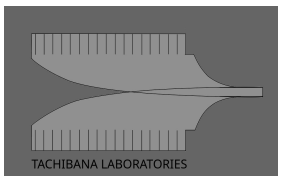
The Python Pilgrim Part One: The Elementary Bosons (ver. 1.0)

Tachibana Laboratories



"Let me see: four times five is twelve, and four times six is thirteen, and four times seven is -- oh dear! I shall never get to twenty at that rate!" —Lewis Carroll, Alice's Adventures in Wonderland

Preface

Python is not only another way to understand programming, it is an excellent starting point for those wanting to discover new ways to solve problems, and with a bit of grit perseverance your journey will take you to distant and unexpected places. A new journey can feel like awakening into a nonsensical world where all of the rules you know are challenged, and common sense is distorted as it was for every person on the frontier of their own, or humanity's knowledge, but as you will come to learn, you are not without hope. With the right map you will translate the reasoning and common sense you know into the rules you need to learn. You will come to discover that the ability to solve the kinds of problems you will face is not something that has to be learned from nothing, but something that runs parallel to your existing problem solving skills.

So what did Alice mean by this? It rather feels like we are stepping into a world of nonsense from the outset, but sometimes nonsense is just the result of some rule we don't yet understand. Oh, and to be clear x, n and k here are whole numbers, and $x^0 = 1$. "Let me see:..."

$4 \cdot 5 = \mathbf{12}$ in base 18 ($\mathbf{1} \cdot 18^1 + \mathbf{2} \cdot 18^0$), and $4 \cdot 6 = \mathbf{13}$ in base 21 ($\mathbf{1} \cdot 21^1 + \mathbf{3} \cdot 21^0$) and $4 \cdot 7 = \mathbf{14}$ in base 24 and $4 \cdot 8 = \mathbf{15}$ in base 27 ...and so on up to, $4 \cdot 13 = \mathbf{19}$ in base 39 ($\mathbf{1} \cdot 39^1 + \mathbf{9} \cdot 39^0$).

There is a pattern here which is as follows. For every x in base $3+3x$ where the product of $4x$ is a two digit number, the product is computed like $4x = \mathbf{n}(3+3x)^1 + \mathbf{k}(3+3x)^0$ where up until now $n = 1$. So why can't Alice get to 20 such that $n = 2$ and $k = 0$? This can be solved algebraically. Following the rules $4x = \mathbf{20}$ implies that $4x = \mathbf{2} \cdot (3+3x)^1 + \mathbf{0} \cdot (3+3x)^0$. Distributing the factor of 2 we get $4x = 6 + 6x$, and then algebraically manipulating we get $x = -3$ in base -6 which is not a number that can be reached by incrementing n by 1 each time, hence Alice "will never get to 20 at that rate". Negative bases are also not allowed for whole numbers. So Alice was on to something after all. Through the knowledge of certain rules we gained the ability to decode that which initially seemed defy being reasoned with easily.

The objective of this series is to make something alien, familiar. It is the author's goal to establish the fundamentals as clearly as possible to address the things that beginner programmers find confusing. With these fundamentals established the reader should be primed to fill in the blanks themselves with their own diligent research. To aid this step of the process I will include within this guide resources and tips as to how to fill in these gaps (your PDF reader will hopefully highlight something if it is a link to an external resource).

Good luck.

Glossary of common terms and syntax

: A single line comment symbol, put this before text you want to comment and don't want the Python interpreter to interact with

string: Strings are alphanumeric text that can represent any letters, numbers or symbols defined by the character encoding. They can be joined but arithmetic operations cannot be performed upon them.

integer : Negative or positive whole numbers including zero, every integer can be factored into a unique product of prime integers

float: A positive or negative real number, simply put a number that has decimal places as a decimal representation of a fraction of a whole number

boolean: A type of variable that can only be True or False, just like a bit can only be 1 or 0 (binary). Absolutely core to how computers work, but that's another topic.

= : Variable assignment. The object on the right is assigned to the variable on the left as such as x = 2, where integer 2 gets assigned to variable x " " : Anything these quotation marks enclose is defined as a string

\ : Division, numerator on left, denominator on right. Version specific behaviour is document in the Versions section

\\ : Integer division, divides and then rounds down to nearest one's place. Integer dividing a float will still return a float but it will be rounded

* : Multiplication, behaves the same on integers and floats. An integer can be multiplied by a float and this will give a float

+ : Addition (note this has a different definition when used with strings)

+ : Concatenation, which used like "string" + "string" with only string type variables, you will get "stringstring"

- : Subtraction

** : Power operator. For numbers x and y, x**y raises x to the power of y

== : Equivalence comparator, for objects a, b that are equal to each other, if a == b (equals) then the comparator will evaluate to True, else it will evaluate to False

not : Non-equivalence comparator. For objects a, b, that are not equal to each other if not (a == b) (a not equals b) then the comparator will evaluate to True, else it will evaluate to False

<, >, <=, >= : Arithmetic comparators. Less than, greater than, less than or equal to, greater than or equal to

% : Modulo, divides and then gives the remainder, numerator on left, denominator on right. If numerator < denominator then x%y=x. Works on floats

+=, -=, *=, /=, %=: For integers and floats, a+ = b means a = a + b, the same pattern follows for the other arithmetic operators

object : Object has a special meaning in object oriented programming relating to having attributes, methods, being instanced and having the mechanism of inheritance. However more generally an object can be any variable, function/ method or data structure. One sign of an object is that it can be assigned to a variable or passed as an argument

iterate (over) : To iterate is to do something repeatedly, so to iterate over something, like a list for example, is to repeat the same operation of accessing each element of the list sequentially.

iterable : An object that has the property that it can be iterated over. This means that the object can be broken down into elements that comprise it, which are other objects.

mutable/ immutable : a mutable object is one that can be changed as such as a list as it can have its elements deleted, modified and added. An immutable object is one that cannot be changed, like an integer or a string. When performing arithmetic on an integer the integer is not modified, instead a new integer is returned as the result. Immutable objects exist because only they can be hashed, and have greater runtime performance for certain operations

return : Return is a statement that breaks out of a function and passes an object back to the calling function, which can be assigned to a variable. This is explained in greater detail in the Function Definition section under Program Structure

list = [a, b, c] : Initialises a new list. Can contain greater than or equal to zero comma space elements. List can be initialised as empty by leaving the square brackets empty. List can contain any Python compatible typed elements, and data structures

list[x] : Accesses element of a list by index where x is an integer. Can be used to get or set an element

list.append(x) : Adds element x to list. A full list of list functions can be found **here**

tuple = (a, b, c) : A tuple is an immutable data structure that stores comma separated objects. Elements can be accessed by index in the same way as a list, but cannot be altered. Python syntax allows a tuple to be assigned without parentheses

dictionary = {key:value, key:value} : A dictionary is a data structure that consists of comma separated key:value pairs. A key can be any immutable object and a value can be any object

dictionary.get(key) : Returns the value associated with the key object supplied

parameter and argument : the variable that is passed into a function

call is referred to as an argument, and the allowed variables that the function can take are referred to as parameters.

Fundamental rules

Code executes from top to bottom : Each line of code (with the exception of a multi-line statement) is executed one at a time from top to bottom.

