

A green snake with a textured, scaly pattern is coiled around a large, light-colored rock. The background is a dark green, textured surface.

EXERCISES & PROJECTS INCLUDED

Slither Into Python

AN INTRODUCTION TO PYTHON
PROGRAMMING FOR BEGINNERS

- All the basics of Python
- Object oriented programming
- Data structures and algorithms

Contents

.....	1
Chapter 1	7
Prerequisites.....	7
1.1 Learning outcomes and who this book is for	7
1.2 A bit about Python	9
1.3 Our first program	10
Chapter 2 - Integers, Operators and Syntax	11
2.1 Integers.....	11
2.2 Operator precedence.....	13
2.3 Syntax.....	13
2.4 Exercises.....	15
Chapter 3 - Variables, Floats, Input and Printing	16
3.1 - Variable Assignment	16
3.2 - Floating-point types.....	20
3.3 - Getting user input.....	21
3.4 - The print function	23
3.5 - Exercises.....	24
Chapter 4 - Control Flow	26
4.1 - Booleans.....	26
4.2 - The <i>if</i> Statement.....	28
4.3 - The <i>if-else</i> statement.....	30
4.4 - The <i>elif</i> statement	32
4.5 - Nested <i>if</i> statements	33
4.6 - Exercises.....	33
Chapter 5 - Looping & Iterations	35
5.1 - <i>while</i> loops.....	35
5.2 - Looping patterns	38
5.3 - <i>break</i> and <i>continue</i>	40
5.4 - <i>while</i> loop examples.....	42
5.5 - Exercises.....	43
Chapter 6 - Strings	45
6.1 - Introduction to strings	45

6.2 - String functions.....	46
6.3 - String methods	47
6.4 - String indexing	49
6.5 - Immutability	51
6.6 - String slicing	52
6.7 - String operators	54
6.8 - Printing & formatting.....	55
6.9 - Exercises.....	58
Chapter 7 - Linear Search	61
7.1 - What is linear search?	61
7.2 Linear search using while	61
7.3 - Examples	62
7.4 - Exercises.....	63
Chapter 8 - Lists.....	67
8.1 - List literals	67
8.2 - List methods and functions	68
8.3 - List indexing	70
8.4 - References & Mutability	70
8.5 - List operations.....	74
8.6 - List slicing	74
8.7 - Exercises.....	75
Chapter 9 - Basic Sorting Algorithms	79
9.1 - What is sorting?	79
9.2 - Selection sort.....	79
9.3 - Insertion sort	82
9.4 - Comparison of Selection sort and Insertion sort.....	84
9.5 - Exercises.....	87
Input, File & Text Processing.....	88
10.1 - What is file & text processing?	88
10.2 - The Sys module.....	88
10.3 - Reading from standard input.....	90
10.4 - Writing to standard output	93
10.5 - Standard error.....	95

10.6 - Opening & reading files	95
10.7 - Writing to files & closing files.....	96
10.8 - Putting is all together	98
10.9 - Exercises.....	99
Chapter 11 - for loops & Dictionaries.....	103
11.1 - for loops.....	103
11.2 - Dictionaries.....	105
11.3 - Dictionary accessing.....	106
11.4 - Adding and removing entries	107
11.5 - Dictionary operators & methods.....	108
11.6 - Sorting dictionaries	110
11.7 - Exercises.....	111
Chapter 12 - Functions & Modules.....	115
12.1 - What are functions?	115
12.2 - Defining our own functions.....	115
12.3 - Arguments & parameters.....	116
12.4 - Variable Scope	119
12.5 - Default mutable argument trap.....	119
12.6 - What are modules?	121
12.7 - Creating our own modules.....	122
12.8 - Exercises.....	123
Chapter 13 - Binary Search.....	127
13.1 - Number guessing game	127
13.2 - Binary Search.....	128
13.3 - Implementing Binary Search	130
13.4 - Improving our number guessing game.....	131
13.5 Analysis of Binary Search.....	131
13.6 - Exercises.....	133
Chapter 14 - Error Handling.....	135
14.1 - Raising Exceptions.....	135
14.2 - Assertions	136
14.3 - try & except.....	137
14.4 - finally	141

14.5 - Exercises.....	142
Chapter 15 - More on Data Types.....	145
15.1 - Tuples.....	145
15.2 - Sets.....	147
15.3 - Lists & list comprehensions	150
15.4 - Exercises.....	151
Chapter 16 - Recursion & Quicksort.....	154
16.1 - Recursion	154
16.2 - Memoization	156
16.3 - Quicksort part 1	158
16.4 - Quicksort part 2	160
16.5 - Exercises.....	160
Chapter 17 - Object Oriented Programming (OOP)	162
17.1 - Classes & Objects	162
17.2 - Defining a new type	163
17.3 - The self variable	164
17.4 - Exercises.....	166
Chapter 18 - OOP: Special Methods and Operator Overloading.....	169
18.1 - The <code>__init__()</code> method	169
18.2 - The <code>__str__()</code> method	170
18.3 - The <code>__len__()</code> method.....	171
18.4 - Operator overloading	172
18.5 - Exercises.....	175
Chapter 19 - OOP: Method Types & Access Modifiers	177
19.1 - What are instance methods?	177
19.2 - What are class methods?.....	177
19.3 - What are static methods?.....	180
19.4 - Public, Private & Protected attributes.....	181
19.5 - Exercises.....	182
Chapter 20 - OOP Inheritance	186
20.1 - What is inheritance?	186
20.2 - Parent & child classes	186
20.3 - Multiple inheritance	191

20.4 - Exercises.....	192
Chapter 21 - Basic Data Structures	195
21.1 - The Stack.....	195
21.2 - The Queue	199
21.3 - Examples	201
21.4 - Exercises.....	203
Chapter 22 - More Advanced Data Structures	206
22.1 - Linked Lists	206
22.2 - Binary Search Trees (BSTs).....	210
22.3 - Exercises.....	218

Chapter 1

Prerequisites

There are only two things you need to have done before reading this book: have Python installed on your computer. I'm not going to go through the process of doing that, so I've left some links below on how to install it on whatever operating system you're using. We'll be using Python 3.7 throughout this book; however you might already have Python installed if you're on Mac or Linux. I'll also be developing on Windows 10 but that shouldn't make a difference.

Secondly you should have a text editor installed. I recommend Sublime Text as it will provide everything we need to get started. A link to download Sublime Text is also linked below.

Python for Windows: [How to setup Python on Windows 10](#)

Python for MacOS: [How to set up Python on MacOS](#)

Python for Linux (Ubuntu): [How to set up Python on Ubuntu](#)

Sublime Text 3: [Install Sublime Text 3](#)

I also highly recommend becoming familiar with the command line. This is going to be essential if you want to be productive. I know I've said I'll be developing on Windows, but I also recommend you install some Linux distribution (Ubuntu is usually a good choice for beginners). You can install this alongside Windows if that's what you've got installed. All great developers know the Linux command line inside and out and if you work in this industry, you'll be expected to know it. Below are some links on how to install Ubuntu alongside Windows and an introduction to both the Linux and Windows command lines.

How to install Linux alongside Windows: [Dual-boot ubuntu and Windows](#)

Introduction to the Windows Shell: [Intro to Windows Shell](#)

Introduction to the Linux Command Line: [Intro to Linux Command Line](#)

1.1 Learning outcomes and who this book is for

Slither into Python is freely available online and aimed at anyone who wants to learn to program or wishes to learn a thing or two about computer science and has little to no knowledge about either.

No prior programming experience or computer science background is necessary. Unlike any other Python resources I have found (not that they don't exist), they don't explain important computer science concepts such as memory or "how computers work" which I believe is essential to becoming a good programmer (having some basic computer science knowledge can often help you get out of odd situations when programming). In this book I will cover the fundamentals of the Python programming language and introduce these important computer science concepts and theories.

Put simply, *coding* or *computer programming* is about making computers do what you want, Computer Science is about *how* computers do it. Each go hand-in-hand, help you learn faster, improve your overall understanding of a language and makes you a better programmer. This book aims to do exactly that through Python.

Another important skill all great programmers have is *problem solving*. Being able to think about a problem from a different perspective, break it down into an easier more manageable problem or re-frame the problem is unbelievably important.

Thinking computationally is also important (Probably most important. Most people fail because they don't learn to think computationally). Consider making a cup of tea. It's something we do every day without giving it much thought. However, that won't work well for us when it comes to writing computer programs. We need to break problems down into a set of well-defined steps.

```
1. Making a cup of tea:
2. - Boil the kettle
3. - Take a cup from the cupboard
4. - Take a teabag from the cupboard
5. - Put the teabag in the cup
6. - Pour water into the cup
7. - If want sugar:
8.   - Take sugar from cupboard
9.   - Put sugar into cup
10. - Stir the tea
11. - Remove the teabag
12. - If want milk:
13.   - Take milk from fridge
14.   - Pour milk into cup
15. - Stir the tea
16. - Drink tea
```

(Please don't email me about my tea making method!).

I will also explain everything very simply in the beginning of the book and gradually become more technical as to not overwhelm you in the beginning (I found this worked well for me when I was learning to program).

I hope that by the end of this book you will have a solid understanding of the Python programming language, a firm understanding of important computer science

concepts, the ability to put problem solving techniques to good use and most importantly, think computationally.

1.2 A bit about Python

Python is an interpreted, high-level, general-purpose programming language. Now you might be wondering what all that means, so let me explain. An interpreted language is a language in which we don't need to compile a program before we run it. Compiling is the process of translating the code we write into machine code (A language the computer can understand). With interpreted languages we essentially generate the machine code as our program runs (at run-time). This is done with the help of an interpreter. The Python interpreter is just called `python`.

High-level means we are abstracted away from the details of the computer. For the purposes of explanation, a low-language like Assembly is used in specialised applications such as operating systems (In fact, the Apollo 11 Guidance Computer was written in assembly and the source code can be found here: [Apollo 11 Source Code](#)). Below is a snippet of this low-level assembly language which we will re-write in Python in the following section.

```
1. SECTION .DATA
2.     hello:      db 'Hello world!', 10
3.     helloLen:   equ $-hello
4.
5. SECTION .TEXT
6.     GLOBAL _start
7.
8. _start:
9.     mov eax,4
10.    mov ebx,1
11.    mov ecx,hello
12.    mov edx,helloLen
13.    int 80h
14.
15.    mov eax,1
16.    mov ebx,0
17.    int 80h
```

Python is being widely adopted by universities all around the world as the introductory language taught to first year computer science students due to its simplicity. In saying that, it may be an easy to understand language and easy to write language, but it is also extremely powerful. It is used everywhere from automating everyday computing tasks to data analytics and artificial intelligence.

Python has a huge community behind it. If you get stuck on a problem or can't make sense of an error message, I can guarantee you'll find your answer in a matter of minutes.

Python also has a ridiculous number of open source libraries, frameworks and modules available to do whatever you want to do. This makes developing your applications even more straightforward.

To top it all off, if you're planning on making a career out of this, the average salary for a Python developer in the US is in the region of \$92,000.

1.3 Our first program

In the previous section we came across some nasty assembly language code. It's now time to write our first program. We're going to translate that program into Python:

```
1. print("Hello world!")
```

That's it! It really is that simple. Create a new file in your text editor, copy the above code into it and save it as *hello.py*. All Python programs end with the *.py* extension.

Next, to run our program we need to open a terminal window. On Windows, open the file explorer and go to the directory (folder) where you saved the file. On the top of the file explorer window you will see something similar to:

```
This PC > Desktop > my_folder
```

Click into this and type "cmd" and hit enter. This will open a terminal window in that directory.

Finally, to run your program, in the terminal window next to *_C:NAME>*, type the following.

```
python hello.py
```

You should now see "Hello world!" was printed out. If you remember from the previous section, *python* is the interpreter and we're telling it to run our program 'hello.py'

Congratulations! You've written your first of many Python programs. Now I want you to forget the above program. I have it here simply to serve as a boost in motivation. In the next chapter we'll take it right back to the very basics and get you on the road to mastering the Python language.

Chapter 2 - Integers, Operators and Syntax

2.1 Integers

Before we begin, getting to grips with the material in this chapter is important yet it's very easy! There isn't really anything difficult in this chapter. However, I will say, some of the exercises at the end may catch you out! so pay attention to the content.

As I said at the end of the previous chapter, forget that hello world program. In this chapter we're going to start off with the very basics and that means using Python as an integer calculator.

In the first part of this chapter we're going to use something called the Python REPL to write our programs.

- REPL stands for read-evaluate-print-loop:
 - Read one line of input
 - Evaluate the expression
 - Print the resulting value
 - Loop (repeat).

At the command prompt (in the terminal window), type the following (then 'Enter'):

python

You should see something like:

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now lets try some integer calculations by typing them at the REPL prompt:

```
5
5 + 7
5 - 3
4 * 4
```

```
15 // 5
5 * 2 + 3
3 + 2 * 5
(3 + 2) * 5
```

These are known as integer expressions. They are expressions which evaluate to an integer value.

A single value on its own such as the first expression in the above list of expressions is called an integer literal.

An integer is a positive or negative whole number.

So, what have we learned so far?

- We write integers in the obvious way
- We've met some basic operators (+, -, * and //). Integer operators in this case.
- We use * for multiplication and // for division
- There's something funny about * happening before +
- We also use (and) for grouping
- Integer expressions evaluate to Integer values

We can also write negative numbers in the same way we normally would.

$5 + -5$ evaluates to 0.

For completeness, the minus sign before the second 5 in the above expression is called the unary negation operator. It acts on a single value. The + is a binary operator as it operates on two things called operands (5 and -5 in this case).

In computer programming, it is very important that you understand how to read your programs in terms of a programs "parts of speech". So far, we have the following "parts of speech":

- Integer literals
- Integer operators
 - Binary operators: +, -, * and //
 - Unary operators: - (negation)
- Operands: $5 + 3$
- Integer expressions

- Integer values

2.2 Operator precedence

Let's take another look at some integer expressions:

```
3 + 2 * 5
(3 + 2) * 5
```

What did you notice when you entered these expressions into the REPL?

They give different answers. This is because of something called precedence. Precedence defines the order in which operators are evaluated and operators are evaluated from highest precedence to lowest.

The orders of precedence are as follows:

1. (...)
2. *, /
3. +, -

So, (...) happens first, then * and //, then + and -

For completeness, + and - are said to have equal precedence. Similarly, with * and //. However, precedence doesn't tell us the order of evaluation for operators of equal precedence and it can be important!

Something called associativity determines how operators of equal precedence are grouped (in the absence of parentheses).

The integer operators we have met are left associative meaning they are evaluated from left to right.

To demonstrate this, an expression such as $1 + 2 - 4 + 7$ is evaluated as follows:

```
1 + 2 - 4 + 7
(1 + 2) - 4 + 7    # Evaluate 1 + 2 first
((1 + 2) - 4) + 7  # Evaluate 3 - 4 next
(((1 + 2) - 4) + 7) # Evaluate -1 + 7
```

2.3 Syntax

Now let's throw a spanner in the works. In the Python REPL try these expressions:

```
4 + 3 +  
(1 + 1))  
5 + 3 2  
9 + - 3  
8 + + 9  
1 * * 6
```

You've just got your first Syntax Error.

In programming, the syntax of a language is concerned with how we put "characters" of the language together to form valid "words".

Another way of looking at syntax is, the set of rules which govern what is a valid program and what is not.

A syntax error occurs when we don't obey these rules (or form "meaningless words"). It's much the same with English for example. If someone wrote down the word "thwoelaman" you'd probably look at them as if they had two heads and that's essentially how the interpreter is looking at you, it can't make sense of what you've written!

For integer expressions however, the syntax is pretty obvious.

When you write code that doesn't follow the Python syntax and you get a syntax error, the python interpreter usually throws out a very useful error message. For example, if we try to evaluate the following expression:

```
>>> (1 + (// 4))
```

You'll get a syntax error that looks like this:

```
File "<stdin>", line 1  
    (1 + (// 4))  
          ^  
SyntaxError: invalid syntax
```

We are told the syntax error has probably occurred on line 1. We're then shown where on line 1 it has probably occurred. This is shown by the ^ character below the expression we tried to evaluate. Python is quite good at

pointing out where these errors occurred. In fact, Python is quite good at pointing out where most errors occurred compared to many other languages (at least in my opinion).

Important Side Note

I want to point out two other operators. The `**` (power) operator and the `%` (modulus) operator.

The function of the power operator (`**`) is quite obvious but the modulus operator (`%`), maybe not so much. Try these expressions in the REPL:

```
2 ** 2
3 ** 2
4 % 3
13 % 10
```

That's right, the `%` operator returns the remainder after division. In the case of `13 % 10`, this returns the remainder from 13 divided by 10 which is 3.

2.4 Exercises

Question 1

- Is the `**` (power) operator left or right associative? (Try figuring this out in the REPL)

Question 2

- Is the `%` operator left or right associative? (Try figuring this out in the REPL)

Question 3

- Where in the precedence hierarchy do `**` and `%` appear?

Question 4

- How are the following expressions evaluated?
 - `1 + 4 - 5 * 7 // 3`
 - `2 ** 3 * 4 + 8 // 3`
 - `4 % 3 ** 5 - 2 ** 2`

Question 5

- From the following list of integer expressions, which expression(s) have invalid syntax?
 - `((4 * 6 - (4 // 2))`
 - `9 * - 11 + 6`
 - `6 - +4`

Chapter 3 - Variables, Floats, Input and Printing

3.1 - Variable Assignment

In the previous chapter we looked at integer expressions. When the interpreter evaluated an expression it simply moved onto the next expression. What if we wanted to store the resulting value? How would we do this? The answer is using variables.

Variables utilize the computer's memory to store the data. You can think of main memory or memory (RAM) as cubbyholes with each cubbyhole labelled with some identifier (x for example). When we want to store a value, we are essentially telling the computer to put a value Y, into the cubbyhole labelled x. In Python this is called variable assignment. Let's look at this in action.

```
1. x = 7
2. y = 5
3.
4. z = x - y    # The value 2 is now essentially stored in the variable z
```

These are called assignment statements.

`x`, `y`, and `z` are variables.

The syntax for assignment statements is:

```
<variable name> = <value you want to store>
```

For this section you'd be best off by opening up your text editor and a command prompt (terminal window). Create a new file and call it whatever you want but make sure you save it with the `.py` extension.

Copy the three above assignment statements into the file and save it. Then in your command prompt, run the program (`python FILENAME.py`). Nothing

appears to have happened BUT I'll illustrate below what went on in the background.

The Python interpreter considers each statement in turn from top to bottom, evaluates the expression on the right-hand side, allocates an area of memory for the new variable `x` and stores the value `7` in the newly allocated memory slot. We'll start with the first statement `x = 7` and consider what happens in memory:

1) Memory:

```
-----  
| | | | | | | | | # Expression on right-hand side is  
evaluated  
-----
```

2) Memory: x

```
-----  
| | | | | | | | | # Allocate an area of memory for x  
-----
```

3) Memory: x

```
-----  
| | | | 7 | | | | | # Store the value 7 in this area  
of memory  
-----
```

Move to next statement

expression # Evaluate right-hand side of

4) Memory: x y

```
-----  
| | | | 7 | | | | | # Allocate an area of memory for y  
-----
```

```

5) Memory:      x      y
   -----
   |  |  |  | 7 |  |  | 5 |  |  # Store the value 5 in this area
of memory
   -----

   # Evaluate the right-hand side of next expression (z = x - y)
   # In this case it involves variables so...
   # Look up x in memory and replace x in the expression with the value
from memory
   # (z = 7 - y)
   # Look up y in memory and replace y in the expression with the value
from memory
   # (z = 7 - 5)
   # Evaluate the expression (z = 2)

6) Memory:      x      y      z
   -----
   |  |  |  | 7 |  |  | 5 |  |  # Allocate an area of memory for z
   -----

7) Memory:      x      y      z
   -----
   |  |  |  | 7 |  |  | 5 | 2 |  # Store the value 2 in this area
of memory
   -----

```

And so on, executing each assignment statement in turn. However, we could imagine that another statement exists that reads `x = 9`

In this case we 're-use' (or update) the memory slot as shown below

```
1) Memory:      x          y    z
-----
|  |  |  | *7* |  |  | 5 | 2 |  # Memory already allocated for x,
re-use it.
-----

2) Memory:      x          y    z
-----
|  |  |  | 9 |  |  | 5 | 2 |  # The value at memory location x
is now 9
-----
```

Now we can use this to do some useful things, for example, lets calculate the surface area of earth (approx.).

```
1. radius_of_earth = 6371
2. pi = 3.14
3. surface_area = 4 * pi * (radius_of_earth ** 2)
```

IMPORTANT NOTE:

- This brings me to an important point. Naming your variables with informative names such as `radius_of_earth` is really good practice, will save you many headaches and if another developer was to come along and take a look at your code, they wouldn't be left scratching their heads.
- You may have also seen me using `#` beside some code. These are called comments. They are ignored by the Python interpreter and are there simply for your benefit. It is also good practice to comment your code. Although the above program is simple and doesn't really require any comments, you could imagine a program that is hundreds or thousands of lines long (that's quite common) and comments would help future developers who were to come along figure out exactly what your code is trying to do. Also, if you were to write some code, leave it for a year and come back to it, having comments helps you understand what you were doing the previous year.

3.2 - Floating-point types

In the previous piece of code about computing the surface area of the earth you may have noticed that I had the statement `pi = 3.14`

We haven't seen these yet. This is a new type (like integer). They are called floating-point numbers (decimals numbers to the average person). In Python this is the `float` type.

```
>>> type(3.14)
<type 'float'>
```

Floats work pretty much the same as integers (which I'm sure you'd expect). We can perform the same calculations with them. Float expressions also evaluate to float types.

```
>>> 2.0 // 1.0
2.0
>>> 2.0 * 4.0
8.0
# etc....
```

I don't know about you, but that division symbol seems strange. Why not just use a single `/`?

Well you can! `//` is for floor division (digits after decimal point are removed). Let's look at how `/` and `//` work:

```
>>> 100 // 3
33
>>> 100 / 3
33.333333333333336
```

We can see here that when using integers, `//` rounds down to a whole number (`int`) but when we use `/` the result is converted to a `float`. This is called type casting.

Similarly:

```
>>> 100.0 // 3.0
33.0
```

```
>>> 100.0 / 3.0
33.333333333333336
```

This is something you need to watch out for when you're doing division. If you use `/` with integer operands and the resulting value contains digits after the decimal place, you may run into an error if the next part of your code is expecting an integer.

We also have all the other standard operators we've met (float operators in this case).

We can also write negative floats in the expected way:

```
-3.14
```

3.3 - Getting user input

Our programs so far haven't really been that interesting and they aren't very beneficial. We can't interact with them when they're running. Most applications these days allow the user to give input (whether that be entering data or clicking options around a screen). In this section we're going to look at how you can get input from the user as your program executes.

To do this we use a function, namely `input()`. We'll get to functions later in this book so don't worry about them, just know that when we type `input()`, our program will wait for the user to input something at the command prompt.

We can also store the input from the user. This is done as follow:

```
>>> x = input()
Hello    # User types this
>>> x
'Hello'
```

By default, `input()` returns a string (`str`), that's what the `' '` are around the `Hello` shows. This is another type which we will get to later but its just textual data (nothing too scary).

Essentially, `input()` takes in a line from input (we'll get to this later), converts the line into a string and returns it.

We can see that from the above code it might look like our program has frozen (it hasn't). We can add a user-friendly prompt that shows when asking the user for input. Let's take a look at a more user-friendly program.

```
>>> name = input("Enter your name: ")
Enter your name: John
>>> name
'John'
```

Perfect! We can now have the user input data into our program.

What if we don't want the user to input a string? What if we want an integer or a float?

There are a couple of ways to handle this situation, let's take a look at them:

```
# Method 1
x = eval(input())

# Method 2
x = int(input())
y = float(input())
```

Method 1 encapsulates the `input()` function in another function called `eval()`.

`eval()` is able to infer the type the user has inputted (if the user inputs 3, then `eval()` will cast the string '3' to an `int`. The same happens if the user inputs a float. I don't like this `eval()` function as it isn't very safe but it works fine here.

I prefer the second method, but we will get an error and our program will crash if the user inputs something that can't be cast to an `int` such as 'hello'.

```
>>> x = int(input())
Hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hello'
>>>
```

We get a `ValueError`. We'll look at different types of errors a little later on. We'll also look at being able to handle these situations (hence why I prefer the second method).

3.4 - The print function

In this section we're going to take another look at that function we came across in the introduction section where we wrote the 'Hello world!' program.

To clarify things for now, you can think of a function as a machine that takes some input, does something with that input, then spits out a value.

In this case, `print()` takes in something (an argument) and prints it to output (we'll also get to this later).

This, in my opinion, is one of the most useful functions Python has to offer. As we progress through this book and things start to get a little more difficult, I can bet you'll be using this quite a lot.

Let's take a look at how the `print()` function works:

```
123456>>> print("Hello")
Hello
>>> print(3.14)
3.14
>>> print("Hello again")
Hello again
```

Nothing too complicated! For now, though, let's recap on what we've covered so far.

Recap:

- We've figured out how to store data within our program (variable assignment)
- We've looked at a new type, the `float`.
- We've discovered that division can be done two ways:
- `//` or floor division
- `/` or your regular old division
- We've learned how to take input from the user with the `input()` function
- We've learned how to print information to the terminal window with the `print()` function.

Let's look at a program that puts all of this together! (Text editor time)

```
1. name = input("What is your name? ")
2. print("Hi " + name)
3. number = int(input("What number do you want to be squared? "))
4. print(number ** 2)
```

In the line `print("Hi " + name)`, the `+` joins the string "Hi" and name together to make a single string, this is called string concatenation and we'll come to this later in the strings chapter.

3.5 - Exercises

Important Note:

The solutions to some of these exercises require thinking outside-the-box. Don't expect to arrive at a solution immediately!

Question 1

Assume a population of rabbits doubles every 3 months. Further assume the current population is 11 rabbits i.e. `number_of_rabbits = 11`

Write a Python program to calculate the rabbit population after 2 years and print the result to the terminal.

Question 2

Extend the above program to allow the user to input the number of rabbits and the time period after which you want to know the rabbit population e.g. 4 years

Question 3

Assume the user is going to input 2 numbers, `x` and `y` for example. Write a Python program that will swap the numbers stored in these variables e.g.

```
# Assume the user has input these
x = 3
y = 11
# Your code to swap here
# After, the variables should be:
# x = 11
# y = 3
```


Question 4

The goal of this task is to derive a single **assignment statement** for the following sequences such that, if the variable x has the first value in a sequence, then x will have the next value in the sequence after the assignment statement is executed. And if the assignment statement were executed repeatedly, then x would cycle through all of the values in the sequence.

```
# EXAMPLE
```

```
# Sequence:
```

```
# 0 1 2 3 4 5 6 7 ...
```

```
#SOLUTION
```

```
x = x + 1
```

Sequence 1:

```
0 2 4 6 8 10 12 ...
```

Sequence 2:

```
0 -1 -2 -3 -4 -5 -6 ...
```

Sequence 3:

```
6 -6 6 -6 6 -6.....
```

Sequence 4 (Difficult):

```
-6 -3 0 -6 -3 0 -6 -3 0 -6 -3 0...
```

Chapter 4 - Control Flow

4.1 - Booleans

Important Note:

In that chapter I introduce mathematical logic, in particular, *truth tables*. You will need to have a solid understanding of truth tables in order to make the most of this chapter. Having a clear understanding of this section of logic will also help you get rid of some weird bugs in your programs further down the line.

If you can remember back to the example of making a cup of tea, we had a step that went something like:

```
- if want milk:
    - take milk from fridge
    - pour milk
```

The above snippet allowed us to decide. In programming this construct is called *control flow*. We either wanted milk or we didn't. In order to implement this kind of thing in our programs we need to introduce a new *type* called the *bool* which is short for *Boolean*.

A *Boolean* is a type which has two values: *true* or *false*.

```
```python >>> type(True) <class 'bool'>
```

```
type(False) <class 'bool'> ```
```

This brings me to an important point. In Python, different values of different types can be *truthy* or *falsey*.

For example, with integers, `0` is falsey while any other integer is truthy. With floats, `0.0` is falsey while any other float is truthy and finally with strings, the empty string (`''`) is falsey while any other string is truthy.

```
>>> False == 0
True
>>> False == 1
False
>>> False == 0
True
>>> False == 0.1
False
```

We can also cast any other value to a *bool* using the `bool()` function if the value can be interpreted as a truth value.

Booleans also have operators. These are different to the operators we've seen with integers and floats. These are *Boolean operators*.

### OR Truth Table

P	Q	P OR Q
True	True	True
True	False	True
False	True	True
False	False	False

### AND Truth Table

P	Q	P AND Q
True	True	True
True	False	False
False	True	False
False	False	False

### NOT Truth Table

P	NOT P
True	False
False	True

Here are a few examples in Python

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
>>> True and True
True
>>> True and False
False
```

```
>>> not True
False
>>> not False
True
```

Again, knowing these truth tables off like the back of your hand is crucial. You'll be dealing with these a lot!

## 4.2 - The *if* Statement

In this section we're going to look at adding control flow to our programs. We'll start by looking at the *if* statement. Before we do though, I want to look at something called *comparison operators*. These operators allow us to compare things.

Comparison Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

They're pretty simple to understand so let's take a look at them in action:

```
>>> 1 == 1
True
>>> 1 == 2
False
>>> 10 != 4
True
>>> 4 != 4
False
>>> 11 > 9
True
>>> 13 < 23
True
>>> 16 < 2
False
>>> 13 < 13
False
>>> 13 <= 13
True
```

I think you get the gist. So, let's get onto the *if* statement.

```
1. wants_drink = input("Do you want a drink with your meal? (yes or no): ")
2. if wants_drink == "yes":
3. print("That will cost an extra £1.50")
4. print("Thank you! Enjoy your meal")
```

If we run the above code with two different types of input, we can get two different results:

```
USER ANSWERS YES
Do you want a drink with your meal? (yes or no): yes
That will cost an extra £1.50
Thank you! Enjoy your meal

USER ANSWERS NO
Do you want a drink with your meal? (yes or no): no
Thank you! Enjoy your meal
```

The above code reads quite clearly as well, almost like English. If the user says yes to wanting a drink, then print that it will cost more.

The syntax for *if* statements is as follows:

```
if <CONDITION>:
 # Do something if condition evaluates to True
```

A couple of important things to note on the syntax. It is always *if* followed by the condition, then a colon (:). For the block of code that follows, it must be indented!! I like to use tabs however you can use spaces. Right now, I'm going to tell you to pick a number of spaces. It's typical to use 4 spaces. For tabs, 1 tab will suffice.

Let's look at another example:

```
1. my_number = int(input("Enter a number: "))
2. if my_number <= 10:
3. print("First line")
4. print("Second line")
5. my_float = 4.3
6. print("Goodbye")
```

The above program does nothing useful, in fact it's complete garbage but it shows the syntax for *if statements* quite well.

Anything that is indented after the *if statement* is called a code block. Everything after the *if statement* such as the `my_float = 4.3`, will be executed regardless of `my_number` being less than or equal to 10.

You'll probably make a lot of syntax mistakes in the beginning, we all did so don't let that discourage you at all, it's completely normal. Even the most experienced developers make syntax mistakes every now and again.

Although the *if statement* is useful, what if we're in a situation where we wanted to do one thing if one condition passed, otherwise, do something else? Next, we're going to look at the *if-else statement*.

## 4.3 - The *if-else* statement

In the previous section we learned how to add some very basic flow control to our program however we imagined a scenario in which we wanted to do something else if the condition for the *if statement* didn't pass.

Let's look at how to do that:

```
1. my_age = int(input("Enter your age: "))
2.
3. if my_age >= 18:
4. print("You can enter")
5. else:
6. print("You are too young to enter")
7.
8. print("Goodbye")
```

It again, is quite easy to read the above program. When we execute the above program we could have two different outputs:

```
SCENARIO 1
Enter your age: 33
You can enter
Goodbye

SCENARIO 2
Enter your age: 14
You are too young to enter
Goodbye
```

The syntax again is obvious:

```
if <CONDITION>:
 # Do something if the condition was True
else:
 # Do something else!
```

### **REMEMBER:**

Don't forget the colon at the end of each statement and don't forget to indent the code block. Each statement in the code block must also be indented the same number of spaces!

Let's look at another example. In this example we want to write a program that will tell us if an integer is even or odd:

```
1. number = int(input("Enter a number: "))
2. if (number % 2) == 0:
3. print("Your number was even")
4. else:
5. print("Your number was odd")
```

Remember back to the chapter on integers, we had the modulus (%) operator? To refresh your memory, this operator returned the remainder after division.

The == comparison operator takes two *operands*; in the above case the left operand was an expression. This will evaluate to a value which is then checked to see if it is equal to zero. If it is, then it is an even number.

We can also combine comparison operators and Boolean operators, let's look at an example of how this is done:

```
1. number = int(input("Enter an even number between 50 and 100: "))
2. if (number % 2) == 0 and number < 100 and number > 50:
3. print("You've entered an even number between 50 and 100")
4. else:
5. print("I don't care about your number")
```

The condition might look nasty but it actually reads quite easily:

"if the number is even AND the number is less than 100 AND the number is greater than 50"

We can even simplify it a bit. Remember I told you that values could be truthy or falsey? Well let's look at our even number program again.

```
1. ...
2.
3. if (number % 2) != 0:
4. print("Your number was odd")
5. else:
6. print("Your number was even")
7.
8. ...
```

Note that we are saying if the remainder doesn't equal zero in the condition.

We could simplify this to:

```
1. if number % 2:
2. print("Your number was odd")
3. else:
4. print("Your number was even")
```

If `number % 2` is 0, then we don't enter the code block for *if* because 0 is falsey.

**REMEMBER:** We execute the statements inside the *if* block only if the condition evaluates to True. Also, just to note, *else* doesn't take a condition. It automatically executes if the condition on the *if statement* fails.

## 4.4 - The *elif* statement

Our programs are now getting a little more complex and we have some decent control flow but they're acting as if there's only two options. What if there were 3 or more choices? The *elif statement* can solve this for us!

The *elif statement* allows us to check multiple expressions. *if...elif...else* statements are checked in sequential order, evaluating each condition and executes the code block for the first condition that evaluates to *true*.

The syntax is as follows:

```
if <condition 1>:
 # Do something
elif <condition 2>:
 # Do something else
elif <condition 3>:
 # Do something else
.
.
.
elif <condition N>:
 # Do something else
else:
 # Do something else
```

We can have as many *elif statements* as we please.

Let's write a program that calculates the grade of a student based on their mark:

```
1. mark = int(input("Enter a mark: "))
2.
3. if mark >= 70:
4. print("You got a first")
5. elif mark >= 50 and mark < 70:
6. print("You got a second")
7. elif mark >= 40 and mark < 50:
8. print("You got a third")
9. else:
10. print("You failed!")
```

These *elif statements* are pretty straight forward.

If you are coming from another language, Python does not provide switch/case statements as do other languages, but we can use *if...elif...else* to mimic them.



