
import numpy as np
vec = np.random.randn(3,1)
print(vec)
```

In NumPy the shape of an n-dimensional vector is considered to be $(n,)$. This means there is no second dimension and so there is no concept of a row or column vector. As such, the transpose operation loses its meaning. This subsequently means that it is not possible to multiple a column vector by a row vector, normally this operation would lead to matrix of shape (n, n). To avoid this and other related issues it is highly recommended to default to implement a vector in NumPy using a matrix instead, but ensuring that one of the dimensions has only 1 entry to form either a column or row vector.

10.4 Other Operations

We are also going to introduce four more operations and describe how they can be applied to matrices. These operations provide insight into how other, similar, operations are also applied. We concentrate on `max` and `min` as well as `argmax` and `argmin`.

The `max` and `min` operators can be applied to both vectors and matrices. For a more general description we describe how this operates on a matrix. These two functions will find the minimum or maximum value in the entire matrix, and return that number. Two similar, but slightly different operators are `argmax` and `argmin` which finds the first *argument* that has the maximum or minimum value respectively. These functions can also be applied to a particular axis of a matrix.

Before we discuss how these operations are applied to a matrix let's first describe a matrix. Let's consider that we have a two-dimensional matrix and

it has 3 rows and 4 columns. We write this out as

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \end{bmatrix} \tag{10.3}$$

In NumPy the first (0-th) axis is considered to be the rows and the second (1-st) axis is considered to be the columns.

If we apply our operations over an axis then it will return a result for the size of the other axes (excluding the current one). This is because the operation is applied across the axis in question and a result is returned. More concretely if we apply our operation over the 0-th axis (the rows) and we have 3 rows and 4 columns we will get a result by applying the operation across the **row** for each **column**. Let's show this with some code by defining a 3×4 matrix and take the `min` and `argmin` on this matrix.

```
import numpy as np
x = np.array([[1,2,3,4],[2,3,1,4],[9,8,7,6]])
print("Matrix X\n", x)
print("Minimum value for each column", np.min(x,axis=0))
print("Index for the minimum value for each column", np.
    argmin(x,axis=0))
```

The output of the above is given below.

```
Matrix X
 [[1 2 3 4]
 [2 3 1 4]
 [9 8 7 6]]
Minimum value for each column [1 2 1 4]
Index for the minimum value for each column [0 0 1 0]
```

10.5 Practical Work

10.5.1 Matplotlib

One of the key components of machine learning is both the ability to visualise data and visualise results. In python there is a nice library called `matplotlib`, it's a really handy library with a number of functions. Let's start with some basic visualisations and introduce a couple of handy numpy functions. First, we need to setup the x axis of a plot, for this there are lots of options but let's introduce `np.linspace(<start>, <finish>, <numvals>`. In this case `<start>` is the starting value and `<finish>` is the final value while `<numvals>` is the number of points to equally divide over that range. Then the y axis, we'll use the sine function in numpy (`np.sin()`).

Now we need to put these together in a plot using `matplotlib`, first let's import this library `import matplotlib.pyplot as plt`. Once you have imported the library you can use it, let's see what that looks like:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 6*np.pi, 100)
y = np.sin(x)

# create a figure
plt.figure()
plt.plot(x,y)
plt.xlabel('x'); plt.ylabel('sin(x)')
plt.title('basic sine plot')
plt.show()
```

1. Now plot three different functions on the same graph; `sin(x)`, `sin(x**2)`, and finally `cos(x)`. Each of these plots needs their own label `plt.plot(x,y,label='<your label>')`, and their own colour `plt.plot(x,y,'r')` will plot in red.

2. Sometimes this can get a little messy with multiple plots on the same graph. Use `plt.subplots(...)` to plot the three different functions from exercise 1 using a single column and three rows (i.e. `plt.subplots(3,1)`).

10.5.2 Matrices and Vectors

Numerical Python (Numpy) is a library that contains almost all matrix and vector mathematical operations. It allows us to create and change matrices

and vectors however we wish. But we need to be careful of these two objects, matrices and vectors, as the way you use them can cause issues in your code.

```
import numpy as np
x = np.array([1,2,3])
print(x)
print(type(x))
print(x.shape)

>>> [1,2,3]
>>> <class 'numpy.ndarray'>
>>> (3,)
```

You should notice that this only has a single dimension `(3,)`, meaning that this is a vector, not a matrix. If we try to transpose (`x.T`) this vector we will end up with the same size. This vector property can be detrimental to different functions, so be careful.

How do we create a matrix?

```
import numpy as np
x = np.array([[1,2,3]]) # adding [] to show an extra
    dimension
print(x)
print(type(x))
print(x.shape)

>>> [[1,2,3]]
>>> <class 'numpy.ndarray'>
>>> (3,1)
```

This matrix has three rows and one column. If we transpose this we will get one row and three columns.

1. Create a vector and add, subtract, multiply, and divide an integer to it.

2. Create a matrix and add, subtract, multiply, and divide an integer to it.

3. Create a matrix with random integers of size 5×4, with a lower value of 0 and a higher value of 10. You will need to use `np.random.randint(<lower value>, <higher value>, size=[rows, columns])`. Print this matrix, what is the maximum value it seems to store? Run this a number of times and compare.

4. What happens if we try to add a vector of size `(4,)` (number of columns) to this matrix? Create a vector `[1,2,3,4]` and add it to your matrix. What about subtraction, multiplication, and division? What about a vector of `(5,)` (number of rows)?

5. From the above exercise, you should have seen that vectors create some issues. Now, create a matrix of size `(5,1)` and add, subtract, multiply, and divide it to the matrix.

6. Create a second matrix of size 5×4 and use different arithmetic operators on the two matrices.

Finally, what about matrix multiplication (dot product)? For this to work you will need to the internal dimensions to be the same, so create a new matrix of size 4×5 and use the @ symbol for matrix multiplication. You should get a 5×5 matrix as the output.

7. Some other important operators on matrices include: `np.min`, `np.max`, `np.mean`, `np.std`, `np.argmax`, `np.argmin`. Using your created matrix from exercise 3 use these different operators. What do each of them do? What's the difference between `np.max` and `np.argmax`?

8. the `np.argmax()` and `np.argmin()` functions on vectors give you a single number rather than a row and column. To interpret this value we will use two extra mathematical operations.

```
import numpy as np
x = np.random.randint(0,10,size=[5,4])
# get the index of the largest value in x
ind = np.argmax(x)
# get the row using floor divide
row = ind//x.shape[1]
# get the column using the modulo command
col = ind%x.shape[1]
print(row, col)
```

Play around with argmax and argmin on a 2d matrix.

9. Finally, we can use these values on the rows or columns, as an example of `np.mean()`:

```
x = np.random.randint(0, 10, size=[5,4])
print(np.mean(x,axis=0))
print(np.mean(x,axis=1))
```

So the axis is what we calculate our statistic over, in line 2 we are calculating our mean over all the rows in a column. And the opposite is true for line 3.

Use the other matrix operations on your matrix using different axis.

Chapter 11

Deeper Into Classes

Classes enable the idea of object oriented programming (OOP). We have previously seen how they can be used to implement the idea of inheritance as well as polymorphism. Relatively simple examples of these were provided. We now extend our discussion and provide further practical examples of how to exploit the ideas of OOP, in particular polymorphism, to provide enhanced behaviour for our classes. But, before we provide examples of enhanced behaviour we first describe how classes are used more generally in Python.

11.1 All Variables are Classes

In Python, all variables are in fact classes and they all inherit from a class called `object`. In essence `object` is the base class of all classes in Python. You can see this in practice by making a variable with an integer and querying it using the `help` function in a Python terminal. You can bring up a Python terminal on the command line by typing `python3` or by calling the `python3` app[1]. In Figure 11.1 you can see an example of the output from using `help` on a Python `int` object. The definition of `int` is that it is a class which inherits from object (`class int(object)`). We have been exploiting the fact that every variable is a class already in Python when we made use of the `print` function. This is because in the background the `print` function is making use of the `str` method in each class.