Functions that aren't called aren't run : Just because code executes from top to bottom doesn't mean that a function will run automatically just because it is defined at the top of your script. A function has to be called in order to run, and when it is called the code within the function will run from top to bottom. To "call" a function is to have your code tell that function to run. Maybe the function name (function()) is written on a line that runs when it is reached, or it is assigned to return a value to a variable when reached. The print() function is an easy way to understand this, try it out.

Variable assignments that aren't reached don't happen : Maybe you'll have a variable operation/ assignment in some branch of an if statement. If that branch isn't reached due to the conditions of its execution, that variable operation/ assignment will not happen.

Variable and function names cannot contain spaces : No spaces allowed, if you want a space, use an underscore (variable_name), or camel case (variableName)

The CONTENTS of a new block must be indented : Following the line on which a control statement as such as an if statement, or a for loop, was initialised, the code executed by that control statement must be indented by pressing *tab*. Your code will NOT run if this is not so, indentation is essential for scope definition, and is analogous to {} in many other languages. NOTE, if your script is properly indented but you are still getting compiler errors, you may have saved your script in an editor that converts tabs to spaces, or has misconfigured the tab size (or something weird like this). If this happens to you, open your script in Sublime Text (it's guiltware, but it's free!), and click on the area in the bottom right of the window where it says "Tab Size: n" where n is a positive integer. Make sure "Tab Size" is 4, and click on "Convert Indentation to Tabs".

Line spacing : Python has no functional requirements of how much or little space is required between each line of code. Although for readability it is good practice, and even recommended by some programming environments to put two spaces between each function block. Spaces elsewhere are really a matter of preference when it comes to writing code for your own personal use.

All iterable data structures (except for dictionary which is indexed by key) are indexed from 0 as the first element : So for example if you have a list of seven elements, it will be indexed from 0 to 6. Do remember this as it is a common problem early on to run into the horrifying visage of "IndexError: list index out of range", which means that you tried to interact with an element at an index that doesn't exist.

Control statements can be nested in other blocks : Conditional statements and loops can be placed inside of each other. Just put the initialising line of the control statement on the same indent level as the body of the statement it is in, and then indent the body of your new statement

as you would with the body of any block

A function must almost always be called with exactly the same number of arguments, as it has parameters, and in the same order : Yes, there are ways to allow a function to accept a varying number of arguments, but for the applications you will encounter while getting the nuts and bolts down, make sure you observe this rule (but do feel free to experiment with more advanced features)

Put new statements on new lines : Put each successive variable assignment, operation or control statement on a new line. Code is executed from top to bottom, so naturally the lines of code are interpreted in the order in which they are defined from top to bottom. There are a few exceptions to this rule as such as variable assignment with tuple unpacking.

Multi-line statement : Some Python interactive development environments will have a right hand margin instead of wrapping the line of code onto the next line. You can write over this margin but, for the sake of code readability, it is considered good practice not to. In order to not write over the margin you can continue a statement onto the next line by putting the "\" character at the end of each line you want to continue onto the next line, and then continue your code on the next line. Python allows you to spread code out over multiple lines if it is encapsulated in "()" or "[]" without having to use "\". **Examples here.**

Variable assignment is evaluated right to left : The right hand side is executed completely before being assigned to the left hand side

Subsequent assignment of same variable replaces that variable : if on one line you define a variable with some value and then you assign a variable with the same name below it with some value, the second definition of that variable will replace the value of the first definition. You are overwriting it. This is useful if you want to update the value of a variable.

Statements evaluate from left to right: For some statement involving operations on objects, evaluation of those objects occurs from left to right as **elaborated upon here.**

Arithmetic and logical operators follow a strict order of operations: Beyond the order in which operations are computed which you may be familiar with, Python has some other operations which also follow an order of execution. They are listed **here.**

Introduction

What is programming?

You speak a language, and when you do you parse your thoughts and feelings into a spoken language using rules that other people can understand, and then using a similar set of rules you parse words back into your mind which results in thoughts and feelings. A programming language is much like a spoken human language, however the subject you are communicating with, a computer, is not intelligent and flexible (the subject of artificial intelligence aside). This means that a programming language only has a fairly small vocabulary, or set of base rules. This is not because there is only a small amount of ideas that can be communicated through them, it is to make the set of rules that can define all functionality as small and clear as possible.

In essence, programming is using this very refined vocabulary in order to define instructions called code, code that explains to the computer what you want it to do. Your code will run in a certain order determined by other code, and the rules of the language, and in a certain way determined by how you implement it, and the internal logic of the rule. It is your goal as a programmer to understand these rules so that you may find creative ways to communicate with your computer as well, so go and paint your masterpiece.

What is a rule?

A rule is the logic that tells you how something will *always* behave, and in Python, rules come in a few flavours, but they all have the common factor that they dictate a very clearly defined and inflexible behaviour. All of the critical rules in Python will be covered in more detail in other sections, but the most common and important categories of rules are:

- **Conditional rules:** Rules that govern statements that tell the computer what to do if an operation on code evaluates to true or false (the *if* statement)
- **Loop rules**: These are the rules that govern how you can control the length of a loop. In Python there are two simple kinds of loops. *for loops* are usually for looping as many times as they are elements in an iterable object, or numbers in a range. *while loops* are to run until a condition is met. More on this is in the Control Flow section.
- **Arithmetic rules:** Legal operations on numbers, as well as the order that operations are carried out on when there are multiple operations being used in an algebraic expression (using binary operators like +, -, /, *)
- **Order of execution rules:** Rules that govern what code gets run and when. There are two factors that affect this. Factor 1: the order in which blocks of code run in based on when code is called (based on a function call, or the branch the code takes from an *if statement*. Factor 2: the order that each line of code runs in as specified by the interpreter (top to bottom).
- **Scope rules**: How and when code should be indented in order to correctly define a block, also when a variable can be accessed in relation to when it was declared.
- **Type mixing rules**: How Python will interpret an attempt to mix types. Python is what is called strongly typed, and will not automatically covert, say, an integer to a string if you try to concatenate an integer and a string. However, you can populate a data structure as such as a lot with different types so long as they are at different indices of the list
- **Index rules**: For an interable object as such as a list, an index always starts at zero. Attempting to access and index that is greater than one less than the length of the iterator (because the final index of an interator is always iterator length - 1) will result in an error, this is explained under fundamental rules. There is also a rule that says that a negative index will wrap around backwards from the end of the iterator, but because you can't have an index of "-0", your index of "-1" will now be the last index, and your first index will be "-n" where n is the length of the list. A list index less than "-n" cannot be accessed due to it "starting" at -1 just exactly the same way that a list index greater than (n-1) cannot be accessed due to it starting at 0.

In mathematics there are rules you will know of such as:

- $a + b = b + a$ (integer addition with commutativity)
- $a = a$ (equality)
- if $a \neq b$ then $a - b \neq b - a$ is equivalent to if $a = b$ then $a - b = b - a$ (non-commutativity of subtraction)

These rules are set in stone, and not proved, they are called axioms. Axioms are simply an assertion made such that there exists a starting point for a system of logic in which things can be proven. The things that are proven are called theorems. The statement "$a^2 = b^2 + c^2$ where a, b, c are the real-valued lengths of the sides of a right angle triangle" is a statement of a theorem, specifically the Pythagorean theorem. It is such because it can be shown that the conclusion follows from the

hypothesis using logical arguments based on axioms that lead inextricably from one argument to the next. It is not just known from inspection. Statements like "a number subtracted from another number is the positive or negative difference between those two numbers on the number line", is not an axiom or a theorem, it is a definition. While it gives a particular mental model of what subtraction is it doesn't assert a universal truth about the property of subtraction. For example it doesn't describe *when* the difference is negative or positive.