## 4.5 - Nested *if* statements

What if we have a condition and based on that condition, we need to check other conditions? Well we can do that by *nesting* `if...elif...else` statements inside other `if...elif...else` statements.

Let's look at an example program to convert the numbers between 0 and 3 to their word equivalents:

```
1. number = int(input("Enter a positive number: "))
2.
3. if number >= 0:
4. if number == 0:
5. print("Zero")
6. elif number == 1:
7. print("One")
8. elif number == 2:
9. print("Two")
10. elif number == 3:
11. print("Three")
12. else:
13. print("You entered a number greather than three")
14. else:
15. print("You entered a negative number")
```

This is a toy example but nested `if...elif...else` statements are used everywhere, and you'll come across them frequently so spend time becoming familiar with writing them!

They're also quite straight forward so I'm going to get straight into some exercises so you can practice these things.

## 4.6 - Exercises

### Important Note:

The solutions to some of these exercises require thinking outside-the-box. Don't expect to arrive at a solution immediately!

### Question 1

Write a program that takes three inputs, `a`, `b` and `c` and prints out whether or not those numbers form a right-angled triangle. The formula you'll need is

$$c = \sqrt{a^2 + b^2}$$

## Question 2

Fizz buzz is a game in which the following rules apply:

- any number which is divisible by 3 is replaced with **fizz**
- any number which is divisible by 5 is replaced with **buzz**
- any number which is divisible by both 3 and 5 is replaced with **fizz-buzz**
- any other number is just the number itself

Examples:

```
1 = 1
2 = 2
3 = fizz
4 = 4
5 = buzz
6 = 6
.
.
15 = fizz-buzz
```

\* Remember what I said about how if...elif...else statements are evaluated sequentially (first to last) until the first condition passes

In your solution you should expect the user to enter a single number and either print the number, fizz, buzz or fizz-buzz

## Question 3

Write a program which takes in six numbers, **x1**, **y1**, **r1** and **x2**, **y2**, **r2** (which are the x and y coordinates of the centre of a circle and that circles radius) and print out whether or not the two circles overlap.

You're going to need the formula for the distance between two points to solve this which is:

$$d(P,Q)=\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$

The rest requires some problem solving! Perhaps drawing out the scenario on paper might help?

# Chapter 5 - Looping & Iterations

## Important Note:

This chapter may be a fair bit more challenging than the previous chapters and there is a lot to take in here.

## 5.1 - *while* loops

Let's say we wanted to do something over and over again. At the moment we'd have to write out the same piece of code over and over. This isn't practical. What if we wanted to do something 100 times or 10,000 times?

In this chapter we look at looping and iteration, in particular, the *while* loop. While loops are syntactically very similar to *if statements*.

```
IF STATEMENTS
if <condition>:
 # Do something
 # And another thing

WHILE LOOPS
while <condition>:
 # Do something
 # And another thing
```

As with *if statements*, the *<condition>* is a *Boolean expression*. The body also contains a sequence of statements. Similarly, the extent of body of the loop is determined by the indentation of the statements.

The way they work however is slightly different and they operate as follows:

1. Evaluate the condition.
2. If the condition evaluates to *False*, then skip the body of the *while loop* and execute the subsequent statements.
3. Otherwise, if the condition evaluates to *True*:
4. Execute the body of the loop
5. Return to step 1.

Let me illustrate this with an example

```
1. x = True # <- Assign True to x (x contains True)
2. while x == True: #
3. print("Hello") #
4. x = False #
5. print("x is now false") #
6.
7. -----
```

```

8.
9. x = True #
10. while x == True: # <- Evaluate the expression (evaluates to True)
11. print("Hello") #
12. x = False #
13. print("x is now false") #
14.
15. -----
16.
17. x = True #
18. while x == True: #
19. print("Hello") # <- Print "Hello" to the terminal window
20. x = False #
21. print("x is now false") #
22.
23. -----
24.
25. x = True #
26. while x == True: #
27. print("Hello") #
28. x = False # <- Assign False to x (x contains False)
29. print("x is now false") #
30.
31. -----
32.
33. x = True #
34. while x == True: #
35. print("Hello") #
36. x = False # <- We've reached the end so re-evaluate condition
37. print("x is now false") #
38.
39. -----
40.
41. x = True #
42. while x == True: # <- Evaluate the expression (it fails, so exit loop)
43. print("Hello") # (If it didn't we'd go around the loop again.)
44. x = False #
45. print("x is now false") #
46.
47. -----
48.
49. x = True #
50. while x == True: #
51. print("Hello") #
52. x = False #
53. print("x is now false") # <- Print "x is now false" to terminal window

```

Let's look at another example.

```

1. i = 0
2. while i < 5:
3. print(i)
4. i = i + 1
5.
6. # THE OUTPUT
7. 0
8. 1
9. 2
10. 3
11. 4

```

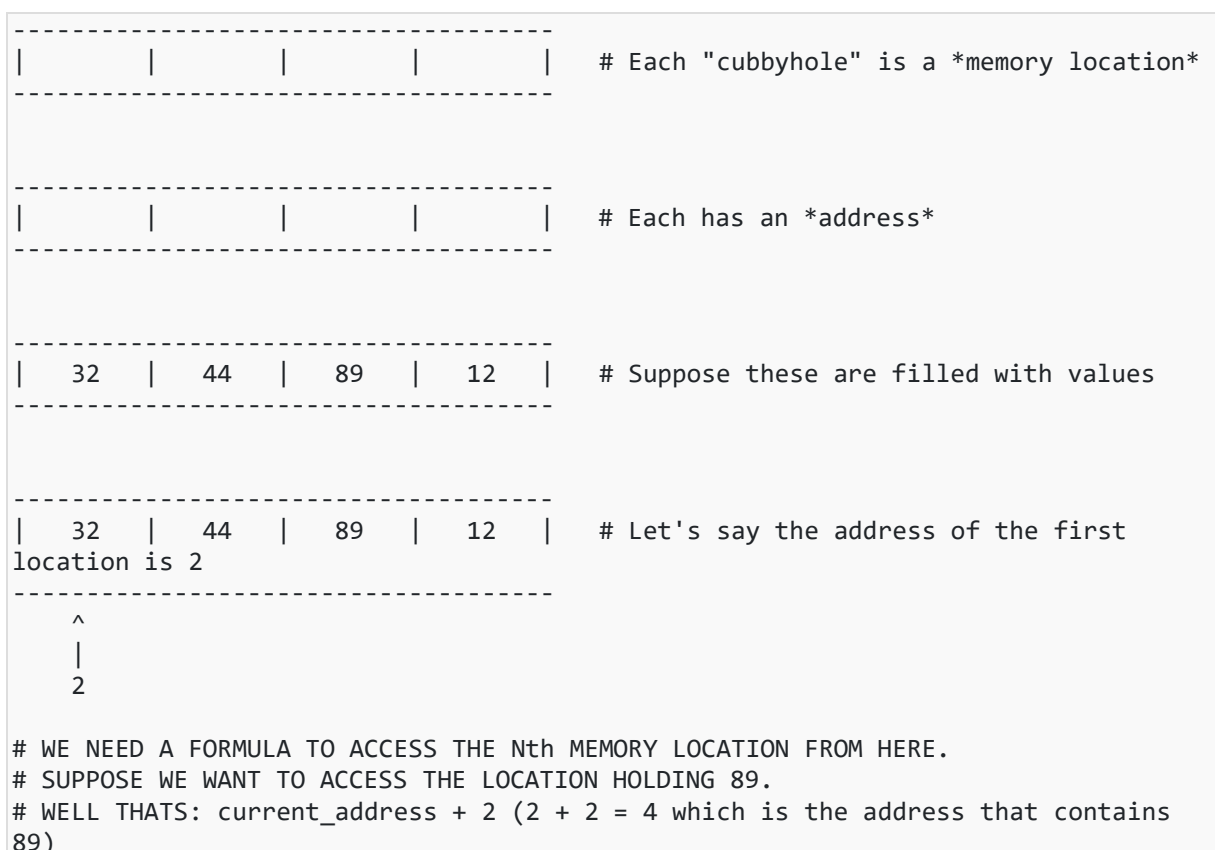
This might look a little confusing but let me explain what's going on. The variable `i` is called a counter (or loop variable) and it controls the number of times the loop iterates.

- The variable `i` is initially 0 and the condition evaluates to `True`, so the body of the loop is executed.
- Print the value of the variable `i` to the terminal and increment `i` by 1. (`i` is now 1 instead of 0).
- Evaluate the condition again. It is still `True` as `i` is still less than 5.
- Print the value of the variable `i` to the terminal and increment `i` by 1. (`i` is now 2).
- Repeat the above steps until `i` is 5 in which case the loops condition is `False` so the loop terminates and we execute any statements which follow.

What if we were to have the condition as `i < 10000`? Our 4 lines of code would still hold, and the program would count from 0 to 9999.

You may be wondering why we counted from 0 as it seems unnatural but there are good reasons for this. The explanation for it may be a little bit too technical for you at this stage but let me give an example that hopefully explains it.

Remember I said you can think of memory as cubbyholes? Well, lets take that example:



```
THIS FORMULA NEEDS TO BE GENERAL SO IF WE WANTED THE FIRST LOCATION
THAT WOULD BE: current_address + 0 (2 + 0 = 2 which is the address of first
location)
```

We don't actually count from 0 we start *indexes* (which we will get to later) with 0. It's just good practice to count from 0 in loops like this as we usually use a value such as *i* to *index* into lists (we'll get to these later) and I recommend getting used to it now. If this still confuses you, it will make complete sense when we get to *lists* a little later on.

It's very rare to see counters beginning at something like 30 and where counters do begin at odd or large number like these, there is usually a very, very good reason for doing so.

Let's look at another example. We'll write a program that allows the user to input 3 numbers and we'll add them up and print the result.

```
1. total = 0
2.
3. i = 0
4. while i < 3:
5. total = total + int(input())
6. i += 1
7. print(total)
```

**Important:** Note how I wrote `i += 1`. This is shorthand for `i = i + 1`. We could have also shortened the first statement of the loop body to `total += int(input())`.

## 5.2 - Looping patterns

You may have noticed in the previous *while loop* examples we had a re-occurring pattern (or structure) to our loops.

There are two patterns we usually follow when writing *while* loops. These are:

1. Do-something-n-times
2. Do-while-not-condition

I'll get to the second pattern in a minute but for now I'll look at the first pattern.

We have been using the first pattern in the previous examples which is as follows:

```
while i < N:
 # Do something
 i += 1
```

In the above pattern, we know the value of *N* prior to evaluating the loops condition. It may have come from user input or it may be the result of some previous calculation.

As a *naming convention*, it's good practice to name the *counter* as *i* and if *i* is being used as a variable name somewhere else, then *j* or *k* will do.

Here's another example which may seem to go against this pattern upon thinking about the problem but what if we wanted to count backwards, so instead of 0, 1, 2, we wanted 2, 1, 0 (You're probably thinking we should do *i = i - 1*)?

Here is how it's done but the user inputs the number we're counting backwards from:

```
1. n = int(input())
2.
3. i = 0
4. while i < n:
5. print(n - i - 1)
6. i += 1
```

Great! We're still conforming to the pattern.

Let's look at the second looping pattern, the "Do-while-not-condition" pattern. With this pattern we don't know how many times we'll go around the loop before it terminates.

It's structure is as follows:

```
while not <condition>:
 # Do something
```

Each pattern has their uses so let's look at an example where this pattern could be used.

Suppose the user inputs numbers some number of times, we don't know how many times yet, and our condition is that they can't input a negative number:

```
1. number = int(input())
2.
3. while not number < 0:
4. print(number)
5. number = int(input())
6. print("You entered a negative number")
```

In this example we continuously ask the user for a number and print it. We continue like this until they input a negative number in which case the condition fails, and the loop terminates.

To recap:

- We use "Do-something-N-times" pattern when we know how many times, we need to go around the loop
- We use "Do-while-not-condition" pattern when we don't know how many times, we need to go around the loop

## Important Note:

If you are using the "Do-something-N-times" pattern, don't forget to increment `i`! Forgetting to increment the value of this variable may lead to something called an *infinite loop* and these are bad! If you are doing a counting loop for example, like the one we have seen above, and you're counting to 10 and forget to increment `i`, your program will count past 10 and continue on forever. Infinite loops can lead to your computer slowing down or even worse, freezing. So be careful!

## 5.3 - *break* and *continue*

In this section I'm going to discuss two new statements. The `break` statement and the `continue` statement. There is a bit of a debate around when and where you should use them and what is best practice regarding the use of these statements, but I'll talk about that a little later on in this section.

The `break` statement is used to exit the loop by skipping the rest of the loop's code block without testing the condition for the loop. `break` interrupts the flow of the program by breaking out of the loop and continues to execute the rest of your program.

The `break` statement is used as follows:

```
while <condition 1>:
 # Statements
 if <condition 2>:
 break
 # More statements
Rest of your program
```

Here's an example of it in action:

```
1. number = 10
2.
3. i = 0
4. while i < number:
5. if i == 5:
6. break
7. print(i)
8. i += 1
9. print("Exiting program")
10.
11. # OUTPUT
12. 0
13. 1
14. 2
15. 3
16. 4
17. Exiting program
```

I think it's quite self-explanatory how `break` works so let's take a look at the `continue` statement.



The `continue` statement is used to skip the remaining statements in the body of the loop and start the next iteration of the loop. Unlike the `break` statement, `continue` doesn't terminate the loop, it just jumps back to the start.

The `continue` statement is used as follows:

```
while <condition 1>:
 # Statements
 if <condition 2>:
 continue
 # More statements
Rest of your program
```

Here's an example of it in action:

```
1. number = 5
2.
3. i = -1
4. while i < number:
5. i += 1
6. if i == 2:
7. continue
8. print(i)
9. print("Exiting program")
10.
11. # OUTPUT
12. 0
13. 1
14. 3
15. 4
16. Exiting program
```

Notice how this has kind of messed up our looping pattern as we had to put `i += 1` as the first statement and we had to initialize `i` to -1.

Now this is a trivial case but `continue` has its uses. Let's look at when and where these statements should be used. For now, though, I recommend avoiding them and don't use them as a crutch to get out of a loop. For the most part, in this book, if you find yourself using these statements there's usually something wrong with your loop and you should think about the problem a bit more.

When used at the start of a loop's code block, they act like *preconditions*. A precondition is a condition placed on something (loop, function, etc) which must hold true at the beginning of that "something" in order for it work correctly.

When placed at the end of the loop's code block, they act like *postconditions*. A postcondition is a condition placed on something which must hold true after the execution of that "something" in order to be satisfied that it has executed correctly.

Now this next one might be a bit confusing so don't get too caught up in it, I'm putting it here for completeness.

These statements may also be used as *invariants*, in particular, *loop invariants*. An invariant is a statement that is placed in a code segment that is repeated (loops for example) and should be true each time about the loop. Invariants are often used in loops and recursion (which we will get too later).

This part may be confusing. If an invariant is placed at the beginning of a while loop whose condition doesn't change the state of the computation (that is, it has no side effects), then the invariant should also be true at the end of the loop. When used in object-oriented programming (which we'll also get to later), invariants can refer to the state of an object.

For now, though, avoid them, especially in the exercises later on.

## 5.4 - while loop examples

As I've said before, **while** loops can be confusing when starting out with them so I'm going to give some more examples of them in action before we move onto exercises.

### Example 1

The first example problem is to check if a given number is a prime number assuming the given number is greater than or equal to 2. Here's the code:

```
1. n = int(input())
2.
3. i = 2
4. while i < n and (n % i) != 0:
5. i += 1
6. if i < n:
7. print("Your number is not a prime number")
8. else:
9. print("Your number is a prime number!")
```

Try and figure out why I have the if/else statements by working it out on some paper with small numbers.

### Example 2

The next example is going to be an extension to the Fizz-Buzz program you've written before. There is a slight change though. Last time you wrote a program that prints the number, fizz, buzz or fizz-buzz for a given number. This time I'm going to write a program that prints the sequence up to a given number.

```
1. n = int(input())
2.
3. i = 1
4. while i < n:
5. if i % 3 == 0 and i % 5 == 0:
6. print("fizz-buzz")
7. elif i // 3 == 0:
```

```

8. print("fizz")
9. elif i // 5 == 0:
10. print("buzz")
11. else:
12. print(i)
13. i += 1

```

Notice the order the if/elif/else statements go in. Since in Python statements are executed sequentially, we need to check that the number isn't divisible by both 3 and 5 first.

### Example 3

In this example program we're going to write a program that allows the user to continuously enter numbers until they enter the number 0. When they enter the number 0, we'll print out the total of all the other numbers they inputted.

```

1. number = int(input())
2.
3. total = 0
4. while not number == 0:
5. total += number
6. number = int(input())
7. print(total)

```

If you now feel a bit more comfortable with *while loops* then let's look at some problems for you to solve!

## 5.5 - Exercises

### Important Note:

The solutions to some of these exercises require thinking outside-the-box. Don't expect to arrive at a solution immediately!

These exercises take a step up in difficulty so think carefully about your solutions.

### Question 1

Write a program that prints the first  $n$  numbers of the Fibonacci sequence. The Fibonacci sequence is defined as:

$$F_n = F_{n-1} + F_{n-2} \quad F_0 = 0, F_1 = 1$$

That is, the next number in the sequence is the sum of the previous two numbers and the first two numbers in the sequence are 0 and 1. A part of the sequence is shown below:

```
10, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

## Question 2

Write a program that prints a sequence such as the one shown below. The number of stars in the widest part should be equal to the number a user input:

```
INPUT
7

OUTPUT
*
**

**
*
```

## Question 3

Write a Python program that reads in a single integer from the user. Then read in an unknown number of integers, each time printing whether or not the number was higher or lower than the previous. You can assume that every number is different and you can stop when the number entered is 0. Here is an example input and output:

```
INPUT
5
2
9
7
0

OUTPUT
lower
higher
lower
Exiting...
```

# Chapter 6 - Strings

## 6.1 - Introduction to strings

In this chapter we're going to take a closer look at strings. We have met them various times throughout the book and we know they represent textual data.

Manipulating textual data is common in programming so becoming comfortable working with strings is essential!

In Python we represent strings as a sequence of characters enclosed by " " or ' '. For example:

```
1. my_string = hello, world! # This is not a string
2.
3. my_string = "hello, world!" # This is a string!
4. my_string = 'hello, world!' # This is also a string!
```

Python also offers a large set of operations, methods and functions for working with strings, some of which we'll meet later on in this chapter. Working with strings in Python is also quite easy compared to other languages.

There are a couple of things I'd like to point out before we jump onto string methods. Firstly, I want to take another quick look at the print function.

*print()* converts each of its arguments to strings, inserts a space character in the output between each of its arguments then appends a **newline** character and writes the result to standard output (in our case the terminal window).

When I say argument I mean the following:

```
print("This", "and", "that")

OUTPUT
"This and that"
```

In the above code, each of the strings is an argument and you can have as many arguments as you like (we'll look at arguments in more detail later in the functions chapter).

When working with textual data (strings) we have some special characters. Some of these are:

## Special Character Meaning

<code>\n</code>	Newline
<code>\r</code>	Carriage return
<code>\t</code>	Tab

And here is what they do:

```
>>> print("Hello\nWorld")
Hello
World

>>> print("Welcome to my\rhome")
Homeome to my

>>> print("Apples\tOranges")
Apples Oranges
```

The `\t` adds a tab, the `\n` will move to a new line and `\r` will essentially take whatever comes after it, and overwrite whatever comes before it up to the length of whatever comes after it. Don't worry about it too much I've only encountered it a handful of times.

We probably won't encounter the tab or carriage return characters again throughout this book but I want to warn you about the newline character. Just because you can't see it doesn't mean it's not there so if you are ever getting unexpected output the newline character might be causing it! But we'll also look at dealing with it a little later on.

There is also a special string called the *empty string*. This is represented as `""`.

## 6.2 - String functions

As I've said previously, Python provides lots of functions for working with strings, so let's take a look at some of the most important ones now!

### Method What it does

<code>len()</code>	returns the length of the string
<code>str()</code>	Returns the string representation of an object
<code>chr()</code>	Converts an integer to a character
<code>ord()</code>	Converts a character to an integer

Let's look at these in action:

```
>>> len("Hello")
5
>>> len("Hello\nWorld") # NewLine character counts!
11
```

```
>>> str(5 + 5)
'10'
>>> chr(65)
'A'
>>> ord("B")
66
```

`len()` and `str()` work as expected. We cast a type to a string using the `str()` function.

But what are `chr()` and `ord()` doing?

At the most basic level, computers store information as numbers. To represent characters, a translation scheme has been devised which maps each character to its representative number.

The simplest of these schemes is called **ASCII** which you may have heard of. It stands for American Standard Code for Information Interchange and it covers the common Latin characters you're probably used to working with.

`chr()` and `ord()` are used for translating between codes and characters.

## 6.3 - String methods

We're now going to look at string methods. They're kind of like functions except they're specific to strings. For example, `len()` works on many different *types*.

The syntax for a method is as follows:

```
1 object.method(<arguments>)
```

Let's look at some of the most commonly used string methods:

Method	Meaning
<code>s.capitalize()</code>	returns a copy of <code>s</code> with the first character capitalized
<code>s.title()</code>	returns a copy of <code>s</code> with the first letter of each word capitalized
<code>s.upper()</code>	returns a copy of <code>s</code> with all alphabetic characters capitalized
<code>s.lower()</code>	returns a copy of <code>s</code> with all alphabetic characters converted to lowercase
<code>s.isalnum()</code>	returns True if <code>s</code> is nonempty and all its characters are alphanumeric (letters and numbers) and False otherwise.
<code>s.isalpha()</code>	returns True if <code>s</code> is nonempty and all its characters are alphabetic and False otherwise

Method	Meaning
--------	---------

s.isdigit()	returns True if s is nonempty and all its characters can be casted to integers and False otherwise
s.strip()	returns a copy of s with the whitespace on the left-hand side and right-hand side removed
s.lstrip()	returns a copy of s with the whitespace on the left-hand side removed
s.rstrip()	returns a copy of s with the whitespace on the right-hand side removed.

Let's look at these in action:

```
>>> s = "tHiS is A StRIng"
>>> s.capitalize()
"This is a string"

>>> s = "tHiS is A StRIng"
>>> s.title()
"This Is A String"

>>> s = "tHiS is A StRIng"
>>> s.upper()
"THIS IS A STRING"

>>> s = "tHiS is A StRInt"
>>> s.lower()
"this is a string"

>>> s = "937 ThIs is A STRing"
>>> s.isalnum()
True

>>> s = "93 *** ThIs is A STRing"
>>> s.isalnum()
False

>>> s = "937 ThIs is A STRing"
>>> s.isalpha()
False

>>> s = "ThIs is A STRing"
>>> s.isalpha()
True

>>> s = "This is not a digit"
>>> s.isdigit()
False

>>> s = "76324"
>>> s.isdigit()
True

>>> s = " This is a string "
>>> s.strip()
```



```

"This is a string"

>>> s = " This is a string "
>>> s.lstrip()
"This is a string "

>>> s = " This is a string "
>>> s.rstrip()
" This is a string"

```

For some of these methods, when we don't pass any arguments, they have a default behaviour. However, we can change this by passing arguments to the function. Let me show you how that works:

```

>>> s = "This is my stringggggg"
>>> s.rstrip("g")
"This is my strin" # rstrip() now removes 'g' instead of whitespace
to the right-hand side

```

Make sure you become familiar with these, you'll probably use them often (some more often than others).

## 6.4 - String indexing

**Important Note:** This section is incredibly important so pay attention to it!

Remember we talked about memory and counting from 0 for *indices*? Well let's combine those two ideas here.

I mentioned earlier in the chapter that strings are a sequence of characters. In memory they are represented as:

H	E	L	L	O	# Each memory location contains one character
---	---	---	---	---	-----------------------------------------------

H	E	L	L	O	# Each location also has an address
13	14	15	16	17	# These are hypothetical addresses

H	E	L	L	O	# We can get the address of 'E' as 13 + 1
13	14	15	16	17	# Similarly for 'H' as 13 + 0

# The 0 above is an "index" so is the 1!

We can index strings in a similar way! The syntax for indexing into a string is as follows:

```
>>> s = "Hello"
>>> s[0]
"H"
>>> s[1]
"e"
>>> s[2]
"l"
>>> s[3]
"l"
>>> s[4]
"o"
>>> s[5]
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Notice in the last line we try to index to string at position 5 but we get an **IndexError** and we're told the index is out of range. This is because we start counting from 0 and 0-4 are the 5 indices of the string.

So, the final index of string *s* is: `len(s) - 1` and we can index using this: `s[len(s) - 1]`

String indexes may also be negative numbers and will specify locations relative to the end of the string.

```
>>> s = "Hello"
>>> s[-1]
"o"
>>> s[-2]
"l"
>>> s[-3]
"l"
>>> s[-4]
"e"
>>> s[-5]
"H"
123H E L L O # String
0 1 2 3 4 # Positive indices
-5 -4 -3 -2 -1 # Negative indices
```

### Recap for this section:

- An index is a position.
- Using indices, we can reference characters at particular positions.
- For indices, we always start counting from 0.
- The first character is at index 0.
- The last character is at the index `len(s) - 1`

## 6.5 - Immutability

In Python, every variable holds an instance of an object and there are two types of objects, **mutable** and **immutable**. So far, we have only met **immutable** types i.e ints, floats, bools and strings. An object being immutable means its value can't be changed once created.

For example, doing something such as:

```
10 = 3
True = False
"Hello" = "World"
```

This wouldn't make any sense and it's probably quite obvious why. However, now that we know about string indexing, doing something such as:

```
s = "string"
s[0] = "b"
```

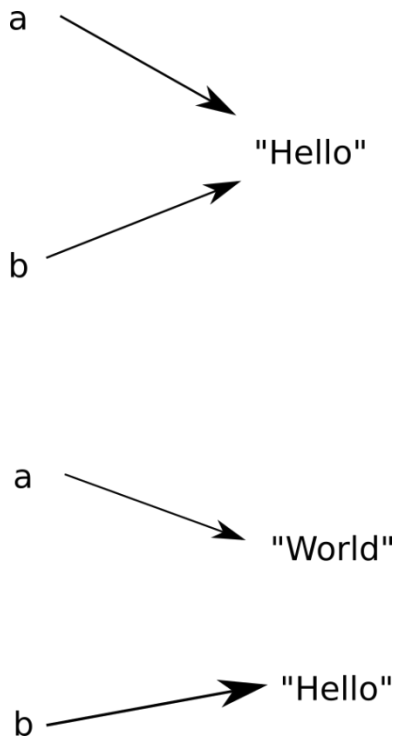
This might seem reasonable. However, it's not and there is good reason for it. What if we tried something such as:

```
s = "string"
s[0] = "bl"
```

Now that might also seem reasonable but remember how strings are stored in memory. When they are created, they have a length and trying to change that length such as above would cause issues on a lower level (not that it can't be done, it's just not worth it in 99% of cases). Making strings immutable types increases performance and security but don't worry about that for now. Just know you can't change a strings value once it's created.

To get a better understanding of how strings are stored in memory and how they're *referenced*, look at the code below, then the diagram that follows.

```
a = "Hello"
b = a
a = "World"
```



From the first part of the diagram, I said before `a` contains "Hello", however that isn't the case. That was for simplicity. What's actually happening is, Python maintains a *namespace* of mappings from variables to objects they refer to. After `a = "Hello"` is executed, a mapping from `a` to the value "Hello" is added to the namespace. `a` now points to the memory location that holds the value "Hello". We say that `a` contains a reference to the value "Hello" stored in memory.

When the line `b = a` is executed, what happens is, `b` now points to the same memory location that `a` points to and when `a = "World"` is executed, a new string instance is created and saved in some new memory location and the location that `a` points to is updated.

Hence, this behaviour is what makes strings immutable. Try to gain an understanding of this and we'll come back to it and look at it in more detail later.

## 6.6 - String slicing

We have seen how to get characters at specific positions of a string using indexing e.g. `s[0]` gets the character at the first position of string `s`.

What if we wanted to get the characters from the first to third position of the string "Hello" i.e. "Hel".

Well we can! We use **slicing** to do so. Slicing works very similarly to indexing except, rather than just specifying the start index, we also give the end index we want.

Here's a couple of examples:

```
>>> s = "Hello"
>>> s[0:3]
'Hel'

>>> s[0:1]
'H'
```

The syntax is: `string[ <start> : <end> ]`

Notice how slicing works though. Using the first example, slicing returns a new string with everything from `s` at the start index, up to but not including the end index.

If we wanted everything after the first letter in `s`, we can omit the end index. We can also omit the first index and leave the end index which will give us everything from index 0 up to the specified end index

```
>>> s = "Hello"
>>> s[1:]
'ello'
>>> s[:3]
'Hel'
```

We can also omit both indexes, in which case we'll get the entire string returned:

```
>>> s = "Hello"
>>> s[:]
'Hello'
```

Note that the *colon* must remain in either case.

The indices may also be variables:

```
>>> first = 2
>>> last = 4
>>> s = "Hello"
>>> s[first:last]
'LL'
```

The indices may also be negative numbers and will specify locations relative to the end of the string:

```
>>> s = "Hello"
>>> s[:-1]
'Hell'
>>> s[-4:]
```

Python also offers extended slicing for strings. This adds a third parameter when slicing the string. This third parameter indicates the step size and the syntax is `s[first:last:step]`.

This is done as follows:

```
>>> s = "This is a string to be sliced"
>>> s[::1]
'This is a string to be sliced'
>>> s[::2]
'Ti sasrn ob lcd'
>>> s[1:8:2]
'hsi'
```

The step size can also be negative (This is useful for reversing a string)

```
>>> s = "This is a string to be sliced"
>>> s[::-1]
'decils eb ot gnirts a si sihT'
>>> s[::-2]
'dcl bo nrsas iT'
```

## 6.7 - String operators

Strings also have operators. These are the `+` operator and the `*` operator. They don't function the same as integer or float operators.

The `+` operator is called the concatenation operator and is used to join strings together.

```
>>> s1 = "Hello"
>>> s2 = "World"
>>> s1 + s2
'HelloWorld'
```

The `*` operator is called the replication operator and is used to replicate a string.

```
>>> s = "Hello"
>>> s * 3
'HelloHelloHello'
```

These operators also have precedence associated with them:

```
>>> s = "This is a string"
>>> s + ' ' * 3
'This is a string '
>>> (s + ' ') * 3
'This is a string This is a string This is a string '
```

There is also another special operator called the `in` operator. This will return a Boolean based on whether a substring is contained within a string.

```
>>> s = "This is a string"
>>> "This" in s
True
>>> "That" in s
False
```

## 6.8 - Printing & formatting

We looked at the `print` function a little while back. It is quite simple in how it works. Takes something as an argument, tries to convert it to a string then prints it to standard output (We'll look at standard output a little later, for now though, it's just the terminal window).

Sometimes we want our output to look nice and have better readability. Let's look at an example with bad readability.

```
1. i = 0
2. while i < 12:
3. print(i, '* 15 = ', i * 15)
4. i += 1
5.
6. # OUTPUT
7. 0 * 15 = 0
8. 1 * 15 = 15
9. 2 * 15 = 30
10. 3 * 15 = 45
11. 4 * 15 = 60
12. 5 * 15 = 75
13. 6 * 15 = 90
14. 7 * 15 = 105
15. 8 * 15 = 120
16. 9 * 15 = 135
17. 10 * 15 = 150
18. 11 * 15 = 165
```

We can see that as the numbers become larger in the output, the spacing becomes off and lines start to jut out. We will look at how to make this look nice a little later in this section.

To achieve better formatting on output we use the `format()` operator. This operator *prepares* a string for printing.

The syntax is as follow: `'{}'.format(x)`.

The string on which `format` operates (`'{}'`) is called the *format string*. The `{}` is called a placeholder or (replacement field). Inside the placeholders we specify how data should be displayed and the arguments to the format method are

the items of data that will be inserted into each placeholder and formatted according to the placeholder's *format commands*.

The general syntax for a *format command* is `{[: <align> <minimum width> <.precision> <type>]}` where square brackets indicate optional parameters.

Let's look at how this in action:

```
>>> pi = 3.14159265359
>>> print('{:.3f}'.format(pi))
'3.142'
```

Let's break down the format command here (`{:.3f}`).

The `.3` is the precision. In this case we want pi to be formatted to 3 decimal places. The `f` is the type which stands for floating point type.

We can have multiple placeholders in the format string:

```
>>> pi = 3.14159265359
>>> e = 2.71828182845
>>> print('The number pi is: {:.3f} and the number e is {:.5f}'.format(pi, e))
'The number pi is: 3.142 and the number e is 2.71828'
```

The first argument to be formatted is matched with the first placeholder and the second argument to the second placeholder and so on.

We can also have nested placeholders:

```
1. e = 2.71828
2.
3. i = 0
4. while i < 6:
5. print('{:.{}f}'.format(e, i))
6. i += 1
7.
8. # OUTPUT
9. 3
10. 2.7
11. 2.72
12. 2.718
13. 2.7183
14. 2.71828
```

Nested placeholders are matched from left to right as well. A way of helping to think about this is, each time we encounter an opening curly brace, we will be inserting the next argument.

Another option we have to the format command is the minimum width.

We can use this minimum width to solve the problem we had with the output from the 15 times tables.



```

1. i = 0
2. while i < 12:
3. print('{:2d} * {:2d} = {:3d}'.format(i, 15, i*15))
4. i += 1
5.
6. # OUTPUT
7. 0 * 15 = 0
8. 1 * 15 = 15
9. 2 * 15 = 30
10. 3 * 15 = 45
11. 4 * 15 = 60
12. 5 * 15 = 75
13. 6 * 15 = 90
14. 7 * 15 = 105
15. 8 * 15 = 120
16. 9 * 15 = 135
17. 10 * 15 = 150
18. 11 * 15 = 165

```

The minimum width will pad out the argument with whitespace to the specified minimum width.

Notice that we have `d` instead of `f`. This means an integer. We also do not specify a precision which is indicated by a `.` followed by the precision number.

We can also specify an alignment. Alignments are specified with `^` for centred, `<` for left justified and `>` for right justified.

Let's look at the star example, this is by no means a good solution, it's simply to show the use of the alignment format command:

```

1. i = 0
2. while i < 6:
3. print('{:^10s}'.format('* '*i))
4. i += 1
5.
6. i -= 2
7. while i > 0:
8. print('{:^10s}'.format('* '*i))
9. i -= 1
10.
11. # OUTPUT
12. *
13. * *
14. * * *
15. * * * *
16. * * * * *
17. * * * * *
18. * * *
19. * *
20. *

```

As you can see in the format command, we specify the string to be centred with a minimum width of 10 and the type to be `s` which is a string.

## 6.9 - Exercises

### IMPORTANT NOTE:

I can't show you everything there is to python. The book would go on forever. A really important skill all good programmers have is being able to know what to look for when they run into a problem, they can't solve themselves. For example, I might want to be able to do something with a string, but I don't know how. Knowing where to look to find out the answer is super important. You may have even come across a website called *stack overflow* by now. This is going to be a really good resource for you. Therefore, some of the questions in these exercises may require you to go and search for methods to help you arrive at your solution.