[1]This will depend on how you have installed Python on your operating system.

```
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> my_test_int = 8
>>> help(my_test_int)
Help on int object:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
```

Figure 11.1: An image of the output from using `help` on an integer (`int`) object in Python3.

11.2 The `str` method in classes

Often when we've been using `print` function we have used it to display the state of our variables. But, every variable is actually an object and so what happens is that every class defines its own way to dispay itself as a string using a method called `str`. This means the `print` function was just calling this `str` method for each object; this is an example of polymorphism where the result of the `str` method is overridden (defined) by each class.

The `str` method for each class can be overriden, that is its definition can be changed for each class. An example of how we can do this ourself is given below.

```
1  class MyFirstClass:
2    def __init__(self):
3      self.iteration = 0
4      self.value = 10
5
6    def add_to_value(self, value_to_add):
7      self.value = self.value + value_to_add
8      self.iteration = self.iteration + 1
9
10   def __str__(self):
11     out_str = "Value: " + str(self.value)
12     out_str += " at iteration " + str(self.iteration)
13     return out_str
```

In the above code you can see that we have provided our own `str` method, even more complicated multi-line strings are possible if we desire. Again, this

is possible because all variables in Python are in fact objects (an instance of a class). We now go further and describe how another statement, the `with` statement, makes use of this fact.

11.3 The `with` statement again

The `with` statement is an example of using defined behaviour in classes from Python. The `with` statement fits the pattern of opening and closing access to something. An example of this is to open a file that we read from or write to, more generally this idea is the idea of a stream that we read from or write to.

The `with` statement assumes that the methods `__enter__` and `__exit__` are defined by the class that we use. In the context of file manipulation, `__enter__` defines how to open the stream (file) while `__exit__` defines how to close the stream (file). A concrete example is to look at what's going on with the File class and we've put a concise version of this below.

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)

    def __enter__(self):
        return self.file_obj

    def __exit__(self, type, value, traceback):
        self.file_obj.close()
```

Above, the `__enter__` method exploits the initialisation of this instance of the object by returning the already opened file object; note the `__init__` function has stored this information is `self.file_obj`. The `__exit__` method then handles the closing of the file object which in this case is to call `close` on `self.file_obj`. Let's do another example which doesn't rely on a file stream.

Another example is to make a class that records how long a block of code takes to execute. Let's call this class `MyTimer`. It will record a name to make it easier for us to understand. Then when we *enter*, which is the start of the `with` statement, we record that as the start time. When we exit, we take the difference of the start and the stop (current) time. We then output this to the screen using the `print` statement. If we were to write this in Python code it would look something like the code below.

```
import time

class Timer(object):
  def __init__(self, name):
    self.name = name

  def __enter__(self):
    self.start_time = time.time()

  def __exit__(self, type, value, traceback):
    out_str = "Elapsed time"
    out_str += " for " + self.name
    out_str += ":"
    print(out_str, (time.time() - self.start_time))
```

We now even deeper into how classes work in Python by describing how they exploit the concept of *decorators* to provide `get` and `set` operators.

11.4 Classes Again: `get` and `set`

Classes in Python provide a flexible way to define *properties* of a class. This is done by enabling methods to get, set and delete the values of the *properties* (effectively an attribute). This let's us build operations related to setting, getting and deleting attributes. An example of why this is useful is that we can provide extra checks when setting an attribute or even incorporate dependencies on other attributes before it gets set. To achieve this, Python makes use of decorators.

To define a function to *get* an attribute we use the `@property` before the function. The `@property` is a decorator, associated with this particular decorator is the potential to define `get`, `set`, and `delete` operators; in these notes we do not cover use of `delete`. Now that we've defined that the property exists we can define how to `set` the value of this property by using `@<property_name>.setter` as a decorator before the function which sets the value of the property. Example code snippets for these two concepts is given below for a `Train` class where we are setting the current speed of the train.

```
class Train:
  def __init__(self):
    self.current_speed = 0

  @property
  def current_speed(self):
    return self.my_current_speed

  @current_speed.setter
  def current_speed(self, value):
    self.my_current_speed = value

this_train = Train()
print(this_train.current_speed)
this_train.current_speed = 10
print(this_train.current_speed)
```

There is obviously a complexity in using the idea of using a *property* (e.g. the *get* and *set* approach for classes) so when is this useful?

An advantage of using a property is that we can easily make the set function more complex. For instance, if there is a defined upper limit on the speed of a train we can ensure this when setting the current speed by adding a check within the setter method. We demonstrate this idea in the code snippet below.

```
  @property
  def current_speed(self):
    return self.my_current_speed

  @current_speed.setter
  def current_speed(self, value):
    self.my_current_speed = value if value <= self.max_speed
   else self.max_speed
```

In this section we have describe two parts to properties in Python. There are actually three parts to the `@property`:

- *getter* (defined with @property)
- *setter* `@<property_name>.setter`
- *deleter* `@<property_name>.deleter`

We encourage the interested reader to explore more on this topic by exploring the Python documentation.

11.5 Practical Work

11.5.1 Deeper into String Creation

In this section of the practical we will look at four different methods for printing out ints and floats.

str()

The first method is the simplest form of creating strings from ints or floats.

```
s0 = 'This is an integer: ' + str(1)
s1 = 'This is a float: ' + str(3.14159)

print(s0)
print(s1)

>>> This is an integer: 1
>>> This is a float: 3.14159
```

%

```
s2 = 'This is an integer: %d\nThis is a float: %f'%(1,
    3.14159)
s3 = 'This is an integer: %04d\nThis is a float: %0.02f'%(1,
    3.14159)

print(s2)
print(s3)

>>> This is an integer: 1
This is a float: 3.14159
>>> This is an integer: 0001
This is a float: 3.14
```

format

```
s4 = 'This is an integer: {}\nThis is a float: {}'.format(1,
    3.14159)
s5 = 'This is an integer: {:04d}\nThis is a float: {:0.02f}'.
    format(1, 3.14159)

print(s4)
print(s5)

>>> This is an integer: 1
```

```
This is a float: 3.14159
>>> This is an integer: 0001
This is a float: 3.14
```

f string

```
i = 1
fl = 3.14159
s6 = f'This is an integer: {i}\nThis is a float: {fl}'
s7 = f'This is an integer: {i:04d}\nThis is a float: {fl:0.02
    f}'

print(s6)
print(s7)

>>> This is an integer: 1
This is a float: 3.14159
>>> This is an integer: 0001
This is a float: 3.14
```

1. Play around with each of these types of string assignments and work out which one you are more comfortable using.

11.5.2 Class Strings

We have used the print statement to print out all kinds of things in this unit so far. We have also built classes to do different things. In this section we will combine these two tasks into one, we will be able to call print on a class.

To do this we need to create the `__str__(self)` method within a class. Here is the pseudo code you will use.

```
class rectangle():
  def __init__(self, *args):
    # if there are two values store it as a rectangle
    # if there is only one argument it's a square.
    # remember you will need to store the height and width as
    members of the class.

  def calculate_circumference(self,):
    # calculate the circumference of the rectangle/square

  def calculate_area(self,):
    # calculate the area of the rectangle/square

  def __str__(self,):
```

```
    # return a string that includes the height, width
    # if it's a square or rectangle
    # the circumference and the area
    # try to do this in a single string using %, format, or
    fstring

  def str(self):
    # return a string that includes the height, width
    # if it's a square or rectangle
    # the circumference and the area
    # try to do this in a single string using %, format, or
    fstring

obj = rectange(<your inputs>)
s0 = str(obj)
print(s0)
print(obj)

>>> ???
>>> ???
```

1. Implement the above pseudo code into a working class and find out what the output to print is. What is the difference between `str()` and `__str__()`?

11.5.3 Class With

Let's implement the timer class from the lecture.

1. Implement the timer class, in the `__exit__` method using one of the string assignment techniques from above to do it in one line.

2. Using the `with` command find out how long it takes to iterate over 1000, 10000, and 10000 epochs in a for loop.

11.5.4 Class Get/Set

We will now use `getters` (`@property`) and `setters` (`@<member name>.setter`) for members/attributes in our rectangle class.

1. Using the lecture as an example, add `getters` and `setters` for our height and width members. Then use them within your code and see how this impacts your `__str__` function when printing.

11.5.5 Calling a class

Previously, to calculate the area and circumference we would have to call them explicitly from the object.