The thing about axioms, and theorems proven from axioms is that they both always function the same way with no exceptions because the truth of those theorems follows from the assertions that the axioms are true. The ways that theorems can be built on top of axioms via proofs is a lot like the way that you will define, and verify how things work while programming. Instructions that are built out of axioms will always follow those axioms, however instructions being defined by yourself might not always do what you expect them to if their implementation means something different from what you intended. This is where it is up to you to test and verify your creation.

Under **almost** every **circumstance** computers do not make mistakes. This is the beauty of programming languages being built out of axioms and proven rules. You can rely on the fact that you have made an error to learn and correct your error and not worry about whether something has gone wrong under the hood.

What is code?

If you think of your code like a list of instructions then it is easy to imagine that the process of programming is just telling your computer which rules to follow and when. What we will refer to as an instruction is an implementation of a singular, or potentially many rules that tell the Python interpreter that the computer should do certain things. The term *instruction* has no formal definition within Python and is just a word I am using to describe how you implement your own code from rules that are defined within the language. What I mean by implementing an instruction based on potentially many rules can be explained through a simple example. I will formally introduce the following concepts later but for now I will describe them in simple terms for the sake of gaining a small amount of immediate insight:

```
1 x = 2
2 x = x + 2
3 print(x)
```

This instruction assigns the integer 2 to the variable x, and then next adds integer 2 to the variable x's value, and then assigns the sum of x + 2 to variable x overwriting it with the new value. Finally the print() function is called with the variable x as the argument. The following is printed:

```
1 4
```

As you can see a new instruction has been built out of the rules defined by Python in order to sum two integers then print the result. This is how Python code is implemented, sequential operations on objects that define some greater functionality than its parts could offer alone. The outcome of this code is equivalent to the following:

```
1 x = 2 + 2
2 print(x)
```

or

```
1 print(2+2)
```

There are many ways to achieve the same things depending on your needs, but the rules implemented by Python that you build your code out of always hold just the same. So in essence the fundamental process of programming is knowing the rules and then using them in order to build your code. It might seem intimidating to have to learn all of the rules, but I assure you that there are not really that many to learn at all. As a handy reference I have compiled a list of rules and concepts at the beginning of this text which define everything you'll need to know to get a feel for programming. There are of course more rules and concepts, but they are auxiliary to this stage in your learning. For those of you who want to delve into those further topics I will include a brief summary of what they are at the end of this text.

Why would I learn Python and what is it good for?

See I got these glasses
So they kick my ass
But I'll kick their asses
When I get to class

How do I start?

Linux: Python 2 or Python 3 are probably already installed, but if not install them with your distro's package manager, in most cases you won't have to add or any repositories to do so. Vim, PyCharm and Sublime Text (A WARNING: Sublime Text has converted my tabs to spaces by default in the past, so be sure to check that first if you have problems) are good editors. I recommend running your program from the terminal, directions on how to do this are under Program Structure: Running your script

Windows: Download from **here** and install. The last time I used Python on windows it also installed the editor Idle, which can be used to launch your program (but it's not very good) so I'd use PyCharm, Notepad++ or Sublime Text (same warning as above). Using random text editors is technically an option, but they might not be configured properly for Python's indentation requirements.

MacOS: https://www.python.org/downloads/mac-osx/

Other: https://www.python.org/download/other/

The Python interpreter

The Python interpreter is sort of like a bilingual dictionary that translates the code of Python into lower-level code that is more native to the computer's hardware. More generally it produces a target language from a source language (Python). Because computer hardware only knows the language of binary digital signals in the form of different voltages, your code must go through several layers of abstraction in order to go from Python to machine code, and the Python interpreter is the system that parses your code and translates them between these layers. I won't go into this, or how an interpreter/ compiler really works but just know that it is the machinery that translates Python into something computer hardware understands.

Program structure

Each Python script has its operation defined by the interaction of a fairly small set of elements. I will begin with presenting a very simple script that incorporates common elements you will encounter before learning about conditionals, and loops. Also note that this does not include elements of class structures, as classes are not covered in this particular guide, but may be covered in the future.

```
#!/usr/bin/env

import module


def function(parameter):
    ''' function prints the string "Hello, world!" and returns nonsense
        param: parameter: a parameter called "parameter" that is a variable
        containing the string "Hello, world!"
    return: nonsense : returns a variable called nonsense which contains
        the string "Pi is exactly 3!"
    '''
        print(parameter)
        nonsense = "Pi is exactly 3!"
        return nonsense


text_argument = "Hello world!"
function_output = function(text_argument)
teapot = "I'm a teapot."

print(teapot)
print(function_output)
```

This script will output the following:

```
Hello world!
I'm a teapot.
Pi is exactly 3!
```

Let's analyse this script piece by piece.

Shebang

```
#!/usr/bin/env
```

This is referred to as a shebang. In UNIX like operating systems when a Python script is made executable, and with certain software like the Python launcher for Windows, this line of code defines the location of the default interpreter. Defining an interpreter when running a script from and interactive development environment, or terminal will override this variable. If you run the script from the terminal using Python 2, or Python 3 your calls will look like these respectively (note pilgrim $ is just a placeholder terminal prompt just like "C:\ ...\current directory>" in Windows command line, it's not part of the command, yours may look different) :

```
pilgrim $ python yourscript.py
pilgrim $ python3 yourscript.py
```

Import

```
import module
```

The import statement is used to make the features of external modules available to your script. Note that "module_name" is just a placeholder name for a real module, if you try to import a module that does not exist on your system you will get an error. If you want to install a new module onto your system, I recommend using "pip". Much information is available **here**, although I will mention that to install Python version specific packages, do the following:

```
#Python 2
pilgrim $ python -m pip install module_name
#Python 3
pilgrim $ python3 -m pip install module_name

#pip won't install in a directory owned by root by default so if it
#complains use:
pilgrim $ pip install --user module_name
```

Note that a lot of packages are only compatible with one version of Python, and certain packages can be incompatible with each other.

Function definition

```
def function(parameter):
    ''' function prints the string "Hello, world!" and returns nonsense
        param: parameter: a parameter called "parameter" that is a variable
        containing the string "Hello, world!"
    return: nonsense : returns a variable called nonsense which contains
        the string "Pi is exactly 3!"
    '''
        print(parameter)
        nonsense = "Pi is exactly 3!"
        return nonsense
```

This is called a function definition. A function is a block of code that has been defined to encapsulate some specific sort of operation. Functions, as evidenced by their name, are defined on the basis of what they do, meaning that it is standard practice to define a function for each clear and distinct purpose that is needed for the code. Think of them like tools, you wouldn't want to combine a bunch of tools together in most cases because it would be awkward, heavy and hard to use. You want one tool to perform a specific task. Functions can also be called from one another in order to provide a service when that role is outside of the scope of the current function. As you can see from the above definition, this function is defined as being able to accept a single parameter, it prints that parameter to the command line, assigns a string to a variable, then returns that variable to the calling code. All of this will be elaborated on in the Functions section.