There is a lot to take in from this chapter. Strings are important and used all the time. Becoming comfortable with them is equally important.

### Question 1

Write a program that takes as input, a single integer from the user which will specify how many decimal places the number `e` should be formatted to.

Take `e` to be 2.7182818284590452353602874713527

```
EXAMPLE INPUT
4
EXAMPLE OUTPUT
2.7183
```

### Question 2

Write a program that will take as input, a string and two integers. The two integers will represent indices. If the string can be sliced using the two indices, then print the sliced string. If either or both integers are outside the string's index range, then print that the string cannot be sliced at those integers.

Assume that the integers cannot be negative.

```
EXAMPLE INPUT
"This is my string"
2
9
EXAMPLE OUTPUT
```

```
'is is m'

EXAMPLE INPUT
"This is a string"
10
22

EXAMPLE OUTPUT
'Cannot slice string using those indices'
```

### Question 3

When you sign up for accounts on website or apps, you may be told your password strength when entering it for the first time. In this exercise, you are to write a program that takes in as input, a string that will represent a password.

Assume a password can contain the following:

- digits
- lowercase letters
- uppercase letters
- special characters (take these to be: \$, #, @, and ^)

A passwords strength should be graded on how many of the above categories are contained in the password. The password should be given a score of 1 to 4.

If the password is greater than or equal to strength 3 (contains characters from 3 of the above categories) then you should print the strength and that the password is valid. Otherwise print the strength and that the password is not valid.

Python comes with built-in documentation. To access this, at the command prompt type `pydoc str`. This will return the documentation for the `str` type. Here you will find all methods strings have to offer.

```
EXAMPLE INPUTS
978
hjj
jKl
nmM2
r@num978LL
LLLL

EXAMPLE OUTPUTS
1
```

1  
2  
3  
4  
1

## Question 4

Write a program that takes 3 floating point numbers as input from the user: a starting radius, radius increment and an ending radius. Based on these three numbers, your program should output a table with a spheres corresponding surface area and volume.

The surface area and volume of a sphere are given by the following formulae:

$$A = 4\pi r^2$$

$$V = \frac{4}{3}\pi r^3$$

Hint: Formatting is key here

```
EXAMPLE INPUT
1
1
10
EXAMPLE OUTPUT
```

Radius	Area	Volume
1.0	12.57	4.19
2.0	50.27	33.51
3.0	113.10	113.10
4.0	201.06	268.08
5.0	314.16	523.60
6.0	452.39	904.78
7.0	615.75	1436.76
8.0	804.25	2144.66
9.0	1017.88	3053.63
10.0	1256.64	4188.79

# Chapter 7 - Linear Search

## 7.1 - What is linear search?

In this chapter you're going to code your first algorithm. We'll take a look at the linear search algorithm.

Linear search (or sequential search) is a method to find an element in something. That something may be a string, a file or a list. The element we are searching for could be anything (a letter, a number, a word, etc).

(Usually) we are searching for the position of the element we want. Therefore, in some cases, the element we are searching for may not exist and we need our algorithm to be able to handle that.

Linear search is commonly used in computer science and is an easy algorithm to understand and write and it's perfect for us at this stage.

## 7.2 Linear search using while

In technical terms, we are searching for the position of some element Q that satisfies some property P.

In the general case, we can achieve this as follows:

```
i = 0
while i < N and not P:
 i += 1
```

What we want is the position of the element Q that satisfies some property P so we continue to search as long as that property P is not satisfied and we stop when it is.

Let's look at an example to help clarify. In this example we want to find the position of the first occurrence of the letter "W":

```
1. s = "Hello World"
2.
3. i = 0
4. while i < len(s) and s[i] != "W":
5. i += 1
```

What we do here is, begin at index 0 ("H") and continue through the string, checking each index until "W" is found and that is linear search.

Now we have another problem. There are two conditions in the while loop. Either "W" is found, and we exit the loop or we search the entire string and "W" is not found. How do we know which condition caused the loop to terminate?

If condition 1 (`i < len(s)`) is `True` then "W" was found as we clearly didn't reach the end of the string OR condition 1 is `False` in which case we did reach the end of the string and "W" was not found.

We can check this simply:

```
1. s = "Hello World"
2.
3. i = 0
4. while i < len(s) and s[i] != "W":
5. i += 1
6.
7. if i < len(s):
8. print("The letter 'W' was found at position " + str(i) + " in the string")
9. else:
10. print("The letter 'W' was not found in the string")
```

This is usually how linear search works; however, we can have more complicated versions which we will get to in the next section.

## 7.3 - Examples

Important Note:

If you have some previous experience of programming, you may be wondering why I'm using while loops to do this? I'm doing it this way to promote computational thinking and, in the future,, we will move away from this approach.

In this section I'm going to go through some more examples of linear search using while loops, some of which may get a little complicated so spend some time going through them.

For our first example, let's recreate the `lstrip()` string method.

```
1. s = input()
2.
3. i = 0
4. while i < len(s) and s[i] == " ":
5. i += 1
6.
7. if i < len(s):
8. print(s[i:])
9. else:
10. print("No alpha-numeric characters exist in this string")
```

The above program will strip all leading white space characters.

Let's look at a more difficult example. For this problem we want the user to input a string and print out that string, removing all leading and trailing whitespace and only one white space character between each word.

```
INPUT

" This is a string with lots of whitespace "

OUTPUT

"This is a string with lots of whitespace"
```

This is difficult to solve and can remember being given this problem when I was learning to program. To solve this problem, we'll need to nest while loops inside each other.

```
1. s = input()
2. output = ""
3.
4. i = 0
5. while i < len(s):
6. # Find a non whitespace character (start of a word)
7. while i < len(s) and s[i] == " ":
8. i += 1
9.
10. j = i
11. if i < len(s):
12. # Find the end of that word
13. while j < len(s) and s[j] != " ":
14. j += 1
15.
16. # Build up a string from the original without the excess whitespace
17. output += " " + s[i:j]
18.
19. i = j + 1
20.
21.
22. # Print out the final string
23. print(output[1:])
```

I'm not going to explain this, I want you to work through it. Take a simple input string to help you walk through the program.

## 7.4 - Exercises

Important Note:

The solutions to some of these exercises require thinking outside-the-box. Don't expect to arrive at a solution immediately!

## Question 1

Write a program that takes a string from the user and prints the first number encountered along with its position.

You can assume the number is a single digit

```
EXAMPLE INPUT
"This is chapter 7 of the book"
```

```
EXAMPLE OUTPUT
"7 16"
```

```
EXAMPLE INPUT
"There are no digits here"
```

```
#EXAMPLE OUTPUT
"No digits in string"
```

## Question 2

Building upon the previous exercise, write a program that takes a string as input from the user and prints the first number encountered along with the starting position of the number.

Working through this question on paper would be a good idea!

```
EXAMPLE INPUT
"This chapter has 1208 words in it!"
```

```
EXAMPLE OUTPUT
"1208 17"
```

```
EXAMPLE INPUT
"There is no numbers here"
```



```
EXAMPLE OUTPUT
```

```
"No numbers in string"
```

### Question 3

Write a program that takes a string as input from the user. The string will consist of digits and your program should print out the first repdigit. A repdigit is a number where all digits in the number are the same. Examples of repdigits are 22, 77, 2222, 99999, 444444. You can also assume that each number will have differing digits unless it is a repdigit.

```
EXAMPLE INPUT
```

```
"34 74 86 34576 47093 3 349852 777 9082"
```

```
EXAMPLE OUTPUT
```

```
"777"
```

```
EXAMPLE INPUT
```

```
"98 3462 245 87658"
```

```
EXAMPLE OUTPUT
```

```
"No repdigit in string"
```

### Question 4

If you open a browser and go to any web page that has text on it and press Ctrl+f, a search box will pop up. You can type any word in this search box and all occurrences of that word on the page are highlighted. This doesn't use linear search; it uses something called regular expressions (don't worry about what they are). As well as the occurrences of the word being highlighted, you are also told how many times the word occurs.

You are to write a program which mimics this behaviour. Your program should take a string as input from the user and then a second string as input which is the word the user wants to search for. Your program should print out how many times the word occurs in the string.

Assume that it still counts if the word you're searching for is embedded in another word, for example if the user was searching for the word "search":

search and searching <- you can count this as 2 occurrences

```
EXAMPLE INPUT
```

```
"In this type of search, a sequential search is made over all items one by one"
```

```
"search"
```

```
EXAMPLE OUTPUT
```

```
2
```

```
EXAMPLE INPUT
```

```
"There wont be an occurrence here"
```

```
"python"
```

```
EXAMPLE OUTPUT
```

```
0
```

# Chapter 8 - Lists

## Important Note:

Just a heads up, there's a lot to cover in this chapter and it can get quite technical in places (especially section 8.4) so pay particular attention to that section, it's really important!

## 8.1 - List literals

Lists are, on the surface, the simplest data structure Python has to offer. They are also one of the most important and are used to store elements.

A list is a sequence of elements in which the elements can be of any type (integers, floats, strings, even other lists).

Lists are of type `list`

The most basic kind of list is the *empty list* and is represented as `[]`. The empty list is *falsey*

```
>>> empty_list = []
>>> empty_list == False
True
```

In a list, elements are separated by commas.

```
my_list = [1, 2, 3, 654, 7]
```

This list contains 5 elements.

We can mix the types within the list, for example:

```
my_list = [True, 88, 3.14, "Hello", ['Another list', 45938]]
```

Notice how the last *element* is another list

In Python, a list is a *collection type* meaning it can contain several objects (strings, ints, etc..) yet still be treated as a single object.

Lists are also a *sequence type* meaning each object in the collection occupies a specific numbered location within it (which means elements are ordered).

Lists are also *iterable* so we can loop through their elements using indices.

They sound a lot like strings, don't they?

However, they differ in two significant ways:

- A list can contain objects of different types while a string can only contain characters
- A list is a *mutable* type (We'll get to this later).

## 8.2 - List methods and functions

We can get the length of a list in the same way we do with strings, using the `len()` function.

```
>>> my_list = [23, 543, "hi"]
>>> len(my_list)
3
```

We also have three other quite important functions, these are `sum()`, `min()` and `max()` and they work as you'd expect.

```
>>> my_list = [1, 2, 3, 4, 5, 6]
>>> sum(my_list)
21
```

`min()` and `max()` also work on strings, as strings have a lexicographical ordering, 'a' being the min and 'z' being the max:

```
>>> my_list = [6, 324, 456, 2, 6574, -452]
>>> max(my_list)
6574
>>> min(my_list)
-452
>>> min("string")
'g'
>>> max("string")
't'
```

We can also sort a list using the `sorted()` function. The `sorted()` function works by converting a *collection* to a list and returning the sorted list.

```
>>> student_grades = [88, 23, 56, 75, 34, 23, 84, 63, 52, 77, 96]
>>> sorted(student_grades)
[23, 23, 34, 52, 56, 63, 75, 77, 84, 88, 96]
```

In computer science, *searching* and *sorting* are two big problems and are important topics.

We can *append* and *pop* elements to and from the end of a list. Append is add and pop is remove. We can do this because lists are *mutable*.

```
>>> my_list = [1, 5, 8]
>>> my_list.append(99)
>>> my_list
[1, 5, 8, 99]
```

```
>>> my_list.pop()
99
>>> my_list
[1, 5, 8]
>>> my_list.pop(2)
5
>>> my_list
[1, 8]
```

Notice that when we pop, we are *returned* the number we popped. By default, if no arguments are passed to `pop()` then the last element is popped. We can also pass an index to `pop()` and it will pop the element at that index and *return* it to us.

We can store the value that is returned after calling `pop()` in another variable

```
>>> my_list = [1, 5, 9]
>>> x = my_list.pop()
>>> x
9
```

If we have a list of strings or characters, we can join them together using the `join()` function:

```
>>> my_list = ["O", "r", "a", "n", "g", "e", "s"]
>>> "".join(my_list)
'Oranges'
```

The string before the join function is called the *joining string* and is what will be between each element of the list when joining them together. In this case we want nothing (the empty string).

However, if we had words, we might want to join them together with spaces:

```
>>> my_list = ["Hello", "World.", "This", "is", "a", "string."]
>>> " ".join(my_list)
'Hello World. This is a string.'
```

### Side Note:

We can break a string into a list using the `split()` function:

```
>>> s = "This is my string"
>>> my_list = s.split()
>>> my_list
['This', 'is', 'my', 'string']
```

The `split()` function takes an argument which is the character we want to split the string on. If no argument is passed, then it defaults to splitting on whitespace.

## 8.3 - List indexing

As we could with strings, we can also index into lists the same way.

```
>>> my_list = [3, 5.64, "Hello", True]
>>> my_list[2]
"Hello"
>>> my_list[0]
3
```

I've talked about how we can have other lists as elements inside lists. In Python and many other languages, lists inside lists are useful for representing many types of data (matrices, images, spreadsheets and even higher dimensional spaces). These are typically referred to as *multidimensional arrays* or *multidimensional lists*.

We can index into these inner lists too!

```
>>> my_list = [[1, 2, 3], [7, 8, 9]]
>>> my_list[0][1]
2
```

This works as follows, we first select the embedded list we want then we select the element from that list. In the above example, we want the list at index 0 ([1, 2, 3]) then we select the element at index 1 (2).

We can 'burrow' down into embedded lists this way.

```
>>> my_list = [[[1, 2, 3], [4, 5, 6]], [7, 8, 9]]
>>> my_list[0][1][1]
5
```

We can extend this to strings inside lists

```
>>> my_list = ["My string", "Another string"]
>>> my_list[0][1]
'y'
```

## 8.4 - References & Mutability

We looked at immutable types earlier on. Now we're going to look at mutable types.

A mutable type (or mutable sequence) is one that can be changed after it is created. A list is a mutable sequence and can therefore be changed in place.

Let's refresh our memory on how strings couldn't be changed after they were created

```
>>> s = "my string"
```

```
>>> s[0] = "b"
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

We get an error. Take a look back at the chapter on strings (Chapter 6) and revise the diagram that exists in that chapter.

I'm going to introduce the `is` keyword now. The `is` keyword compares object IDs (identities). This is similar to `==` which checks if the objects referred to by the variables have the same content (equality).

To give an analogy, let's say a clothing company is mass producing a certain type of shirt. Let's take two shirts as they are coming off the line.

We would say `shirt_1 == shirt_2` evaluates to `True` (They are both the same color, size and have the same graphic printed on them)

We would say `shirt_1 is shirt_2` evaluates to `False` (Even though they look similar, they are not the same shirt).

Mutability and Immutability is quite like that.

Let's look at some more Python examples:

```
>>> s = "Hello"
>>> t = s
>>> t is s
True
>>> t == s
True
```

These variables, `s` and `t` are referencing the same **object** (The string "Hello"). What if we try to update `s`?

```
>>> s = "Hello"
>>> t = s
>>> t is s
True
>>> s = "World"
>>> t
"Hello"
```

The variable `s` is now referencing a different object and `t` still references "Hello". As strings are immutable we must create a new *instance*. We cannot change it *in place*.

With lists however, things operate a little differently.

```
>>> a = [1, 3, 7]
>>> b = a
```

```

>>> b is a
True
>>> a.append(9)
>>> a
[1, 3, 7, 9]
>>> b
[1, 3, 7, 9]
>>> a is b
True

```

The behavior has changed here and that is because the object pointed to by `a` and `b` is *mutable*. Modifying it does **not** create a new object and the original reference is **not** overwritten to point to a new object. We do not overwrite the reference in the variable `a`, instead we write **through** it to modify the mutable object it points to.

There are however some little tricks thrown in with this behavior. Consider the following code:

```

>>> a = [1, 3, 7]
>>> b = a
>>> a = a + [9]
>>> a
[1, 3, 7, 9]
>>> b
[1, 3, 7]

```

What's going on here? That goes against what we just talked about doesn't it? Well, obviously not, the Python developers wouldn't leave a bug that big in language. What's happening here is we executed the following code: `a = a + [9]`. This doesn't write *through* `a` to modify the list it references. Instead a new list is created from the *concatenation* of the list `a` and the list `[9]`.

A reference to this **new** list overwrites the original reference in `a`. The list referenced by `b` is hence unchanged.

That's not all the little tricks thrown in. There's one more subtlety. Consider the following code:

```

>>> a = [1, 3, 7]
>>> b = a
>>> a += [9]
>>> a
[1, 3, 7, 9]
>>> b
[1, 3, 7, 9]

```

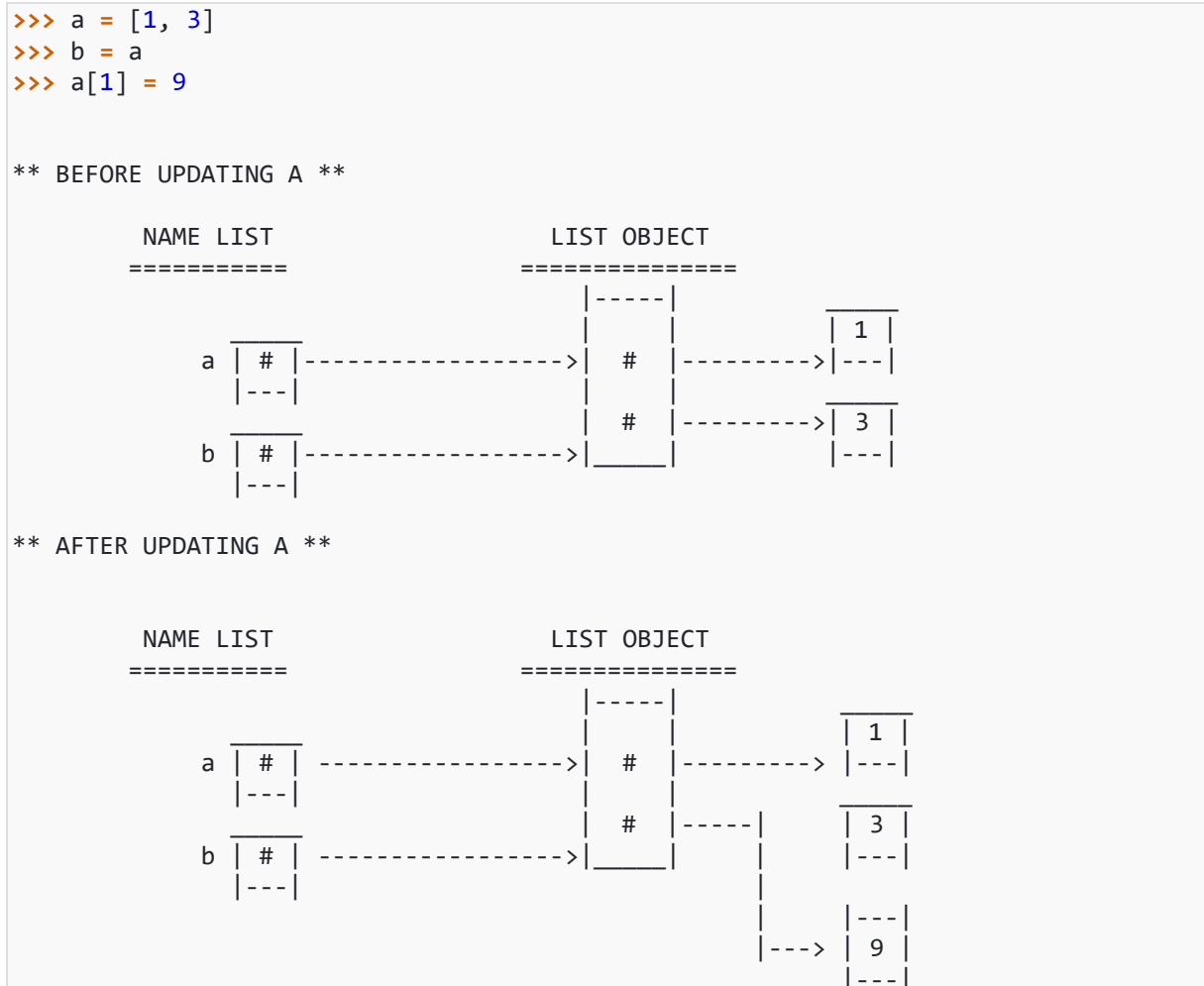
It turns out `x += y` isn't always shorthand for `x = x + y`. Although with immutable types they do the same thing, with lists they behave very differently. It turns out the `+=` operator when applied to lists, modifies the



list **in-place**. This means we write *through* the reference in `a` and append `[9]` to the list.

**Writing through references** is illustrated below.

Firstly, consider the code, then the diagram:



So what's happening in the second part of this diagram? The `#`'s represent references.

`a` is a reference to a list object which contains references to integers.

`b` references the same list object. But when we "change" `a`, we're clearly not changing it. We're updating a reference within it. While `b` just points to the list, `a` has changed what's inside it hence `b`'s reference is not affected.

We can still see the `3` floating around in the second part of the diagram. This will be picked up by something called the *garbage collector* (A garbage

collector keeps memory clean and stops it from becoming flooded with stuff like the unreferenced 3).

It's important to keep a mental image of this.

## 8.5 - List operations

We've seen two operations (Now knowing they're different) in the previous section. The `+` and `+=` operators. I'll briefly run over them again.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4, 5, 6]
>>> a += b
>>> a
[1, 2, 3, 4, 5, 6]
```

We also have the `*` operator, which does as expected:

```
>>> a = [1, 2, 3]
>>> a * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

We also have a new operator. This is the `in` operator. The `in` operator is a *membership test* operator and returns `True` if a sequence with a specified value is in an object

```
>>> a = [3, 7, 22]
>>> 4 in a
False
>>> 3 in a
True
```

The `in` operator can be applied to any *iterable* type (lists, strings, etc).

## 8.6 - List slicing

List slicing works exactly how it does with strings. This makes sense as both strings and lists are *sequence* types).

```
>>> a = [1, 2, 3, "Hello", True]
>>> a[3:]
["Hello", True]
>>> a[-4: -2]
[2, 3]
```

Extended slicing also works exactly the same as it does with strings.

```
>>> a = [1, 2, 3, "Hello", True]
>>> a[::-1]
[True, 'Hello', 3, 2, 1]
```

```
>>> a[::2]
[1, 3, True]
```

### Important:

Remember in section 8.4 we wrote something like:

```
>>> a = [1, 2, 3]
>>> b = a
```

If we updated `a`, then `b` would also be effected. What if we wanted to make a copy of `a` and store it in `b`? We can use slicing to do that as slicing returns a new object

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b is a
False
```

We can see now they do not reference the same object so updating one will not affect the other.

## 8.7 - Exercises

**Important Note:** We've learned a lot up to this point. Your solutions should be combining (where necessary) everything we've learned so far. If you are unsure of something, don't be afraid to go back to previous chapters. It's completely normal not to remember everything this early on. The more practice you have the more of this stuff will stick!

### Question 1

Write a program that takes a string as input from the user, followed by a number and output each word which has a length greater than or equal to the number.

The string will be a series of words such as "apple orange pear grape melon lemon"

You are to use the `split()` function to split the string

```
EXAMPLE INPUT
"elephant cat dog mouse bear wolf lion horse"
5

EXAMPLE OUTPUT
elephant
mouse
```

```
horse
```

You may use the following code at the beginning of your program

```
my_words = input().split()
num = int(input())

YOUR CODE HERE
REMEMBER: my_words is a List
```

## Question 2

Write a program that takes a string as input from the user, followed by another string (the suffix) and output each word that ends in the suffix.

The first string will be a series of words such as "apples oranges pears grapes lemons melons"

You are to use the `split()` function to split the string

```
EXAMPLE INPUT
"apples oranges pears grapes lemons melons"
"es"

EXAMPLE OUTPUT
apples
oranges
grapes
```

Hint: Your solution should be generalized for any suffix. Don't assume the length of the suffix will be 2

You may use the following code at the beginning of your program

```
words = input().split()
suffix = input()

YOUR CODE HERE
REMEMBER: words is a List
```

## Question 3

Write a program that builds a list of integers from user input. You should stop when the user enters 0.

Your program should print out the list.

```
EXAMPLE INPUT
3
5
6
7
0

EXAMPLE OUTPUT
```

```
[3, 5, 6, 7]
```

## Question 4

Building on your solution to question 3, taking a list built up from user input, ask the user for two more pieces of input. Integers this time.

The integers will represent indices. You are to swap the elements at the specified indices with each other.

```
EXAMPLE INPUT
6
3
9
0

* LIST SHOULD NOW BE: [6, 3, 9] *

1
2

EXAMPLE OUTPUT
[6, 9, 3]
```

You may assume that the indices will be within the index range of the list.

## Question 5

Write a program that builds a list of integers from user input. Your program should then find the smallest of those integers and put it at the first index (0) in the list. Input ends when the user inputs 0.

```
EXAMPLE INPUT
10
87
11
5
65
342
12
0

LIST SHOULD NOW BE: [10, 87, 11, 5, 65, 342, 12]

EXAMPLE OUTPUT
[5, 87, 11, 10, 65, 342, 12]
```

This problem is a little harder, think about your solution!

## Question 6

### **\*\* THIS IS A HARD PROBLEM \*\***

Write a program that takes two *sorted lists* as input from the user and merge them together. The resulting list should also be sorted. Both of the input lists will be in increasing numerical order, the resulting list should be also.

```
EXAMPLE INPUT
2
6
34
90
0 # END OF FIRST INPUT
5
34
34
77
98
0 # END OF SECOND INPUT

EXAMPLE OUTPUT
[2, 5, 6, 34, 34, 34, 77, 90, 98]
```

Don't assume both lists will be the same length!

# Chapter 9 - Basic Sorting Algorithms

## 9.1 - What is sorting?

We've learned a lot so far and its now time to put what we've learned to good use. We're now going to look at your first important algorithms.

Sorting is the process of arranging items systematically. In computer science, sorting refers to arranging items (whatever they may be) in an ordered sequence.

Sorting is a very common operation in applications and developing efficient algorithms to perform sorting is becoming an ever more important task as the amount of data we gather grows at an exponential rate.

Sorting is usually used to support making lookup or search efficient, to enable processing of data in a defined order possible and to make the merging of sequences efficient.


In this chapter we're going to look at two of these sorting algorithms: Selection sort and Insertion sort.

## 9.2 - Selection sort

Selection sort is a general-purpose sorting algorithm. In order for selection sort to work we must assume there exists some sequence of elements that are order-able (integers, floats, etc.).

Selection sort is an in-place sorting algorithm which means we don't build a new list, rather, we rearrange the elements of that list.

Before we begin, we need to look at some terminology:

- Sorted subarray - The portion of the list that has been sorted
- Unsorted subarray - The portion of the list that has not yet been sorted
-  - The list we are sorting

In selection sort, we break the list into two parts, the sorted subarray and the unsorted subarray and all of the elements in the sorted subarray are less than or equal to all the elements in the unsorted subarray. We also begin with the assumption that the entire list is unsorted.

Next, we search the entire list for the position of the smallest element. We must search the entire list to be certain that we have found the smallest element. If there are multiple occurrences of the smallest element, we take the position of the first one. We then move that element into the correct position of the sorted subarray. We repeat the above steps on the unsorted subarray

This is illustrated below

```

6 3 9 7 2 8
|||=====| # FIND THE POSITION OF THE SMALLEST
 UNSORTED PART # ELEMENT IN THE LIST AND SWAP WITH
6

--

2 3 9 7 6 8 # 2 IS NOW IN THE CORRECT POSTION
|===||=====| # FIND THE POSITION OF THE SMALLEST
 UNSORTED PART # ELEMENT IN THE LIST AND SWAP WITH
3
 # 3 IS THE SMALLEST (NO SWAP)

--

2 3 9 7 6 8 # 3 IS NOW IN THE CORRECT POSITION
|=====||=====| # FIND THE POSITION OF THE SMALLEST
 SORTED UNSORTED PART # ELEMENT IN THE LIST AND SWAP WITH
9
 PART

--

2 3 6 7 9 8 # 6 IS NOW IN THE CORRECT POSITION
|=====||=====| # FIND THE POSITION OF THE SMALLEST

```



```

7 SORTED PART UNSORTED # ELEMENT IN THE LIST AND SWAP WITH
 # 7 IS THE SMALLEST (NO SWAP)

--

2 3 6 7 9 8 # 7 IS NOW IN THE CORRECT POSITION
|=====||=====| # FIND THE POSITION OF THE SMALLEST
 SORTED PART UNSORTED # ELEMENT IN THE LIST AND SWAP WITH
9

--

2 3 6 7 8 9 # 8 IS NOW IN THE CORRECT POSITION
|=====||| # WE HAVE REACHED THE END OF THE
LIST
 SORTED PART # THE LIST IS NOW SORTED

```

Does this seem somewhat familiar? If you did the exercises from the previous chapter, exercise 5 was basically this process!

It's time to code selection sort! Let's look at the code for it below:

```

1. i = 0
2. while i < len(a):
3. p = i
4. j = i + 1
5. while j < len(a):
6. if a[j] < a[p]:
7. p = j
8. j += 1
9.
10. tmp = a[p]
11. a[p] = a[i]
12. a[i] = tmp
13.
14. i += 1

```

That's it! Let's break this down as there is a lot going on here

- The outer while loop is controlling our sorted subarray.
- The inner while loop is searching for the next smallest element

- The three lines above `i += 1` are swapping the smallest element into the correct position.
- The process then repeats.

A skill that all programmers have is being able to step through code and figure out how it's working. I encourage you to do the same with this algorithm (pen and paper and walking through an example)

Another good way to figure out what's going in the middle of some code is to stick in a `print()` statement to print out what variables hold what values at that current time. Another good way is to set break points. I'm not going to show you how to do this as it's generally a feature of the text editor you're using so look up how to set break points for your editor and how to use them! It's a skill you'd be expected to have if you worked in this field.

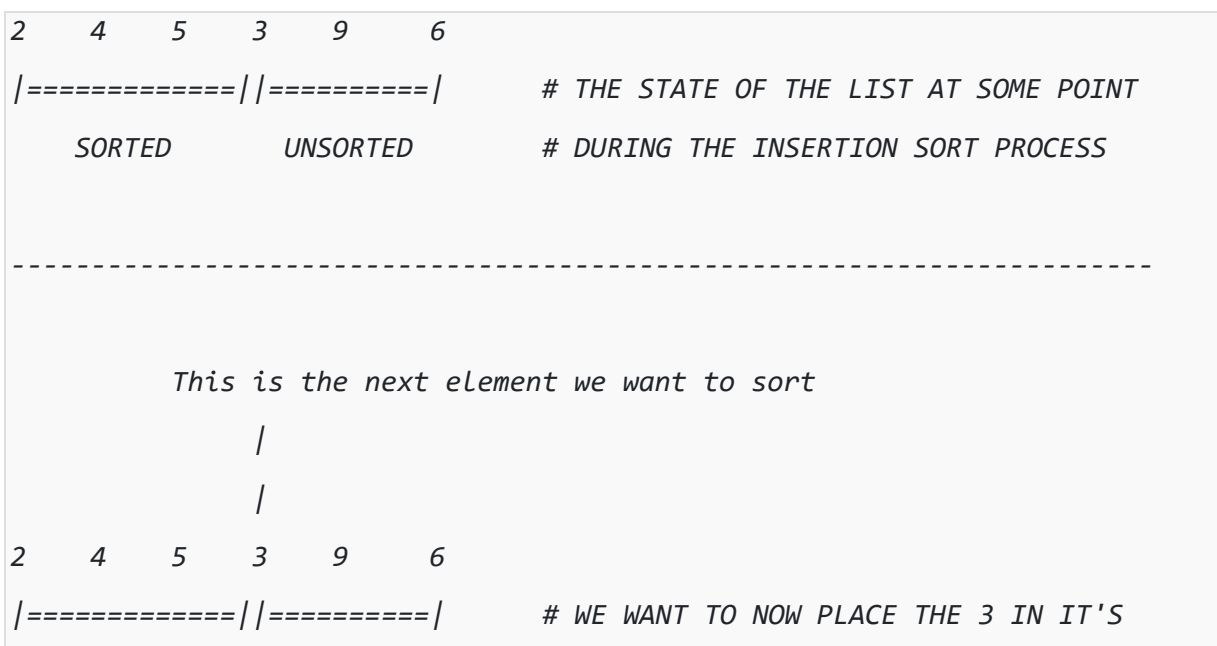
## 9.3 - Insertion sort

Insertion sort is another general purpose, in place sorting algorithm.

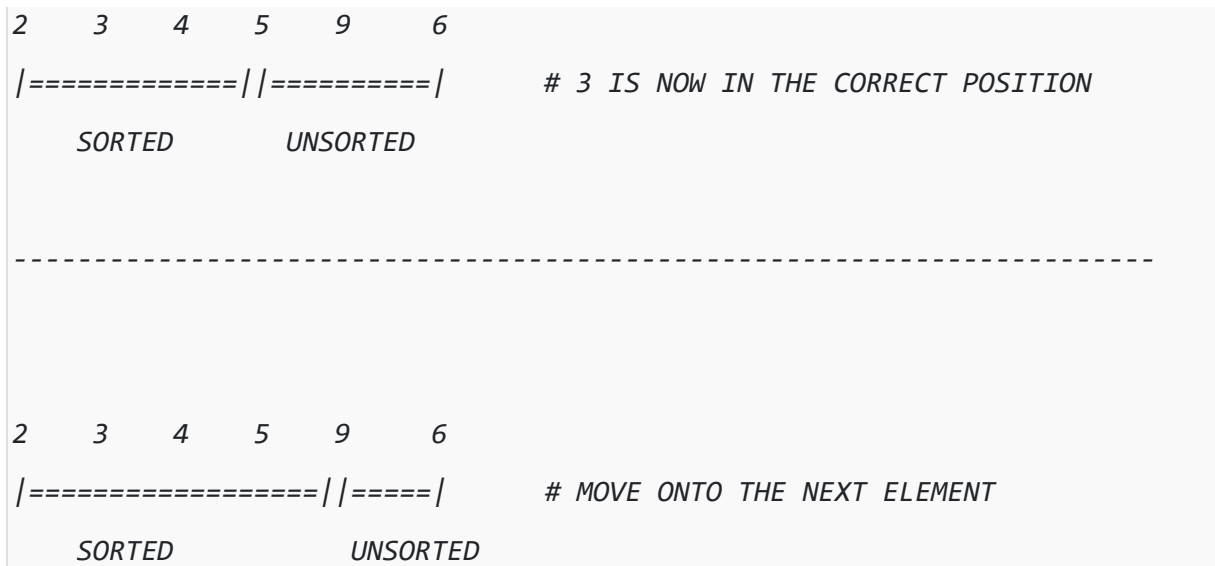
To contrast how insertion sort works compared to selection sort:

- Selection Sort: Select the smallest element from the unsorted subarray and append it to the sorted subarray of the list.
- Insertion Sort: Take the next element in the unsorted subarray and insert it into its correct position in the sorted subarray.