```
obj = rectangle(10, 20)
circ = obj.calculate_circumference()
area = obj.calculate_area()
```

Or alternatively, we could create a function that does both for us

```
class rectangle():
  ...

  calculate_stats(self,):
    return self.calculate_circumference(), self.
   calculate_area()

obj = rectangle(10,20)
c, a = obj.calculate_stats()
```

There is another option for us to use to calculate all important information, or perform certain important applications. Here, we will introduce `__call__()`.

```
class rectangle():
  ...

  def __call__(self,):
    # return the relevant information
    # do relevant important tasks

obj = rectangle(10,20)
<variables> = obj()
```

1. Implement the `__call__()` function in your rectangle code. It should return the circumference and the area at the least.

Chapter 12

Images and Basic Operations

Images are an interesting example of data which can be manipulated in multiple programming languages. In particular, images are usually represented as a multi-dimensional array and a variety of operations can be performed on them ranging from thresholding them, converting them into other images or representations through to performing colour conversions and applying operations (e.g. kernels) upon them. To do this we will use the `scikit-image` library which provides a consistent basis for simplified access (e.b. abstraction) to many of these operations.

12.1 Images

Images can be represented by matrices. Often they are represented as a three-dimensional (3D) matrix consisting of height (h), width (w), and number of channels (c). Height and width represent the physical aspects of the image, in terms of pixels, which the channels is required for colour images. To represent colour in an image we generally use 3 values which describe the mixing of red (R), green (G) and blue (B) colours. This is often referred to as an RGB image.

When dealing with images the order in which spatial elements are stored is important. Generally, the first elements in the matrix refer to the height (h), the second the width (w) and the third is the number of channels (c). In other words, it is stored as a three-dimensional matrix consisting of height $h \times w \times c$. It is important to understand that when representing an image in this multi-dimensinoal matrix format $(0, 0)$ is actually in the top-left of the matrix/image and is not the bottom left that is usually the case for cartesian coordinates. Let's look at example of this by accessing an image in Python using the `scikit-image` library.

To load and save images we can make use of the input/output (IO) functionality of `skimage`. Displaying images can then be achieved by using `matplotlib`, in particular using `imshow`. This will then dsiplay valid images, grayscale or colour. When executing the code from command line or in a prompt you will also need to force the drawing to the screen by calling `show`. An example snippet of code is given below.

```
import skimage, skimage.io
import matplotlib.pyplot as plt

c_fname = "test.jpg"
c_img = skimage.io.imread(c_fname)
plt.imshow(c_img)
plt.show()
```

From the above code, if we look deeper into the properties of `c_img` we will see the following. It has the shape of $h \times w \times c$ with the particular values of $1,010 \times 1,790 \times 3$. It is an `ndarray` which is a multi-dimensional array (e.g. matrix) which means that we can apply any of the `numpy` operations on this. This includes addition, subtraction, multiplication, division, and much more.

12.1.1 Generating Image Masks: boolean images

Quite often we want to obtain or to make use of image masks. An image mask is a binary representation of the image for instance to perform segmentation and declare if a pixel, or region, is of interest. An example of this could be to find all the pixels that we consider to represent vegetation based on their colour.

A simple way to process an image is to see if any of the values are above, or below, a pre-defined threshold. Similar to comparative operator on integers or floats, this would lead to boolean results, either `True` or `False`. The overall result would then be an image of boolean values which is often called an image mask. Let's go through how this kind of operation could occur what would happen to the resultant image.

How can we convert an RGB image to a mask? Normally we would do this based on an algorithm which given the colour value will provide the binary response of that value being the colour (e.g. `1` or `True`) or not being the colour (e.g. `0` or `False`). To represent the algorithm we will call it a function $g(\mathbf{x})$ where $\mathbf{x}$ is the input which in this case are the R, G, and B values or a 3-dimensional vector; the function $g(.)$ is not the same as a coding `function` in Python. The resultant mask will be of size $h \times w \times 1$ and use binary values to represent each element. Normally, we would drop the last

dimension and just refer to this as a $h \times w$ binary image.

Let's make a really simple algorithm or function that only looks at the green value to decide if there is vegetation or not. We will use a threshold where if the green value is greater than 200 then this is definitely green like vegetation would be. An example of this kind of algorithm is given below.

```
def generate_green_mask(c_img, threshold=200):
  out_img = np.zeros((c_img.shape[0], c_img.shape[1]), dtype=
    'bool')

  for h in range(c_img.shape[0]):
    for w in range (c_img.shape[1]):
      if c_img[h,w,1] > threshold:
        out_img[h,w] = True
  return out_img

mask_of_green = generate_green_mask(c_img, threshold=200)
plt.imshow(mask_of_green, interpolation='none')
plt.show()
```

It's interesting to consider some of the potential failure points of the code above. For instance, if the image was normalised to be in the range $[0 \ldots 1]$ would the code still work? Finally, once we have this mask what are some of the ways that we can use it?

12.1.2 Using Image Masks

An interesting thing we could do with the mask is to manipulate the image itself. We could, for instance, suppress or remove those elements of the image which are not vegetation by using the mask. This could be written as a for loop as shown below assuming that we have the mask as `c_mask` and the image is an RGB image `c_img`

```
masked_img = np.zeros(c_img.shape, dtype='uint8')

for h in range(mask_of_green.shape[0]):
  for w in range (mask_of_green.shape[1]):
    if mask_of_green[h,w]:
      masked_img[h,w,:] = c_img[h,w,:]

plt.imshow(masked_img)
plt.show()
```

However, the above code is inefficient because we run a for loop in Python over the image and explicitly set the data that exists.

There are several other ways to write the previous code more efficiently. This is in terms of less lines of code as well as computation time is given as an example of how to do this is given below.

```
masked_img = c_img
masked_img[c_img[:,:,1]<200,:] = np.array(([0,0,0]))
plt.imshow(masked_img)
plt.show()
```

The reason that the above code is more efficient is that the in-built functions in `numpy` are calling libraries made in C or a similarly efficient language. This avoid the computational issues that exist when running code using Python.

12.1.3 Extracting Part of an Image

Previously, we generated a mask and then blanked out the parts of the image that didn't want to see. An alternative to this is to extract a patch or sub-region of an image. To do this, we need to be able to index within the matrix and obtain only those elements that we want to obtain. For example we might only want a rectangular part from the center of the image. In such a case we can use matrix slicing by providing the index ranges that we want to use to obtain the region. Note that slicing also works on lists in Python.

For slicing a range of values is given. The range of values needs to be from the smallest value of the range to the largest value of the range. Something like the following

```
my_array[smallest_value:largest_value]
```

You must also include a valid range for all of the elements to extract and the range can be a single dimension. If we think about this for a colour image with three channels, we could extract just the red channel (1st channel) using the following code snippet.

```
my_colour_image[smallest_height:largest_height,
                smallest_width:largest_width,
                0]
```

Note that for clarity we have split the code for slicing over multiple lines; this is possible in many languages including Python. The resulting matrix will only have two dimensions as the third channel is just one element, we will see an example of this later on. If instead of taking only the red channel but we want to include all channels we can use a shortcut which is `:` as shown in the code snippet below.

```
my_colour_image[smallest_height:largest_height,
                smallest_width:largest_width,
                :]
```

Below, is a snippet of code to do this for an example image.

```
import matplotlib.pyplot as plt
from skimage import data

original = data.astronaut()
print("The size of the original data is: ", original.shape)
center_h, center_w = int(original.shape[0]/2), int(original.
    shape[1]/2)
region_size = [80,60] # height, width
subregion = original[center_h-int(region_size[0]/2):center_h+
    int(region_size[0]/2),
                     center_w-int(region_size[1]/2):center_w+
    int(region_size[1]/2),
                     :]
print("The size of the subregion data is: ", subregion.shape)

fig, axes = plt.subplots(1, 2, figsize=(8, 4))
ax = axes.ravel()

ax[0].imshow(original)
ax[0].set_title("Original")
ax[1].imshow(subregion)
ax[1].set_title("Subregion")

fig.tight_layout()
plt.show()
```

The above code, in addition to producing two images also outputs the following.