Docstring

```
    ''' function prints the string "Hello, world!" and returns nonsense
        param: parameter: a parameter called "parameter" that is a variable
            containing the string "Hello, world!"
        return: nonsense : returns a variable called nonsense which contains
            the string "Pi is exactly 3!"
    '''
```

This is called a Docstring. It is used to document elements of your code, in the case it is for the function() function. A Docstring isn't necessary for your code to run but it is a good practice to document your functions so that it can be easily understood. In python 3.7, if a Python script is opened from within the Python environment (by calling the single argument 'python' from the command line) the following functionality is available.

```
print(function.__doc__)
```

```
function prints the string "Hello, world!" and returns nonsense
    param: parameter: a parameter called "parameter" that is a variable
        containing the string "Hello, world!"
    return: nonsense : returns a variable called nonsense which contains
        the string "Pi is exactly 3!"
```

This same print function can be made available in Python 2.7 with the following import statement:

```
1 from __future__ import print_function
```

Function call

```
1 function_output = function(text_argument)
```

The function call is how a given function is ran. In this case the function call is on the right hand side of a "=" operator as, in addition to calling the function, we want the function's *return* value to be assigned to the variable "function_output"

Running your script

If you want to try some of these things out immediately, in Linux based operating systems you open a terminal and cd to the directory your script is in, then do the following. Using Python 2, or Python 3 your calls will look like these respectively:

```
pilgrim $ python yourscript.py
pilgrim $ python3 yourscript.py
```

Variables

In very high level terms a variable is like a container that you can store data inside of. A variable can store objects, which belong to three main categories:

1. Typed Data:
   - x = 12 (x is set to the integer 12)
   - car = "The Homer" (car is set to the string "The Homer")
   - state = True (state is set to the boolean value "True")
   - not_pi = 22/7 (not_pi is set to the floating point number that the expression 22/7 evaluates to)
2. Data Structures:
   - week_days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"] (sets the variable week_days to the list containing the string representations of the week days)
   - age_of_my_dogs = {"Barney":6, "Balthazar": 8, "Lenny":3} (sets the variable age_of_my_dogs equal to the dictionary whose entry is a map between the names of dogs as strings, and their ages as integers)
   - primes = ("two", 3, 5, "seven", 11) + ("thirteen") (sets the variable primes equal to the result of concatenating two tuples together, which will result in the first tuple with the element "thirteen" added to the end)
3. Classes (note, these classes are all made up, and this topic won't really be covered because it can get pretty weird, but it's important to be aware of)
   - my_cat_oscar = Cat(12, "Oscar") (creates an instance of the class Cat() called my_cat_oscar and sets its age to the integer 12, and its name to the string Oscar)
   - oscar_age = my_cat_oscar.get_age() (calls the method get_age() from the instance of Cat() called my_cat_oscar, and method get_age() retrieves the age that was assigned to the class variable in Cat() when it was creates, in the case 12)

When a variable stores one representation of data in the form of typed data (in the case of python an "object"), that object can only have one type. This, however, does not mean that you cannot store data *structures* inside of variables that can contain multiple pieces of data with multiple types. As you can see above, the tuple stored in "primes" contains strings and integers. In Python there is no notion of variable declaration and initialisation, just variable *assignment*. This means that you can assign variables on the fly, and there is no need to tell Python what type the variable can hold in advance.

Assignment

Python follows a strict rule for assigning a value to a variable. An assignment always follows a right to left assignment order where the variable is on the left, and that object being assigned to the variable is on the right. Unlike many programming languages, variables do not have to be declared or initialised before they can be used. In fact the concepts of variable declaration and initialisation do not quite exist in Python, instead the term *assignment* is used. Also unlike many other programming languages you do not need to declare the type of the variable, a variable does not have a fixed type despite its value having a type. These properties allows you to assign objects of different types to the same variable. Variable assignment resembles the following:

```
1  #not real code, just an illustration of assignment
2  variable <- data
3
4  #real code assigning the integer 1 to the variable x
5  x = 1
6
7  #this can be proven with the print() function
8  x = 1
9  print(x)
10
11 1   #the output to the terminal
12
13 #it is exactly the same for different object types
14 list = [1, 2, 3]
15
16 true_or_false = True
17
```

```
capital_cities = {"New Zealand":"Wellington", "South Korea":"Seoul"}

#variables can be rewritten with different data types
taxicab = 1729
taxicab = "one thousand seven hundred and twenty nine"
print(taxicab)

one thousand seven hundred and twenty nine
```

Beware however, there is the exception that when using a variable that refers to a mutable data structure, that variable must be assigned the value of its data structure before it is used. For example:

```
my_list = []
my_dictionary = {}
my_set = ()
```

From here you can build up equation-like expressions that have their value assigned to the variable on the left, these expressions are what takes your code from the level of just representing data, to the level of manipulating it and actually implementing meaningful problem solving tools. The expression on the right hand side of the variable assignment can contain an unlimited amount of typed data like strings, ints, floats, booleans etc, data structures like lists, and tuples, as well as function calls, so long as they conform to type mixing restrictions.

You can technically have as many of these types of objects as you want, but with the hard restriction that data types can't be mixed with the exception of numerical types. There is also the notion that you should group together operations in a meaningful manner so that your code makes sense to read. For this reason it is not overly common to perform a long chain of operations on a single line, and variables containing results of operations should be separated in a meaningful manner.

```
# we will assume the variables that appear on the right
# have already been given values, this is just to what is possible
cosine_theta = math.sin(math.pi/2 - theta)

radial_acceleration = v**2/r

palendrome = "Pythonidae" + string_reverse("Pythonidae")

complex_modulus = math.sqrt((x.real)**2+(-x.imag)**2)

host_dictionary["8D:F5:E5:62:B0:DE"] = "10.0.0.2"

#the two statements below are equal
new_value += (x+7)/(y*10)
new_value = new_value + (x+7)/(y*10)
```

There is also a concept in Python called multiple assignment, or tuple unpacking, it is where an element of a tuple is assigned to an variable in the corresponding position in a comma separated sequence. It is performed as follows:

```
#assigns x = 1, y = 2
x, y = (1, 2)

#the same thing can be done without parentheses
#as they are not necessary for a tuple
x, y = 1, 2

#extending this with an assignment
z = (1, 2)
x, y = z

#sometimes you will have a function that returns
#a tuple, the process is the same
x, y = tuple_function()
```

Assignment can also be done to indices of certain mutable objects like lists as seen above with host_dictionary. The following will demonstrate how a variable can be assigned to a particular index of a list.

```
a = 1
b = 2

list = ["a", "b"]
list[0] = 2

# where the variable a at index zero now equals 2
print(list[0])

# which results in:
2

# the same can be done with our shortcut +=, -= type of operations

a = 1
b = 2

list = [a, b]
list[0] += 2

# where the variable a at index zero now equals 3 after the operation
print(list[0])

# which results in:
3
```