I'll illustrate this with a semi complete example:



SORTED			UNSORTED			# CORRECT POSITION
<hr/>						
			3			
2	4	5	_	9	6	
=====			=====			# TAKE 3 OUT OF THE LIST
SORTED			UNSORTED			
<hr/>						
			3			
2	4	5	_	9	6	
=====			=====			# IS 3 < 5? YES, SO MOVE 5 UP
SORTED			UNSORTED			
<hr/>						
			3			
2	4	_	5	9	6	
=====			=====			# IS 3 < 4? YES, SO MOVE 4 UP
SORTED			UNSORTED			
<hr/>						
			3			
2	_	4	5	9	6	
=====			=====			# IS 3 < 2? NO, PLACE AFTER THE 2
SORTED			UNSORTED			
<hr/>						



It's time to code insertion sort! Let's take a look at the code for it, I'll give the explanation as comments:

```

1. i = 1 # Assume the first element (a[0]) is sorted
2. while i < len(a): # Same as with selection sort
3. v = a[i] # The value we want to insert into the correct position
4. p = i # The position the element should be inserted into to
5. while p > 0 and v < a[p-1]: # While value is < element to left
6. a[p] = a[p-1] # Move the element to left up
7. p -= 1 # Decrement p (move one position left)
8. a[p] = v # Found correct position so insert the value
9.
10. i += 1 # Move to next element

```

## 9.4 - Comparison of Selection sort and Insertion sort

In this section I'm going to talk about the algorithmic complexity of the two algorithms. This is a really important section. If you ever do a technical interview, in 99.9% of cases you'll be asked to write some code or an algorithm and give it's algorithmic complexity. You need to know this stuff and it isn't just for Python, this applies to any language! I'm going to explain this in the simplest way possible while still giving you a good understanding of the topic.

Selection sort and Insertion sort are what we call quadratic sorting algorithms.

This means they both have a big O time complexity of:

$O(n^2)$

Time complexity refers to the time it takes for the algorithm to complete.

There are generally three categories:

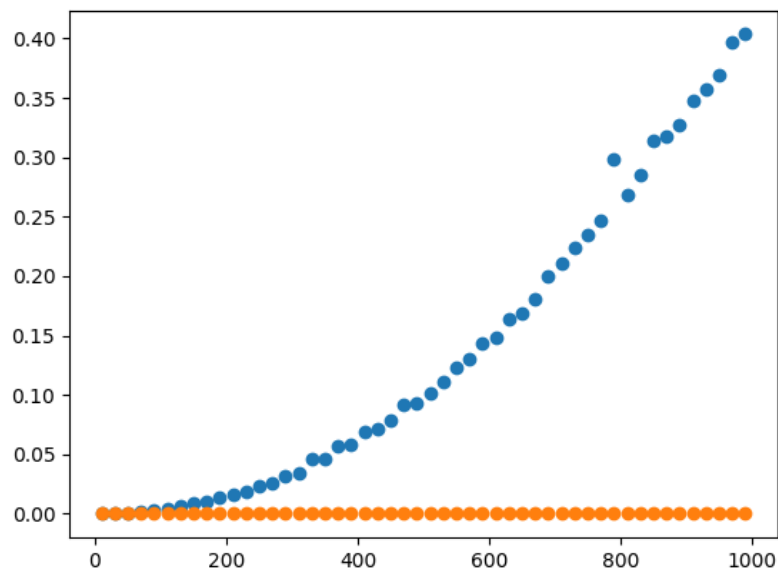
- Best case scenario - For example with insertion sort, let's say our list is already sorted and we try to sort it, then we are in a best case scenario as we don't need to move any elements around and the algorithm finishes quickly.
- Average case: This is how the algorithm performs in terms of time on average
- Worst case scenario: This is how the algorithm would perform in terms of time if our list was completely scrambled and out of order.

Big O deals with the worst case and it is the case we are usually concerned with!

There's a lot of maths behind this and there's a whole topic of computer science related to it so I'm not covering it in detail here.

Anyway, when we say that both of these algorithms have a time complexity of  $O(n^2)$ , we are essentially saying that the performance of the algorithms will grow proportionally to the square of the size of the input (lists in this case). Another way to think of  $O(n^2)$  is that if we double the size of our input, it will quadruple the time taken for the algorithm to complete. These are just estimates. It doesn't mean that two algorithms with time complexities of  $O(n^2)$  will both take the exact same time to complete. You can look at this as a guide as to how your algorithm will perform in a general sense.

I've taken the time to run some algorithms with different time complexities on lists of varying sizes and we can look at how the algorithm takes longer to complete as the size of the list grows. You can see this in the diagram below:



## Time complexity

In this diagram, the y-axis represents how long it took the algorithm to finish in seconds and the x-axis shows the number of elements in the list.

The blue line is for an  $O(n^2)$  algorithm and the orange is for an  $O(n)$  algorithm (Linear time algorithm).

I have a fast computer, so it doesn't appear that the orange line is changing (it is, just very slowly).

However, it is very clear that the  $O(n^2)$  algorithms become very slow when the size of the input becomes large.

In the real world, a list of size 1000 is small and selection sort or insertion sort wouldn't cut it. However, that doesn't mean they don't have their uses. In fact, in practice, selection sort and insertion sort outperform the faster algorithms on small lists and some of the fast sorting algorithms will actually switch to these  $O(n^2)$  algorithms when they are nearing the end of the sorting process. It turns out the whole process finishes faster when this is done (in some cases).

We also have something called space complexity and this deals with how much memory is taken up by an algorithm. Both of these algorithms have  $O(1)$  space complexity. This means they use constant memory. They use constant memory as they are in-place sorting algorithms. They don't create additional lists to assist during the sorting process.

There is usually a trade-off between time and space complexity. As you can see here, we have constant memory (This is good) but quadratic time complexity (This is bad). We could write some algorithm that is faster but takes more memory. It depends on the problem and the computing resources we have.

## 9.5 - Exercises

There won't be any coding exercises in this chapter, just theory questions. They are just as important to understand and get right though!

### Question 1

What is the time complexity of:

1. Insertion sort
2. Selection sort

Give your answer in Big O notation

### Question 2

What is the space complexity of:

1. Selection sort
2. Insertion sort

Give your answer in Big O notation

### Question 3

Give two examples of when we might use selection sort or insertion sort in the real world?

### Question 4

If we ended up in a situation in which the input was sorted but we didn't know and we try to sort the input, why might we prefer insertion sort over selection sort?

You may have to go and search for the answer to this.

# Input, File & Text Processing

## 10.1 - What is file & text processing?

File processing is the process of creating, storing and accessing a files content. Up until now we've only been able to temporarily store data in our programs in objects such as lists. In the real world, this wouldn't be much use. We need some way of being able to save data permanently. To do this, our programs need to store data on a hard disk. This is persistent storage and the data in this storage will be able to survive a system reboot or our program crashing. If we reboot our system, RAM is cleared and we lose our data. We've been storing data in RAM until now.

When doing file processing we follow these steps:

- Open the file
- Read and/or write to the file
- Close the file

Up until now we've been using `input()` and `print()` for input and output (I/O). These are high-level interfaces to the underlying operating-system services and sometimes we need finer control, especially when we want to read and write files.

Python implements file objects as general interfaces to data streams. These file objects allow us to read and write to and from standard input and standard output (this is what `input()` and `print()` do and we'll get to standard input and output later). The file objects also allow us to read and write files.

Text processing on the other hand is the processes of manipulating textual data. Text processing is a very common task in computing. Python has great support for text processing and we've even met some of the methods that Python provides to allow us to do this. These are `split()`, `strip()` and `join()` to name a few.

## 10.2 - The Sys module

The sys module provides information about constants, functions and methods of the Python interpreter. A module is a file consisting of Python code. A



module can define functions, classes and variables. To access a module you must import it into your code using the `import` keyword.

```
import sys
```

```
YOUR CODE
```

Through the `sys` module we can access arguments that are passed to our script at the command line. Lots of Python scripts need access to these arguments. We get access to them via `argv` (`sys.argv`).

`argv` is shorthand for argument vector. This is a list containing the command-line arguments passed to the script. The first element in this list is the script itself. The arguments for the script come after the name of the script. Let's look at how this works. Create a new Python file and save it as `myscript.py` and enter the following code:

```
import sys
```

```
print(sys.argv)
```

Then, at the command-line type the following, then Enter:

```
1$ py myscript.py arg1 arg2
```

If you are on linux or macOS, type: `python3 myscript.py arg1 arg2`

The `$` at the beginning is the command prompt, don't type this.

You will get the following output:

```
$ py myscript.py arg1 arg2
```

```
['myscript.py', 'arg1', 'arg2']
```

Let's use this in a more constructive way. Create a new file called `calculator.py` and paste in the following code:

```
1. import sys
2.
3. total = 0
4.
5. i = 1
6. while i < len(sys.argv):
7. total += int(sys.argv[i])
8. i += 1
9. print(total)
```

At the command line type the following, then Enter:

```
$ py calculator.py 5 3 6
14
```

We can pass the numbers we want to add together to the script. This saves us having to ask the user for input each time we want new input. Remember though, `argv` is a list of strings and that's why we had to cast the arguments to integers in the above code.

The `sys` module also provides us with three standard file objects. These are: `stdin`, `stdout`, and `stderr`. These stand for standard input, standard output and standard error.

In the sections that follow we will look at each of them.

## 10.3 - Reading from standard input

Standard input is stream data (usually text) going into a program. In Python, we access standard input through the `sys` module which provides standard input (`stdin`) as a file object.

We have various methods available to us as to how we read from standard input.

Important: The following methods, unlike `input()` and `print()`, will never add or remove newline characters. We will have to handle newline characters ourselves.

The first is the `read()` method. It is the most basic of the three. `read()` will read the entire contents of the file and the entire contents will be assigned to a single string.

```
12345import sys

contents = sys.stdin.read()

print(contents)
```

Important: We can pass a file to our program using a command-line operator called the input redirection operator. When we use this operator, we redirect input from our keyboard to the text file.

The syntax for this method is `file.read()` where `sys.stdin` is a file (file object here).

You should create a text file and fill it up with some text. I recommend you have about 10 lines of text. Each line can be a single word if you like.

To run the above code, type the following at the command line. Make sure the text file and the Python script are in the same directory. Type the following, then Enter.

```
$ py myscript.py < input.txt
This is line one.
I'm line two
And I'm line three
```

This works the same on Linux, Windows and Mac.

This is the output we get. Yours will look different depending on what you have in your file. As you can see we've printed out that file you just created.

It reads the file in as a single string and the string looks like so:

```
"This is line one.\nI'm line two\nAnd I'm line three"
```

The second method we have available to us is the `readlines()` method. This is a little more complicated as it stores each line as an element in a list.

```
1. import sys
2.
3. contents = sys.stdin.readlines()
4.
5. print(contents)
```

The syntax for this method is `file.readlines()` where `sys.stdin` is a file (file object here).

Now we run our script the same way as before:

```
$ py myscript.py < input.txt
['This is line one.\n', 'I'm line two.\n', 'And I'm line three']
```

As you can see our text file is now stored in a list, with each line being an element in the list. Notice how the newline characters haven't been removed.

This method is good as we can now do some manipulation on each line of text more easily.

Rather than using the input redirection operator we can just read lines from terminal.

You can run the above script again, this time omitting the redirection operator.

Important: With files we eventually reach the end of them. This is indicated in the file by EOF (end of file). We don't need to worry about how this works as it's handled at a lower level. When we do not redirect standard input to read from a file, we do however, need indicate EOF at the command line. On Windows this is done by pressing ctrl+z. On Linux this is indicated by pressing ctrl+d. When you run the above script again and omit the redirect, you will be prompted to continuously input lines of text. When you are finished press ctrl+z or ctrl+d.

```
$ py myscript.py
line one
line two
final line # PRESS ENTER, THEN CTRL+Z OR CTRL+D TO INDICATE EOF.
['line one\n', 'line two\n', 'final line\n']
```

The third method available to us is the `readline()` method. This is arguably better than `read()` and `readlines()` as it only reads a single line from standard input. With the two previous methods, we could imagine a situation where a text file is really, really big and storing the entire contents of the file in memory might be a waste of valuable memory.

We can still read in many lines of input using this method, we'll just need a loop. Let's look at how that is done.

```
1. import sys
2.
3. line = sys.stdin.readline()
4. while line:
5. print(line.strip()) # Strip the newline character as print() will add one.
6. line = sys.stdin.readline()
```

The syntax for this method is `file.readline()` where `sys.stdin` is a file (file object here).

Running this while redirecting standard input to a file yields:

```
$ py myscript.py < input.txt
This is line one.
```

```
I'm line two.
And I'm line three
```

I want to take another look at the `read()` method. We can actually control how much of the input is read. If we redirect input to a file, we can limit how much of the file we read at any given time. Similarly, if we don't redirect, we can limit how much of user input through the console we read. We do this by passing an integer as an argument to the method. This integer represents how many characters we want to read.

This is done as follows:

```
1. import sys
2.
3. contents = sys.stdin.read(6)
4.
5. print(contents)
```

Running the code and redirecting standard input to a text file would yield:

```
$ py myscript.py < input.txt
This i
```

As you can see, we passed the integer 6 to the read function, telling it to only read 6 characters.

Remember: Whitespace counts as a character and so does the newline character!

Get used to these methods! This is how we'll be the primary way of taking input into our functions from now on!

## 10.4 - Writing to standard output

Standard output refers to the streams of data that are produced by programs. These standardized streams make it very easy to send output from these programs to a devices' display monitor, printer, etc.

We write to standard output by using the `write()` method on the `sys.stdout` file object. The behaviour of `write()` has changed since Python 2.x. In Python 3, when we write to standard output, we are returned the number of characters that were written.

Let's look at that in action in the Python REPL:

```
>>> import sys
>>> sys.stdout.write("Hello")
Hello5
>>>
```

We can see that we wrote "Hello" to standard output and we were returned the number of characters that were written (5). The reason the 5 is on the same line is because, unlike `print()`, we must append the newline character ourselves:

```
>>> import sys
>>> sys.stdout.write("Hello\n")
Hello
5
>>>
```

We can also redirect the standard output stream to a file using the output redirection operator (`>`). Let's look at how this is done.

```
1. import sys
2.
3. sys.stdout.write("Hello World!\n")
```

Then at the command-line run:

```
1$ py myscript.py > output.txt
```

If you check the directory where your script is stored, you should now see a file called `output.txt`.

Open it and view its contents, it should contain what we just wrote to the output stream.

When we redirect standard output to a file, a file will be created if it doesn't exist already.

We also have the `writelines()` method available to us which is not all that dissimilar to `readlines()` except it writes instead of reads.

```
1. import sys
2.
3. my_lines = ["Line one\n", "Line two"]
4. sys.stdout.writelines(my_lines)
```

Then, at the command-line, run:

```
1$ py myscript.py > output.txt
```

If you open output.txt you'll find that it contains the strings in the list above on two separate lines.

## 10.5 - Standard error

There is still one standard stream we have not covered yet and that's standard error (*stderr*). It again is a file object in python so we can write to it. It is typical for programs to write error messages and diagnostics to standard error. Typically, standard error is outputted to the terminal which started the program.

Having the standard output stream solves the semi predicate problem by allowing output and errors to be distinguished. We can redirect standard error to a different output stream to standard output. (an error log file for example).

That's all I want to say on standard error. We won't have to worry about it again in this book. It's just good to know about all the standard data streams we have access to.

## 10.6 - Opening & reading files

In this section we're going to look at open and reading from files that are stored in permanent storage. Although the standard data streams from the previous section are files, they were stored in RAM and were already open.

When reading a file, we initialize a file object that acts as a link from the program to the file stored on the disk.

To open a file we use the *open()* function. The *open()* function takes two arguments: The filename and the mode in which the file is to be opened. The syntax is: *open(<filename>, <mode>)*

There are various modes in which we can open a file, we're only concerned with four of them here.

The four modes we'll be dealing with are:

1. *'r'* - This mode opens a file for reading only.

2. `'w'` - This mode opens a file for writing. If the file doesn't exist, it creates a new file with the specified file name. If the file does exist and we write to it, anything that was previously in the file is overwritten.
3. `'a'` - This opens a file in append mode. If the file doesn't exist, it creates a new file with the specified file name. If it does exist, we add to the file rather than overwriting it.
4. `'x'` - This opens a file for exclusive creation, failing if it already exists and throwing a `FileExistsError`.

Once the file is open, we can read its contents using the methods from section 3.4: `read()`, `readlines()` and `readline()`.

Let's look at an example of opening a file from the disk and reading in its contents:

```
1. my_file = open('input.txt', 'r')
2. content = my_file.readlines()
3. print(content)
```

Make sure you have a file called "input.txt" saved in the same directory as your script and for demo purposes make sure it has a few lines of text in it.

At the command-line, run:

```
$ py myscript.py
['This is line one.\n', "I'm line two.\n", "And I'm line three"]
```

You can now see that the contents of your file have been read in successfully. It is in list form as we used the `readlines()` method.

## 10.7 - Writing to files & closing files

In the previous section we looked at opening files. Now let's look at how to write to them and close them.

As you may have guessed, we write to files using the `write()` method we met in previous sections. To write to a file though, it must be opened in write mode.

Let's look at how to write to file:

```
1. my_file = open('output.txt', 'w')
2. my_file.write("I am being written to a file\n")
```

Run this script:



```
1$ py myscript.py
```

Now open the file called output.txt. Don't worry if it didn't exist, one will be created. You should now see the line "I am being written to a file" contained in that file.

Now we need to close the file.

When a file is opened, the operating system allocates memory to track that file's state and the operating system cannot reallocate that memory until the file is closed. If we don't close open files, then we are using up memory unnecessarily.

When your program exists, the file is closed automatically but maybe we don't want to exit straight after we are finished reading and writing to a file.

We close files by using the `close()` function. This is demonstrated below:

```
1. my_file = open("output.txt", "w")
2. my_file.write("Hello\n")
3. my_file.close()
```

The file has now been closed and memory can be reallocated for other uses.

Having to open files and remember to close them can become a bit of a pain. We can use something called the `with` statement to handle this for us.

The `with` statement clarifies code that previously would use `try...finally` blocks to ensure that clean-up code is executed. Don't worry about `try` and `finally` we'll get to them a little later but for now, we can use `with` to handle the closing of files for us.

```
1. with open("data.txt", "w") as my_file:
2. my_file.write("Some text here")
3.
4. # THE REST OF YOUR PROGRAM HERE
```

So, what is going on here? We can see the `with` statement, but we also have an `as` statement. The `as` statement is used to assign the returned object to a new identifier. In this case, we open a file and are returned a file object. The file object is then assigned to the variable `my_file` using the `as` statement.

We then do whatever we need to do. In this case we write to the file. When we exit the `with` block the file is closed. Opening and closing files this way, is usually how most people do it.

Another use of the `as` statement which may clarify things is, for example, we were importing the `sys` module and we didn't like how long the name "`sys`" was (I know this is a silly example but some modules have long names), we could import it as such:

```
1. import sys as s
2. content = s.stdin.read()
```

Now, every time we want to refer to the `sys` module, we call it by its alias, `s`.

## 10.8 - Putting is all together

We've learned a lot in this chapter so let's look at an example that puts everything together.

Let's assume we have a text file that contains many lines. Each line has a student's name followed by the mark they received for a specific class. Here is a sample from that text file:

```
Liam 84
Noah 33
William 43
James 37
Logan 59
```

What we want to do is process this file and output to a new file whether each student passed or failed the class.

The output should look like this:

```
Liam PASS
Noah FAIL
William PASS
James FAIL
Logan PASS
```

We want to pass the input filename and output filename as arguments to the script and we take a fail to be a mark less than 40.

Here's how we might do that:

```
1. import sys
2.
3. src = sys.argv[1] # The input source file
4. dst = sys.argv[2] # The file to output to
5.
6. with open(src, 'r') as fin, open(dst, 'a') as fout: # Open in append mode
7.
8. student = fin.readline().strip() # Strip the newline character
9. while student:
10.
11. student_data = student.split() # ['Liam', '84'] for example
12.
13. name = student_data[0]
14. mark = int(student_data[1])
15.
16. if mark < 40:
17. grade = "FAIL"
18. else:
19. grade = "PASS"
20.
21. fout.write('{:s} {:s}\n'.format(name, grade)) # Write to the output fi
22. le
23. student = fin.readline().strip() # Get the next student
```

Notice how we can open multiple files at the same time!

We run this as follows:

```
$ py grades.py input.txt output.txt
```

Your output should now be:

```
Liam PASS
Noah FAIL
William PASS
James FAIL
Logan PASS
```

File processing is a common task, as I've said, so become used to it, you'll probably be doing it a lot!

## 10.9 - Exercises

Important Note: These questions tougher than previous ones and get considerably more difficult as they go on, especially question 4. Don't be discouraged, however. Stick at them. If you manage to complete these 4

questions, you're on your way to becoming a great developer and problem solver!

## Question 1

Write a program that multiplies an arbitrary number of command-line arguments together. The arguments will be numbers (convert them first).

Your program should be run as follows:

```
$ py calc.py 5 99 32
```

To test that your solution is correct, you should get the following output

```
$ py calc.py 8 8 9 3 2
3456
```

## Question 2

Write a program that reads in lines from standard input (no redirection) and outputs them to standard output (no redirection).

## Question 3

Write a program that reads a text file and outputs (using `print()` is fine) how many words are contained within that file. The name of the text file should be passed as a command-line argument to your script.

Your program shouldn't consider blank lines as words and you need not worry about punctuation. For example, you can consider `"trust,"` to be a single word.

Your program should be run as follows:

```
1$ py num_words.py input.txt
```

To test that your solution is correct, use the [Second Inaugural Address of Abraham Lincoln](#) as your input text and your program should output: 701

## Question 4

Write a program that reads in the contents of a file. Each line will contain a word or phrase. Your program should output (using `print()` is fine) whether or not each line is a palindrome or not, "True" or "False". The name of the text file should be passed as a command-line argument to your script.

A palindrome is a word or phrase that is the same backwards as it is forwards. For example, "racecar" is a palindrome while "AddE" is not.

Your program should not be case sensitive: "Racecar" should still be considered a palindrome. Spaces should NOT affect your program either. Consider removing white space characters.

Your program should be run as follows:

```
$ py palindrome.py input.txt
```

To test that your solution is correct, use the following as your input text:

```
racecar
AddE
HmLLpH
Was it a car or a cat I saw
Hannah
T can arise in context where language is played wit
Able was I ere I saw Elba
Doc note I dissent A fast never prevents a fatness I diet on cod
```

Using the above input, your output should be:

```
True
False
False
True
True
False
True
True
```

## Question 5

**\*\* THIS QUESTION IS HARD \*\***

Like question 3, write a program that reads a text file. This time your program should output how many unique words are contained within the file.

This time you do need to care about punctuation and your solution should not be case sensitive. For example, `"trust"` and `"trust,"` and `"Trust"` should be considered the same word, therefore only one occurrence of that word should be recorded as unique and if you were to come across `"trust"` again, then don't record it.

Your program should be run as follows:

```
$ py unique_words.py input.txt
```

To test if your solution is correct, use the [Second Inaugural Address of Abraham Lincoln](#) as your input text and your program should output: 343

Hint 1: Take a look at the `string` module. In particular, `string.punctuation`. If you import this, you may be able to use it to your advantage. Import it as follows `from string import punctuation`. Figure out how `string.punctuation` works.

Hint 2: The solution to this exercise may make use of some string methods that we looked at back in the strings chapter.

Hint 3: Making use of a second list might be a good idea!

# Chapter 11 - for loops & Dictionaries

## 11.1 - for loops

So far, we have been using `while` loops to iterate over lists and strings. I purposely chose to not mention `for` loops yet as using while loops help with computational thinking in the beginning. However, we have had enough practice and if you completed the final question from the last chapter, you're more than ready to move on.

In this chapter we'll be looking at `for` loops.

`for` loops are used when you have a block of code which we need to execute a known number of times (kind of similar to our do-something-n-times loop pattern). When using `for` loops we assume we don't need to check for some condition, and we need to iterate over every value in the collection or sequence.

The `for` loop makes use of the `in` operator. The `in` operator checks for the presence of something in some collection.

The syntax for the `for` loop is as follows:

```
for <loop_var> in <collection>:
 # Do something
```

I'll show you this working in an example and then walk through how it works:

```
1. names = ["Tony", "John", "Jim"]
2. for name in names:
3. print(name)
```

Running this in the terminal will give:

```
$ py myscript.py

Tony

John

Jim
```

In the above example, `name` is the loop variable. We can pick any variable name for this and I have chosen `name` as it is appropriate as we are looping through `names`.

The loop variable is a temporary variable and its role is to store a given element for the duration of the given iteration. Let's step through our example and see how this works.

1. We pick the first element from the list and check if it exists
2. It does, so we assign it to a variable called `name`.
3. We enter the body of the loop and do any work we need to (printing the name in this case).
4. We go back to start of the loop and repeat steps 1 to 4.
5. We will get to a point when there is no "next element", in which case we exit the loop.

What if we were in a case where we didn't have something to iterate over but we wanted to execute a block of code a fixed number of times?

Python provides two built-in functions for doing this. They are the `range()` function and the `xrange()` function.

The `range()` function takes three parameters, a start and stop and a step. The start and step are optional. The `range()` function builds a sequence based on these parameters.

Let's look at it in action:

```
1. for i in range(5, 10):
2. print(i)
```

Running this would give us:

```
$ py myscript.py
5
6
7
8
9
```

As you can see, we start at 5 and go up to but not including the `10`. If we omit the start parameter, then it defaults to 0 and I'm sure you can guess at this point what the step parameter does from your knowledge of it in extended slicing.



The range function has built up a sequence from 5 to 9 and we iterate over that using the for loop.

## 11.2 - Dictionaries

So far, we've met lots of different types. We have also met Lists and Strings. These two types are examples of data structures. In this section we're going to look at another Python built-in data structure called the dictionary. It is of type `dict`.

Dictionaries are collection types but not sequences. That is, its elements aren't ordered like they are in a list or string.

Dictionaries are sometimes referred to as maps or hashmaps.

A dictionary is a collection of key-value pairs. A dictionary implements a mapping from a key to a value.

This is illustrated below:

KEYS	VALUES
/	/
/ "Tony" - ----- -> "457-2344356" /	/
/	/
/ "Adam" - ----- -> "359-5550983" /	/
/	/
/-----/	/-----/

In this example, we implement a phone book using a dictionary. If we look up the key `"Tony"`, we are led to the value `"457-2344356"`.

The underlying implementation of a dictionary is designed in such a way that given a key; we can retrieve the associated value with great efficiency.

The Big O complexity for the lookup operation is  $O(1)$ . It is constant. It doesn't matter how large the dictionary is, the time for searching for a value is always the same. Magic, right? Take it as magic, I won't be going into how it does so in this book as it's a little too advanced for beginners, but you're more than welcome to look up how it's done.

We also do not know the order in which key-value pairs are stored in a Python dictionary so don't write programs that rely on its order. If you have a dictionary that is the same each time you run your program, the order will be different each time.

The syntax for building a dictionary is as follows:

```
my_dictionary = {<key1> : <value1>, <key2> : <value2>, ..., <keyN> : <valueN>}
```

For the above phone book example, that would be done as follows:

```
phonebook = {"Tony": "457-2344356", "Adam" : "359-5550983"}
```

 Note the syntax for dictionaries uses curly braces rather than square brackets.

With dictionaries, a key can be any immutable type. The values can be of any type, including other dictionaries.

The empty dictionary is: `d = {}`.

## 11.3 - Dictionary accessing

We can index access a dictionary by keys in order to retrieve values.

This is done as follows:

```
>>> phonebook = {"Tony": "457-2344356", "Adam" : "359-5550983"}
>>> phonebook["Tony"]
'457-2344356'
```

If we try to access a dictionary based on a key that doesn't exist within the dictionary, we get an error. This error is known as a `KeyError` in Python.

```
>>> phonebook = {"Tony": "457-2344356", "Adam" : "359-5550983"}
>>> phonebook["Bob"]
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
KeyError: 'Bob'
```

Remember, dictionaries are not sequenced types so we cannot access by position. We can however use integers as keys but even then, they are just that; integers.

```
>>> my_dict = {1: "one", 3: "three", 2: "two"}
```

```
>>> my_dict[3]
'three'
>>> my_dict[0]
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
KeyError: 0
```

If we have a key-value pair in which the value is a list, we can index the list as follows:

```
>>> my_dict = {"first": [2, 4, 8]}
>>> my_dict["first"][0]
2
```

## 11.4 - Adding and removing entries

We can also add new entries into an existing dictionary. When we add a new entry, we are adding a new mapping of a key-value pair into the dictionary.

The syntax for doing this is:

```
dictionary[<key>] = <value>
```

Let's look at that in action

```
>>> my_dict = {}
>>> my_dict["one"] = 1
>>> my_dict
{"one":1}
```

As we can add entries to the dictionary without a new dictionary being created every time, dictionaries are mutable types.

Remember that the keys must be a mutable type and values can be of any type. This will catch a lot of people out.

We can remove entries from the dictionary by using the `del` statement. This is done as follows:

```
>>> phonebook = {"Tony": "457-2344356", "Adam" : "359-5550983"}
>>> del phonebook["Tony"]
```

In the above example we are saying, delete the entry from the dictionary whose key is *"Tony"*.

## 11.5 - Dictionary operators & methods

We can get the length of a dictionary the same way we do with strings and lists:

```
>>> phonebook = {"Tony": "457-2344356", "Adam" : "359-5550983"}
>>> len(phonebook)
2
```

This returns the number of key-value pairs stored in the dictionary.

We can use the *in* operator to test whether or not a key exists within the dictionary

```
1. phonebook = {"Tony": "457-2344356", "Adam" : "359-5550983"}
2. if "Tony" in phonebook:
3. print(phonebook["Tony"])
4. else:
5. print("Key doesn't exist")
```

The reason I introduced dictionaries in this chapter is because we use *for* loops to cycle through each of a dictionary's keys.

```
1. phonebook = {"Bill": "457-2344356", "Adam" : "359-5550983"}
2.
3. for name in phonebook:
4. print("{}'s number is {}".format(name, phonebook[name]))
```

The above code would produce the following output:

```
Bill's number is 457-2344356.
```

```
Adam's number is 359-5550983.
```

We can retrieve a list of a dictionary's keys or values by using either the *keys()* method or *values()* method respectively.

```
>>> phonebook = {"Bill": "457-2344356", "Adam" : "359-5550983"}
>>> phonebook.keys()
dict_keys(['Bill', 'Adam'])
>>> phonebook.values()
dict_values(['457-2344356', '359-5550983'])
```

These aren't really lists as we can't index them but we can iterate over them using a `for` loop.

I want to briefly introduce another data structure. It's called the tuple. I won't go into detail on it just yet but it's basically an immutable list. We can index into it but can't change it (much like a string).

The reason I need to let you know about it here is because we can retrieve a list of tuples using the dictionary method `items()`.

This works as follows:

```
>>> my_dict = {"one": 1, "two": 2}
>>> my_dict.items()
dict_items([('two', 2), ('one', 1)])
```

We can see the tuples have the form `(e1, e2, ..., en)` where `e1...en` are elements.

Much like a list with parenthesis instead of square brackets.

We can iterate over these too. This works as follows:

```
my_dict = {"one": 1, "two": 2}

for (k, v) in my_dict.items():
 print("Key: {} and Value: {}".format(k, v))
```

Remember, `dict.items()` returns a 'list' of tuples with each tuple containing two elements, the key and the value. When we say `for (k, v) in my_dict.items()` we are essentially saying for each tuple in the list of tuples, assign `k` to the element in the first position in the tuple and `v` to the element in the second position. I'll illustrate this below:

<code>(k,</code>	<code>v)</code>
<code> </code>	<code> </code>
<code>v</code>	<code>v</code>
<code>("one",</code>	<code>1)</code>

`k` is assigned to `"one"` and `v` is assigned to `1`.

Running the above code gives the following output:

```
Key: two and Value: 2
```

```
Key: one and Value: 1
```

Notice how they're not in the same order as they appear when we created the dictionary.

## 11.6 - Sorting dictionaries

We can sort a dictionary based on the values of its keys using the `sorted` method. Let's look at how this can be done:

```
phonebook = {"Bill": "457-2344356", "Adam" : "359-5550983"}

for (k, v) in sorted(phonebook.items()):
 print("Key: {} and Value: {}".format(k, v))
```

Running this would give the following output:

```
Key: Adam and Value: 359-5550983
Key: Bill and Value: 457-2344356
```

As you can see, the dictionary has been sorted based on the keys. Since the keys are strings, they are sorted lexicographically.

We can also sort on values, however, we don't have the knowledge to do that yet. We'll come back to that in the functions chapter.

We can however reverse the order they are sorted in. The `sorted()` function has an optional parameter called `reverse`. We can set this to `True` and the dictionary will be sorted in reverse order based on the keys.

```
phonebook = {"Bill": "457-2344356", "Adam" : "359-5550983"}

for (k, v) in sorted(phonebook.items(), reverse=True):
 print("Key: {} and Value: {}".format(k, v))
```

This will give us the following output:

```
Key: Bill and Value: 457-2344356
Key: Adam and Value: 359-5550983
```

## 11.7 - Exercises

### Question 1

Write a program that reads a file. The file will contain a number of items stocked in a shop and the number of each item available. Your program should parse the file and build a dictionary from this it, then output the stock available in alphabetical order and in a specific format. You can use the following as your input text file:

```
Oranges 12
```

```
Apples 10
```

```
Pears 22
```

```
Milk 7
```

```
Water Bottles 33
```

```
Chocolate Bars 11
```

```
Energy Drinks 8
```

Your program output should look like this:

```
Apples : 10
```

```
Chocolate Bars : 11
```

```
Energy Drinks : 8
```

```
Milk : 7
```

```
Oranges : 12
```

```
Pears : 22
```

```
Water Bottles : 33
```

Hint: Find out the length of the longest key.

### Question 2

Write a program that reads a text file. The file is a list of contact details for people. With each contact you're given the name, phone number and email address. Your dictionary should be a mapping from contact names to email and phone number for that contact. Your program should then ask for user input. The input should be a single name. If the name can be found in the contact list then output the name and contact details for that person, otherwise output *No contact with that name.*

The contact list file is:

```
Liam 345-45643454 Liam@mymail.com
Noah 324-43576413 noah.doe@gmail.com
Tony 987-56543239 tony@hr.mycompany.com
Bert 654-99275234 bert.hertz@support.hertz.co.uk
```

If the user then enters the following names in this order:

```
Tony
Noah
John
Annie
Bert
```

Your program should give the following output:

```
Name: Tony
Email: tony@hr.mycompany.com
Phone: 987-56543239

Name: Noah
Email: noah.doe@gmail.com
Phone: 324-43576413

No contact with that name

No contact with that name

Name: Bert
Email: bert.hertz@support.hertz.co.uk
Phone: 654-99275234
```

These details should be printed one by one as the user enters the names, not in bulk as I have above.

### Question 3



In the previous chapter we tried to count how many words were contained within a piece of text. In this question you are to do the same except you should output how many times each word occurred in the text.

Again, punctuation matters. This time we want to strip any punctuation surrounding a word. Your program should not be case sensitive either, *Hello* and *hello* should count as the same word. However, something like *John* and *John's* should not be counted as the same word. In other words, you are stripping leading and trailing punctuation characters.