```
The size of the original data is:  (512, 512, 3)
The size of the subregion data is:  (80, 60, 3)
```

In the above example we have shown how to perform slicing by providing a range of values. In particular, the range of values used was given by the following

```
center_h-int(region_size[0]/2):center_h+int(region_size[0]/2)
```

which goes from the center point based on the height (`center_h`) and then up half the region size (`-int(region_size[0]/2)`) through to the center point based on the height (`center_h`) and then down half the region size (`+int(region_size[0]/2)`); keep in mind that the top-left index of this matrix (h, w) is $(0, 0)$. It is important to keep in mind that when indexing matrices an integer value has to be provided. For this reason you can see that sometimes the code explicitly converts values to integers using `int`.

12.2 Kernels on Images

A common operation that is applied to an image is something called a kernel. Effectively a kernel is an operation applied to every part of an image and usually takes a spatial area (window) of information into account. An example of such an operation is to blur an image this applies a Gaussian function across a spatial area (window) for all the pixels in an image, in simple terms think of it as smudging an image. Before showing how to do Gaussian blur kernel let's start with a simpler linear kernel known as a box linear filter.

The box linear filter takes the average value over a window and replaces the current pixel with that value. This operation, over one channel, for a $H \times W$ kernel can be written as,

$$\frac{1}{H * W} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{12.1}$$

where in this case we have $H = 3$ and $W = 4$. This kernel would then be applied to every pixel in the image for each channel separately. An issue arises when the kernel is to be applied close to the border of the image, this is because the filter will be applied outside the bounds of the image. There are several ways to overcome this issue ranging from using only the available valid values through to reflecting the existing pixels (like a mirror) to generate "artificial" values on which the full kernel is applied. Below we provide pseudo-code for how to apply a kernel, like the averaging kernel above, only where there is sufficient existing information for it to be applied in image.

```
import matplotlib.pyplot as plt
from skimage import data
import numpy as np

kernel_h, kernel_w = (11, 11)
my_kernel = np.ones((kernel_h, kernel_w)) / (kernel_h*
    kernel_w)

original = data.astronaut()
new_image = np.zeros(original.shape, dtype='int')
for h in range(5, original.shape[0]-6):
  for w in range(5, original.shape[1]-6):
    for c in range(original.shape[2]):
        sub_region = original[h-5:h+6,w-5:w+6,c]
        new_image[h,w,c] = int((my_kernel * sub_region).sum()
    )

print("The size of the original data is: ", original.shape)
print("The size of the filtered image data is: ", new_image.
    shape)

fig, axes = plt.subplots(1, 2, figsize=(8, 4))
ax = axes.ravel()

ax[0].imshow(original)
ax[0].set_title("Original")
ax[1].imshow(new_image)
ax[1].set_title("Filtered Image")

fig.tight_layout()
plt.show()
```

The above pseudo-code is very simplistic and slow. This is because it consists of embedded for loops. Also, it has hard coded values for the extent of the kernel in the for loops. The interested reader should consider how this could be recoded to remove this reliance on hard coded values and even how to remove or simplify the embedded for loops.

Fortunately, in Python and many other languages libraries are available to implement this and many other filters over images. In `skimage` this is available through `skimage.filters` and we provide a code snippet for doing this with a Gaussian filter to achieve Gaussian blur.

```
import matplotlib.pyplot as plt
from skimage import data
import skimage, skimage.filters

original = data.astronaut()
print("The size of the original data is: ", original.shape)
filtered_img = skimage.filters.gaussian(original, sigma=5,
    channel_axis=-1)
print("The size of the filtered image data is: ",
    filtered_img.shape)

fig, axes = plt.subplots(1, 2, figsize=(8, 4))
ax = axes.ravel()

ax[0].imshow(original)
ax[0].set_title("Original")
ax[1].imshow(filtered_img)
ax[1].set_title("Filtered Image")

fig.tight_layout()
plt.show()
```

In the above example the value for `sigma` represents the size of the Gaussian filter and is given as $\sigma = 5$. This results in a kernel of size $(2*\sigma+1 \times 2*\sigma+1)$.

12.3 Colour Spaces

Representing colour effectively can be very important to perform a range of tasks. One example is to estimate skin colour to estimate where people are all the way through to estimating from drone imagery where in a field there is vegetation vs not vegetation for agricultural applications. Considerable work has gone into what are some methods to represent colour and for perceptual purposes there are many better colour spaces than RGB. One example of a better colour space is the Lab colour space which also consists of three channels ($C = 3$) and was developed to mimic how humans perceive colour and is a complicated non-linear representation. In this case, the L or intensity aims to match the perception of lightness while a and b represent chrominance values. The a represents the red-green value and b represents the blue-yellow value. There are many such transformations and rather than discuss them in detail here we instead demonstrate how to access them easily via the skimage library.

In `skimage` accessing the transformations to these colour is made simple by using in-built functions. We use a code snippet from `https://scikit-image.org/docs/dev/auto_examples/color_exposure/plot_rgb_to_gray.html` to show how we can convert an RGB image to a grayscale image.

```
import matplotlib.pyplot as plt

from skimage import data
from skimage.color import rgb2gray

original = data.astronaut()
grayscale = rgb2gray(original)
print("The size of the original data is: ", original.shape)
print("The size of the grayscale data is: ", grayscale.shape)

fig, axes = plt.subplots(1, 2, figsize=(8, 4))
ax = axes.ravel()

ax[0].imshow(original)
ax[0].set_title("Original")
ax[1].imshow(grayscale, cmap=plt.cm.gray)
ax[1].set_title("Grayscale")

fig.tight_layout()
plt.show()
```

The above code other than providing the colour image and the grayscale image also provides the following output.