Another very important variable assignment is assigning the return value of a function to a variable, as was seen in the earlier section. If a function definition ends with the return statement, followed by the object to be returned, the variable that function's value is assigned to gets set to that return value. For example:

```

def increment_a_number(arg):
'''output = increment_a_number(arg) -> function does operation on arg,
result of operation is assigned to modified_arg (not essential
just for clarity) -> return modified_arg ->
output = modified_arg (modified_arg doesn't
literally take the place of increment_a_number(arg) in the written code,
increment_a_number(arg) just takes on the value of modified_arg when it
has run, and returned modified_arg) '''

    modified_arg = arg + 1
    return modified_arg


output = return_a_number(1)

print(output)

#the result printed to the terminal
2
```

Functions

Function definition

```
1  def function(parameter):
2      ''' function prints the string "Hello, world!" and returns nonsense
3          param: parameter: a parameter called "parameter" that is a variable
4          containing the string "Hello, world!"
5      return: nonsense : returns a variable called nonsense which contains
6          the string "Pi is exactly 3!"
7      '''
8          print(parameter)
9          nonsense = "Pi is exactly 3!"
10         return nonsense
```

You will recognise this function definition from the Program Structure section. A function can consist of the following elements. Some elements are optional.

- def statement : this tells python that you are declaring a function
- function name : a descriptive name of what the function does, must not contain spaces
- function parameter field: this is the () that goes to the right of the function name, it is necessary
- function parameter OPTIONAL : this is what goes between (), this variable is so that you can make a variable outside of the function available within the function. If a function has a parameter, you must pass in a corresponding argument when you call it. Note that all functions will have () as the characters directly following the function name to the right with no spaces between them, or them and the function whether or not you have defined them to accept a parameter.
- the colon ":" : this comes directly after the parameter, and tells the interpreter that the function's block scope will begin on the next line
- function body : this is all of the code that goes inside of the function, a function that is called without a body will result in an error
- return statement and value OPTIONAL : see how in our example when we call the function function() it looks like this:

```
1  function_output = function(text_argument)
```

This means that our function has an output, and the output is defined by the object that is to the right of the return statement, and that output gets assigned to the variable function_output (you can call it whatever you want this is just an example). In our example it is the variable "nonsense", but you can also return strings, numbers, and data structures like lists.

So putting it all together you have a function with and without a parameter:

```
1  def function_name(parameter):
2      #function contents
3
4  def function_name():
5      #function contents
```

Function call

```
function_output = function(text_argument)
```

In this section of the code, a call to a function named "function" is made. This function takes the argument "text_argment" because it is defined as taking one parameter. The value of the function that is returned is then assigned to the variable named "function_output" once, and only once the code inside of that function has reached, and executed the return statement. If that return statement is not reached, or does not exist, then nothing will be assigned to the variable.

```
#valid configurations for function calls as defined by you
function_output = function(text_argument)
function_foo()
function_bar(argument)

#built-in functions: behave like normal functions but have reserved
    names
print(argument)
abs(numerical_var)
len(object)
range(stop)
range(start, stop, step)
```

Here we can see there are four different basic configurations of a function call:

1. function() : The function stated on its own like this just calls the function, running it
2. function_output = function() : The function output is now assign to a variable, provided that function returns a value
3. function_with_args(some_argument) : The function is called on its own like in 1. but, let's assume it has been defined to accept one or or more arguments, so we have to give it as many arguments as there are parameters in its definition
4. function_with_args_output = function_with_args(some_argument) : Same as 2., but with the conditions of 3.

Stating the function name on its own followed by the closed parentheses "()" tells the interpreter to run that function, also known as "calling" the function. Functions can be called with arguments inside of the parentheses, and the output of the function can be assigned to a variable on the same line that it is called, as it was in the example. In the lines below the function call and variable assignment example just above, there are some other ways that functions can be called, as well as a now familiar inbuilt function.

Data types and data structures

In a very high level programming-oriented way, a data type essentially defines the rules regarding which operators can be used on which object, and also defines what happens when an operator is used on an object of a particular type. Python is a "strongly typed" programming language which means that there are rules in place that enforce the mixing of data types and data structures through operators. This means that you cannot concatenate a string with an integer (like "two" + 11), this will result in an error. However you can use arithmetic between different types of numbers, for example 2 + 2.0 is valid between an integer and a float, or $2+(1+2j)$ is valid between an integer and a complex number. This follows for all numerical types. In Python types can be split into two important categories, data types and data structures. Before you approach this section I will give some instruction on how to understand the mathematics that follows. It is simple mathematics, and I believe it is very useful for a deeper understanding of the concepts, but I understand that it may appear completely alien to some. The syntax and jargon is as follows:

- set: A set is a collection of unique entities called "elements" which follow some rule about their inclusion in the set. Often a set is unordered but some have rules about their order, and looks like this $A = \{1, 3, 5, 2, 9\}$. For example in the set of integers, as a consequence of the rules that govern its order you get the next integer by adding one to the previous one, in fact you can do this forever as there are an infinite amount of integers. Some sets are finite and some are infinite.
- $\mathbb{Z}, \mathbb{R}, \mathbb{C}$ are the set symbols for integers, real numbers, and complex numbers respectively.
- set size: Set size, also known as cardinality is a measure of the number of elements in a set. For set $A = \{1, 3, 5, 2, 9\}$ $|A| = 5$
- set builder notation: It is useful to be able to define a set in one line, this is what this notation does. $\{x \in \mathbb{Z} : 2 \leq x \leq 6\} = \{2, 3, 4, 5, 6\}$ from left to right means for an element x in ($\in$) the set of integers ($\mathbb{Z}$) such that (:) 2 is less than or equal to x is less than or equal to 6, the resulting set is equal to $\{2, 3, 4, 5, 6\}$
- $[x, y]$ means the closed interval of x and y. It is like saying you have three integers x, y, z such that $x \leq z \leq y$ meaning z exists in that interval and can be equal to its end points
- (x, y) means the open interval of x and y. It is like saying you have three integers x, y, z such that $x < z < y$ meaning z exists in that interval, but strictly between its end points

Data types

Numerical types

- **integer** (max size: unbounded (see below); immutable: yes; arithmetic: yes (algebraic, bitwise))
  Integers are negative or positive whole (discrete) numbers including zero that exist in the interval $(-\infty, \infty)$. All integers follow a strict order, and increment by one at a time with no number in between two consecutive integers. This means that for any closed interval $[x, x+1]$ of consecutive integers $x, x+1 \in \mathbb{Z}$ it follows that $|[x, x+1]| = 2$. This means that the size of the set of this interval is exactly equal to the number of integers in it. Integers being whole numbers have some important properties:
  - All algebraic operators are compatible with integers
  - All algebraic comparators are compatible with integers
  - Bitwise operations can be performed on integers, and their result is returned in base 10
  - Some integers are called "prime numbers", all of which are positive. They are special because their only factors are one, and themselves. Some examples of prime numbers are 2, 3, 5, 7, 11. There are an infinite number of prime numbers, but there is no efficient formula for finding them.
  - Every integer has a unique prime factorisation, meaning every integer can be represented by a totally unique product of prime numbers
  - Every integer divided by one of its prime factors has remainder zero
  - Integers other than zero are even or odd. All even integers can be represented by the form $\{x, k \in \mathbb{Z} : x = 2k\}$, all odd integers can be represented by the form $\{x, k \in \mathbb{Z} : x = 2k + 1\}$
  - To divide any even integer by two will result in a remainder of zero. Any odd integer divided by two will result in a remainder of one

  By default integers in Python are represented in base 10, which is the numerical base which you are most likely used to thinking in and speaking of, however there are many **other** ways of representing positive and **negative** values. One you may be familiar with is binary (base 2) in which all numbers can be represented by "1" and "0". The rules are that bases must be positive, and for all characters x that represent a position of a value in base n, $0 \leq x < n$, hence why the only values in base 2 are "1" and "0". To define an integer in a specific base the following syntax is required:
  - binary prefix: 0b or 0B (e.g. 0b101010)

- octal prefix: 0o or 0O (e.g. 0o00371)
- hexadecimal prefx: 0x or 0X (e.g. 0x5F3759DF)

Technically in Python an integer's max size is $\leq 2^{63} - 1$, but Python converts integers to another data type called a "long" when it exceeds this size, and longs have unbounded size.