Your program should be run as follows:

```
$ py dict_count.py input.txt
```

To test that your solution is correct, use the [Second Inaugural Address of Abraham Lincoln](#) as your input text and your program should output (this is a sample from the out):

```
.
.
.
the : 58
oath : 1
of : 22
presidential : 1
office : 1
there : 2
is : 6
less : 2
occasion : 2
for : 9
an : 3
extended : 1
address : 2
than : 4
.
.

```

.

You need not format the output. The length of your dictionary should be 701.

## Question 4

Write a program that takes two dictionaries and outputs their intersection. The intersection of two dictionaries should be a third dictionary that contains key-value pairs that are present in both of the other two dictionaries. You can hard code these two dictionaries in. They don't need to be read from anywhere.

Run your script as follows:

```
$ py intersection.py
```

To test if your solution is correct, you can run your program with the two following dictionaries:

```
d1 = {"k1": True, "k2": True, "k3": True, "k4": True}
d2 = {"k6": True, "k2": True, "k5": True, "k1": True, "k8": True, "k4": True}
```

Your program should output the following dictionary:

```
intersection = {"k2": True, "k1": True, "k4": True}
```

# Chapter 12 - Functions & Modules

## 12.1 - What are functions?

In Python, a function is some code that takes input, performs some computation and produces an output. In programming, a function is a reusable block of code which performs a specific task.

We can call a function multiple times during a program's execution. We have done this many times throughout this book already.

Some of the common functions we have met are `len()` and `print()`. When we want to know the length of something we pass that something to the `len()` function for example.

When we want to print something to the terminal window we call the `print()` function.

We have also met methods. Methods are functions that are associated with an object. For example the `format()` method operates on strings. Methods are implemented the same way as functions and are essentially functions that operate on associated objects, but we'll get back to those later.

## 12.2 - Defining our own functions

Most of the time, the Python built-in functions are not enough. Let's take our linear search algorithm for example. We may want to search a string multiple times throughout our program. Until now we would have had to write out the linear search algorithm over and over again. Programmers don't like to repeat themselves so if they find themselves repeating the same piece of code, they'll put that code into a function. This means, whenever we want to use that piece of code we can just call it like we do with `print()` for example.

We define our functions using the `def` keyword. Let's take a look at a function that prints "Hello, World!" to the screen.

```
1. def hello():
2. print("Hello, World!")
```

Above we have defined, using `def`, a new function called `hello()`. When we call our function, the functions code block will execute, in this case we just

print `"Hello, World!"` to the screen. This function can be used as follows within our code:

```
1. def hello():
2. print("Hello, World!")
3.
4. hello()
```

The last line above calls our function. This should be familiar as you've called other functions many times now!

The syntax for defining functions is:

```
def func_name(arg1, arg2, ..., argn):
 # function code
 return result
```

I'll explain what `arg1, arg2, ..., argn` is in the next section but let's take a look at the `return` statement.

In our first function we didn't get a result back, it simply printed something to the screen. Let's write a function that will `return` the text "Hello, world!" to us.

```
1. def hello():
2. return "Hello, World!"
```

We can use this function as follows:

```
1. def hello():
2. return "Hello, World!"
3.
4. x = hello()
5. print(x)
6.
7. # ALTERNATIVELY
8. print(hello())
```

In this function, the string `Hello, World!` is handed back to the function caller. Since the function returns a value, its caller is expected to collect that value. Above we see the caller collects the returned value and stores it in `x`. You can see that the `print()` function also collects the value returned by the `hello()` function.

## 12.3 - Arguments & parameters

We've seen functions such as `len()` that take arguments such as a string or dictionary and return its length.

In the previous section we looked at the syntax for a function. In that, we had something like:

```
def myFunc(arg1, arg2, arg3):

#
```

The `arg1`, `arg2`, `arg3` are called the functions parameters. A function parameter is like a variable that is local to the function thus cannot be referenced outside of that function. Another name of a parameter is a formal parameter. If our function has one formal parameter, then we must pass a value to the function when we call it. This value is called an argument. Another name for an argument is an actual parameter.

Let's look at an example of a function that adds two numbers:

```
1. def add(x, y):
2. result = x + y
3. return result
4.
5. print(add(5, 7))
```

In the above function, `x` and `y` in the function definition are called the parameters.

When we call the function, we pass `5` and `7` to the function. These are the arguments that we supplied to the function.

The variable `result` is also local to the function and cannot be referenced from outside the function.

Arguments and parameters are, by default, matched based on position. In this example, the first argument (`5`) is copied into the first parameter (`x`).... and so on.

Earlier on we looked at a function that printed hello world. Functions that do not return any values are called procedures.

A procedure changes the state of our program. For example, they may update values of variables, print something to the screen or update a file.

Functions that return a value inspect the state of our program. They return a value based on their inspection.

Remember the `split()` method? I said if we don't pass any arguments to this method then it will split a string based on whitespace. This is called a default argument.

A default argument is a parameter that assumes a default value if one is not provided when the caller invokes the function.

Below is an example of a function with default arguments:

```
1. def print_circle_info(r, x=0, y=0):
2. print("(x, y) coordinates: {}, {}\nRadius: {}".format(x, y, r))
```

In the above function, we must pass a radius as an argument when calling it. We don't however have to pass an x or y coordinate. If we don't pass an x or y coordinate, our function defaults these values to 0.

If we do pass x and y coordinates, then the values we pass will be assigned to the parameters x and y.

We can also define functions that have a variable number of arguments.

```
1. def my_function(*argv):
2. for arg in argv:
3. print(arg)
4.
5. my_function("first", "second", "even a third")
```

The output from this calling this function would be:

```
first
second
even a third
```

In the above example, `argv` is some arbitrary name we chose as the parameter name. The `*` beforehand indicates that it is a variable length parameter.

We also have keyword variable length parameters.

```
1. def my_function(**kwargs):
2. for k, v in kwargs:
3. print("{} : {}".format(k, v))
4.
5. my_function(first="Hello", second="World")
```

The output would be as follows:

```
second : World
first : Hello
```

As you can probably tell, `kwargs` is a dictionary. We indicate that it is a keyword variable length parameter by the `**` before the variable name.

Both normal and keyword variable length parameters must be put at the end of the parameter list when defining your functions parameters. There is good reason for this. If we had `def func(x, y, *argv, z)` we wouldn't know where `*argv` ended.

I'm going to say, be cautious when using variable length parameters. They should only be used when you have a good reason to use them (and that isn't very often).

## 12.4 - Variable Scope

Not all variables are accessible from all parts of your program. The part of a program where a variable is accessible is called its scope. A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file and any file which imports that file.

Global variables can have unintended consequences because of their "range" (they can be accessed from anywhere). There are only very special circumstances in which we should use global variables in software we make in real life.

Variables which are defined inside code blocks are local to that block. A block is a construct that delimits the scope of any declaration within it. In Python, a variable defined inside a function is local to that function. It is accessible from when it is defined until the end of that function.

The formal parameters of a function act like local variables. However, assignments to a parameter can never affect the associated argument unless they are a mutable type, which you've seen in the previous section.

## 12.5 - Default mutable argument trap

Consider this code and its output:

```
1. def add_to_list(word, word_list=[]):
2. word_list.append(word)
3. return word_list
4.
5. def main():
6. word = "apple"
7. tlist = add_to_list(word)
8. print(tlist)
```

```

9. word = "orange"
10. tlist = add_to_list(word, ['pear'])
11. print(tlist)
12. word = "banana"
13. tlist = add_to_list(word)
14. print(tlist)
15.
16. main()

```

```

$ py trap.py
['apple']
['pear', 'orange']
['apple', 'banana']

```

Let's look at why this is strange behaviour. We see that the second parameter to `add_to_list()` is optional (indicated by the default value `[]`). If no argument for this parameter is supplied when calling the function then that parameter takes on the value of `[]`, the empty list.

The first time we call the function and pass it `apple`, we get back `['apple']`. This is fine and it's what we expected. The second time we call the function and pass it `orange` and a list of `['pear']`, we get back the list `['pear', 'orange']`. This is also fine and what we expected. The third time however, we get some unexpected behaviour.

Why is `apple` in the list if we called the function and passed `banana` and no list. Surely we should just get back `['banana']`? This is called the default mutable argument trap and no, it's not a bug.

The empty list is initialised only once by Python. It is initialised when the `def` for that function is first encountered. This means the list has memory and anything added to it will stay there.

The takeaway from this is to not use mutable types as default arguments. Instead, work around it as follows:

```

1. def add_to_list(word, word_list=None):
2. if word_list is None:
3. word_list = []
4. word_list.append(word)
5. return word_list
6.
7. def main():
8. word = "apple"
9. tlist = add_to_list(word)
10. print(tlist)
11. word = "orange"
12. tlist = add_to_list(word, ['pear'])
13. print(tlist)

```



```
14. word = "banana"
15. tlist = add_to_list(word)
16. print(tlist)
17.
18. main()
```

This will give:

```
['apple']
['pear', 'orange']
['banana']
```

And this is what we expected the first time.

`None` is a special type. It is an object that indicates no value and it is an immutable type. In fact, `None` is returned from a function if there is no return statement in that function i.e a procedure.

## 12.6 - What are modules?

I've briefly mentioned modules before and I said, a module is simply a file consisting of Python code. A module can define functions, classes and variables. A module can also contain runnable code.

Modules allow you to logically organize your Python code. Grouping code into modules makes it easier to understand and use your code. It is not uncommon for a software project to span hundreds of files.

You can import modules using the `import` statement which we have seen when importing `sys` and `string`.

We can import specific attributes from modules using the `from` keyword e.g. `from string import punctuation`. We can also import all attributes from a module using `*` e.g. `from string import *`.

When we import a module, the Python interpreter searches for the module in the following manner:

- The Python interpreter searches the current directory.
- If it's not there, then the interpreter searches each directory in the shell variable `PYTHONPATH`
- Finally, if not there then the interpreter searches the default path. On Linux, this is normally `/usr/local/lib/python`.

## 12.7 - Creating our own modules

We can create our own modules to logically organize our code. Let's say we want to create a module that contains some maths functions. We do this as follows:

```
1. # maths.py
2.
3. def add(x, y):
4. return x + y
5.
6. def multiply(x, y)
7. return x * y
8.
9. def main():
10. print(add(x, y))
11. print(multiply(x, y))
12.
13. if __name__ == "__main__":
14. main()
```

The `add()` and `multiply()` functions are fine and we have come across similar functions before. The `main()` function we have also come across in a previous section but the `if __name__ == "__main__":`, we have not come across before.

When you execute a program directly from the command line (as you've been doing), the Python interpreter sets a special variable for that script. That variable is `__name__`. When a script is executed directly, that variable is set to `"__main__"`, otherwise, if it is being imported for example, then it is set to the name of the module, `"math"` in this case.

We include the `if` statement to check this variable as sometimes we want to test our code or have some way of demoing our module. If we run it directly then the `main()` function is executed. Otherwise, if we import this module, then the `main()` function is not executed.

It is good practice to include this check in your scripts. This check is also usually located at the end of the script.

The way Python handles the `__name__` variable has changed in Python 3.7. It is now under [PEP 567](#). On the surface nothing has drastically changed.

## 12.8 - Exercises

When completing these exercises, use the following pattern when writing your scripts:

```
1. # functions and constants
2.
3. def main():
4. # Test code
5. # Anything else that shouldn't happen when your code is imported
6.
7. if __name__ == "__main__":
8. main()
```

If you are asked to write module, then you should save the module in the same directory as your testing script.

### Question 1

Write a module named `sorting.py` that contains functions that implement both selection and insertion sort i.e your module (`sorting.py`) should contain a `selection_sort(a)` function and an `insertion_sort(a)` function. Your function should return the values, not print them.

Your module should be imported and run as follows in another script called `test.py`:

```
1. import sorting
2.
3. a = [5, 6, 3, 8, 7, 2]
4. print(sorting.selection_sort(a))
5. a = [5, 6, 3, 8, 7, 2]
6. print(sorting.insertion_sort(a))
```

```
$ py test.py
```

```
[2, 3, 5, 6, 7, 8]
```

```
[2, 3, 5, 6, 7, 8]
```

### Question 2

Write a module called `arithmetic.py`. This module should include the following functions:

- `add()`
- `multiply()`

- `divide()`
- `subtract()`

The `add()`, `multiply()` and `subtract()` functions should be able to handle an arbitrary number of arguments. `divide()` should only take 2 arguments. Your function should return the calculated values, not print them.

Your module should be imported and run as follows in another script called `test.py`

```
1. from arithmetic import *
2.
3. print(add(4, 6, 7, 2, 3))
4. print(add(3, 2))
5.
6. print(subtract(5, 6, 8, 9))
7. print(subtract(3, 2))
8.
9. print(divide(6, 3))
10.
11. print(multiply(2, 3))
12. print(multiply(2, 3, 3))
```

```
$ py test.py
```

```
22
5
-18
1
2
6
18
```

### Question 3

Write a function called `length()` that mimics the `len()` function. Your function should work for strings, dictionaries and lists. `length()` should only take 1 argument and return the length of the data-structure passed to it.

Your function should be tested with the following:

```
1. a = [5, 3, 4, 1, 2, 3]
2. print(length(a))
3.
4. a = []
5. print(length(a))
6.
```

```

7. d = {}
8. print(length(d))
9.
10. d = {"one": True, "two": True}
11. print(length(d))
12.
13. s = "This is a string"
14. print(length(s))
15.
16. s = ""
17. print(length(s))

```

```
$ py test.py
```

```

6
0
0
2
16
0

```

## Question 4

Write a function called `fib()` that calculates and returns the n-th Fibonacci number. Your function should take 1 argument, the Fibonacci number you want to calculate.

Your function should be tested with the following:

```

1. print(fib(3))
2. print(fib(0))
3. print(fib(1))
4. print(fib(10))
5. print(fib(13))

```

```
$ py test.py
```

```

3
1
1
89
377

```

Remember:

$\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$

$\text{fib}(0) = 1$

fib(1)=1

## Question 5

Write a function called `read_file()` which takes a single argument, a filename (as a string), and returns the contents of the file in list form with each element in the list being a single line from the file.

Your function should be called as follows:

```
1. lines = read_file("input.txt")
```

## Question 6

Write a function procedure called `rep_all()` that takes 3 arguments, a list of integers and 2 numbers. Your procedure should replace all occurrences of the first number with the second.

Calling your function as follows and printing the list afterwards should produce the following:

```
1. a = [4, 2, 3, 3, 7, 8]
2. rep_all(a, 3, 10)
3. print(a)
```

```
$ py test.py
```

```
[4, 2, 10, 10, 7, 8]
```

# Chapter 13 - Binary Search

## 13.1 - Number guessing game

Let's assume we have a sorted array of length 100,000. Further assume somebody has chosen a number from that array, at random and we don't know what it is. We're given 20 attempts to guess the answer. This may seem impossible and that we only have luck on our side.

We can use the knowledge we have so far to write a function that guesses this, and we have a couple of ways of doing it.

Let's look at two ways we might attempt this. The first is linear search.

```
1. from random import randint
2.
3. arr = [x for x in range(0, 100000)]
4. secret = randint(0, len(arr))
5.
6. i = 0
7. while i < 20 and i != secret:
8. i += 1
9.
10. if i < 20:
11. print("The secret number is " + str(i))
12. else:
13. print("Hard luck, you didn't guess the secret number")
```

The first line imports the `randint()` function from the `random` module which takes two integer arguments and returns a random integer between the arguments.

The second line builds up the array using something called a list comprehension. We'll get back to these soon.

The third line chooses that random number as the secret number and the rest of the code is standard linear search to try find the secret.

Obviously, this is a terrible idea as the secret number must be between 0 and 20 which has a 20/100000 chance of happening.

Another approach we could take is, taking 10 random guesses.

```

1. from random import randint
2. arr = [x for x in range(0, 100000)]
3. secret = randint(0, len(arr))
4.
5. guess = randint(0, len(arr))
6. i = 1
7. while i < 20 and guess != secret:
8. guess = randint(0, len(arr))
9. i += 1
10.
11. if i < 20:
12. print("The secret number is " + str(i))
13. else:
14. print("Hard luck, you didn't guess the secret number")

```

Better approach? Maybe, but we can do much better. In fact, we can guess the number within 20 guesses every time without fail. We use a new algorithm called Binary Search to do that.

## 13.2 - Binary Search

One of the assumptions made in the previous section was that the list was sorted. Knowing that the list is sorted allows us to take advantage of that fact. In this section we'll look at how we do that using a new searching algorithm called binary search.

Let me illustrate how binary search works:

```

2 4 9 12 34 35 77
|_____| # The number we were trying to guess is in here

2 4 9 12 34 35 77
| low | high
|_____| # We will call these two position low and
high

2 4 9 12 34 35 77
| low | high
|_____| # The number is somewhere between low and
high

2 4 9 12 34 35 77
| low | high # We're also able to calculate the index
for

```



```

|_____| # the number in the middle of low and high

2 4 9 12 34 35 77
| low | middle | high # We're also able to calculate the index
for
|_____|_____| # the number in the middle of low and high

#
Assume the number we're searching for is 4
#

2 4 9 12 34 35 77
| low | middle | high # We're also able to calculate the index
for
|_____|_____| # the number in the middle of low and high

2 4 9 12 34 35 77
| low | middle | high # As the list is sorted we can check if
the
|_____|_____| # number at 'middle' is <= or > 4

2 4 9 12 34 35 77
| low | middle | high # In this case 4 is < 12 so we don't need
to
|_____|_____| # bother searching anything above middle
index

2 4 9 12
| low | high # So we cut that portion of the list out
|_____| # and make middle the new high

2 4 9 12
| low | high # Repeat the process,

```

```
|_____|
condition # continuously halving the list until the

that low < high fails i.e low == high
```

So, our approach here is:

- Find the midpoint between *high* and *low*
- Adjust *low* or *high* depending on whether what we're searching for is less than or equal to ( $\leq$ ) or greater than ( $>$ ) the midpoint value.

We calculate the midpoint by getting the average of low and high i.e.  $(low + high) // 2$ . Notice we're doing integer division here, so we'll round down if there is a decimal place.

In the next section we'll look at implementing binary search

## 13.3 - Implementing Binary Search

In the previous section we looked at how binary search works. Now we need to code it.

Roughly, only 10% of developers are able to code binary search properly. There's a little cop out in the middle of it so we need to be careful so that our solution works correctly in all cases.

Below is the commented implementation of binary search:

```
1. def binary_search(arr, elem):
2. low = 0 # Define initial low value.
3. high = len(arr) # Define initial high value.
4.
5. while low < high: # The condition that must hold true.
6. mid = (low + high) // 2 # Calculate the mid index.
7.
8. if arr[mid] < elem: # Check if the value is in first or second half
9. low = mid + 1 # Update low value if mid val < elem (in second
10. half).
11. else:
12. high = mid # Otherwise update high value (elem in first ha
13. lf).
14. return low # Return the position of the element
15. # we're searching for.
```

It is important that *mid* and *high* are never equal. This is one place where some developers mess up when coding this algorithm.

We must add `1` when updating the low value because if `low == mid`, we will end up in an infinite loop and that is bad news! This is another place where developers mess up.

Since binary search has its cop outs like those above, I recommend you learn it off by heart.

## 13.4 - Improving our number guessing game

We can now take this binary search algorithm and modify it slightly to fit our game. We're going to add a check that we don't do more than 20 iterations of the binary search algorithm i.e. halve the list more than 20 times.

```
1. from random import randint
2.
3. arr = [x for x in range(0, 100000)]
4. secret = randint(0, len(arr))
5.
6.
7. low = 0
8. high = len(arr)
9. i = 0
10. while low < high and i < 20:
11. mid = (low + high) // 2
12.
13. if arr[mid] < secret:
14. low = mid + 1
15. else:
16. high = mid
17. i += 1
18.
19. if i < 20:
20. print("The secret number is " + str(low))
21. else:
22. print("Hard luck, you didn't guess the secret number")
```

Our improved version of the game will guess (find) the secret number within 20 attempts every time.

This is important because we get guess the answer correctly, in the worst case, in 20 attempts, whereas our previous approaches, the worst case would have been 100,000 attempts!

## 13.5 Analysis of Binary Search

In this section I'm going to look at Binary Search's runtime complexity much like I did with Insertion Sort and Selection Sort.

Insertion Sort and Selection Sort had a runtime complexity of:

$O(n^2)$

Binary Search has a runtime complexity of:

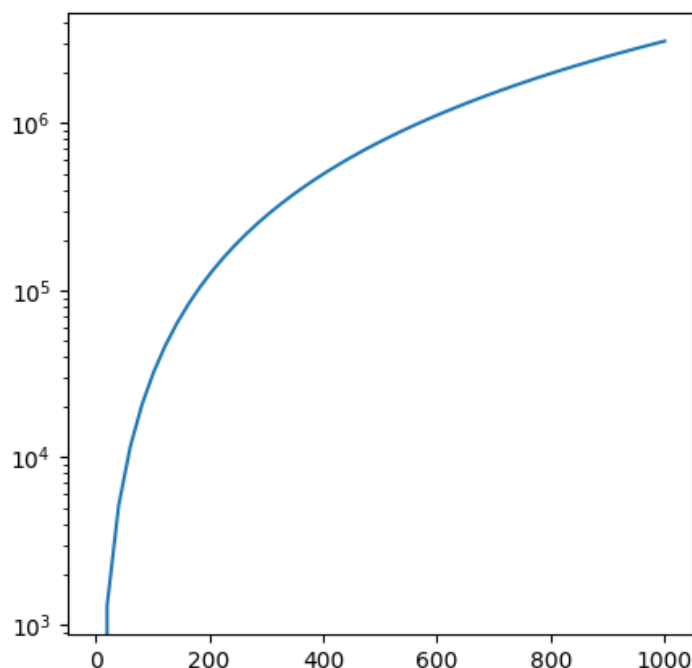
$O(\log(n))$

More specifically:

$O(\log_2 n)$

It is a little trickier to tell what the runtime of an algorithm like this is. As a general rule of thumb, if we halve the input upon each iteration then it will have a logarithmic runtime (or at least, a logarithmic component to its runtime complexity).

Recall the graph of the Insertion Sort and an  $O(n)$  algorithm. Here is the graph of an algorithm with binary search's runtime complexity.



You can clearly see that as the input gets much larger, the time it takes for the algorithm to complete begins to level off and increasing the input size begins to have a smaller and smaller effect on the algorithm's performance.

Binary Search's space complexity is also  $O(1)$ , which is constant memory, as we don't create any copies of the list when searching.

This is considered a very efficient algorithm as its runtime complexity is close to constant and the space complexity is constant.

## 13.6 - Exercises

### Question 1

Answer the following questions:

- Why must we add 1 to *Low* when updating its value?
- What is the runtime complexity of Binary Search
- What is the space complexity of Binary Search?

### Question 2

Using what you've learned in this chapter, write a function called *insert()* that uses an efficient algorithm (hint, hint...) and takes a list and an element as input and returns the index at which that element should be inserted into the list.

For example, calling the insert function as such should return the following:

```
1. insert([1, 3, 4, 5], 2)
2.
3. # RETURNS
4. 1 # i.e. 2 should be inserted at index 1
```

### Question 3

Again, using what you've learned, write a function called *contains()* that takes two arguments, a list and an element and returns whether or not that element is contained within that list.

For example:

```
contains([1, 3, 5, 7], 3)
```

```
RETURNS
```

```
True
```

```

```

```
contains([1, 3, 5, 7], 2)
```

```
#RETURNS
```

```
False
```

# Chapter 14 - Error Handling

## 14.1 - Raising Exceptions

In Python you'll run into many kinds of errors and I'm sure you've ran into plenty so far!

We've seen syntax errors which arise from improper syntax e.g. too many closing brackets. You'll also run into something called exception errors. Exception errors arise whenever syntactically correct Python code results in an error. There are many types of exception errors, to name a few, you'll get an `ImportError` when you try to import a module that cannot be found, you'll get a `ZeroDivisionError` when the second operand of the division or modulo operator is zero. If you run into an error that doesn't fall into a specific category of exception, Python will throw a `RuntimeError`.

Sometimes in our programs we want to raise an exception if a specific condition occurs. This condition might not be one that will crash our program, but we as programmers don't want it occurring.

To raise an exception, we use the `raise` statement. The condition we are raising the exception for may be one that is illogical. For example, we may have a program that changes the speed of a car. It would be illogical for our car to go at a negative speed as there is no such thing.

We would raise the exception as follows:

```
1. def decrease_velocity(curr_vel, decrease):
2. return curr_vel - decrease
3.
4. def main():
5. car_velocity = 20
6. inp = int(input())
7. while inp:
8. car_velocity = decrease_velocity(car_velocity, inp)
9. if car_velocity < 0:
10. raise Exception("Car cannot have a negative speed")
11. inp = int(input())
12.
13. if __name__ == "__main__":
14. main()
```

Running this with the following inputs would give the following output:

```
$ py change_vel.py
5
6
7
5
Traceback (most recent call last):
 File "test3.py", line 14, in <module>
 main()
 File "test3.py", line 10, in main
 raise Exception("Car cannot have a negative speed")
Exception: Car cannot have a negative speed
```

As you can see, we raised an exception with our own, custom error message.

We can even raise specific types of exceptions. A `ValueError` may be suited to this example and we simply replace `Exception` with `ValueError`.

We may also raise exceptions where we expect errors to occur as they normally would. We would raise an exception in this case as to provide a more detailed error message.

But, as a general rule of thumb, an exception is raised when a fundamental assumption of the current code is found to be false.

## 14.2 - Assertions

If you remember back to the beginning of this book, I talked about pre and post conditions. A pre-condition is a condition that always holds true prior to the execution of some code. Preconditions may be used in functions to enforce a contract between a function and its invoker. Let's assume we have some function that requires a list as an argument. We want to assert that the argument passed to the function is a list and not something else like a string.

We implement assertions in python using `assert`

```
1. def sorter(arr):
2. assert(type(arr) == list)
3. # Do the sorting
4.
5. def main():
6.
7. arr = [6, 3, 5, 1, 8]
```



```

8. sorter(arr)
9.
10. print("We've made it this far")
11.
12. arr = "Hello"
13. sorter(arr)
14.
15. if __name__ == "__main__":
16. main()

```

running the above code produces the following output:

```

$ py assert.py
We've made it this far
Traceback (most recent call last):
 File "test3.py", line 16, in <module>
 main()
 File "test3.py", line 13, in main
 sorter(arr)
 File "test3.py", line 2, in sorter
 assert(type(arr) == list)
AssertionError

```

As you can see our assertion was `True` when we passed the list, so the program continued as normal. When we passed a string to the sorter function, the assertion returned `False` and our program exited and raised an `AssertionError`.

## 14.3 - try & except

In this section I'm going to talk about handling exceptions that we may encounter in our code. In Python, to handle exceptions we use a `try` and `except` block. We catch the errors using these constructs.

The way `try` and `except` blocks work are not all that different from an `if` and `else` statement. Inside the `try` block we put code that should run as "normal". If an exception is raised in the `try` block, then we catch this exception in the `except` block. By catching the exception, we can handle the error without our program crashing.

Let me give two examples. One where we don't try to catch the exception and one where we do.

```

1. filename = input()
2. with open(filename) as f:

```

```

3. lines = f.readlines()
4.
5. print(lines)

```

This would produce the following error:

```

$ py read_file.py

Traceback (most recent call last):

 File "test3.py", line 1, in <module>
 with open("IDontExist.txt") as f:
FileNotFoundError: [Errno 2] No such file or directory: 'IDontExist.txt'

```

Our program has crash. What if we wanted to handle this appropriately and allow the user to enter the filename again? We use a `try` and `except` block to do this.

```

1. filename = input("Enter a file name: ")
2. while filename:
3. try:
4. with open(filename) as f:
5. print(f.readlines())
6. filename = input("Enter a file name: ")
7. except FileNotFoundError:
8. print("That file doesn't exist in the current directory")
9. filename = input("Enter a file name: ")

```

Now if we run our code:

```

$ py read_files.py

Enter a file name: test.txt

That file doesn't exist in the current directory

Enter a file name: input.txt

['This is line one.\n', "I'm line two.\n", "And I'm line three"]

Enter a file name: wrongfile.txt

That file doesn't exist in the current directory

```

We can see, rather than our program crashing, we handle the `FileNotFoundError` in a correct manor and allow the user to re-enter a filename.

Python has an exception hierarchy. This is shown below:

```

BaseException

+-- SystemExit

+-- KeyboardInterrupt

```

```

+-- GeneratorExit
+-- Exception
 +-- StopIteration
 +-- StopAsyncIteration
 +-- ArithmeticError
 | +-- FloatingPointError
 | +-- OverflowError
 | +-- ZeroDivisionError
 +-- AssertionError
 +-- AttributeError
 +-- BufferError
 +-- EOFError
 +-- ImportError
 | +-- ModuleNotFoundError
 +-- LookupError
 | +-- IndexError
 | +-- KeyError
 +-- MemoryError
 +-- NameError
 | +-- UnboundLocalError
 +-- OSError
 | +-- BlockingIOError
 | +-- ChildProcessError
 | +-- ConnectionError
 | | +-- BrokenPipeError
 | | +-- ConnectionAbortedError
 | | +-- ConnectionRefusedError
 | | +-- ConnectionResetError
 | +-- FileExistsError
 | +-- FileNotFoundError
 | +-- InterruptedError
 | +-- IsADirectoryError

```

```

| +-- NotADirectoryError
| +-- PermissionError
| +-- ProcessLookupError
| +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
| +-- NotImplementedError
| +-- RecursionError
+-- SyntaxError
| +-- IndentationError
| +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
| +-- UnicodeError
| +-- UnicodeDecodeError
| +-- UnicodeEncodeError
| +-- UnicodeTranslateError
+-- Warning
 +-- DeprecationWarning
 +-- PendingDeprecationWarning
 +-- RuntimeWarning
 +-- SyntaxWarning
 +-- UserWarning
 +-- FutureWarning
 +-- ImportWarning
 +-- UnicodeWarning
 +-- BytesWarning
 +-- ResourceWarning

```

We were being quite specific with our exception in the previous example. Assume another error might occur somewhere in our program and we are unsure of what it might be, but we still want to allow the user to enter a

filename after it occurs. We can have multiple except clauses after the `try` block.

```
1. filename = input("Enter a file name: ")
2. while filename:
3. try:
4. with open(filename) as f:
5. print(f.readlines())
6. filename = input("Enter a file name: ")
7. except FileNotFoundError:
8. print("That file doesn't exist in the current directory")
9. filename = input("Enter a file name: ")
10. except:
11. print("Unexpected error occurred")
12. filename = input("Enter a file name: ")
```

Now, if we run into any other error, we have handled it with a more general exception.

We also handle exceptions we explicitly raise in the same way:

```
1. inp = int(input())
2. try:
3. if inp < 5:
4. raise ValueError
5. else:
6. print(inp)
7. except ValueError:
8. print("Input shouldn't be less than 5")
```

## 14.4 - finally

Usually, if we have a `try` and `except` block, we'll try to do something specific that may be prone to errors. In the example from the previous section, that specific something was opening a file. We shouldn't have to ask the user for input again in the `try` block. We also shouldn't have to ask the user to enter the filename again after we handled the exception in the `except` block.

Instead, we want some way of cleaning up afterwards. In Python, we do this with the `finally` statement.

This is demonstrated below:

```
1. filename = input("Enter a file name: ")
2. while filename:
3. try:
4. with open(filename) as f:
5. print(f.readlines())
6. except FileNotFoundError:
7. print("That file doesn't exist in the current directory")
8. except:
9. print("Unexpected error occurred")
10. finally:
```

```
11. filename = input("Enter a file name: ")
```

The *finally* block is always executed even if no exceptions were raised in the *try* block. Although our code is prone to errors from the users input, we can handle that correctly thus making our code more robust.

## 14.5 - Exercises

### Question 1

Write a program that takes a number from a command-line argument and print out the multiples of that number up to 10. The user should also be able to specify a formatting flag and/or shortening flag at the command line. As an example, take a look at a couple of different ways a user could run the program:

```
$ py times_tables.py 15
0 * 15 = 0
1 * 15 = 15
2 * 15 = 30
3 * 15 = 45
4 * 15 = 60
5 * 15 = 75
6 * 15 = 90
7 * 15 = 105
8 * 15 = 120
9 * 15 = 135
10 * 15 = 150

$ py times_tables.py -f 15
0 * 15 = 0
1 * 15 = 15
2 * 15 = 30
3 * 15 = 45
4 * 15 = 60
5 * 15 = 75
```

```
6 * 15 = 90
7 * 15 = 105
8 * 15 = 120
9 * 15 = 135
10 * 15 = 150

$ py times_tables.py -f -s 15
0
15
30
45
60
75
90
105
120
135
150
```

In the second example, notice the `-f` flag and, the `-s` flag in the third example.

Your program should be able to handle the following situations that may raise exceptions:

- User enters a flag that doesn't exist e.g. `-d`
- User enters a flag but no number
- User doesn't enter any command-line arguments
- User doesn't enter a number, instead they enter a string ("hello" for example)
- User enters the same flag two or more times.
- User enters two or more numbers

This program may take some time to implement.

## Question 2

When should we use the following?

- *raise*
- *finally*
- *try* and *except*
- What is a precondition?
- What is a postcondition?



# Chapter 15 - More on Data Types

## 15.1 - Tuples

A tuple is essentially an immutable list. Tuples, like strings, cannot be modified after creation. However, a tuple is different to a string in that while strings consist solely of characters, a tuple can contain objects of any type (like a list).

To create a tuple, we use the comma operator:

```
>>> my_tuple = 2, 4, 8, 16
>>> my_tuple
(2, 4, 8, 16)
```

We typically encapsulate a tuple in parenthesis as it makes things easier to read.

```
>>> my_tuple = ("Hello", 6, 3.14)
>>> my_tuple
('Hello', 6, 3.14)
```

To create a tuple with a single value you must include a comma!

```
>>> t = (4,)
>>> type(t)
<class 'tuple'>
>>> v = (4)
>>> type(v)
<class 'int'>
```

We can concatenate tuples using the `+` operator and repeat tuples using the `*` operator:

```
>>> t = (2, 4, 6, 8)
>>> u = (1, 3, 5, 7)
>>> t + u
(2, 4, 6, 8, 1, 3, 5, 7)
>>> t * 2
(2, 4, 6, 8, 2, 4, 6, 8)
```

We can also slice and index in the same way we do with strings:

```
>>> t = (5, 4, 3, 2, 1)
>>> t[3]
2
>>> t[::-1]
(1, 2, 3, 4, 5)
>>> t[1:3]
(4, 3)
```

We can also iterate over tuples using loops. I'm going to use the `for` loop here, but you can use `while` too:

```
>>> chars = ("a", "b", "c")
>>> for c in chars:
... print(c)
...
a
b
c
```

We can use the `in` operator to check if a value exists in a tuple

```
>>> t = ("John", "Liam", "Tony")
>>> print("Tony" in t)
True
```

And lastly, as expected, we get an error if we try to change a tuples contents:

```
>>> t = (1, 2, 3)
>>> t[0] = 4
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

If you remember back to the chapter on dictionaries, the `d.items()` method returns a list of tuples.