```
The size of the original data is:  (512, 512, 3)
The size of the grayscale data is:  (512, 512)
```

What can be seen here is that the number of channels for the original image is $C = 3$ which is expected for an RGB image. The number of channels for the grayscale image is missing and so implicitly this is $C = 1$ which makes sense because we have removed the colour information and converted to a single channel. There are a range of other colour conversions available such as `rgb2lab` and `lab2rgb`.

12.4 Practical Work

In Python image processing and manipulation is made easy by a number of libraries (numpy, skimage, sklearn). In this practical, we will see how to load an image, manipulate the image, change its colour space, and save it. For this, you will need to download the images available at:
`https://uni-bonn.sciebo.de/s/8rF2TVnzstHal5V`.

12.4.1 Loading Images

1 Use the `imread` function from the `skimage.io` library to load in the image `sb20_rgb.png` into a variable named `rgb`. What is the `type` and `dtype` of `rgb`.

You should have seen that `rgb` has the `dtype` of `uint8`. What is the maximum and minimum value of `rgb`? Based on this knowledge convert the image from $0 \rightarrow 255$ to $0 \rightarrow 1$. Print out the resulting minimum and maximum values of the new `rgb` variable. How would we print these values out for each channel in the RGb image?

Finally, for this exercise, print out the shape of the RGB image. You should see the size $(480, 640, 3)$.

12.4.2 Excess Green Filtering

1 Next we will segment the image based on the excess green equation. This equation is:

$$exG = 2 \times G - R - B, \tag{12.2}$$

where G represents the green channel, R the red channel, and B the blue channel of the RGB image. To calculate this value, you will need to use the $[:, :, channel]$ notation to get the R, G, or B channels respectively. In this case the ":" indicates that you are using all rows and columns in the numpy matrix.

Once you have calculated this value print out the shape of `exG` and the minimum and maximum values. What do you notice about these values?

2 Next, you will create a binary mask based on this `exG` matrix. You should notice that the values of this matrix range between -0.17 and 0.38. Create a new mask to the same size and structure as the `exG` matrix, to do this you will use the `np.zeros_like()` function.

Next we will fill out this binary mask with 0's where the `exG` mask is below a certain threshold and 255 when above or equal to a certain threshold. To

do this you will need to produce a binary assignment within the new mask such that $newmask[exG >= \tau] = 255$. Where τ is a threshold that you will set, I would start at 0.0 and slowly refine this value. To refine this value you will save *newmask* using the `imsave` function from the `skimage.io` library. You will need to do this for each of the thresholds that you have set, the result is purely based on your visual interpretation of the output mask.

12.4.3 Masked Manipulation and IoU

1 In this exercise, we will compare a manually annotated binary mask to that produced by the `exG` threshold approach. First, load the image `sb20_smap.png` which is the semantically annotated ground truth for the the RGB image from the first exercise. Next, use `np.unique()` to print out the unique integer values in the loaded image. You should get the values $[0, 1, 2, 3, 4, 5, 6]$.

2 Now we will mask our RGB image based on the semantic map we just loaded. Create a new RGB image using `rgb2 = rgb.copy()`. Then for all unique values in the semantic map (for loop) apart from 0 we will colour in `rgb2` with random colours. To get the random colours use `np.random.randint(100, 255, [3,])/255.0`, where the three relates to the three channels in the RGB image.

In this for loop if the smap is equal to the current unique value than `rgb2` will be equal to the current random colour. Finally, we will create a new image based on the new `rgb2` image and the previous `rgb` image. Such that,

$$outrgb = \alpha \times rgb + (1 - \alpha) \times rgb2, \tag{12.3}$$

where α will be set to 0.5, but you can play around with this value. Save this created image and see if you can tell the different plants apart based on this segmented image.

3 Comparing masks is an important part of computer vision research, particularly with segmentation masks. To do this we use the intersection over union (IoU) metric,

$$IoU = \frac{M \cap GT}{M \cup GT}, \tag{12.4}$$

where M is your predicted mask and GT is the ground truth mask. $\cap$ indicates the and function (`np.logical_and()`) and the $\cup$ indicates the or function (`np.logical_or()`). Using this function calculate the IoU of the exG mask versus the semantic mask.

12.4.4 Colour Space Conversion and Filtering

1 A problem with RGB images is that they are an additive colour space. This means that the three channels are added together to produce the colour of each pixel. Other colour spaces such as Lab, HSV, or HSL disambiguate the colour information. In this exercise we will use the Lab colour space, where the 'L' channel indicates the illumination in the image, 'a' represents the progression from red to green, and the 'b' is the progression from yellow to blue.

First convert the `rgb` image from the first exercise to the Lab colour space using the `rgb2lab` function from the `skimage.color` library. For your information the 'L' channel ranges from $0 \rightarrow 100$, and the 'a' and 'b' channels range from $-127 \rightarrow 128$. Print out the minimum and maximum values from the different channels in the Lab image you just created to make sure these ranges are correct.

2 Green colour filtering using the 'a' channel of the Lab image. Create a new mask image with the same height and width of the Lab image (`np.zeros()`). Next we will use the threshold method from above to filter out the mask image based on the 'a' channel.

$$Lab\_mask[Lab[:,:,1] <= \tau] = 255 \tag{12.5}$$

where, τ is the threshold you will alter. Save the image and alter the threshold until you visually feel like you have the best mask. Start with a threshold of 0 and move it lower from there.

3 Compare your Lab based mask to the annotated semantic mask using the IoU metric. Which one is better, exG or Lab?

12.4.5 Edge Detection

In this final exercise we will perform edge detection using a traditional kernel approach. We will do this by using the sobel kernels:

$$f_v = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \qquad f_h = \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

where f_v looks for vertical lines and f_h looks for horizontal lines in the image.

1 Load in the image `plaid.jpg` and convert it to a grayscale image. Create two matrices, one for the horizontal edge results and one for the vertical edge results. These two matrices should have the same shape as the grayscale input image after conversion.

Using two embedded `for loops` create a sliding window that scans the input plaid image. Each iteration should sum the convolution of the two kernels (f_v and f_h) on the sliding window of the image. The summed result will go into the appropriate result matrix at the correct index (depending on the for loop locations). Remember that you have a filter size of 3×3 so you will need to either start and finish at the right location or pad the original image.

Once you have created your horizontal and vertical edge maps you need to calculate the magnitude:

$$mag = \sqrt{h_{map}^2 + v_{map}^2} \tag{12.6}$$

where h_{map} and v_{map} are the horizontal and vertical edge maps calculated from the sliding window and kernel operations.

2 Now you need to plot these results, for this you will need `matplotlib.pyplot`. Plot the resulting h_{map}, v_{map}, and mag for this image.

3 Finally, we will see if our technique is similar to an inbuilt library for the same function. You will need to import the `sobel` function from the `skimage.filters` library. Once you have done this it's as simple as running this function with the grayscale image as the input. Once you have calculated this map, compare your created version with this output and plot both using `matplotlib`.

Chapter 13

Data Structures

Data structures are a critical part when doing programming. They provide a way to store and organise your data. Ideally, they should be designed so that you can perform operations on the easily and efficiently. You've already been introduced to some data structures available in Python.

In Python, `lists` and `dicts` are two frequently used data structures. The previous chapters have shown us several examples of how these can be used to store and access a range of data. Also, we've seen the idea of `classes` which is another way to store data and to also provide greater flexibility in how we can access, change and delete that data.

In this chapter, we will briefly discuss some of the tradeoffs between using things like `dicts` vs a `class` for storing our data. Furthermore, we'll discuss some other common data structures, like queues, that we haven't looked at previously.

13.1 Data Holder: list vs dict vs class

It is possible to hold data in many structures but commonly we use either a `dict` or `class` to do this. We would normally use a `dict` if only data is stored and all operations on the data are performed externally. Often a `dict` is preferred when the data has associated names and there are different values that we'd like to store. Let's consider the example of storing some an image and its filename.

We could store an image and its filename in either a `list` or a `dict`. If we stored this information in a list `list` we would have to remember the order of what we stored because the 0-th (or first) index could the image or the filename as shown below.

```
my_data = []
my_data.append(img)
my_data.append(fname)
```

The downside of using a numbered index is that it's ambiguous, particularly to other programmers, and can be easily forgotten. By contrast a dictionary can be indexed via a keyword which can be self explanatory. An example would be to store the filename using the keyword `filename` and the image using the keyword `image` as shown below.

```
my_data = {}
my_data['image'] = img
my_data['filename'] = fname
```

The `dict` has the advantage of being self-explanatory in terms of how we reference the information we want to access. Therefore, for easily named data we would normally prefer a `dict` over a `list`. This raises the question, when do we want to increase the complexity of our data structure and use a `class`?

Generally, if the data should be manipulated in addition to being stored then we should consider a `class`. To make this tradeoff more easy to understand let's consider an example. If we're holding data like information for how well a student performed in a test, their accuracy. We would likely use a class so that we could accumulate the statistics and then do the final operation all in a self-contained manner. An example of this is shown below.

```
class Accuracy:
  def __init__(self):
    self.num_total = 0
    self.num_correct = 0

  def add_to_value(self, num_samples, num_correct):
    self.num_total += num_samples
    self.num_correct += num_correct

  def final_accuracy(self):
    return self.num_correct / self.num_total
```

We do this so it is clear and easy to understand how to accumulate the data and what the final operation to calculate the final score is. The `class` could have further additions to make it easier to accumulate, but we leave this to the interested reader to implement.

The above are some examples of when to use a `dict` over a `class`. Now we consider two examples that highlight the flexibility of `lists` in terms of a queue and how to implement a circular list.

13.2 FIFO Queue

In programming we sometimes want to store things which are then processed later. This can be considered to be something of a ToDo list or a **queue** of tasks. Frequently, the order in which we put things into this queue is important and we want to process the oldest item first. Such a queue is called a first-in-first-out (FIFO) queue and occurs in many interactions. A practical example would be queuing at a shop.

The idea of a FIFO queue can be implemented in Python by using a `list`. When we add elements to a `list` we `append` them which means they get put onto the back or the end of the queue. We show this idea by adding 4 items to a list in the code below.

```
my_list = []
print(my_list)
my_list.append('item1')
print(my_list)
my_list.append('item2')
print(my_list)
my_list.append('item3')
print(my_list)
my_list.append('item4')
print(my_list)
```

The result of the above code is that we start with an empty list (`[]`) and then gradually build it up. But, `'item1'` is always at the start or head of the list.