- **float** (max size: 2.2250738585072014e-308 to 1.7976931348623157e+308; immutable: yes; arithmetic: yes (algebraic only))

  A floating point number, or float is Python's numerical representation of a "real" number which represents a continuous value on the number line. A float, or real number is a number that exists in the interval $(-\infty, \infty)$ with the extra property that every real can have an infinitely large amount of decimal places. The implication is that for any closed interval $[x, x+1]$ of consecutive reals $x, x+1 \in \mathbb{R}$ its cardinality (set size) $|[x, x+1]| = |\mathbb{R}|$. It might seem perplexing that a proper subset of a set has the same size as it, but both sets are *uncountably infinite*, which is a consequence of there being an infinite number of numbers between any two numbers. The interval and the set both being uncountably infinite implies that they have the same and equal cardinality. HOWEVER, while this is true of the reals, the size of floats are constrained by their implementation, and also computer memory, so by default floats are constrained to a certain interval of values. Some facts about floats:
  - All algebraic operators are compatible with floats, even modulo has an implementation which is described below
  - All algebraic comparators are compatible with floats, HOWEVER, due to limits of precision of floats, two floats that appear identical may not be the same. To determine the equality of two floats with a given level of precision use **this function**. Note you will have to import "math" module
  - Using the print() function, Python will truncate the amount of a float's decimal places to a certain level by default, this can be controlled using **formatting**
  - For an arbitrarily large float, you can use float('inf'). This is not a number, but an object with a property that it is greater than any other actual float. Similarly -float('inf') is smaller than any other float that can be defined
  - Primes exist for the reals because the integers are a subset of the reals, but floats with decimal places have no unique prime factorisation
  - The concept of odd and even doesn't apply to non-integer valued numbers, although Python considers things like $2 == 2.0$ to evaluate to true, so testing parity by modulo 2 will return a zero remainder for floats that are equivalent to even integers. This is because the integers are a subset of the real numbers and Python wishes to treat them as such.
  - Dividing two floats may produce a non-zero decimal value and will not round down to the nearest integer.
  - Despite float division returning a decimal value, the modulo operator used between floats will treat its quotient like an integer and return a float valued remainder. For example $3.0/2.0 = 1.5$ however $3.0\%2.0 = 1.0$.
- **complex numbers** (max size: 2.2250738585072014e-308 to 1.7976931348623157e+308 for each part; immutable: yes; arithmetic: yes (algebraic))
  A complex number is a type of number that consists of two parts, a real part and an imaginary part, and looks like $z = x + yi$ where x is the real part and iy is the imaginary part. Imaginary means that it is a number that is a real multiple of i, where i is the solution to $\sqrt{-1}$. I won't delve into its properties or applications as they are a bit too far off map for somebody learning how to program, but I will mention that the existence of the imaginary unit i is due to the fact that there is no real x such that $x^2 = -1$ so itfs inverse $\sqrt{-1}$ is otherwise undefined for the reals. The existence of i means that $i^2 = -1$ can be defined such that $i = \sqrt{-1}$.
  - To assign a complex number to a variable use the syntax z = a + bj (that is "j" for juliet, as Python uses engineering notation). Note that you must multiply j by something, z = 1 + j will return an error, do z = 1 + 1j instead
  - a complex number can also be defined via the constructor complex(real, imaginary). For example x = 1 + 2j equivalent to x = complex(1, 2)
  - To access the real part of a complex number assigned to variable z use z.real
  - To access the imaginary part of a complex number assigned to variable z use z.imag
  - complex numbers have no well defined ordering, so the comparator operators are not compatible with them

Non-numerical types

- **string** (max size: unbounded; immutable: yes; arithmetic: no):
  Strings are the common programming way of representing written text. When a string is defined or referenced it is enclosed between "" or '', in Python they both have the same functionality. A string can be added or "concatenated" with another string via the "+" operator, in order to concatenate strings, place "+" between each string you want to concatenate. There is no upper limit to how many strings you can perform this operation on. This operation works

if and only if all objects being concatenated are strings, if you want to concatenate a string with another type with, for example, variable name x you will have to invoke the str(x) function to convert that value into a string. A string is converted to lower level code by what is called a character encoding, it is like a dictionary that knows the binary representation of a given character. Take note that Python 2.x and 3.x use different default character encoding, however for learning purposes it is unlikely that this will be a problem. Be aware that some sources will have a different character encoding if you are using strings from somewhere external to Python as an input to your code. One example is using the output of a Linux terminal in your code.

- **boolean** (max size: not applicable; immutable: yes; arithmetic: logical, bitwise)
  In boolean logic there are only two values: True and False. In some contexts they are represented as 1 and 0, and sometimes other things, but they are always distinct and separate values where one is the negation of the other. All objects in Python have a boolean value associated with them and will be treated by boolean operators accordingly. This boolean value is purely in relation to certain properties of the object alone and not a result of some kind of comparison operation.
  - objects that evaluate to True by default
    - variables that have been assigned legal values of a supported data type other than 0 or 0.0
    - string variables that are non-empty
    - variables that have been assigned a boolean value of True
    - data structures that contain legal values of supported data types
  - objects that evaluate to False by default
    - variables that have been assigned 0 or 0.0
    - empty string variables
    - variables that have been assigned a boolean value of False
    - data structures that contain zero elements
    - variables with type None

  Boolean values tell you the outcome of comparisons made between objects. In Python all comparators evaluate to either True or False based on the values being compared. The outcomes of these comparisons can be evaluated together with the boolean operators implemented by python. These three operators are "and", "or" and "not. The outcome of each group of statements evaluated together with boolean operators also returns a boolean value of True or False. This is very important for the implementation of conditions within flow control as such as "if statements" and "while loops". All boolean operators have an order of precedence, operations that are placed inside of parentheses "()" are always evaluated before other operations. The order of precedence goes: first: (), second: not, third: and, fourth: or. In the following examples in this section I will just state truth values rather than assigning them to variables in order to simplify things e.g. (True and True), (True or False), not(True and True). Their truth tables and further explanations will follow.

p	q	p and q	p	q	p or q	p	not p
1	1	1	1	1	1	1	0
1	0	0	1	0	1	1	0
0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	1

- and: for statements evaluated together with "and", all statements must evaluate to True, else the group of statements evaluated will evaluate to False. For example:
  - (True and True) equals True
  - (True and False) equals False
  - (True and False) and True equals False
- or: for statements evaluated together with "or", at least one statement must evaluate to True, else the group of statements will evaluate to false. For example:
  - (True or False) equals True
  - (False or False) equals false
  - (True and False) or True equals True
  - (True and False) or False equals False
  - (True and False) or (True or False) equals True
- not: "not" is the boolean negation operator which means that it inverts the value of the statement it is applied to.
  - not False equals True

– not True or False equals False

Be careful though as the not operator has an effect upon a collection of statements enclosed in parentheses that is not immediately obvious. The not operator affects the operator within parentheses as demonstrated in these examples:

– not(True and True) is equal to (not True) or (not True)
– not(True or True) is equal to (not True) and (not True)

These two results are called De Morgan's laws. They may not be obvious at a glance but perhaps upon some truth testing you can see their consistency and understand their logic.