```
>>> d = {"a": 10, "b": 11, "c": 12}
>>> for (k, v) in d.items():
```

```
... print(k, v)
...
a 10
b 11
c 12
```

## 15.2 - Sets

A set is a collection of objects of arbitrary type. The set's objects are called its members. Sets are an unordered type (like dictionaries). They are also iterable. An important feature of sets is that they only contain one copy of a particular object i.e. duplicates are not allowed. Sets are mutable types.

Sets are generally used for membership testing and removing duplicates. They usually hold a collection of similar items.

To take full advantage of sets, you'll need to have a good understanding of set theory. I won't be covering set theory in this book so you can read up on it here: [Introduction to Set Theory](#). You may however be familiar with basic set theory as you covered it in school. Set's are very important and are one of the data types that creep up in interviews.

In Python, a set is created using the `set()` function.

```
x = set() # s is now an empty set
```

We can also create a set using comma separated values enclosed in curly braces:

```
x = {1, 2, 3, 4}
```

Note that we cannot use `{}` to create an empty set as this would create an empty dictionary.

We use the `in` operator to check for membership:

```
>>> x = {2, 4, 6, 8}
>>> 6 in x
True
```

When we use the `in` operator on strings and lists, the runtime complexity is  $O(n)$ . When we use this operator on sets it's on average  $O(1)$ . This is the same with dictionaries. It is therefore very fast to access.

To add a new member to a set we use the `add()` method:

```
>>> x = {2, 4, 6}
>>> x.add(8)
>>> x
{8, 2, 4, 6}
```

Notice how the set doesn't have an order.

We can get the length of a set using the `len()` function:

```
>>> x = {2, 4, 6}
>>> len(x)
3
```

We can also create a set from other objects:

```
>>> s = set("abcdefghijk")
>>> s
{'d', 'e', 'i', 'h', 'k', 'g', 'j', 'f', 'a', 'b', 'c'}
>>> x = set([2, 4, 6, 8])
>>> x
{8, 2, 4, 6}
```

Remember: Duplicates are removed from sets!

```
>>> my_list = [2, 2, 6, 4, 5, 4, 4, 9]
>>> my_set = set(my_list)
>>> my_set
{9, 2, 4, 5, 6}
```

This is particularly useful when searching for unique elements in something.

We can get the intersection of two sets A and B using the `intersection()` method. This will return a set which contains the elements that are common to both A and B:

```
>>> A = {2, 3, 4, 5, 6, 7}
>>> B = {4, 5, 6, 10, 34, 22, 1}
>>> A.intersection(B)
{4, 5, 6}
```

We can get the union of two sets A and B using the `union()` method. The union of two sets is the set of elements in both A and B:

```
>>> A = {1, 3, 3, 7}
>>> B = {2, 3, 6, 8}
>>> A.union(B)
{1, 2, 3, 6, 7, 8}
```

We can get the set difference between two sets A and B using the `difference()` method. The A set difference B is the set of elements in A but not in B. Likewise, B set difference A is the set of elements in B but not in A:

```
>>> A = {1, 2, 3, 4, 5, 6}
>>> B = {4, 5, 6, 7, 8, 9}
>>> A.difference(B)
{1, 2, 3}
>>> B.difference(A)
{8, 9, 7}
```

We can also check if set B is a subset of A using the `issubset()` method. Set A is a subset of set B if every member of A is also a member of B:

```
>>> A = {2, 4, 6}
>>> B = {1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> A.issubset(B)
True
>>> B.issubset(A)
False
```

We can check if A is a superset of B using the `issuperset()` method. B is a superset of A if every member of A is also a member of B:

```
>>> A = {2, 4, 6}
>>> B = {1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> B.issuperset(A)
True
>>> A.issuperset(B)
False
```

## 15.3 - Lists & list comprehensions

We've seen how to create lists and you should be used to doing it at this point. However, consider the following task:

Write a program that builds a list that contains all the even numbers from a second list.

The task of building one list by processing elements from another list is a very common task in computing. Until now we have had to use some kind of loop to iterate over all the elements in the second list and pick out the even numbers. Python has provided a short-cut for this called a list comprehension. You have seen me using a list comprehension in a previous chapter to build a list of 1,000 integers.

In this section we're going to look at how to create list comprehensions.

A list comprehension is a short-cut for building one list from another.

The syntax for a list comprehension is: `[expression for-clause condition]`.

For example, to create a list that contains every number from 0 to 1,000 we could use the following list comprehension:

```
lst = [x for x in range(0, 1000)]
```

The above list comprehension reads: "Add x to the new list for each x in the list of elements from 0 to 1000".

To solve the task, I mentioned at the beginning of this section we could use the following list comprehension:

```
>>> my_lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> evens = [x for x in my_lst if not x % 2]
>>> evens
[2, 4, 6, 8, 10, 12, 14]
```

In the above list comprehension, `x` is the expression, `for x in my_lst` is the for-clause and `if not x % 2` is the condition. It reads as follows: "Add x to the new list for each x in my\_lst if x modulus 2 does not equal 0".

Remember, 0 is falsey, therefore, `if not x % 2` is True if `x % 2` is equal to 0.

We can use a list comprehension to build up a list of squared values from another list using the following list comprehension:

```
>>> vals = [1, 2, 3, 4, 5, 6]
>>> squares = [x ** 2 for x in vals]
>>> squares
[1, 4, 9, 16, 25, 36]
```

Let's look at a list comprehension that will square the even numbers and leave any odd numbers the same:

```
>>> vals = [1, 2, 3, 4, 5, 6, 7, 8]
>>> evens_squared = [x ** 2 if not x % 2 else x for x in vals]
>>> evens_squared
[1, 4, 3, 16, 5, 36, 7, 64]
```

The above list comprehension reads as: "Add the square of x if x is even otherwise just x for each x in vals"

List comprehensions are so common to stumble across and you'll often see them being used if you research any problems online. They are generally a construct only found in Python. If you move on to Java for example in the future, you won't be able to use list comprehensions as they don't exist in Java.

Comprehensions don't just apply to lists. They can be used as short-cuts for building a new collection from another collections. We therefore have set and dictionary comprehensions:

```
>>> my_set = {x for x in range(10)}
>>> my_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> my_dict = {x:True for x in range(5)}
>>> my_dict
{0: True, 1: True, 2: True, 3: True, 4: True}
```

Practice with these! Use them when you can (which will probably be often). Making use of comprehensions is the Pythonic way of writing Python code.

## 15.4 - Exercises

Download the following file (copy this page to a text file): [Dictionary](#).

You'll be using this for some of the exercises.

## Question 1

During a study, several software engineers were asked what their top 3 favourite programming languages were.

Write a program that determines the number of unique languages given as answers during the study. Below is the input file and each line being an answer given by a software engineer:

```
java python c++
python scala c
java haskell javascript
javascript python c
c c++ c#
c# haskell java
```

Your program should be run as follows and give the following output:

```
$ py unique.py input.txt
8
```

## Question 2

Write a list comprehension that creates a list containing all the odd numbers between 0 and 10,000.

## Question 3

Using the dictionary you downloaded earlier, write a program that builds a list containing all the words that are longer than or equal to 18 characters. You should use a list comprehension to do this!

Your program should be run as follows:

```
$ py Long_words.py < dictionary.txt
```

Note: This may take some time to run as the input file is large!

## Question 4



Using the dictionary, you downloaded earlier, write a program that builds a list containing all the words that contain every vowel (a, e, i, o and u). For example, the word "equation" should be in the list. You should make good use of a list comprehension to do this!

Your program should be run as follows:

```
$ py vowels.py < dictionary.txt
```

## Question 5

Using the dictionary, you downloaded earlier, write a program that builds a list containing all the words that contain exactly 4 a's and end in 'ian'. For example, the word "alabastrian" should be in the list. You should make good use of a list comprehension to do this!

Your program should be run as follows:

```
$ py two_conditions.py < dictionary.txt
```

## Question 6

Using the dictionary, you downloaded earlier, write a program that builds a list containing all the words in the dictionary whose reverse also occurs. For example, the words "lager" and "regal" should be in the list. You should make good use of comprehensions to do this!

Your program should be run as follows:

```
$ py reverse.py < dictionary.txt
```

Tip: You may want to think about your solution for this. Your program may take a very long time to run if you don't implement a more efficient solution!

# Chapter 16 - Recursion & Quicksort

## 16.1 - Recursion

Before I go any further, I want to say this, writing a recursive solution is DIFFICULT when starting out. Don't expect it to work first time. Writing a recursive solution involves a whole new way of thinking and can in fact be quite off putting to newcomers but stick with it and you'll wonder how you ever found it so difficult.

So, what exactly is recursion? A recursive solution is one where the solution to a problem is expressed as an operation on a simplified version of the same problem.

Now that probably didn't make a whole lot of sense and it's very tricky to explain exactly how it works so let's look at a simple example:

```
1. def reduce(x):
2. return reduce(x-1)
```

So, what's happening here? We have a function called reduce which takes a parameter *x* (an integer) and we want it to reduce it to '0'. Sounds simple so let's run it when we pass 5 as the parameter and see what happens:

```
>>> reduce(5)
RuntimeError: maximum recursion depth exceeded
```

An error? Well lets take a closer look at whats going on here:

reduce(5) calls reduce(4) which calls reduce(3).....Thus our initial call reduce(5) is the first call in an infinite number of calls to reduce(). Each time reduce() is called, Python instantiates a data structure to represent that particular call to the function. This data structure is called a stack frame. A stack frame occupies memory. Our program attempts to create an infinite number of stack frames which would require an infinite amount of memory which we don't have so our program crashes.

To fix this problem we need to add in something referred to as the base case. This is simply a check we add so that our function knows when to stop calling itself. The base case is normally the simplest version of the original problem and one we always know the answer to.

Let's fix our error by adding a base case. For this function we want it to stop when we reach 0.

```
1. def reduce(x):
2. if x == 0:
3. return 0
4. return reduce(x-1)
```

So now when we run our program we get:

```
>>> reduce(5)
0
```

Great it works! Let's take another look at what is happening this time. We call `reduce(5)` which calls `reduce(4)`....which calls `reduce(0)`. Ok stop here, we have hit our base case. Now what the function will do is return 0 to the previous call of `reduce(1)` which returns 0 to `reduce(2)`....which returns 0 to `reduce(5)` which returns our answer. There you go, your first recursive solution!

Let's look at a more practical example. Let's write a recursive solution to find  $N!$  (or  $N$  factorial):

$N$  factorial is defined as:  $N! = N * (N-1) * (N-2)..... * 2 * 1$ .

For example,  $4! = 4 * 3 * 2 * 1$

Our base case for this example is  $N = 0$  as  $0!$  is defined as 1. So, let's write our function:

```
1. def factorial(n):
2. if n == 0:
3. return 1
4. return n * factorial(n-1)
```

OK let's see that in action. We'll call our function and pass 4 in for  $n$ :

`factorial(4)` calls `factorial(3)` ..... which calls `factorial(0)` which is our base case so we return 1 back to the previous call of `factorial(1)` which takes our 1 and multiplies it by 1 which evaluates to 1 which passes that back to `factorial(2)` which takes our 1 and multiplies it by 2 which evaluates to 2 which passes that back to `factorial(3)` which multiplies 3 2 to get 6 which passes that back to `factorial(4)` which multiplies 4 6 which evaluates to 24 which is returned as our answer.

These are very simple cases and not super useful but being able to spot when a recursive solution may be implemented and then implementing that solution

is a skill that will make you a better programmer. For certain problems recursion may offer an intuitive, simple, and elegant solution.

For example, you may implement a recursive solution to find the number of nodes in a tree structure, count how many leaf nodes are in a tree or even return whether a binary search tree is AVL balanced or not. These solutions tend to be quite short and elegant and take away from the long iterative approach you may have been taking.

As I've said, recursion isn't easy but stick with it, it will click and seem so simple you'll wonder how you ever found it so difficult.

## 16.2 - Memoization

We've written the solution to the Fibonacci problem a couple of times throughout this book. When writing those solutions, we've used an iterative approach. We can, however, take a recursive approach. It may be difficult to recognize when a recursive solution is an option, but this is a great example of when a recursive solution is an option!

Let's look at a solution to the following problem:

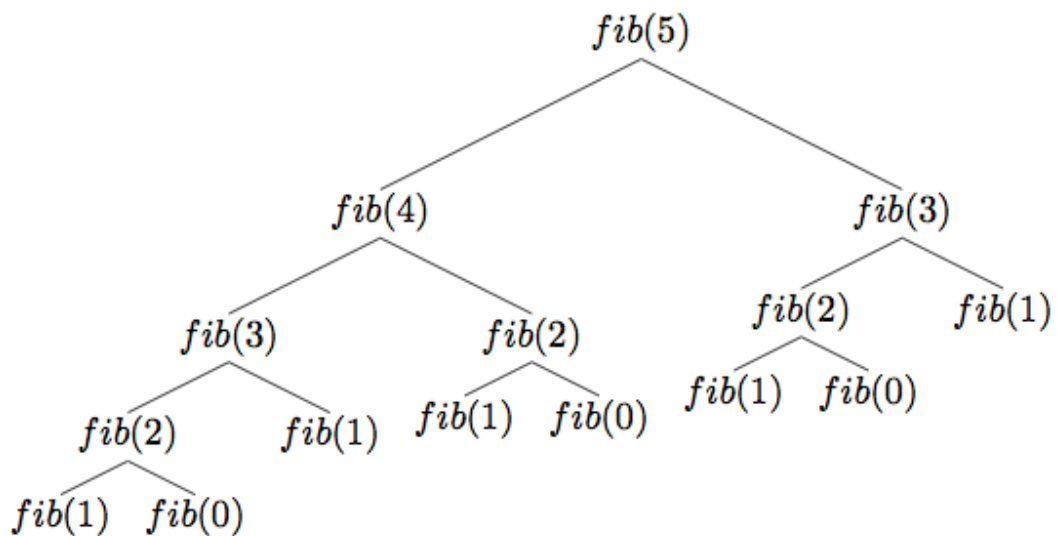
Write a program that takes an integer *n* and computes the n-th Fibonacci number:

```
1. def fib(n):
2. if n <= 1:
3. return 1
4. else:
5. return fib(n-1) + fib(n-2)
```

That's much simpler! In this example we'll take fib(0) to be 1 and fib(1) to be 1 also. Therefore, our base case is if n is 0 or 1 we return 1. The recursive case comes straight from the definition of the Fibonacci sequence, that is:

$$\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$$

However, you may have noticed that if you input anything greater than (roughly) 35 - 40 your program begins to really grind to a halt and take a VERY long time and anything above 45~50, well... go make a cup of coffee and come back and it MIGHT be finished. What's causing this? Let's look at a diagram of the first few recursive calls to *fib()*.



We can see that when we call `fib(5)` we, at various points in our functions execution, call `fib(1)` 5 times! and `fib(2)` 3 times! This is unnecessary. Why should we have to recompute the value for `fib(1)` over and over again?

We can reduce the number of times we need to calculate various values by using a technique called memoization. Memoization is an optimization technique used to speed up our programs by caching the results of expensive function calls. A cache is essentially a temporary storage area and we can implement a cache in our `fib()` function to store previously calculated values and just pull them out of the cache when needed rather than making unnecessary function calls.

We can use a dictionary to implement a cache. This will give us  $O(1)$  for accessing and will drastically improve the performance of our function.

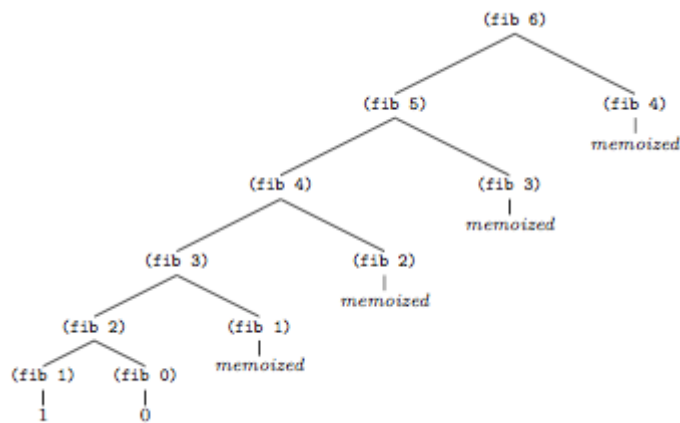
Let's look at how that's done:

```

1. def fib(n, cache=None):
2. # Avoid default mutable argument trap!
3. if cache == None:
4. cache = {}
5. # Base case
6. if n <= 1:
7. return 1
8. # If value not in cache then calculate it and store in cache
9. elif not n in cache:
10. cache[n] = fib(n-1, cache) + fib(n-2, cache)
11. # Return the value
12. return cache[n]

```

Let's look at our tree now, this time for *fib(6)*.



## 16.3 - Quicksort part 1

In this section we'll look at sorting lists of numbers.

To achieve this, I'll first cover quicksort's core operation: partitioning. It works as follows:

```
>>> A = [6, 3, 17, 11, 4, 44, 76, 23, 12, 30]
>>> partition(A, 0, len(A)-1)
>>> print(A)
```

```
[6, 3, 17, 11, 4, 23, 12, 30, 76, 44]
```

So, what happened here and how does it work? We need to pick some number as our pivot. Our partition function takes 3 arguments, the list, the first element in the list, and the pivot. What we are trying to achieve here is that when we partition the list, everything to the left of the pivot is less than the pivot and everything to the right is greater than the pivot. For the first partition seen above, 30 is our pivot. We'll always take our partition to be the last element in the list. After the partition we see some elements have changed position but everything to the left of 30 is less than it and everything to the right is greater than it.

So, what does that mean for us? Well it means that 30 is now in its correct position in the list AND we now have two easier lists to sort. All of this is done in-place, so we are not creating new lists.

Let's look at the code:

```
1. def partition(A, p, r):
2. q = j = p
3. while j < r:
4. if A[j] <= A[r]:
5. A[q], A[j] = A[j], A[q]
6. q += 1
7. j += 1
8. A[q], A[r] = A[r], A[q]
9. return q
```

The `return q` at the end isn't necessary for our partition but it is essential for sorting the entire list. The above code works its way across the list `A` and maintains indices `p`, `q`, `j`, `r`.

`p` is fixed and is the first element in the list. `r` is the pivot and is the last element in the list. Elements in the range `A[p:q-1]` are known to be less than or equal to the pivot and everything from `A[q-1:r-1]` are greater than the pivot. The only indices that change are `q` and `j`. At each step we compare `A[j]` with `A[r]`. If it is greater than the pivot it is in the correct position, so we increment `j` and move to the next element. If `A[j]` is less than `A[r]` we swap `A[q]` with `A[j]`. After this swap, we increment `q`, thus extending the range of elements known to be less than or equal to the pivot. We also increment `j` to move to the next element to be processed.

I encourage you to work through an example with a small list on paper to make this clearer.

## 16.4 - Quicksort part 2

Now onto the quicksort part. Remember it is a recursive algorithm so it will continuously call on `partition()` until there is nothing left to partition and all elements are in their correct positions. After the first partition we call `partition()` again, this time we call it on the list of elements to the left of the pivot and the list of elements to the right of the pivot.

Let's look at the code:

```
1. def quicksort(A, p, r):
2. if r <= p: # If r <= p then our list is sorted
3. return
4. q = partition(A, p, r) # partition incoming list
5. quicksort(A, p, q-1) # call quicksort again on everything to left of pivot
6. quicksort(A, q+1, r) # call quicksort again on everything to right of pivot
7. return A
```

It's that simple. All we do here is check if the index of the pivot, is less than or equal to the index of the start of our list we want to partition. If it is we return as whatever list was passed does not need to be partitioned any further.

Otherwise, we partition the list A, and call quicksort again on the two new sub lists.

On line 3, we return without specifying anything to return. This is because the quicksort function is a procedure.

Quicksort works best on large lists that are completely scrambled. It has really bad performance on lists that are almost sorted. Or in Big-O notation, the best case (scrambled) is  $O(n \log(n))$  and in the worst case, (almost or completely ordered list) is  $O(n^2)$ .

Again, I encourage you to try this on paper with a simple list. It will help clarify what is going on.

## 16.5 - Exercises

Important: These exercises are quite difficult! Make use of problem-solving techniques to help you arrive at a solution. Sketching out what's happening on paper, it's a good way to help you out!



Some of these questions are the kind of questions you can expect if you get a recursion question when interviewing at companies such as Google, Facebook or Amazon.

### Question 1

What is:

- Recursion?
- Memoization?

### Question 2

Write a recursive function that adds up all the numbers from 1 to 100.

### Question 3

Write a recursive function that takes an integer as an argument and returns whether that integer is a power of 2. Your function should return *True* or *False*.

### Question 4

Write a recursive function that takes a string as an argument and returns whether that string is a palindrome. Your function should return *True* or *False*.

### Question 5

Write a recursive function that takes a list of integers as an argument and returns the maximum of that list.

Hint: Thinking recursively, the maximum is either the first element of the list or the maximum of the remaining list!

# Chapter 17 - Object Oriented Programming (OOP)

## 17.1 - Classes & Objects

So far, we've met many of Python's built-in types such as Booleans, integers, dictionaries etc. These are all class types. This means that any instance of a string, list, float etc, is an object of the class string, list or float. In other words, every string object, e.g. *"HeLLo"*, is an instance of the string class. An object is an implementation of a type

In this chapter we're going to look at a programming paradigm called object-oriented programming (OOP). This paradigm is based on the concept of objects, which contain data, in the form of attributes and functions (called methods).

In OOP, programs are designed by making them out of objects that interact with one another. An important feature of objects is that they use their methods to access and modify their attributes. For example, if we wanted to model a *Person*, they might have an *age* attribute and a *name* attribute. Each year they get one year older so they may have a *change\_age()* method to update their age.

The general syntax for invoking an object's method is *object\_name.method\_name(arguments)*. We have done this multiple times. For example, calling the *count()* method on a string:

```
>>> s = "I am a string object, an instance of the string class"
>>> s.count('a')
5
```

Most of the time, Python's built-in types are not enough to model data we want, so we built our own types (or classes) which model the data exactly how we need.

A good way to think of this is that a class is a cookie cutter and an instance of that class is the cookie we cut out.

## 17.2 - Defining a new type

Let's say we wanted to model *Time*. Python's built in data types are not going to be enough, to easily and logically model this. Instead, we would create our own *Time* class. We define a new class using the *class* statement.

```
1. class Time:
2. pass
```

We have now defined a new class called time. As a side note, *pass* basically means "do nothing". I have it there so our program will not crash.

Save the above code to a file called *my\_time.py* and import it as follows:

```
>>> from my_time import Time
>>> t = Time()
>>> type(t)
<class 'time.Time'>
>>> isinstance(t, Time)
True
>>> print(t)
<time.Time object at 0x000001A7C6902F60>
```

We can see that *t* is of type *Time* and *t* is an instance of *Time* and we can see it is an object stored at the memory location *0x000001A7C6902F60*.

We want to represent time in *hours*, *minutes*, and *seconds*. We can therefore define a method that initialises the time of *Time* objects. We also want to define a *display()* method that will print out the time:

```
1. class Time:
2.
3. def set_time(time_obj, hours, minutes, seconds):
4. time_obj.hours = hours
5. time_obj.minutes = minutes
6. time_obj.seconds = seconds
7.
8. def display(time_obj):
9. print("The time is {}:{}:{}".format(time_obj.hours,
10. time_obj.minutes,
11. time_obj.seconds))
```

In the above class definition, the *set\_time()* method requires four arguments. The first is the time object we want to initialise and the other three are the hours, minutes, and seconds we want to set for our time object.

As shown above, we access the hours, minutes and seconds attributes using the period operator e.g `time_obj.hours = hours`. This is saying, for the time object that was passed, set its hours attribute to the hours passed as an argument.

In the `display()` method, we access the hours, minutes, and seconds using the period operator.

Now let's see how we can use this class:

```
>>> from my_time import Time
>>> t = Time()
>>> Time.set_time(t, 11, 32, 45)
>>> Time.display(t)
'The time is 11:32:45'
>>> t2 = Time()
>>> Time.set_time(t2, 22, 43, 17)
>>> Time.display(t2)
'The time is 22:43:17'
>>> Time.display(t)
'The time is 11:32:45'
```

We can see we have two different time objects. We create an object of the `Time` class by calling the class as if it were a function. Calling the class as a function will instantiate an instance of the class and return a reference to it.

## 17.3 - The self variable

As seen in the previous section, we had to call methods by referring to the class in which it belongs to e.g `Time.display(t)`. However, we didn't have to do that when invoking methods on Python's built in types e.g `s.count('a')`. It turns out that this is just shorthand for `str.count(s, 'a')`.

We can adopt the same approach for our `Time` class:

```
>>> from my_time import Time
>>> t = Time()
>>> t.set_time(11, 32, 45)
>>> t.display()
```

```
'The time is 11:32:45'
```

Note how we only have to pass three arguments to the `set_time()` method rather than the four we declared. This is because when we invoke an instance method on an object, that object is automatically passed as the first argument to the method. Therefore, we supply one less argument than the number we declared in the method definition. Python will automatically supply the missing object as the first argument on our behalf.

By convention, this first parameter, which we've been calling `time_obj`, is named `self`. This refers to the instance on which the method is acting. This makes the methods `set_time()` and `display()` instance methods.

Instance methods are methods that act upon a particular instance of an object. Its first parameter is always the object upon which it will operate. By default, all a classes' methods are instance methods unless we declare them as class methods which we will get to in a later chapter.

We can therefore re-write our class as follows:

```
1. class Time:
2. def set_time(self, hours, minutes, seconds):
3. self.hours = hours
4. self.minutes = minutes
5. self.seconds = seconds
6.
7. def display(self):
8. print("The time is {}: {}: {}".format(self.hours,
9. self.minutes,
10. self.seconds))
```

And we can use it as follows:

```
>>> from my_time import Time
>>> t = Time()
>>> t.set_time(11, 32, 45)
>>> t2 = Time()
>>> t2.set_time(22, 34, 18)
>>> t.display()
'The time is 11:32:45'
>>> t2.display()
'The time is 22:34:18'
```

I'm going to leave it at that for this chapter. There is a lot to take on here and its really important stuff! The rest of the book will be focused solely on object-oriented programming. So, make sure you understand the material in this book and complete the following exercises.

## 17.4 - Exercises

### Question 1

Write a class that models a *Lamp*. The lamp class will have two methods. The first will initialise the lamp and is called *plug\_in()*. This will set the *is\_on* attribute which is a boolean to *False*. The second method is called *toggle()*. If the lamp is off, it will turn it on (set *is\_on* to *True*) and if the lamp is on, it will turn it off (set *is\_on* to *False*).

Your class may be imported and used as follows:

```
>>> from my_lamp import Lamp
>>> la = Lamp()
>>> la.plug_in()
>>> print(la.is_on)
False
>>> la.toggle()
>>> print(la.is_on)
True
>>> la.toggle()
>>> print(la.is_on)
False
```

### Question 2

Write a class that models a *Circle*. The circle class will have four methods. The first will initialise the circle. The *initialise()* method should take three parameters, *x\_coord*, *y\_coord*, and *radius*. This will set the circles x and y coordinates and its radius. The next method is called *calc\_area()*. It should calculate the area of the circle (Look up the formula). The third method is called *calc\_perimeter()* and should calculate the circumference of the circle (Look up the formula). The fourth method is *overlaps()*. It should take as an argument, another circle and print out whether or not the two circles overlap.

Your class may be imported and used as follows:

```
>>> from my_circle import Circle
>>> c1 = Circle()
>>> c1.initialise(2, 2, 4)
>>> c2 = Circle()
>>> c2.initialise(3, 2, 2)
>>> c1.calc_perimeter()
25.13
>>> c2.calc_perimeter()
12.57
>>> c1.calc_area()
50.27
>>> c1.overlaps(c2)
True
```

Hint: Your definition of the overlaps methods should be `overlaps(self, other_circle)`

### Question 3

Write a class that models a students grades for particular modules. The class will have four methods. The first should initialise the student. The student should have a name and age. The grades should be modelled as a dictionary in which the key is the name of the module and the value is the grade. This should initially be empty. The second method is called `add_module()` and should add a module to the student object. The third is called `update_module_grade()` and should update the grade for a particular module. The final method is called `show_grades()` and should print out the students modules and the grade associated with each module.

Your class may be imported and used as follows:

```
>>> from my_student import Student
>>> s1 = Student()
>>> s1.initialise("Noah", 19)
>>> s1.add_module("Python")
>>> s1.update_module_grade("Python", 87)
```

```
>>> s1.add_module("Networks")
>>> s1.update_module_grade("Networks", 77)
>>> s1.show_grades()
```

*Noahs grades are:*

*Python: 87*

*Networks: 77*

Hint: Make sure you're initialising the grades dictionary correctly!



# Chapter 18 - OOP: Special Methods and Operator Overloading

## 18.1 - The `__init__()` method

In the previous chapter we looked at how to define new types by creating classes. We also had to initialise our instance objects using methods like `initialise()`. We don't have to do this with Python's built-in types such as strings.

For example, to initialise a list we could just type `my_list = [1, 2, 3]`.

We can define a special method called `__init__()` (that's double underscore). This is called a constructor. If we provide that method for our class, then it is automatically called when we create an object. We can therefore replace any initialisation methods we had previously defined with `__init__()`. This allows us to create and initialise our object in one step.

Let's look at a `Point` that models a point.

```
1. class Point:
2. def __init__(self, x, y):
3. self.x = x
4. self.y = y
5.
6. def display(self):
7. print("X coordinate: {}\nY coordinate: {}".format(self.x, self.y))
```

We can now use our point class as follows:

```
>>> from my_point import Point
>>> p1 = Point(3, 4)
>>> p1.display()
X coordinate: 3
Y coordinate: 4
```

As you can see, our `__init__()` method was called automatically when we created an instance of the point class.

If the `__init__()` method takes two arguments (excluding self), then we must pass two arguments, otherwise we'll get an error.

```
>>> from my_point import Point
>>> p1 = Point()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
```

```
TypeError: __init__() missing 2 required positional arguments: 'x' and 'y'
```

We can however, set default values for these arguments. This will allow us to initialise an instance of `Point` without passing the arguments:

```
1. class Point:
2. def __init__(self, x=0, y=0):
3. self.x = x
4. self.y = y
5.
6. def display(self):
7. print("X coordinate: {}\nY coordinate: {}".format(self.x, self.y))
```

We can now use our class as follows:

```
>>> from my_point import Point
>>> p1 = Point()
>>> p1.display()
X coordinate: 0
Y coordinate: 0
```

What happens when we define this special method and create an instance is:

- Python creates an empty instance of the `Point` class.
- This empty object is passed, along with `3` and `4` to the `__init__()` method.
- `__init__()` initialises the object with the supplied arguments
- A reference to the initialised object is created and assigned to `p1`.

Remember, special methods are never called directly!

## 18.2 - The `__str__()` method

The `__str__()` method is another special function. We can *override* it which means we can change its implementation and hence it's behaviour.

When we call `print()` on a Python built-in such as a string or a list, we get a nicely formatted, human readable representation of the object.

Under the hood, the `list` built in type implements this `__str__()` method and hence, allows us to print our lists in human readable format.

We can *override* this method to allow us to format and print our objects as we see fit. Therefore, when we call `print()` on an object of a class we've defined, we'll no longer get something like `<time.Time object at 0x000001A7C6902F60>` printed to the terminal.

We do this by returning a formatted string. Let's look at how we might do this for the `Point` class. We could now replace our `display()` function and replace it with `__str__()`, that way we can just print our point instance.

```
1. class Point:
2. def __init__(self, x=0, y=0):
3. self.x = x
4. self.y = y
5.
6. def __str__(self):
7. return "X coordinate: {}\nY coordinate: {}".format(self.x, self.y)
```

We can now use our class as follows:

```
>>> from my_point import Point
>>> p1 = Point(2, 3)
>>> print(p1)
X coordinate: 2
Y coordinate: 3
```

The return value of this special method must be a string object.

## 18.3 - The `__len__()` method

As you can probably guess, the `__len__()` method allows us *override* the default implementation of the `len()` function to define what the length of our own types are. For example with a `Point`, the length may be the distance from the origin ( the point (0, 0)). However, we must return an integer. We will get an error otherwise. So, we'll have to round either up or down. Python does this in order to make the `len()` function predictable. In this example, we'll just cast our return value to an integer:

```
1. class Point:
2. def __init__(self, x=0, y=0):
3. self.x = x
4. self.y = y
5.
6. def __str__(self):
7. return "X coordinate: {}\nY coordinate: {}".format(self.x, self.y)
8.
9. def __len__(self):
10. distance_from_center = ((self.x)**2 + (self.y)**2)**0.5
11. return distance_from_center
```

We can now call the `len()` function on instances of our `Point` class.

```
>>> from my_point import Point
>>> p1 = Point(24, 33)
>>> len(p1)
40
```

The actual distance of this point from the center is 40.8044...

It is unfortunate that we can't return a floating-point number, therefore if we want real accuracy we'll have to implement a different method that we'll have to call on the instance.

## 18.4 - Operator overloading

There are a tonne of special methods in Python, I'm not going to cover them all, however, there are some special methods that are called when we use operators. For example, when we use the comparison operator `==` on two integers, a special method called `__eq__()` is called.

When *method overriding* is done on methods that define the behaviour of operators, we call that: *operator overloading*.

We can overload the default behaviour of the `==` method, so that it applies to our `Point` class. This is done as follows:

```
1. class Point:
2. def __init__(self, x=0, y=0):
3. self.x = x
4. self.y = y
5.
6. def __str__(self):
7. return "X coordinate: {}\nY coordinate: {}".format(self.x, self.y)
8.
9. def __len__(self):
10. distance_from_center = ((self.x)**2 + (self.y)**2)**0.5
11. return int(distance_from_center)
12.
13. def __eq__(self, other):
14. return ((self.x, self.y) == (other.x, other.y))
```

In this case, we overload the `__eq__()` method by comparing two tuples. If the two tuples, each containing the x and y coordinates of points, are equal, then our two points are equal. We can use this as follows:

```
>>> from mytime import Point
>>> p1 = Point(2, 2)
>>> p2 = Point(2, 2)
>>> p1 == p2
True
>>> p3 = Point(2, 3)
>>> p1 == p3
False
```

We could also add two points together to get a new point. In this case we would overload the `__add__()` method. The `__add__()` method implements the functionality of the `+` operator, not the `+=` operator. Therefore, we must return a new object as the `+` operator is not for in-place addition.

Let's look at that:

```
1. class Point:
2. def __init__(self, x=0, y=0):
3. self.x = x
4. self.y = y
5.
6. def __str__(self):
7. return "X coordinate: {}\nY coordinate: {}".format(self.x, self.y)
8.
9. def __len__(self):
10. distance_from_center = ((self.x)**2 + (self.y)**2)**0.5
11. return int(distance_from_center)
12.
13. def __eq__(self, other):
14. return ((self.x, self.y) == (other.x, other.y))
15.
16. def __add__(self, other):
17. new_x = self.x + other.x
18. new_y = self.y + other.y
19. return Point(new_x, new_y)
```

Now we can use this method as follows:

```
>>> from my_point import Point
>>> p1 = Point(2, 3)
>>> p2 = Point(5, 2)
>>> p3 = p1 + p2
>>> print(p3)
X coordinate: 7
Y coordinate: 5
```

We can overload the `+=` operator, which is for in-place addition. This way we don't need to return a reference to a new point object and assign that to a new variable `p3`, instead we update the object's (`p1` in this case) data attributes.

This is done by overloading the `__iadd()` method.