```
[]
['item1']
['item1', 'item2']
['item1', 'item2', 'item3']
['item1', 'item2', 'item3', 'item4']
```

For lists in Python we can remove and get the value for the head (first item) of the list. This is achieved by calling `pop` as shown in the code below.

```
my_list = []
my_list.append('item1')
my_list.append('item2')
my_list.append('item3')
print(my_list)
curr_item = my_list.pop(0)
print(curr_item)
my_list.append('item4')
print(my_list)
```

Where the above code would lead the following result where `'item1'` has been removed and returned to us.

```
['item1', 'item2', 'item3']
item1
['item2', 'item3', 'item4']
```

This is the set of operations that enables us to use a `list` as a FIFO queue. The process is as follows:

- **Adding** to the queue is achieved by using `append`.
- **Retrieving** the most relevant (oldest) item from the queue is achieved by using `pop`.

13.3 Circular List

There is often data that has a repeating pattern. Let's take an everyday example of this using the days of the week which is a repeating pattern that we could store in a list as follows.

```
days_of_the_week = ['Sunday', 'Monday', 'Tuesday', 'Wednesday
    ', 'Thursday', 'Friday', 'Saturday']
```

As there are seven (7) days of the week we could index into this list using a number from $0-6$; this is because indexing in Python starts at 0.

A use for the above list might be to work out on which day of the week something will happen. So someone could tell us that on the 3rd day of the week they will go shopping and we could then index this list to work out which day that is, it's `'Tuesday'`. However, what happens when the index is beyond the range of numbers $0-6$? At this point we would need to start back over at the beginning of the list. How can we do this programmatically?

One solution to the above would be to recognise that we've gone beyond the length of the list and do something. For instance, if the value is above 8 days in the future we could just subtract 7 (the length of the list) and we would get a valid index. Because we could get very large numbers like 25 days in the future we can do this as a while loop as shown below.

```
days_of_the_week = ['Sunday', 'Monday', 'Tuesday', 'Wednesday
    ', 'Thursday', 'Friday', 'Saturday']
index = 25
c_index = index
while c_index >= len(days_of_the_week):
    c_index -= len(days_of_the_week)
print(c_index, days_of_the_week[c_index])
```

This works as expected and gives us that the day would the Thursday. It also provides us with an idea about how to implement a circular loop, however,

in doing so we've employed a while loop and have to keep processing until that finishes.

A simpler solution to the above problem is to recognise that we can use the modulo operation. The modulo operation gives us the remainder of a division which is equivalent to the operation we were implementing using the `while` loop. Concretely, we keep the modulo of the index and use this value and we can apply this to any value even those within the valid range. The code is shown below.

```
1 days_of_the_week = ['Sunday', 'Monday', 'Tuesday', 'Wednesday
    ', 'Thursday', 'Friday', 'Saturday']
2 index = 25
3 c_index = index%len(days_of_the_week)
4 print(c_index, days_of_the_week[c_index])
```

The above code gives us exactly the same answer as before without the need of a `while` loop. This is an example of how considering the structure of our data and the available in-built operations can help us solve the problem at hand. We can now use the same structure to extend this to handle the months in the year as well as other data with a repeated or circular structure.

13.4 Practical Work

13.4.1 List and Dictionary Refresh

1. We have used the FIFO queuing method in a previous practical without actually knowing what it was. Play around with lists and dictionaries popping and adding values to each. Can you do this with a tuple? If not, how can you get around this?

13.4.2 Sets

Sets are immutable storage classes in Python. They store a unique set of values (i.e. duplicate values are not stored).

```
lst = [1,1,2,3,4,5,3,1,2]
st = set(last)
print(st)

>>> {1,2,3,4,5}
```

1. Try using some of the mutable commands on a set, what do you find?

13.4.3 Circular List as Class

1. We will create a class that stores the days of the week and the current day (as an index to the days of the week). As input to your class you will give it the current day. Use, `__add__()` (adding a number) and `__iadd__()` (add and equate) to add to the current day. Any method that has the double underscore we call dunder methods, we have now seen a number of these methods throughout the course. You can add any value to your object but you must be able to calculate the circular list of days.

Also, in this class we will overwrite the `str()` method or implement `__str__()` method (or both) so that we can write out what the current day is as a string.

2. Use the `__eq__()` method to tell if two days of the week objects are on the same day. This method returns a boolean.

3. What other dunder methods could you implement for this class?

Chapter 14

More Operators

Much of this book has concentrated on fundamental elements of programming from variables, conditional statements, and functions through to modules and classes. But, advanced programming languages, like Python, provide a range of operators and commands to facilitate programming. Also, Python, like all languages, has some limitations. For instance, nested loops (e.g. double for loops) are often inefficient in Python and so we present some ways to consider how to simplify/optimise the code in such situations. The intent of this chapter is not to provide an exhaustive guide into extra operations nor to dictate how to optimise all code. Rather it provides a quick deeper insight into some of the available operations and ideas for writing programs with the hope that interested readers can pursue these and explore the other options available within Python.

14.1 Some More Operators in Python

Python provide a range of operators and commands to facilitate programming. The benefit of several of these commands is to reduce the amount of code written with the extra benefit of also improving readability. This is because less code has to be read, provided the people reading the code understand the operators being used. In this section, we cover some of the main operators in Python that lead to more compact and readable code.

14.1.1 ternary operators

The ternary operator is a quick way of defining the content of a variable based on the conditional `if else` structure. Let's take a practical example of this for putting a threshold on the maximum difference between two values.

Normally we would write such code as something like the following.

```
max_abs_diff = 5
x = 20
y = 30
abs_diff = abs(x-y) # absolute difference
if abs_diff > max_abs_diff:
    abs_diff = max_abs_diff
```

Although the above code for the conditional statement is completely correct, this type of conditional statement is so common that an easier way to write it is provided. This easier way to write it is called a ternary and it allows us to write if statements, like the one above, on a single line.

The ternary operator has the following pattern.

```
<var_name> = <value1> if <conditional> else <value2>
```

Both `value1` and `value2` can be any valid variable or pre-defined constant. For our above example we could write our ternary-based solution as the following.

```
abs_diff = abs_diff if abs_diff < max_abs_diff else
   max_abs_diff
```

14.1.2 Zip and Enumerate

Iterating over data structure is something that frequently occurs. Furthermore, a common situation is to want to iterate over a data structure and to simultaneously count how far through the data structure we are. Let's take a practical example of this using numerical data.

If we have two lists, of the same length, called `x` and `y`.

```
y = [0.4,0.5,0.8,0.2,0.8,1.2]
x = [1.2,2.1,3.5,4.4,5.9,6.7]
```

We could jointly iterate over the two lists by using their length and write following code.

```
N = len(x)
for idx in range(N):
    x_val = x[idx]
    y_val = y[idx]
    <do_something>
```

Instead of having to index into the two lists and then `<do_something>` Python provides an operator called zip to combine the two lists.

Zip aggregates or combines iterators (e.g. list) and has the following pattern.

```
zip(a,b,c, ...)
```

It provides something that can be iterated over by combining all of the variables (a, b, c, ...) provided the variables can also be iterated over. To use this pattern with our code snippet from above we could instead write the following.

```
N = len(x)
for x_val, y_val in zip(x,y):
    <do_something>
```

It can be seen that we no longer need to use the index into the lists to get the values and instead we directly iterate over to get the values. This leads to short and easier to write code. Note that the length that will be iterated over will be the length of the shortest iterator (e.g. list).

Enumerate enables us to count as we iterate through something (e.g. lists). It has the following pattern.

```
enumerate(data_buffer, <start_index>)
```

This can be useful in several contexts and for our example we choose the option of reading lines from file. Normally, to count how many lines we have currently read from a file we would have to define and maintain an extra variable as shown below.

```
fp = open("example1_input_file.txt")
data_buffer = fp.readlines()
fp.close()
count = 0
for v in data_buffer:
    print(count, v.strip())
    count += 1
```

Instead, we can use enumerate and remove the need to maintain and to increment this variable.