Data structures

Data structures represent collections of objects, where those objects are stored in correspondence to some sort of index, or key. Unlike variables, data structures must be defined before they can be used.

- **Lists** (max size: ?; immutable: no; ordered: yes)
A list is collection of comma-space elements that are ordered by sequential ordinal indices starting from zero, and are bounded by square brackets. The elements of a list may be objects of any "type". For example you may have strings and integers in the same list. Elements of a list may be any object, including other lists. The form of a list is as follows:

```
linux_distros = ["CentOS", "Debian", "Ubuntu", "Gentoo", "Slackware"]
nested_lists = [1, 2, [3, 4], ["a", "b", 3.14, "banana" ]]
```

– If it has not been already, a list must be initialised before use. For example:

```
# Initialises an empty list
empty_list = []

# Initialises an list with 100 elements, all equal to zero (of
    length 100)
zeroes_list = [0]*100

# Initialises a list containing some existing elements
fruits_list = ["nashi", "persimmon", "plum"]
```

– Lists may be added together, to produce a single list of elements, in the order in which the lists were added. For example: [1, 2] + [3, 4] = [1, 2, 3, 4]
– A list may be multiplied by a positive integer n to produce a new list with n copies of its original elements. For example: ["a", "b"] * 2 = ["a", "b", "a", "b"]. Note that this does not perform any numerical operation on the elements of the list, it just copies then n times.
– lists can be checked for membership: x in ["z", "y", "x"] evaluates to True
– lists can be iterated over with a for loop. For example:

```
for x in [1, 2, 3]:
    print(x)

#would output:
1
2
3
```

For each element in the list, the body of the for loop will be called allowing operations to be performed sequentially for each element of a given list.

– Elements can be accessed by, or assigned to a specific index of a list:

```
# Example: accessing an index
vehicles = ["car", "truck", "boat"]
print(vehicles[0])
# would output:
car

# Example: writing to an index
```

```
vehicles = ["car", "truck", "boat"]
another_vehicle = "plane"
vehicle[2] = another_vehicle
print(vehicles)
# which would output:
["car", "truck", "plane"]

#but be careful that you assign to an existing index:
vehicles[3] = another_vehicle
# would result in:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

As a list is a mutable data structure, its length can be modified on the fly. Hence, if you wish to add, or remove an element from a list rather than replace an element you can do the following:

```
# list.append(object)
vehicles = ["car", "truck", "boat"]

# append element to end of list
vehicles.append("plane")

print(vehicles)

# which would result in:
["car", "truck", "boat", "plane"]

# list.insert(index, object)
vehicles = ["car", "truck", "boat"]

# insert element to given index of list
vehicles.insert(1, "plane")

print(vehicles)

# which would result in:
["car", "plane", "truck", "boat"]

#list.remove(object)
vehicles = ["car", "truck", "boat"]
vehicles.remove("car")
print(vehicles)
# which would result in:
["truck", "boat"]
```

If a list is non-empty, elements can be assigned to the indices of its existing elements by a direct indexing method. Although this is possible, you run the risk of assigning an element to a non-existent index. This is especially important to remember with an empty list, as it contains no indices, and will result in:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

One benefit of directly assigning a value to an index via this method is that you can directly perform arithmetic operations on elements, as seen in the list example in the Assignment section. I will demonstrate once again:

```
# In this example I will simply perform an arithmetic operation on
    an element of a list, then print it

list = [2, 3, 5, 7, 11]
```

```
4
5  list[0] += list[1] # adds element at index 1 to element at index 0
6
7  print(list)
8
9  # which results in:
10 [5, 3, 5, 7, 11]
11
12
13 # In this slightly more complex example each element in the given
       list will be added to itself. Details of the for loop operation
        will be explained in the Data Structures section if you are
       not already familiar
14
15 some_natural_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9 ,10]
16
17 for index, number in enumerate(some_natural_numbers):
18     some_natural_numbers[index] += number
19
20 print(some_natural_numbers)
21
22 # which results in
23 [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

As mentioned above it is possible to make a list an element of a list, and iterate over it with a for loop. This can be performed in a couple of way. If a regular or nested for loop seems alien, I recommend jumping ahead to the *for loop* section under Control Flow.

```
1  # Method 1: nested for loop
2  animals = [["labrador", "boxer", "greyhound"], ["eagle", "parrot",
       "cockatoo"]]
3  for animal in animals:
4      for species in animal:
5          print(species)
6
7  # which would output:
8  labrador
9  boxer
10 greyhound
11 eagle
12 parrot
13 cockatoo
14
15 # Alternatively, if you just wanted to retrieve a sub-list, you
       can simply use a single for loop
16 animals = [["labrador", "boxer", "greyhound"], ["eagle", "parrot",
       "cockatoo"]]
17 for animal in animals:
18     print(animal)
19
20 # which would output:
21 ['labrador', 'boxer', 'greyhound']
22 ['eagle', 'parrot', 'cockatoo']
23
24 # Method 2: direct indexing
25 animals = [["labrador", "boxer", "greyhound"], ["eagle", "parrot",
       "cockatoo"]]
26 print animals[0]
27
28 # which would output
```

```
29 ['labrador', 'boxer', 'greyhound']
```

There are **many other methods, as well as tricks you can perform** on list indices called "list slicing" in order to do things as such as modify, and even reverse the order of elements of a list, but I will leave that up to the reader to investigate as it is beyond an elementary introduction.