```
1. class Point:
2. def __init__(self, x=0, y=0):
3. self.x = x
4. self.y = y
5.
6. def __str__(self):
7. return "X coordinate: {}\nY coordinate: {}".format(self.x, self.y)
8.
9. def __len__(self):
10. distance_from_center = ((self.x)**2 + (self.y)**2)**0.5
11. return int(distance_from_center)
12.
13. def __eq__(self, other):
14. return ((self.x, self.y) == (other.x, other.y))
15.
16. def __add__(self, other):
17. new_x = self.x + other.x
18. new_y = self.y + other.y
19. return Point(new_x, new_y)
20.
21. def __iadd__(self, other):
22. temp_point = self + other
23. self.x, self.y = temp_point.x, temp_point.y
24. return self
```

We can now use the `+=` operator on our points

```
>>> from my_point import Point
>>> p1 = Point(2, 2)
>>> p2 = Point(5, 2)
>>> p1 += p2
>>> print(p1)
X coordinate: 7
Y coordinate: 4
```

Be careful when overloading in-place operators. In this case we use create a temporary point and then update the `self` instance using multiple assignment. Remember when we swapped the value in one variable for another and we had to create a temp variable? We can do that much faster by doing `x, y = y, x`. In this case `x, y` and `y, x` are tuples!

Anyway, after we update the attributes for `self`, we must return `self`.

Below is a list of other operators you can overload.

### Operator Method that can be overloaded

<code>==</code>	<code>__eq__()</code>
<code>!=</code>	<code>__ne__()</code>
<code>&lt;</code>	<code>__lt__()</code>
<code>&lt;=</code>	<code>__le__()</code>
<code>&gt;</code>	<code>__gt__()</code>

## Operator Method that can be overloaded

>=	<code>__ge__()</code>
+	<code>__add__()</code>
-	<code>__sub__()</code>
//	<code>__floordiv__()</code>
/	<code>__truediv__()</code>
*	<code>__mul__()</code>
**	<code>__pow__()</code>
%	<code>__mod__()</code>
+=	<code>__iadd__()</code>
-=	<code>__isub__()</code>
*=	<code>__imul__()</code>
//=	<code>__ifloordiv__()</code>
/=	<code>__itruediv__()</code>
**=	<code>__ipow__()</code>

There are more but we won't need any more than this, in fact we won't use half of these in this book.

## 18.5 - Exercises

### Question 1

Create a `BankAccount` class. The bank account class should have the following attributes: `balance` and `account_owner`. The bank account class should also have `deposit` and `withdraw` methods. You should also be able to `print` the bank account, to display the account owners name and balance. When implemented correctly, you should be able to use the Bank account class as follows:

```
>>> account = BankAccount("John Smith")
>>> print(account)
Owner: John Smith
Balance: $0
>>> account.deposit(200)
>>> print(account)
Owner: John Smith
Balance: $200
>>> account.withdraw(50)
>>> print(account)
Owner: John Smith
Balance: $150
>>> account.withdraw(200) # Not enough in the account to withdraw 200
>>> print(account)
Owner: John Smith
```

## Question 2

Create a `Line` class. The line class should have four attributes: an `x1` and a `y1` coordinate for one point on the line and an `x2` and a `y2` coordinate for the second point on the line. You should be able to do a few things with an instance of a line. You should be able to call the `len` method on a line to show its length, add two lines together, subtract two lines, multiply a line by an integer, `print` the line and compare two lines to see if they are equal. In this case equality means both lines are the same length. When implemented correctly, you should be able to use the `Line` class as follows:

```
>>> lineOne = Line(2, 2, 4, 4)
>>> lineTwo = Line(1, 1, 3, 3)
>>> len(lineOne)
2
>>> lineThree = lineOne + lineTwo
>>> len(lineThree)
4
>>> lineFour = lineOne - lineTwo
>>> len(lineFour)
1
>>> lineFive = lineTwo * 3
>>> len(lineFive)
6
>>> print(lineOne)
Line Details:
Point 1: (2, 2)
Point 2: (4, 4)
>>> print(lineOne == lineTwo)
True
>>> print(lineOne == lineFive)
False
```

Note: The length of the line must return an integer.



# Chapter 19 - OOP: Method Types & Access Modifiers

## 19.1 - What are instance methods?

By the end of this chapter you should be ready for your first project!

We have already met instance methods. They are the type of methods we have been working with in regard to object-oriented programming until this point.

Instance methods are the most common type of methods in classes. They are called instance methods as they act on specific instances of a class. They can access the data attributes that are unique to that class.

For example, if we have a `Person` class, then each instance of a person may have a unique name, age etc. Instance methods have `self` as the first parameter, and this allows us to go through `self` to access data attributes unique to that instance and also other methods that may reside within our class.

I'm not going to provide an example for instance methods as we've seen them time and time again!

## 19.2 - What are class methods?

Class methods are the second type of OOP method we can have. A class method knows about its class. They can't access data that is specific to an instance, but they can call other methods.

Class methods have limited access to methods within the class and can modify class specific details such as class variables which we will see in a moment. This allows us to modify the class's state which will be applied across all instances of the class. They are one of the more confusing method types. Class methods are also permanently bound to its class.

We invoke class methods through an instance or through a class. Unlike instance methods whose first parameter is `self`, with class methods, it's first parameter is not an object, but the class itself. This first parameter is called `cls`. We use a decorator to mark a method as a class method. The decorator is `@classmethod`.

Let's look at an example by going back to our `Time` class. We want a function that can convert seconds to 24-hour time with hours, minutes, and seconds. This method should be bound to the class of `Time` rather than a specific instance.

```
1. class Time:
2. def __init__(self, hours=0, minutes=0, seconds=0):
3. self.hours = hours
4. self.minutes = minutes
5. self.seconds = seconds
6.
7. def __str__(self):
8. return "The time is {:02}:{:02}:{:02}".format(self.hours,
9. self.minutes,
10. self.seconds)
11.
12. # OTHER METHODS CAN GO HERE...
13.
14. @classmethod
15. def seconds_to_time(cls, s):
16. minutes, seconds = divmod(s, 60)
17. hours, minutes = divmod(minutes, 60)
18. extra, hours = divmod(hours, 24)
19. return cls(hours, minutes, seconds)
```

For this example, I've cut out other instance methods that we don't care about for now. The seconds to time class method takes the class as the first parameter and some number of seconds, `s`, as its second parameter.

A side note on the `divmod()` function: The `divmod()` function takes two arguments, the first is the number of seconds we want to convert to 24 hour time, the second will be divided into it (60 in this case). This returns a tuple in which the first element is the number of times the second parameter divided into the first, and the second element of the tuple is the remainder after that division. For example, if we called `divmod(180, 60)`, we'd get back a tuple of (3, 0) in which we use multiple assignment to assign the 3 to the minutes and 0 to the seconds which is what we expect as 180 seconds is 3 minutes.

Back to our class method. When we have calculated our hours, minutes and seconds, we return `cls(hours, minutes, seconds)`. `cls` will be replaced with `Time` in this case. So really, we're returning a new `Time` object.

To see this in action:

```
>>> from my_time import Time
```

```
>>> t = Time.seconds_to_time(11982)
>>> print(t)
'The time is 03:19:42'
```

I also mentioned class variables earlier. Much like before, class variables are bound to a class rather than a specific instance. We can use a class method to check the number of times an instance of the time class has been created for example.

With convention, class variables are usually all uppercase for the variable name. Let's modify our time class above to add a class method called `COUNT` which will count the number of times an instance of the class has been created.

```
1. class Time:
2.
3. COUNT = 0
4.
5. def __init__(self, hours=0, minutes=0, seconds=0):
6. self.hours = hours
7. self.minutes = minutes
8. self.seconds = seconds
9. Time.COUNT += 1
10.
11. def __str__(self):
12. return "The time is {:02}:{:02}:{:02}".format(self.hours,
13. self.minutes,
14. self.seconds)
15.
16. # OTHER METHODS CAN GO HERE...
17.
18. @classmethod
19. def seconds_to_time(cls, s):
20. minutes, seconds = divmod(s, 60)
21. hours, minutes = divmod(minutes, 60)
22. extra, hours = divmod(hours, 24)
23. return cls(hours, minutes, seconds)
```

Note how we have a new `COUNT` variable just before our constructor and inside our constructor, after initializing all the attributes, we increase the `COUNT` class variable by 1. Now we can see how many times instance of our class has been created.

```
>>> from my_time import Time
>>> t1 = Time(23, 11, 37)
>>> t2 = Time()
>>> Time.COUNT
2
```

```
>>> t3 = Time(12, 34, 54)
>>> Time.COUNT
3
```

It may be tricky trying to spot when to use these. As a general rule of thumb, if a method can be invoked (and makes sense to) in the absence of an instance, then it seems that, that method is a good candidate for a class method.

## 19.3 - What are static methods?

Finally, we have a third method type that is called a static method. These are not as confusing as class methods. You can think of static methods just like ordinary functions. Python will not automatically supply any extra arguments when a static method is invoked. Unlike `self` and `cls` with instance and class methods.

Static methods are methods that are related to the class in some way, but don't need to access any class specific data and don't need an instance to be invoked. You can simply call them as pleased.

In general, static methods don't know anything about class state. They are methods that act more as utilities.

We use the `@staticmethod` decorator to specify a static method. We will add a static method to our `Time` class that will validate the time for us, checking that the hours are within the range of 0 and 23 and our minutes and seconds are within the range of 0 and 59.

```
1. class Time:
2.
3. COUNT = 0
4.
5. def __init__(self, hours=0, minutes=0, seconds=0):
6. self.hours = hours
7. self.minutes = minutes
8. self.seconds = seconds
9. Time.COUNT += 1
10.
11. def __str__(self):
12. return "The time is {:02}:{:02}:{:02}".format(self.hours,
13. self.minutes,
14. self.seconds)
15.
16. # OTHER METHODS CAN GO HERE...
17.
18. @classmethod
19. def seconds_to_time(cls, s):
20. minutes, seconds = divmod(s, 60)
21. hours, minutes = divmod(minutes, 60)
```

```

22. extra, hours = divmod(hours, 24)
23. return cls(hours, minutes, seconds)
24.
25.
26. @staticmethod
27. def validate(hours, minutes, seconds):
28. return 0 <= hours <= 23 and 0 <= minutes <= 59 and 0 <= seconds <= 59

```

This static method is simply a utility that allows us to verify that times are correct. We may use this to stop a user from creating a time such as `35:88:14` as that wouldn't make any logical sense.

```

>>> from my_time import Time
>>> Time.validate(5, 23, 44)
True
>>> Time.validate(25, 34, 63)
False
>>> t = Time(22, 45, 23)
>>> t.validate(t.hours, t.minutes, t.seconds)
True

```

It may also be tricky to spot when to use static methods, but again, if you find that you have a function that may be useful but it won't be applicable to any other class yet at the same time it doesn't need to be bound to a instance and it doesn't need to access or modify any class data then it is a good candidate for a static method.

## 19.4 - Public, Private & Protected attributes

In object-oriented programming, the idea of access modifiers is important. It helps us implement the idea of Encapsulation (information hiding). Access modifiers tell compilers which other classes should have access to data attributes and methods that are defined within classes. This stops outside classes from calling methods or accessing data from within another class, or making those data attributes and methods public, in which case all other classes can call and access them.

Access modifiers are used to make code more robust and secure by limiting access to variables that don't need to be accessed by every (or maybe they do in which case they're public).

Until now we have been using public attributes and methods. Classes can call the methods and access the variables in other classes.

In Python, there is no strict checking for access modifiers, in fact they don't exist, but Python coders have adopted a convention to overcome this.

The following attributes are public:

```
1. def __init__(self, hours, minutes, seconds):
2. self.hours = hours
3. self.minutes = minutes
4. self.seconds = seconds
```

By convention, these are accessible by anyone (by anyone I mean other classes).

To "make" these variables private we add a double underscore in-front of the variable name. For an attribute or method to be private, means that only the class they are contained in can call and access them. Nothing on the outside. In the Java language, if you had two classes and tried to call a private method from another class you would get an error.

```
1. def __init__(self, hours, minutes, seconds):
2. self.__hours = hours
3. self.__minutes = minutes
4. self.__seconds = seconds
```

The third access modifier I'll talk about is protected. This is the same as private except all subclasses can also access the methods and member variables. I'll talk about subclasses in the next chapter so don't worry about that now, but by convention, we use a single underscore before the variable name.

```
1. def __init__(self, hours, minutes, seconds):
2. self._hours = hours
3. self._minutes = minutes
4. self._seconds = seconds
```

## 19.5 - Exercises

### Question 1

Write a Python program that contains a class called *Person*. An instance of a *Person* should have name and age attributes. You should be able to print an instance of the class which, for example, should output *John is 28 years old*.

Your class should have a method called `fromBirthYear()` which as parameters should take an age and a birth year. For example, `fromBirthYear('John', 1990)` should return an instance of `Person` whose age is 29 (at the time of writing this book, 2019).

Running your program should produce the following:

```
>>> from my_person import Person
>>> john = Person("John", 28)
>>> print(john)
John is 28 years old
>>> adam = Person.fromBirthYear("Adam", 1990)
>>> print(adam)
Adam is 29 years old
```

Think carefully about what type of method to use.

## Question 2

Write a `Student` class that has initializes a student with 3 attributes: `name`, `grades` and `student_number`. Grades should be a dictionary of modules to grades. The student number of the first student instance should be 19343553. The student number of the second student instance should be 19343554 and so on. You should have three methods, `add_module()` which takes one parameter, a module name, and initialises the grade to 0. The second method should be, `update_module()` which takes two parameters, the module name and the grade for the module. The final method should allow the user to print a student.

Think carefully about how you will implement the increasing student number.

Running your program should give the following output:

```
>>> from my_student import Student
>>> john = Student("John")
>>> john.add_module("Python")
>>> john.update_module("Python", 88)
>>> print(john)
Name: John
Student Number: 19343553
```

Grades:

Python 88

```
>>> adam = Student("Adam")
>>> adam.add_module("Java")
>>> adam.update_module("Java", 60)
>>> print(adam)
```

Name: Adam

Student Number: 19343554

Grades:

Java: 60

### Question 3

Modify the `Student` class to include a method that will validate grades. A grade should be between 0 and 100.

Think carefully about the type of method this should be.

```
>>> from my_student import Student
>>> john = Student("John")
>>> john.add_module("Python")
>>> john.isValidGrade(88)
True
>>> Student.isValidGrade(101)
False
>>> john.update_module("Python", 88)
>>> print(john)
```

Name: John

Student Number: 19343553

Grades:

Python 88

### Project

You are tasked with creating a network simulation application in which two parties exist. We'll call these parties the `Sender` and `Receiver`. There should also exist a process that continuously checks if the `Sender` has any packets of data



to send, if it does, then the process should deliver them to the *Receiver*. The *Sender* should periodically create data packets to be sent (These can be randomly generated strings). The *Receiver* should periodically check if it has received any packets of data, if it has, then it should print them, then delete them.

Hint: You should think carefully about what classes are needed and what methods they should contain. Something called a 'buffer' may be helpful here for your process that moves packets between *Sender* and *Receiver*. Read up about buffers and what they. Think carefully about what Python built-in types might be able to implement a buffer.

# Chapter 20 - OOP Inheritance

## 20.1 - What is inheritance?

In programming we often come across objects that are quite similar or we may see that one class is a type of another class. For example, a car is a type of vehicle. Similarly, a motorbike is a type of vehicle. We can see a relationship emerging here. The type of relationship is an “is a” relationship.

In object-oriented programming we call this inheritance and inheritance helps us model an is a relationship. Inheritance is one of the pillars of object-oriented programming and we’ll be exploring it in this chapter.

There is some important terminology around inheritance that we must clarify before we look any deeper into it. Let’s go back to our example of cars and motorbikes being types of vehicles. If we look at each of these as a class, we would call the vehicle class a base class or parent class and the car and motorbike classes, derived classes, child classes or sub classes (they may be referred to differently depending on what you’re reading, just know they are exactly the same thing).

We can look at inheritance in programming the same way we do in real-life where a child inherits characteristics from its parents, in addition to its own unique characteristics. In programming, a child class can inherit attributes and methods from its parent class.

## 20.2 - Parent & child classes

We can create a parent which will act as a base from which subclasses can be derived. This allows us to create child classes through inheritance without having to rewrite the same code over and over again.

Let’s say we have *Rectangle* and *Square* classes. Many of the methods and attributes between a *Rectangle* and *Square* will be similar. For example both a rectangle and a square have an *area()* and a *perimeter()*. Up until now we may have implemented these classes as follows:

```

1. class Rectangle:
2. def __init__(self, length, width):
3. self.length = length
4. self.width = width
5.
6. def area(self):
7. return self.length * self.width
8.
9. def perimeter(self):
10. return 2 * self.length + 2 * self.width
11.
12. class Square:
13. def __init__(self, length):
14. self.length = length
15.
16. def area(self):
17. return self.length * self.length
18.
19. def perimeter(self):
20. return 4 * self.length

```

And we would have used them as follows:

```

>>> rectangle = Rectangle(2, 4)
>>> square = Square(2)
>>> square.area()
4
>>> rectangle.area()
8

```

This seems a little bit redundant however as both a square and rectangle have 4 sides each and they both have an area and a perimeter. If we look more closely at this situation we notice that a square is a special case of a rectangle in which all sides are the same length.

We can use inheritance to reflect this relationship and reduce the amount of code we have to write. In this case a *Square* is a type of *Rectangle* so it makes sense for a square to inherit from the rectangle class.

Let's look at this code again but this time I have made changes to the square class to reflect this relationship. There are a few new things going on here but I'll explain after.

```

1. class Rectangle:
2. def __init__(self, length, width):
3. self.length = length
4. self.width = width
5.
6. def area(self):
7. return self.length * self.width
8.
9. def perimeter(self):

```

```

10. return 2 * self.length + 2 * self.width
11.
12. class Square(Rectangle):
13. def __init__(self, length):
14. super().__init__(length, length)

```

The first change here is when we define the name of the *Square* class. I have `class Square(Rectangle)`. This means that the *Square* inherits from the *Rectangle* class.

The next change is to the square's `__init__()` method. We can see we are calling a new `super()` method. The `super()` method in Python returns a proxy object that delegates method calls to a parent or sibling of type. That definition may seem a little confusing but what that means in this case is that we have used `super()` to call the `__init__()` method of the *Rectangle* class, which allows us to use it without having to re-implement it in the *Square* class. Now, a square's length is *length* and its width is also *length* as we passed length in twice to the call to the `super()`'s `__init__()` method.

Since we've inherited *Rectangle* in the *Square* class, we also have access to the `area()` and `perimeter` methods without having to redefine them. We can now use these classes as follows:

```

>>> rect = Rectangle(2, 3)
>>> square = Square(2)
>>> print(rect.area())
6
>>> print(square.area())
4
>>> print(rect.perimeter())
10
>>> print(square.perimeter())
8

```

As you can see, the `super()` method allows you to call methods of the superclass in your subclass. The main use case of this is to extend functionality of the inherited method.

With inheritance we can also override methods in our child classes. Let's add a new class called a *Cube*. The cube will inherit from *Square* (which inherits from *Rectangle*). The *Cube* class will extend the functionality of the `area` method to

calculate the surface area of a cube. It will also have a new method called `volume` to calculate the cube's volume. We can make good use of `super()` here to help us reduce the amount of code we'll need.

Let's look at how we do this:

```
1. class Square(Rectangle):
2. def __init__(self, length):
3. super().__init__(length, length)
4.
5. class Cube(Square):
6. def area(self):
7. side_area = super().area()
8. return 6 * side_area
9.
10. def volume(self):
11. side_area = super().area()
12. return side_area * self.length
```

We can now use this class as shown:

```
>>> cube = Cube(2)
>>> print(cube.area())
24
>>> print(cube.volume())
8
```

As you can see, we haven't had to override the `__init__()` method as it's inherited from `Square`. We have however overridden the `area()` method from the parent class (`Square`) on the `Cube` to reflect how the area of a cube is calculated. We have also used the `super()` method to help us with this.

We have also extended the functionality of cube to include a `volume()` method. Again, we have used the `super()` method to help with this.

As you can see, by using inheritance we have greatly reduced the amount of code compared to how we would have implemented these three classes previously.

I just want to go back to overriding the `__init__()` method for a second. Let's assume a *Person* class and an *Employee* class that is derived from *Person*. Everything about an employee is the same as a *Person* but the employee will also have an employee ID.

Let's look at how we would implement these:

```
1. class Person:
2. def __init__(self, name, age):
3. self.name = name
4. self.age = age
5.
6. def show_name(self):
7. print("Name: " + self.name)
8.
9. def show_age(self):
10. print("Age: " + str(self.age))
11.
12. class Employee(Person):
13. def __init__(self, name, age, employeeId):
14. super().__init__(name, age)
15. self.employeeId = employeeId
16.
17. def show_id(self):
18. print("Employee ID: " + str(self.employeeId))
```

We can now use these as follows:

```
>>> person = Person("John", 32)
>>> emp = Employee("Simon", 25, 4532245)
>>> person.show_name()
Name: John
>>> person.show_age()
Age: 32
>>> emp.show_name()
Name: Simon
>>> emp.show_age()
Age: 25
>>> emp.show_id()
Employee ID: 4532245
```

The reason I'm showing you this is because, although I've already shown how to override methods, people get confused when overriding the `__init__()` method to also include new attributes on the derived class.

## 20.3 - Multiple inheritance

So far we've looked at single inheritance. That is, child classes that inherit from a single parent class. Multiple inheritance is when a class can inherit from more than one parent class.

This allows programs to reduce redundancy, but it can also introduce a certain amount of complexity as well as ambiguity. If you plan on doing multiple inheritance in your programs, it should be done with care. Give thought to your overall program and how this might affect it. It may also affect the readability of your programs.

For example:

```
1. class A:
2. def method():
3. # your code here
4.
5. class B:
6. def method():
7. # your code here
8.
9. class C(A, B):
10. pass
11.
12. >>> C.method()
```

Which *method* is the *C* class referring to? The method from A or the method from B? This may make your code harder to read. In Python, when calling *method()* on C, Python first checks if C has an implementation of *method* then it checks if A has implemented *method*. It has in this case so use that. If it hadn't then Python would check each parent class in turn (A, B, ..... ) and so on until it finds the first parent class to have implemented method.

I'm not going to go too deep into multiple inheritance as it's quite self-explanatory on how to use it, so I'm only going to give one example of how it can be used:

```
1. class CPU:
2. def __init__(self, num_registers):
3. self.num_registers = num_registers
4. # Other methods
5.
6. class RAM:
7. def __init__(self, amount):
8. self.amount = amount
9. # Other methods
```

```

10.
11. class Computer:
12. def __init__(self, num_registers, amount):
13. CPU.__init__(num_registers)
14. RAM.__init__(amount)
15.
16. # Other methods

```

This can be used as follows:

```

>>> comp = Computer(16, 32)
>>> print(comp.num_registers)
16
>>> print(comp.amount)
32

```

## 20.4 - Exercises

### Question 1

Create two classes: *Person* and *Employee*. The person class should have *name* and *age* attributes. The employee class should have one additional *employeeId* attribute. The employ should also have *clock\_in* and *clock\_out* methods. When implemented correctly, you should be able to use the two classes as follows:

```

>>> person = Person("John", 33)
>>> employee = Employee("Tom", 42)
>>> employee.clock_in()
Tom has clocked in.
>>> employee.clock_out()
Tom has clocked out.

```

Note: You are to use inheritance to implement the employee class.

### Question 2

Create a *Shape* class. The shape class should have two attributes: *num\_sides* and *side\_length*. It should also have one method called *show\_type()*

Create two other classes: *Triangle* and *Pentagon*. Each of these classes should inherit from the *Shape* class.



Both the *Triangle* and *Pentagon* classes should have *area()* methods.

When implemented correctly, you should be able to use your classes as follows:

```
>>> shape = Shape(7, 2)
>>> shape.show_type()
I am a Shape
>>> triangle = Triangle(3) # This refers to side length
>>> triangle.show_type()
I am a triangle
>>> triangle.area()
3.9
>>> pent = Pentagon(4) # This refers to side length
>>> pent.show_type()
I am a pentagon
>>> pent.area()
27.53
```

Note: The triangle is an equilateral triangle and the pentagon is a regular pentagon. You can find the formulae for both of their areas online.

### Question 3

Create a *Clock* class. It should have three attributes: *hour*, *minute*, and *second*. It should also have a method called *show\_time()* that displays the time in 24 hour time. It should also have a method called *tick()* that advances the time by 1 second.

Create a *Calendar* class. It should have three attributes: *day*, *month*, and *year*. It should have two methods, one called *show\_date()* and the other called *next()* that advances the date to the next day.

Create a third class called *CalendarClock* that inherits from both *Clock* and *Calendar*. It should have one method called *tick()* that advances the time by 1 second, and a second method, *show\_date\_time()* that shows the date and time.

When implemented correctly, you should be able to use your classes as follows:

```

>>> clock = Clock(23, 59, 59)
>>> clock.show_time()
The time is 23:59:59
>>> clock.tick()
>>> clock.show_time()
The time is 00:00:00
>>> cal = Calendar(30, 12, 2019)
>>> cal.show_date()
The date is 30/12/2019
>>> cal.next()
>>> cal.show_date()
The date is 31/12/2019
>>> calClock = CalendarClock(31, 12, 2019, 23, 59, 59)
>>> calClock.show_date_time()
The date is 31/12/2019 and the time is 23:59:59
>>> calClock.tick()
>>> calClock.show_date_time()
The date is 01/01/2020 and the time is 00:00:00

```

Note: There are a few ways of completing this question, however I want you to use multiple inheritance. If you do, then the code for CalendarClock should be quite short.

# Chapter 21 - Basic Data Structures

The backbone of every piece of software you will write will consist of two main things: data and algorithms. We know that algorithms are used to manipulate the data within our software (and that manipulation should be done as efficiently as possible). For this reason, it is important to structure our data so that our algorithms can manipulate the data efficiently.

We have met various data structures already, lists and dictionaries to name a few. You also know that using lists for example, makes it very easy for us to store and manipulate sequences of data (Trust me, without this abstraction it becomes a bit of a pain).

We have also seen that data structures allow us to model systems that we see in the real world. For example, we could design and build a class that allows us to model a library. We could easily use this to build a library management system.

In this chapter we're going to look at two fundamental data structures that are used everywhere in computing. We're also going to look at some examples of how to take advantage of them.

## 21.1 - The Stack

The stack is one of the most fundamental data structures in computing. Every time you run a program, that program takes advantage of a stack. I'll first explain how it works, then I'll explain how your computer takes advantage of it.

A stack is a linear data structure that follows a particular order in which the operations are performed. A stack is a LIFO (Last in First Out) data structure. That is, the last element entered into the stack is the first element that will leave the stack. You can think of a stack as a stack of dinner plates. Plates are stacked, one on top of the other. The last plate to be put on top of the stack will be the first plate to be removed as we can't remove the plate at the bottom (or the stack of plates would fall over and they'd all smash).

A stack (usually) has 4 main operations: push (add an element to the top), pop(remove an element from the top), top (show the element at the top) and

isEmpty (is the stack empty). We can also add a length method and it is usually useful!

Here is an illustration of a stack

```
1. #####
2. # Stack
3. #####
4.
5. |-----| <-- Top
6. |-----|
7. |-----|
8. |-----|
9. |-----|
10. |-----|
11. |-----|
12.
13.
14. #####
15. # If we pop:
16. #####
17.
18. |-----| <-- Pop
19.
20. |-----| <-- New top
21. |-----|
22. |-----|
23. |-----|
24. |-----|
25. |-----|
26.
27.
28. #####
29. # If we push
30. # a new element:
31. #####
32.
33. |-----| <-- Push new
34. |-----| element
35. |-----| (new top)
36. |-----|
37. |-----|
38. |-----|
39. |-----|
40. |-----|
```

The stack is such a fundamental data structure that many computer instruction sets provide special instructions for manipulating stacks. For example, the Intel Pentium processor implements the x86 instruction set. This allows for machine language programmers to program computers at a very low level (Assembly language for example).

A very special type of stack called the "Call stack" or "Program stack", usually shortened to "The stack" is what some of the instructions in the x86 instruction set manipulate.

Let's consider what happens with the following program at a low level (For anyone with a solid understanding of computers this is a little high level but a good example).

```

.
.
.
print(4)
.
.
.

```

In the above Python snippet, we have some set of instructions executing in order (Like we have always seen). At some point we call the print function to output the number 4 to our terminal window. This is where the stack comes into play. Lets look at how our program might be stored in memory (At a high level)

Memory Address	Instruction	
-----		
0x0000001	.....	# The interal code for the print
function might		
0x0000002	.....	# be stored at 0x0...01 to 0x0...03.
0x0000003	.....	
.		
.		
.		
0x0000004	.....	# Some instruction before the print.
0x0000005	print(4)	# The call to print.
0x0000006	.....	# Some instruction after print (The
next		
.		# instruction to be executed).
.		
.		

(I'M AWARE THIS ISN'T WHAT ACTUALLY HAPPENS BUT LET'S GO WITH THIS FOR NOW).

When we run a program, it is loaded into memory as machine code (Code the computer knows how to execute). Instructions are stored sequentially. In this view, we can look at functions as "mini programs" inside our main program. That is, the code for them is stored elsewhere in memory. When we call `print(4)` we are basically saying, jump to where the code for print is and start sequentially executing instructions until the function has completed, then return to the original location and continue on executing where you left off.

Before I explain this, the CPU has many special pieces of memory within it, called registers (These are like tiny pieces of RAM, usually 32-bits or 64-bits in size). One of these registers is called the instruction pointer register often abbreviated to IP. It contains the memory address of the next piece of code to execute.

In this case we execute the instruction at `0x00000004` ( then increase the IP), in the next instruction we call `print(4)` which is at location `0x00000005` yet the code for the print function is stored at `0x00000001` to `0x00000003`. This is where the stack comes into play. We know that when we print the number we want to continue where we left off and execute the instruction at `0x00000006`. In this case we push the return address (`0x00000006`) onto the stack, set the IP to `0x00000001` and start executing the code for the print function. When we're finished, we pop the return address off the stack (`0x00000006`) and place this in the IP. Now our program continues to execute where we left off.

I'm sure you can imagine how recursion works now at this level (essentially a series of push, push, push ..... pop, pop, pop).

Things are a lot more complex than this, but it isn't necessary for the explanation of how stacks are used at the most fundamental levels. Don't worry too much if you didn't understand all of that. It isn't necessary for this book, but it shows just how important the stack is.

Stacks have many high levels uses too, for example, undo mechanisms in text editors in which we keep track of all text changes in a stack. They may be used in browsers to implement a back/forward mechanism to go back and forward between web pages.

Let's look at the code for a stack in Python. We will implement a stack using a list

```
1. class Stack:
```

```

2. def __init__(self):
3. self.stack = []
4.
5. def push(self, elem):
6. self.stack.append(elem)
7.
8. def pop(self):
9. if len(self.stack) == 0:
10. return None
11. else:
12. return self.stack.pop()
13.
14. def isEmpty(self):
15. if len(self.stack) == 0:
16. return True
17. return False
18.
19. def top(self):
20. return self.stack[-1]

```

This is a very simple version of a stack, but we can now use our stack as follows:

```

1. stack = Stack()
2.
3. stack.push(1)
4. stack.push(4)
5. stack.push(2)
6.
7. print(stack.top())
8.
9. print(stack.pop())
10. print(stack.pop())
11. print(stack.pop())

```

We will get the following output if we run this program

```

2 # top
2
4
1

```

When we call the `top()` method, we don't actually remove the top element, we are just told what it is. When we use `pop()` method, we do remove the top element.

We will look at an example of where this can be used a bit later.

## 21.2 - The Queue

Queues in some sense similar to stacks. They are a linear data structure except rather than the operation order being Last in First Out, they are FIFO (First in First Out). They behave exactly like any queue you may think of in real life. A

queue at a supermarket for example, you get in line at the back and wait until you get to the front to leave.

In computing, a queue may be useful in CPU scheduling. In computers, there are many things happening at what seems to be, the same time. For example, you may have music playing as you read an article. There are many things happening here. Your music is being played, your monitor is being updated (refreshed) as you scroll through your article and you're using your mouse wheel to indicate when to scroll. In a simple sense, your CPU can only deal with one thing at a time. Your music application needs time on the CPU to output the next bit of music, as you scroll while this is going on, your mouse driver needs CPU time to tell the computer what to do, then your monitor drivers need CPU time to refresh the pixels on your screen. This all seems to be happening at the same time, except it isn't (computers are just so fast it appears it is). What's actually happening in this situation is each program is quickly using the CPU, then jumping back in the queue. This is happening over and over again. A CPU scheduling algorithm deals with which program gets to go next on the CPU and may implement some variation of a queue to handle this. (A priority queue for example).

They have similar methods for adding and removing except they are `enqueue()` and `dequeue()` respectively.

I think you get the gist of how you may visualize a queue so I'm just going to jump to the code for implementing a queue. Again, we will implement a queue using a list.