```
fp = open("example1_input_file.txt")
data_buffer = fp.readlines()
fp.close()
for i, v in enumerate(data_buffer, 0):
    print(i, v.strip())
```

14.2 Generating Lists and Dictionaries

Generating lists and dictionaries is a common programmatic operation. It is important for pre-initialisation of lista and in the case of dictionaries to generate the dictionary values from some other data structure(s). Below we

highlight some useful common operators that can be used to make these two tasks easier.

14.2.1 List comprehension

List comprehension allows us to generate a new list based on, usually, a simple expression. This enables us to do things like replace a for loop structure when initialising a list and put this onto a single line. What this can enable is code that is quicker to write and also more readable as only one line of code needs to be read. The template of what we write is below.

```
new_list = [an expression for a member in an iterable]
```

We make this concrete by considering that we make a list to represent the 3 times table. Up until now we could make this by doing a for loop as shown below.

```
times_table_3 = []
for i in range(11):
  times_table_3.append(3*i)
print(times_table_3)
```

This would lead to our list having the values

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

List comprehension lets us write what was previously 4 lines of as a single line of code.

```
times_table_3 = [3*i for i in range(11)]
```

14.2.2 Generating Dictionaries

Dictionaries are a common data structure and when we initialise them we frequently use loops. In particular, we will think along the lines of a dictionary whose entries can be a list of values where we loop over some data and progressively add to this list of values. Such a loop could look something like the code below.

```
my_dict = {}
for curr_data, curr_key in some_list:
  if my_dict.has_key(curr_key): # Exists, so append it
    my_dict[curr_key].append(curr_data)
  else: # Doesn't exist yet so initialise it
    my_dict[curr_key] = [curr_data]
```

Although the above code is correct, it is actually inefficient. This is because the `if/else` statement is slower than another construct in Python and that construct is the `try/except` statement. We can replace the if/else with a try/except statement, which is equally clear and correct but considerably faster, see below.

```
my_dict = {}
for curr_data, curr_key in some_list:
  try: # Exists, so append it
    my_dict[curr_key].append(curr_data)
  except(KeyError): # Doesn't exist yet as shown by KeyError
    my_dict[curr_key] = [curr_data]
```

14.3 Errors and Error handling

We frequently experience errors when running code, especially during the early stages of writing/testing our program. Typically, these indicate that we have incorrectly used some feature and the error messages often provide us with helpful information to identify the problem.

```
print("hello world!')
```

Running the above buggy code will lead to the following output:

```
1   File "<stdin>", line 1
2     print("hello world!')
3                          ^
4 SyntaxError: EOL while scanning string literal
```

In line 4, we get the information that the error encountered is of type `SyntaxError`. As the name suggests, this tells us that there is a problem with the syntax of our code, i.e., it is "grammatically" incorrect. This is followed by an error message saying that the end of the line was reached while reading a string. Lines 1 to 3 show additional information about where the error occurred in our original code. All of this information helps us to identify the bug, which was the incorrect use of `'` in an unsuccessful attempt to close a string that was opened by `"`.

14.3.1 Raising Errors

We have the option to `raise`, or highlight that there is, an error in our own program. This makes sure that if certain requirements are not met we do not let our program continue for further execution. This can be done by raising an `Exception`:

```
raise Exception("A helpful error message.")
```

which will lead to the following output:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: A helpful error message.
```

There are many different built-in Exception types in Python, which should be used in appropriate cases. Some common examples are `TypeError`, `ValueError` and `NotImplementedError`. In the following example, we implement the `raise` of a `ValueError` to stop our program in case a given list does not contain a specified number `n`:

```
n = 10
my_list = [1, 2, 3, 4, 5]
if n not in my_list:
    raise ValueError(f"my_list needs to contain the number {n
    }, however, that is not the case: {my_list}")
```

That way, we get the following error, as intended:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: my_list needs to contain the number 10, however,
    that is not the case: [1, 2, 3, 4, 5]
```

We can further implement our own Exception type. To do so, we create a class that inherits from the original `Exception` class, which typically is the parent for the built-in types as well:

```
class MyError(Exception):
    pass
```

14.3.2 Handling Errors

Python does have the functionality to handle errors explicitly, that is, without stopping our program. For this purpose, we have previously made use of the `try/except` statement for a very specific case in Section 14.2.2. In general, it can be used to try a piece of code, and in case of failure, run different code. However, **this can often be dangerous**, as it masks any errors in that place.

```
x = 999
y = 0
try:
    z = x / y
except:
    print("hello there!")
```

In the above code we try to divide `x` by `y`, which is not possible, as we set `y=0`. However, no error occurs, and all we get is the following, in this case totally meaningless output:

```
hello there!
```

The `try/except` statement can be a great feature, if applied in useful ways, which is clearly not the case for the above example. In case of unexpected errors (e.g. errors which were not considered when writing your code) the use of `try/except` can have severe consequences. In the case of the example above, the calculation of `z` has never been executed, which could falsify any calculations down the line and ruin your experiment. **Therefore, never blindly except any errors just to make your code run!**

There are ways to improve the safety of error handling. An example of this would be to parse the Exception and print its message. Modifying the above code to:

```
try:
    z = x / y
except Exception as e:
    print(f"Warning: excepted an error: {e}")
    print("hello there!")
```

additionally provides the error message in the output:

```
Warning: excepted an error: division by zero
hello there!
```

That way, we have the information that something went wrong, including some details. At the same time, our program does not stop and the alternative part gets excecuted. However, in case of our example, this is **still very unsafe**, as a user could easily overlook the warning, the alternative code is still meaningless, and `z` is still not calculated at all!

A further option is to explicitly except only certain, previously considered error types, in combination with suitable alternative code:

```
try:
    z = x / y
except ZeroDivisionError as e:
    y = 0.000000001
    z = x / y
```

Now, we only except the case of accidentally dividing by 0 and provide an alternative that we assume is useful for our use case, dividing by a very small number close to zero instead. At the same time, if any other error occurs, execution is stopped as usual. For example, if a user accidentally defines `y='0'`, as a string, we get the following error:

```
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

14.4 Practical Work

14.4.1 Ternary Operators

Ternary operators are a method of putting an `if else` statement on a single line.

1. Turn the following code into a ternary operator:

```
x = 4
if x >= 4:
  y = 'this'
else:
  y = 'that'

print(y)
```

14.4.2 Zip and Enumerate

Zip

In our previous work if we wanted to print values from two different lists of the same length we would have iterated over the indexes like so:

```
import numpy as np
x = np.random.randint(0,10,size=[1,20])
y = np.random.randint(10,20,size=[1,20])

for i in x.shape[1]:
  print(x[i], y[i])
```

However, there is a way to accumulate the values in a single step: `zip`.

```
for <output variables> in zip(<input variables>):
  print(<output variables>)
```

In this case we join all the variables and output their current value based on the index directly to the output variables.

1. Define three numpy vectors of the same length `(10,)`. Use the zip function to put them all together in a for loop and print their values to the screen. What happens if we use a numpy matrix?

14.4.3 Enumerate

Sometimes when iterating over a storage variable we need the index and the value. In this case we can use the enumerate function.

```
for i, val in enumerate(vector):
  print(i, val)
```

1. Using one of the vectors from above use the enumerate function to output the index and the current value. When do you think this would be helpful?

14.4.4 List Comprehension

List comprehension is a way to simplify list creation. Much like ternary operators it puts everything on a single line.

1. Convert the following code into list comprehension.

```
lst0 = []
for x in range(100):
  lst0.append(x**2)
```

14.4.5 Dictionary Generation

1. Let's create a dictionary from two lists using the zip function. The two lists are as follows:

```
lst0 = ['dog', 'cat', 'this', 'that', 'dog', 'cat']
lst1 = [1, 1212, 3.14, 'house', 2, 42]
```

For each key (`lst0`) we will store the corresponding value (`lst1`) as a list (`append`). Don't forget to check if the key already exists.