```
1. class Queue:
2. def __init__(self):
3. self.q = []
4.
5. def enqueue(self, elem):
6. self.q.append(elem)
7.
8. def dequeue(self):
9. if len(self.q) != 0:
10. return self.q.pop(0)
11.
12. def isEmpty(self):
13. if len(self.q) > 0:
14. return False
15. return True
```

And there you go. A very simple Queue class. Notice how we pop from index 0. Recall that the `pop()` method for lists takes an optional argument which is the index of the list you want to remove an element from. If we don't supply



this optional argument it defaults to -1 (the end of the list, which is what stacks do). Here we remove the element at the start of the list but add elements to the end.

Running the following program, will produce the following output

```
1. queue = Queue()
2.
3. queue.enqueue(1)
4. queue.enqueue(4)
5. queue.enqueue(2)
6.
7. print(queue.isEmpty())
8.
9. print(queue.dequeue())
10. print(queue.dequeue())
11. print(queue.dequeue())
12.
13. print(queue.isEmpty())
```

*False*

*1*

*4*

*2*

*True*

## 21.3 - Examples

The first example I'm going to look at is for stacks. This question is known to have been asked during past Google coding interviews!

The problem is: Given an input string consisting of opening and closing parentheses, write a python program that outputs whether or not the string is balanced.

For example:

*Input: ([{}])*

*Output: Balanced*

*Input: (([)])*

*Output: Unbalanced*

*Input: {()}[[]]}*

*Output: Balanced*

*Input: [()(){}]*

*Output: Unbalanced*

One approach we can take with this problem is to use a stack. Each time we encounter an opening bracket we append that to the stack. When we encounter a closing bracket we pop from the stack and check that the element we popped is the opening bracket for the closing bracket. For example, `[` is the opening bracket for `]`, similarly, `{` is the opening bracket for `}` and so on.

Here is a possible solution. Assume our stack class from earlier exists:

```
1. opening = ["(", "[", "{"]
2. # Note that pairs is a dictionary that maps the closing bracket
3. # to the opening bracket
4. pairs = {")": "(", "]" : "[", "}": "{"}
5.
6. def balanced(input_string):
7. stack = Stack()
8. for bracket in input_string:
9. if bracket in opening:
10. stack.push(bracket)
11. elif stack.isEmpty():
12. return False # Can't pop anything as stack is empty so it's unbalanced
13.
14. elif pairs[bracket] != stack.pop():
15. return False # The brackets didn't match up so it's unbalanced
16.
17. return not stack.isEmpty() # If stack is empty at the end,
 # return True, else, False
```

We can make better use of the pairs map and make our code a bit neater but it won't be as readable so I've taken a slightly longer approach.

Now we can run our code as follows to get the desired output from the example input and output above:

```
1. # ABOVE CODE HERE
2.
3. inputString = input()
4. if (balanced(inputString)):
5. print("Balanced")
6. else:
7. print("Unbalanced")
```

I don't have an example usage of queues as I can't think of anything useful to do with them with the knowledge we've got at the moment. Don't worry though, there will be a good question on queues coming up soon.

## 21.4 - Exercises

### Question 1

You are given two strings  $S$  and  $T$ . Both strings either contain alphanumeric characters or a '#' character. A '#' means backspace. Your goal is to use your knowledge of stacks to design a function that will decide whether both strings are equal after the back space operation has been applied. For example:

*Input:  $S = \text{"xy\#z"}\text{, } T = \text{"xw\#z"}$*

*Output: True*

*Explanation: When the backspace operation is applied, both strings become "xz"*

*Input:  $S = \text{"abcd\#\#"}\text{, } T = \text{"acbd\#\#"}\mathbf{\text{"}}$*

*Output: False*

*Explanation:  $S$  becomes 'ab' and  $T$  becomes 'ac' which are not the same*

*Input:  $S = \text{"z\#\#y"}\text{, } T = \text{"\#d\#y"}$*

*Output: True*

*Explanation: Both strings become 'y'*

*Input:  $S = \text{"er\#\#"}\text{, } T = \text{"u\#u\#"}\mathbf{\text{"}}$*

*Output: true*

*Explanation: Both  $S$  and  $T$  become '' (empty string)*

### Question 2

It is possible to create a queue through the use of two stacks. Implement a queue class like above, except rather than using a list to implement the queue you use two stacks.

This might be a little tricky, you'll have to think about this.

### Question 3

It is also possible to create a stack through the use of two queues. Implement a stack class like the one we have seen earlier, except rather than using a list to implement the stack, use two queues.

This might be a little tricky, you'll have to think about this.

## Mini Project 1

You are to design an RPN calculator. RPN stands for Reverse Polish Notation. An RPN calculator is also known as a postfix calculator. We as humans do math as follows, for example,  $4 + 7$  or we must use BEMDAS rules:  $8 + ((10 * 3) / 2)$ . In RPN, we would represent these as:  $4\ 7\ +$  and the last expression we represent as  $8\ 10\ 3\ 2\ /\ * +$ . What's happening here is the operator (+ is in the prefix position rather than infix position as we are used to). This might look a little confusing but it's actually quite simple and we can use a stack to solve this problem.

Let's look at how this works:

```
Expression: 4 7 +
1) Read the first character (4)
2) This is a number so we push it to the stack
3) Read the next character (7)
4) This is also a number so we push it to the stack
5) Read the next character (+)
6) This is an operator. It's a binary operator as it works on two
 operands, so we
 pop two elements from the stack
7) Add the two numbers together
8) Push the result back onto the stack
```

If we had a unary operator such as negation then we would pop one element from the stack, calculate the result and push it back.

You are to implement the following operators in your calculator:

- Addition: (+)
- Subtraction: (-)
- Multiplication (\*)
- Floor division (//)
- Negation (n)
- Power (e)

Your calculator should read in RPN expressions, one line at a time and evaluate them. There is one additional operator to implement. This is  $p$  which stands for print. If you encounter  $p$  then print the result. You can assume  $p$  will always occur at the end of the input.

Example:

*Input: 4 7 3 + + p*

*Output: 14*

*Infix: 4 + 7 + 3*

*Input: 5 8 \* n p*

*Output: -40*

*Infix: -(5 \* 8)*

*Input: 3 6 1 + + 2 e p*

*Output: 100*

*Infix: (3 + 6 + 1)^2*

Think carefully about this problem. It's even a great beginner project to host on your Github account to showcase your work to the world!

# Chapter 22 - More Advanced Data Structures

In the previous chapter, we learned about some simple yet fundamental data structures with rather simple implementations. In this chapter we're going to take a look at a couple of new data structures. They'll also be a little bit more advanced in their implementation.

## 22.1 - Linked Lists

The first data structure we're going to look at in this chapter is a **Linked List**.

A linked list is a linear data structure where each element is a separate element. Linked List objects are not stored at contiguous locations. Instead the linked list objects are linked together using pointers.

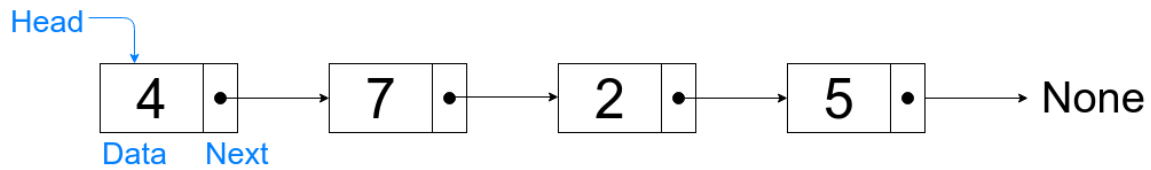
Throughout this book I may have referred to Python lists as arrays and visa-versa. Python lists have a complicated implementation under the hood. This allows for lists to be of arbitrary size, grow and shrink and store many items of various types.

In many other programming languages, an array is much like a fixed size list that stores elements of a specific type. They store elements contiguously in memory. That isn't necessarily the case in Python.

Python is implemented in a language called 'C'. This language is low-level compared to Python. In the C language, Python lists are implemented with a data structure somewhat similar to a linked list. It's a lot more complicated than a linked list but a lot of the concepts and techniques we'll learn here will apply to the C Python implementation of lists.

There are many variations of linked lists but in this chapter, we're going to look at one variation called a **Singly Linked List**.

Below is a diagram of a Singly Linked List:



There are various elements to this so let me explain:

- Head: The head is the first element in the list
- Tail: The tail isn't marked here but its the last element (5 in this case)
- Node: A node consists of two things; The data and a pointer to the next element (Next)

Linked Lists have various advantages over arrays (not Python lists). These include dynamic size (they can grow and shrink as needed much like python lists, whereas arrays can't do this) and ease of insertion and deletion.

When we delete something from a Python list, the list must be resized and various other things must be done (this is done under the hood so we don't worry about this).

There are also some disadvantages to Linked Lists compared to arrays. We can't index them, therefore if we wish to access an element we must iterate through each element, starting from the head, in order until we find the element we are searching for.

From the above diagram we can break the problem of implementing a linked list down into two things:

1. A node class: This will contain the data the node will store and a pointer to the next node.
2. Linked List class: This will contain the head node and methods we can perform on the linked list.

Let's look at the node class:

```
1. class Node:
2. def __init__(self, elem):
3. self.elem = elem
4. self.next = None
```

This is our linked list node class. It's very simple and only contains two attributes. The first is the data the node will store and a *next* attribute. The next attribute is set to **None** by default as it won't point to anything initially.

This concept of a node is very powerful when it comes to data structures. It gives us a lot of flexibility when developing other data structures as we'll see later on.

Let's now look at the first attempt at our linked list:

```
1. class LinkedList:
2. def __init__(self):
3. self.head = None
```

When we initialize our `LinkedList` it will be empty, that is, the head will point to nothing (`None`). That's all we need to start implementing methods.

To get started, we'll take a look at the `add()` method. Firstly, we need to think about how we're going to go about doing this. In our linked list, we'll be adding new elements to the end of the list. Firstly, we need to check if the head is empty. If it is, then we just point the `LinkedList` head to a new node. Otherwise, we need to "follow" this trail of pointers to nodes until one of the node's `next` attribute points to `None`. When we find this node then we update its `next` attribute to point to the element we're trying to add.

Let's look at how that's done:

```
1. class LinkedList:
2. def __init__(self):
3. self.head = None
4.
5. def add(self, elem):
6. if self.head == None:
7. self.head = Node(elem)
8. else:
9. curr = self.head
10. while curr.next != None:
11. curr = curr.next
12. curr.next = Node(elem)
```

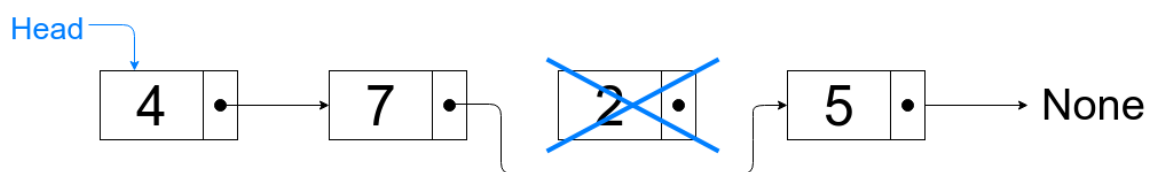
Here we check that the head of the linked list isn't empty. If it has then we assign a new `Node` to the linked list `head`. Otherwise we create a new variable called `curr`. This keeps track of what node we're on. If we didn't do this we would end up updating the linked list's `head` node by mistake. After we create this new variable, we loop through the elements of the list, going from node to node *through* the `next` pointer until we find a node whose `next` pointer is empty. When we find that node, we update its `next` pointer to point to a new `Node`.



This may take a little time to wrap your head around or understand but the more you work with data structures like this or ones similar, the more sense it will make.

Let's look at our deletion method. We have many options when deleting elements from a linked list just like we do with adding them. We could remove the first element, the last element or the first occurrence of an element. Since our linked list can hold many occurrences of the same data, we will remove the first occurrence of an element.

Let's look at how that may be done with the help of a diagram:



As you can see, from this diagram we want to delete the node that contains 2. To do this we find the first node who's next pointer points to a Node that contains the value 2. When we find this Node we just update it's next pointer to point to the same node the Node we want to delete points to. Essentially, we re-route the next pointer to bypass the element we want to delete. This is quite easy to do.

Let's look at how it's done.

```
1. class LinkedList:
2. def __init__(self):
3. self.head = None
4.
5. def delete(self, val):
6. if self.head == None:
7. return
8. if self.head.elem == val:
9. self.head = self.head.next
10. return val
11.
12. curr = self.head
13. while curr.next != None and curr.next.elem != val:
14. curr = curr.next
15.
16. if curr.next == None:
17. return
18. else:
19. curr.next = curr.next.next
20. return val
```

We have to cover a couple of cases here. The first is that the list is empty in which case return nothing. The second is that the head of the linked list is the

element we want to delete. If it is then we make the head of the list the second `Node` in the list (Which may be `None` but that's ok, it just means the list only had one item). Finally, if it is neither of those cases we traverse the list using linear search until we find the first occurrence of the element then we 're-route' the `next` pointer of the `Node` that points to the `Node` we want to delete, to the `Node` the `next` pointer of the `Node` we want to delete points to.

That last part might seem confusing but we're doing what you see in the diagram. It's probably best at this point if you're confused by that to draw this scenario out on paper to help clarify things.

There are a couple of other useful methods we can add here but we'll leave them for later.

Just before I finish up on linked lists, I want to talk a little bit more about the complexities of their methods and why you might use them (or not).

The run time complexity of the add method is  $O(n)$  in this case as we must iterate over each entry in the list to find the last element. The runtime complexity of the deletion method is also  $O(n)$ . Although we don't always iterate over each element in the list, Big-O deals with the worst case, which is going through the entire list. There are optimizations we could make though and we'll get to them later on too.

Linked Lists in Python usually don't have a use case. This is because Python lists are already very well optimized and dynamic. However, in languages such as C or C++, Linked Lists are essential (Where dynamic arrays, like python lists, may not exist). They also pop up in technical interviews so you're best to know them. In fact, if I was hiring an engineer, I'd be very skeptical about hiring one who didn't know how to code a linked list.

## 22.2 - Binary Search Trees (BSTs)

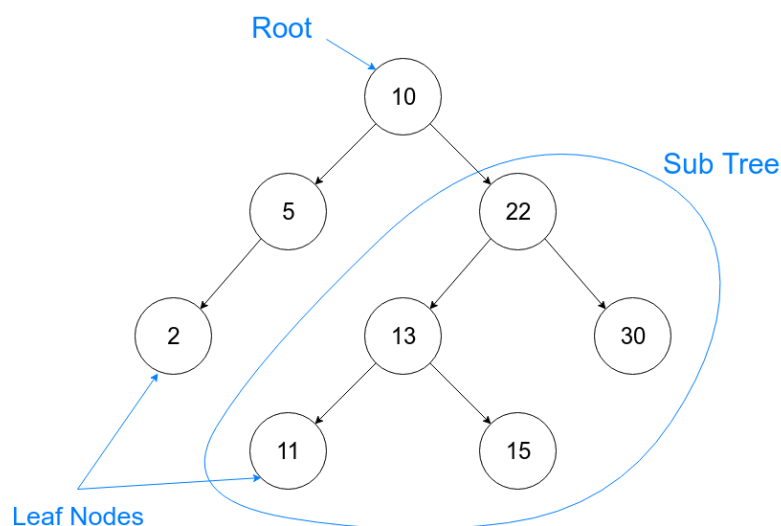
Before I start this section, I want to give you a warning. We will be using recursion quite a lot here. If recursion is not your strong point (which it probably won't be at this point), perhaps brush up on the recursion section. However, this section might be a good place to help you understand recursion. BSTs are what helped me wrap my head around recursion (well, at least that's when recursion clicked for me).

A binary search tree (BST), is somewhat different to the data structures we have met so far. It is not a linear data structure. It is a type of ordered data structure that stores items. BSTs allow for fast searches, addition and removal of items. BSTs keep their items in sorted order so that the operations that can be performed on them are fast, unlike a linked list who's add and remove operations are  $O(n)$ . The add and remove operations on a BST are  $O(\log n)$ . Searching a BST is also  $O(\log n)$ . This is the average case for those three operations.

This is a similar situation to linear search vs. binary search which we looked at earlier.

In computer science, a 'tree' is a widely used *abstract data type* (a data type defined by its behaviour not implementation) that simulates a hierarchical tree structure (much like a family tree), with a root node and subtrees of children with a parent node represented as a set of linked nodes.

A binary tree is a type of tree in which each Node in the tree has *at most*, two child nodes. A binary search tree is a special type of binary tree in which the elements are ordered. A diagram might help you visualize this, so here's one:



There is a lot to take in here so let me explain. The root of this tree is the node that contains the element 10. The big thing labeled Sub Tree is the *right sub tree* of the root node.

10 is the *parent node* to 22 and 5 and similarly, 5 is a child node of 10.

A leaf node is a node that has no children.

As you can see from this diagram, it is a binary tree as each node has at most two child nodes. It also satisfies an important property that makes it a binary search tree. That is, that if we take any node, **10** for example, everything to the right is greater than it and everything to the left is less than it. Similarly, if we look at the node **13**, everything to the right is greater than **13** and everything to the left is less than **13**.

What makes searching fast is this, if we wanted to check if **2** was in the tree, we start at the root node and check if **2** is less than the value at the root. In this case **2** is less than **10** so we know we don't need to bother searching the root node's right sub tree. Essentially what we've done here is cut out everything in that blue circle as the element **2** will not be in there.

Let's look at implementing a binary search tree. Again, the concept of a **Node** is important here. Our **Node** class for a binary search tree will be slightly different than what it was for a linked list. Let's take a look:

```
1. class Node:
2. def __init__(self, elem):
3. self.elem = elem
4. self.left = None
5. self.right = None
```

As you can see here, each node will have an element, a left child and a right child (either or both of which may be **None**).

Let's make our first attempt at a **BinarySearchTree** class:

```
1. class BinarySearchTree:
2. def __init__(self):
3. self.root = None
```

Pretty simple so far. Basically, what we can do if we want to add, remove or search for an element is follow pointers from left sub trees or right sub trees until we find what we are looking for.

This time we can look at the search operation first. It will be a good look at how to use recursion in a practical situation. We either want to return **None** if the element is not found or return the **Node** that contains what we're looking for if the element is found.

```

1. class BinarySearchTree:
2. def __init__(self):
3. self.root = None
4.
5. def search(self, value):
6. return self.recursive_search(self.root, value)
7.
8. def recursive_search(self, node, value):
9. if node is None or node.elem == value:
10. return node
11. if value < node.elem:
12. return self.recursive_search(node.left, value)
13. else:
14. return self.recursive_search(node.right, value)

```

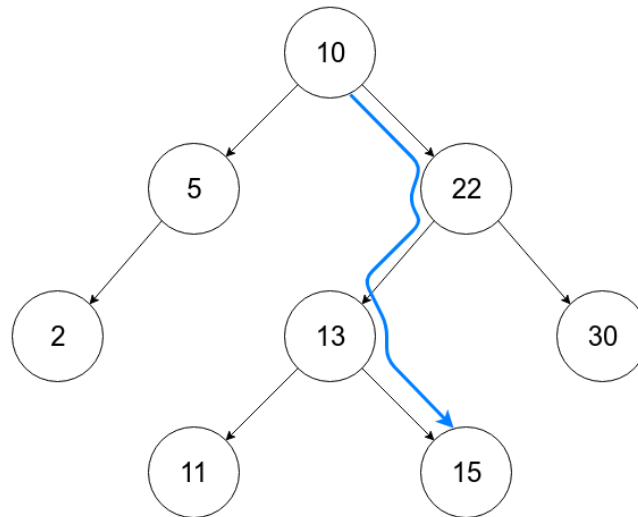
This may seem a little odd as we have two search functions but there is good reason for it. The `search` method allows us to call the `recursive_search` method and pass in the root node, that way, we now have a 'copy' of the root node and not the root node itself so we don't end up in an infinite loop when recursively searching. Again, the recursive nature of this is very difficult to explain so I'd try working through an example on paper to help clarify after the following explanation.

Inside the `recursive_search` method we have a *base case* which checks if we've reached `None` (we didn't find what we're looking for) or we've reached the `Node` that contains what we're searching for.

If we don't pass our base case then we check if the value we're searching for is less than the value at the node we're currently at; if it is then we search the left subtree of that node by calling the `recursive_search` method and passing the current nodes `left` tree. Otherwise, the value we're searching for is greater than the value at the current node, in which case we call `recursive_search` and pass in the current nodes `right` tree.

Remember a node's left or right tree is simply a pointer to a node (we can look at this node as the root node of a sub tree).

For example, if we're searching for the value `15` in the tree from the diagram above, then the path we take looks like this:



Just to clarify, a leaf nodes' `left` and `right` 'trees' are `None`.

Next, we will look at inserting an element into the tree. What we're doing here is following pretty much the same thing we did with search except when we reach a leaf node on our path we will set its left or right pointer to point to a new node (depending on whether or not the value is less than or greater than the value at the child node).

Here's how we will approach this:

- Base case: If the node we are at is `None` then insert the new node here.
- Recursive case:
  - If the value we're inserting is less than the value of the current node then we check if left tree of the current node is `None`; If it is, then insert there, otherwise, call insert again and pass the left subtree.
  - If the value we're inserting is greater than the value of the current node then we check if the right tree of the current node is `None`; If it is, then insert there, otherwise, call insert again and pass the right subtree.

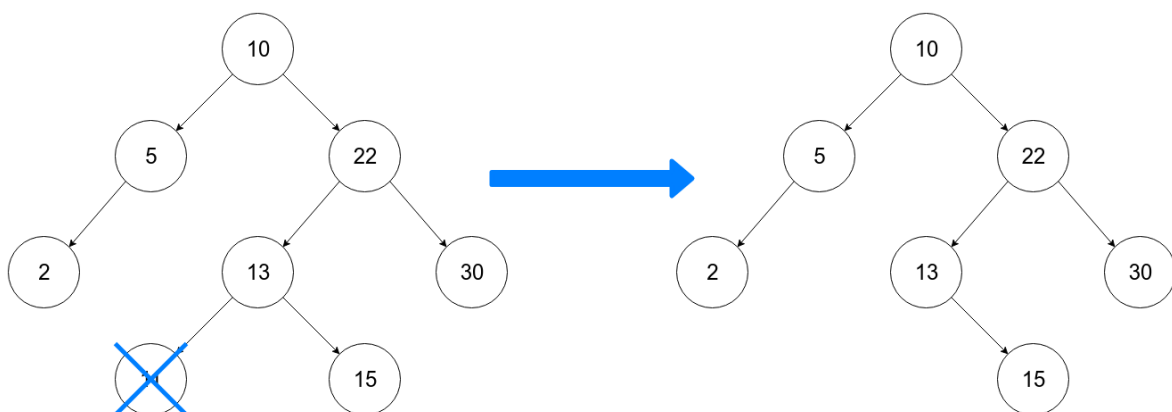
Here's how that's done:

```
1. class BinarySearchTree:
2. def __init__(self):
3. self.root = None
4.
5. def insert(self, val):
6. if self.root is None:
7. self.root = Node(val)
8. else:
9. self.recursive_insert(self.root, val)
10.
11. def recursive_insert(self, node, val):
12. if val < node.elem:
13. if node.left is None:
14. node.left = Node(val)
15. else:
16. self.recursive_insert(node.left, val)
17. else:
18. if node.right is None:
19. node.right = Node(val)
20. else:
21. self.recursive_insert(node.right, val)
```

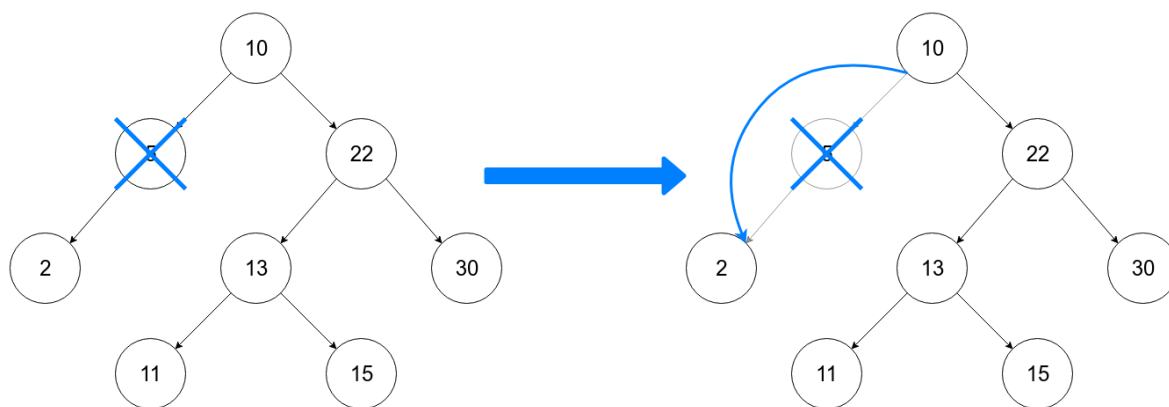
That's it. I hope you can see why recursion is useful here. It makes our code read better and when we're planning this out in our heads, it naturally fits into our algorithm.

Next, we're going to look deleting an element. This is quite a bit more difficult. With insertion, we always insert into one of the leaf nodes but if we're deleting something, then three possibilities arise. The first is the simplest case:

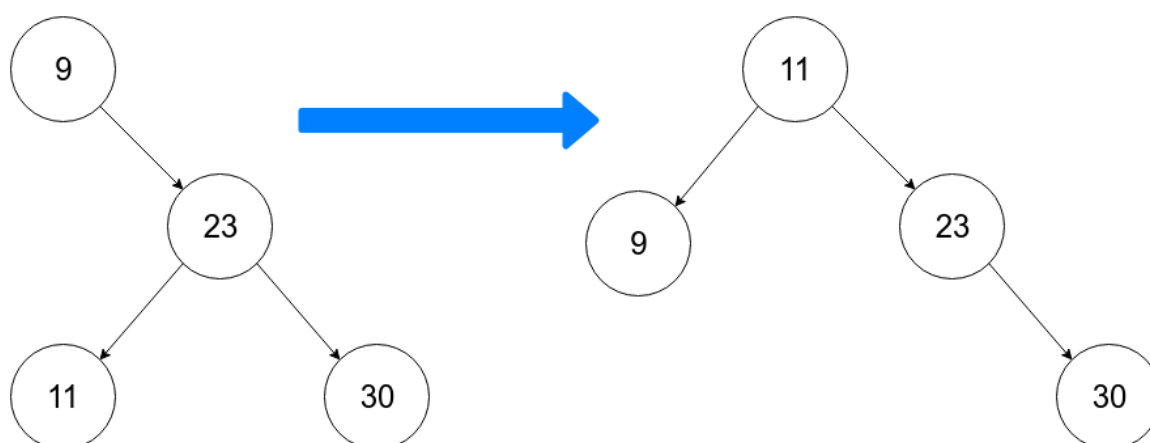
1. The node we're removing has no child nodes (i.e. a leaf node). In this case we can just remove the node.



2. The next case is when we're removing a node that only has one child. Again this is simple enough to deal with. In this case we just cut the node we're removing from the tree and link its child to its parent.



3. The final case is when we're removing a node that has two child nodes. This is the most complex case and requires some smart thinking. There is however, one useful property of BSTs that we can use to solve this problem. That is, the same set of values can be represented as different binary-search trees. Let's consider the following set of values:  $\{9, 23, 11, 30\}$ . We can represent this in two different ways:



Both of these trees are different yet they are both valid. What did we do here though to transform the first tree into the second?

- We started at the root node (9 in this case).
- Searched its right subtree for the minimum element (11 in this case)
- Replaced 9 with 11
- Hang 9 from the left subtree

We can take this idea and apply it to removing elements with two child nodes. Here's how we'll do that:

- Find the node we want to delete
- Find the minimum element in the right subtree



- Replace the element of the node to be removed with the minimum we just found.
- Be careful! The right subtree now contains a duplicate.
- Recursively apply the removal to the right subtree

When applying the removal to the right subtree to remove the duplicate, we can be certain that the duplicate node will fall under case 1 or 2. It won't fall under case 3 (It wouldn't have been the minimum if we did).

There are two functions we'll need here. One is the remove method and the other will find the minimum node. We get a free function here (we will have a `min` operation on our BST!).

Here's how to code it, I'll also comment the code as it's a bit complex:

```

1. class BinarySearchTree:
2. def __init__(self):
3. self.root = None
4.
5. def remove(self, value):
6. return self.recursive_remove(self.root, None, value)
7.
8. def recursive_remove(self, node, parent, value):
9. # Helper function
10. def min_node(node):
11. curr = node
12. while curr.left != None:
13. curr = curr.left
14. return curr
15.
16. # Base case (element doesn't exist in the tree)
17. if node == None:
18. return node
19.
20. # If element to remove is less than current node
21. # then it is in the left subtree. We must set the current
22. # node's left subtree equal to the result of the removal
23. if value < node.elem:
24. node.left = self.recursive_remove(node.left, node, value)
25.
26. # If element to remove is greater than current node
27. # then it is in the right subtree. We must set the current
28. # node's right subtree equal to the result of the removal
29. elif value > node.elem:
30. node.right = self.recursive_remove(node.right, node, value)
31.
32. # If the element to remove is equal to the value of
33. # current node, then this is the node to remove
34. else:
35. print(node.elem, not parent is None)
36. # Not the root node
37. if not parent is None:
38. # Node has no children (CASE 2)
39. if node.num_children() == 0:
40. if parent.left == node:
41. parent.left = None
42. else:

```

```

43. parent.right = None
44. # Node has only one child (CASE 2)
45. elif node.num_children() == 1:
46. # Get the child node
47. if node.left != None:
48. child = node.left
49. else:
50. child = node.right
51.
52. # Point the parent node to the child node of
53. # the node we're removing
54. if parent.left == node:
55. parent.left = child
56. else:
57. parent.right = child
58.
59. # Node has two children (CASE 3)
60. elif node.num_children() == 2:
61. # Get the min node in the right subtree of node to remove
62. smallest_node = min_node(node.right)
63.
64. # Copy the smallest nodes value to the node formerly holding
65. # the value we wanted to delete
66. node.elem = smallest_node.elem
67. self.recursive_remove(node, parent, value)
68. # Node to delete is the root node
69. else:
70. # Only 1 element in the tree (the root) - set the root to None
71. if node.num_children() == 0:
72. node = None
73. # Root has one child - make that the root
74. elif node.num_children() == 1:
75. if node.left != None:
76. self.root = node.left
77. if node.right != None:
78. self.root = node.right
79. elif node.num_children() == 2:
80. smallest_node = min_node(node.right)
81. node.elem = smallest_node.elem
82. self.recursive_remove(node, None, value)
83. return node

```

This is tough. Make sure you understand the delete operation, even if that means going through it over and over again. Use a pen and paper if you need! It is a good question for an employer to ask as it shows how someone approaches a tough problem (pointing out different scenarios, breaking the problem down, etc.).

That is all I want to cover on Binary Search Tree's for now. There are a few more methods we can add, and I'll leave that up to you in the exercises!

## 22.3 - Exercises

**Important:** Some of these exercises are quite difficult! Make use of problem-solving techniques to help you arrive at a solution. Sketching out what's happening on paper is a good way to help you out!

It's time to turn up the heat a bit on the difficulty of the questions. There are some really tough questions in here. Good luck!

## Question 1

We talked earlier about the run-time complexities of the Linked List methods we implemented. I mentioned that we could improve their complexities.

Currently, the `add()` method has a run-time complexity of  $O(n)$ .

Change the implementation of the `add()` method so that its run-time complexity is  $O(1)$ .

## Question 2

Change the implementation of the `add()` or `remove()` methods on the Linked List class in order to achieve a Linked List that behaves like:

- A Stack
- A Queue

Try to give optimized implementations of `add()` and/or `remove()` when doing this.

## Question 3

Another useful method our Linked List class may have is a `length()` method. That is, it returns the number of elements in the linked list.

Implement the `length()` method.

## Question 4

If you thought carefully about your implementation of the `length()` method from the previous exercise then you might be able to skip this question. However, I'm not expecting that you did.

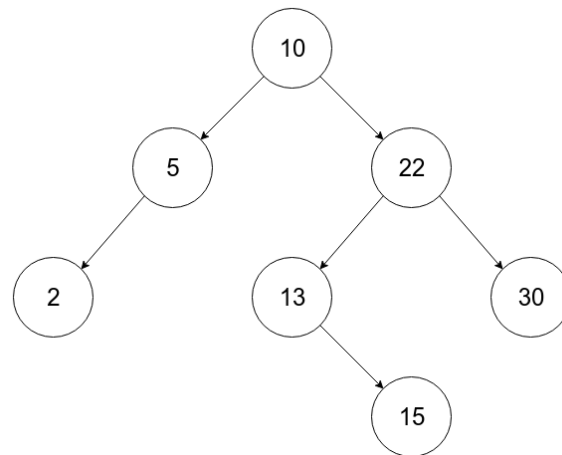
If my guess is correct, your `length()` method iterates over each element in the list, adding 1 to your length each time until you reach the end of the list. This will give your `length()` method a run-time complexity of  $O(n)$ .

We can do better than that though. How might you change the Linked List class so that the run-time complexity of your `length()` method is  $O(1)$ ?

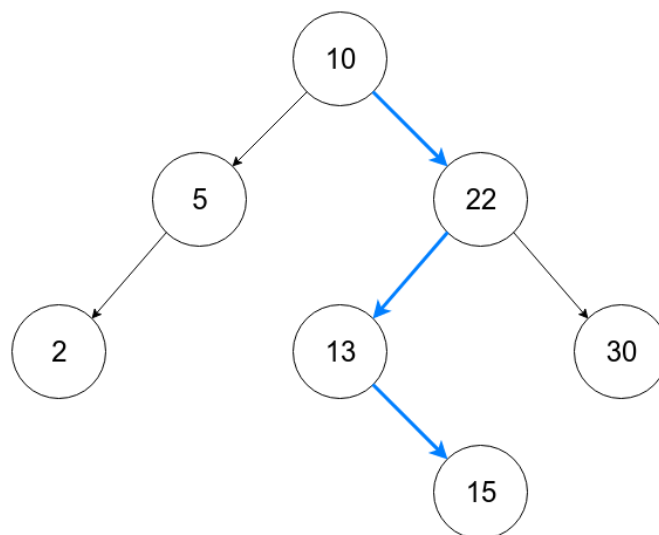
## Question 5

The *height* (depth) of a Tree is the number of edges on the longest path from the root node to leaf node. (An edge is basically the arrows in our diagrams that we've looked at before)

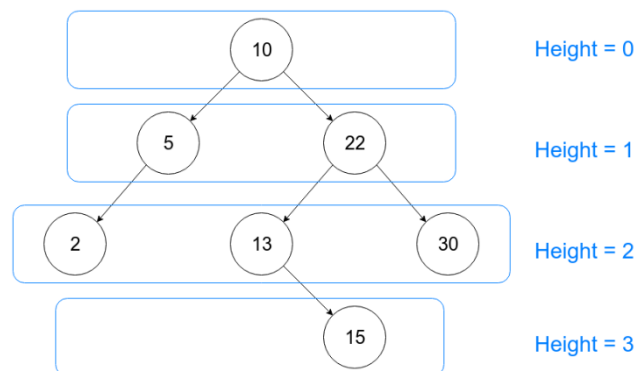
For example, the height of the following tree is 3



The height is determined by the number of edges in the longest path, like the following:



You can also view the height of a tree as such:



You should write a recursive method to find the height of any given binary search tree

## Question 6

### This question is very difficult

Consider what happens when we add elements to the tree in the following order: {2, 5, 7, 12, 13}. We end up with a Linked List. That means that our search and insert is no longer  $O(\log n)$ . It's now  $O(n)$ . This isn't really that great for us.

We can however, make an optimization to our binary search tree so that our search and insert are always  $O(\log n)$ , regardless of what order we enter the elements.

The optimization is that the difference between height of the left and right subtrees is no greater than one for all nodes. When the difference between left and right subtrees height is no greater than 1, we say the tree is **balanced**. This means, if we insert or remove an element and the height between left and right subtrees becomes greater than 1 then we must rearrange the nodes.

This optimization means that our binary search tree will become self-balancing.

This type of self-balancing binary search tree has a special name. It's called an **AVL Tree**.

Remember: This question is very difficult, I don't expect you to be able to answer this, but I encourage you to give it a go!