14.4.6 Errors and Error handling

Run the following code, which will lead to an error, as line 2 expects a list, but we mistakenly provide a tuple:

```
my_data = (1,2,3,4,5)
my_data.append(6)
```

What type of error do you encounter? Use the `try/except statement` to keep your program running despite the error, but still display the original error message. To make this safe, except only the specific occuring Exception type, and come up with some alternative solution for that case. In your solution, make use of Python's built-in function `isinstance(object, type)`, to check if my_data is a tuple.

Chapter 15

Recursion

So far we discussed ideas such as function, conditional, loops, and even classes. However, there is another important way of doing programming which is called recursion. Also, the idea of memory and how we make room for variables has not been covered. Therefore, in this chapter we briefly describe these two ideas and how they can be used in Python.

15.1 Recursion

Recursion is a concept in programming where a function will then call itself, usually iteratively. This means that the result of a function is dependent on a further call to the function itself. In fact, many problems can be expressed as recursion so let's make this concrete with an example by considering how to count down to 0.

If we want to countdown from a positive number all the way down to zero we could use recursion. To do this, we would pass to the function the current count and subtract one. We would then call the same function again but with the decremented (reduced) value. This would look something like the following.

```
def count_down(n):
    print(n)
    if n==0:
        print("Blast off!!")
        return
    else:
         count_down(n-1)
count_down(5)
```

The previous code would provide as output the following.

```
5
4
3
2
1
0
Blast off!!
```

The key element in the above code is line 7 where in the `count_down` function we make a further call to `count_down` but we have now changed the input parameter. Importantly we have also provided a termination criteria. In this example we choose to terminate when `n==0`, on line 3. Without this the code would continue forever. This is conceptually similar to a while loop, where you need to ensure that the function/program can exist from the loop, or in this case the recursive calls. But, there is a large difference between while loops and recursion which we will explain below.

What happens if we keep making recursive calls for a very long time? Let's try this by changing our code to be the following.

```
def count_down(n):
    print(n)
    count_down(n-1)

count_down(5)
```

The code will keep running for a while but then at some point it will give you something like the following message.

```
RecursionError: maximum recursion depth exceeded while
   calling a Python object
```

Why is there a maximum recursion depth?

An issue with recursion is that generally recursion will use more memory. First, when a function is called the state of the current function is added (stored) to something called a "stack". This "stack" stores the current state of the program which includes the state of the current function along with the variables, in memory. Second, when we do recursion we are calling the function again which means that the stack still has the state of teh previous function. Because we now call the function again, new memory is allocated for the currently called function so that it can instantiate its own variables in memory. In the case of recursion, the above two steps repeatedly occur taking more memory each time the function is called. The consequence is that we will have to store, in the "stack", hundreds or thousands of version of the function we are calling. Furthermore, this memory can only be freed until the function is finished and is a major issue when employing recursion.

By contrast, when we use a `for` loop we simply have to store the state of the current variables until the end of the loop and not the full function.

15.2 Stack: as a data structure

We briefly described what a stack is above, but it could also be a useful data structure so we describe this in more detail. A stack is similar to the FIFO Queue data structure but rather than first in first out it becomes last in first out (LIFO). An example of this could be putting bricks on top of each other, or stacking bricks. To take a brick off the stack you take the last brick that you put on and repeat this process until they have all been removed. An example is provided in Figure 15.1 where the stack starts with 3 items (A, B, and C).

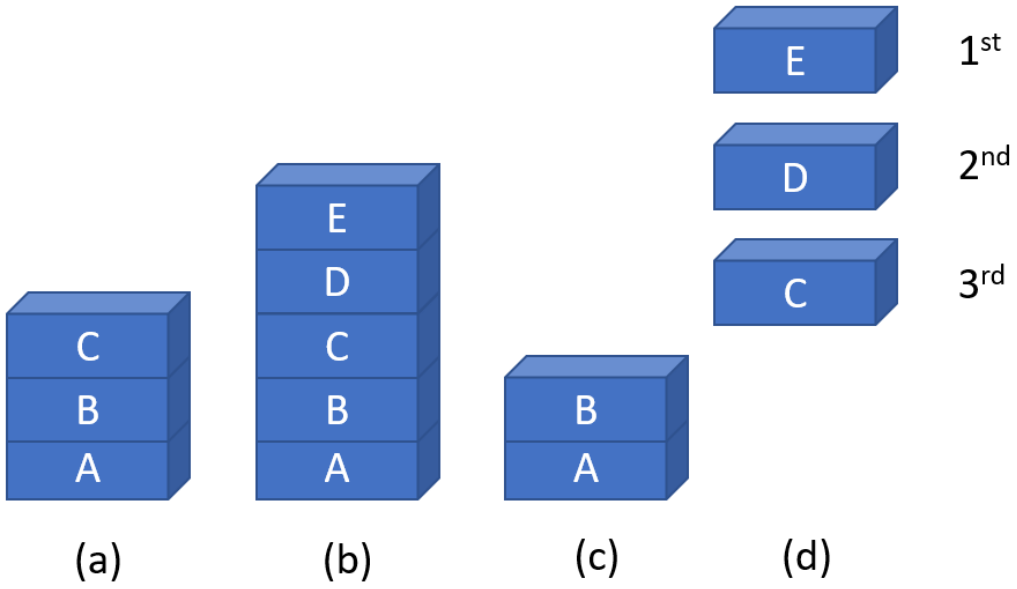

Figure 15.1: An example of a stack. In (a) the stack starts with items A, B, and C. Two more, D and then E, are added in (b). In (c) and (d) the idea of removing 3 items is given where first the top one E is removed, then D followed by C.

Given the above it becomes clear that recursion is a programming principle that should be used with care. In particular, careful thought should be given as to how the recursive process can be exited and also that recursion does not go too many levels this. The second point is important because of the memory implications due to maintaining the stack.

To better understand what a stack or LIFO is let's explore how this would work by making a class. In the code below, we will make a class called a stack that will add elements and remove elements.

```
class Stack():
    def __init__(self):
        self.stack = []

    def add_element(self, elem):
        self.stack.append(elem)

    def remove_element(self):
        return self.stack.pop(-1)

my_stack = Stack()
my_stack.add_element(1)
my_stack.add_element(2)
my_stack.add_element(3)
print(my_stack.remove_element())
```

The above code will print the number `3`, this is because it is last element entered and so is the first one removed.

To provide a contrast below we provide example code for a FIFO class.

```
class FIFO():
    def __init__(self):
        self.list = []

    def add_element(self, elem):
        self.list.append(elem)

    def remove_element(self):
        return self.list.pop(0)

my_fifo = FIFO()
my_fifo.add_element(1)
my_fifo.add_element(2)
my_fifo.add_element(3)
print(my_fifo.remove_element())
```

In contrast the above code will return `1` as this was the first element entered.

For both the FIFO and Stack code we provided above, what are some of the potential issues? As an example, when we call `remove_element` for both classes there is no check to ensure that the stack or list has elements to remove. It is left to the reader to consider how the code could be changed to ensure this is does not lead to an error.

15.3 Practical Work

15.3.1 Recursion - Merge Sort

Merge sort is a common divide-and-conquer algorithm. A divide-and-conquer algorithm splits a large task into many smaller tasks, simplifying the task considerably. Using recursion it sorts the values in a vector into ascending or descending order. This algorithm has two functions `merge`, which merges two vectors and assigns them in their appropriate order. Then the primary `merge sort` which is the recursive part of the algorithm.

The pseudo-code for this algorithm to provide the ascending order is as follows:

```
def merge(x, y)
  assign i,j,k as zero
  assign o as vector to len(x) + len(y)
  while i<len(x) and j<len(y)
    if x[i] < y[j]
      o[k] = x[i]
      k++
      i++
    else
      o[k] = y[j]
      k++
      j++
  if i<len(x)
    assign the remaining x to o
  else
    assign the remaining y to o
  return o
```

```
def merge_sort(x)
  # base case return
  if len(x) == 1
    return x
  # split the vector
  l = round(len(x)/2.)
  top = x[:l]
  bot = x[l:]
  # recursion
  t = merge_sort(top)
  b = merge_sort(bot)
  o = merge(t,b)
  return o
```

Implement this algorithm in Pythoncode using functions. How would we do this in a class?

Bibliography