- **Tuples** (max size: ?; immutable: yes; ordered: yes)
A tuple is very much like a list in terms of its structure and the way in which it is interacted with. You may wonder what the point of them is if they are just like lists, so here is the motivation: tuples are immutable, meaning they cannot be modified once they are created, this means that they have a set size in memory when they are defined, so they have much less resource overhead for your computer. In essence, there is no need for your computer to make provisions for a data structure that can increase the amount of memory it requires with time. While this might not seem like a big deal, it can become important when your tuples become very large, or your program requires many tuples to be stored in memory, and you do not need to write to, or modify their contents in any way. This is useful when returning data from a function in a structured manner, since it is often the case that the program will simply only need to read the data that is returned from a function rather than modify it. A tuple has the following form:

```
1 socket = (port, IP_address)
2 colours = (("orange", "vermilion","copper"), ("blue", "azure", "
    ultramarine"), ("magenta", "fuchsia"))
```

So in summary:

  - Tuples can be defined but they cannot be written to after they have been defined

```
1 primes = (2, 3, 5, 7)
2 x = ("a", "b", (1, 2, "x"))
```

  - Elements of tuples can be read by index

```
1 x = ("a", "b", (1, 2, "x"))
2 print(x[2])
3
4 # will result in:
5 (1, 2, 'x')
```

  - Arithmetic can be performed on tuples. Tuples can be added which RETURNS another new tuple, which contains the elements of the tuples in the order in which they were added. Note, I emphasise that this does not write to a tuple, it defines a new tuple. A tuple can be multiplied by an integer n greater than zero which returns a new tuple containing the elements of that tuple duplicated n times. Both of these operations are equivalent to the same operations on lists.

```
1 # Addition
2 (1, 2, 3) + ("a", "b", "C")
3 # results in
4 (1, 2, 3, "a", "b", "c")
5
6 # Multiplication
7 ("C. elegans")*2
8 # results in
9 ("C. elegans", "C. elegans")
```

  - A tuple is iterable, hence its elements can be iterated over

- **Dictionaries** (max size: ?; immutable: no; ordered: yes in Python 3, no in Python 2)
A dictionary is a data structure that provides a correspondence between a key and a value. Like an actual dictionary that has a word, followed by a definition, the key is like the word, and the value is like the definition. A dictionary is enclosed in curly braces, and each key-value pair in the dictionary is separated by a comma, and are linked by a colon between them. They have the following form:

```
1 # generally:
2 dictionary = {key1:value1, key2:value2, keyn:valuen}
3
4 # as an example we will use throughout this section:
```

```
5 my_computer = {"CPU":"i7 4790k", "GPU":"NVIDIA GeForce GTX 980 Ti", "
    PSU": "Cooler Master 750 watt", "Motherboard": "Asus Maximus VI
    Hero"}
```

So, in summary:

- An empty dictionary must be initialised before use:

```
1 x = {}
```

- Dictionaries are a collection of key-value pairs with form {key:value}
- Dictionaries are mutable, hence their key-value pairs can be added, removed and updated
- Dictionaries can be iterated over
- Dictionaries can be checked to see if they contain a given key
- Order of key-value pairs is not guaranteed. Dictionaries are ordered in Python 3, and unordered in Python 2

Values can be returned from a dictionary in the following ways:

```
1  # We will work with this dictionary for both examples
2  my_computer = {"CPU":"i7 4790k", "GPU":"NVIDIA GeForce GTX 980 Ti", "
       number_of_GPUs":1, "PSU": "Cooler Master 750 watt", "Motherboard":
        "Asus Maximus VI Hero"}
3
4  # Method 1, treating the key like an "index"
5  CPU = my_computer["CPU"]
6  print(CPU)
7  # which would output:
8  i7 4790k
9
10 # Method 2, using the get method of the dictionary data structure
11 GPU = my_computer.get("GPU")
12 print(GPU)
13 # which would output:
14 NVIDIA GeForce GTX 980 Ti
```

Values of a dictionary can be updated by assigning a new value to an existing key in the following way:

```
1  my_computer = {"CPU":"i7 4790k", "GPU":"NVIDIA GeForce GTX 980 Ti", "
       number_of_GPUs":1, "PSU": "Cooler Master 750 watt", "Motherboard":
        "Asus Maximus VI Hero"}
2
3  # To see original value:
4  print(my_computer["GPU"]
5  # would output:
6  NVIDIA GeForce GTX 980 Ti
7
8  # now to update the value associated with key "GPU"
9  my_computer["GPU"] = "GPU":"NVIDIA GeForce GTX 1080 Ti"
10 print(my_computer["GPU"]
11 # would output:
12 NVIDIA GeForce GTX 1080 Ti
```

Key-value pairs can be added or removed from the dictionary in the following way:

```
my_computer = {"CPU":"i7 4790k", "GPU":"NVIDIA GeForce GTX 980 Ti", "number_of_GPUs":1, "PSU": "Cooler Master 750 watt", "Motherboard": "Asus Maximus VI Hero"}

# Adding a key-value pair is as simple as using the approach that was taken to access and element of the dictionary in the "index" way, except this time we are specifying a key that does not yet exist as the "index"
my_computer["display"] = "Dell Ultrasharp"

# Deletion method 1: pop method (dictionary.pop(key))
my_computer.pop("CPU")

# Deletion method 2: del keyword with "index" approach (keywords/statements are a concept we haven't touched on yet, but approximately what is going on should be self-evident)
del my_computer["CPU"]
```

These are the fundamental functions of dictionaries but **more can be found here**

- **Sets** (max size: ?; immutable: no; ordered: no) A set is a collection of unique objects, and usually related objects, to clarify this mean that duplicates are not permitted. In Python sets are unordered with respects to the order in which elements are added to them, and hence are unindexed. Sets have a number of operations that are defined on them that allow us to study the property of sets, as such as their similarity to each other. Some of these operations are:
  - set.intersection() : The intersection of sets is strictly the elements that belong in both of the sets. This operation will return a set containing only those elements. This concept is related to the logical AND
  - set.disjoint() : Returns whether or not two sets have an intersection
  - set.union() : The union of sets is equivalent to joining the two sets. This will return a set containing elements of both sets
  - set.issubset() : A subset is a set whose elements also exist in another set. Python has chosen the definition of "improper subset" meaning that one set can be a subset of another, even if they are identical. This function will return True if a given set is a subset of the set provided as the argument, and False otherwise. Note: the empty set is a subset of any set.

  While this may seem odd, and not very useful, sets are a great data for grouping large numbers of related elements. Say you might have a dictionary whose keys are nations in which certain types of plants grow, and for each nation, you want to keep a collection of a large amount of plants that can be found there, but in addition to this, you want to know which plant species can be found in multiple countries to get an idea of how humans have spread them.

Control flow

So far we have seen how to assign a value to a variable, how to perform operations on values, and how to format your code into logical groupings based on what it does in the form of functions, but we need control flow to make programs. Control flow is broken up into two categories.

The first category is conditional operations, which are like branching decisions where an outcome is based on whether or not a particular condition, or conditions within your code are met. The second category is loop operations, where specified code is run a certain number of times based upon a condition, or conditions. The important characteristic of both of these categories of control flow mechanisms is that they perform some operation based on whether or not conditions are met.

1. **Branching conditional operations**:
   - **if statement**

     The *if* statement is the "head" of the machinery of conditional operations in Python, it is where the block of conditional operations start, and hence it is where the first condition, or conditions are defined by the programmer, and checked by the program. As you can imagine it is called as such because it follows an if x then y style of logic. The following examples will illustrate this.

```
1  is_it_raining = True
2
3  if is_it_raining is True:
4      print("Use your umbrella")
5
6  is_it_raining = True
7  brought_an_umbrella = False
8
9  if (is_it_raining is True and brought_an_umbrella is False):
10     print("You got rained on")
11
12 24hr_time = 900
13 am_I_hungry = True
14
15 if (24hr_time < 1200 and am_I_hungry is True):
16     print("Time to have breakfast") #night breakfast is cool too
```

     As you can see here, multiple (theoretically unlimited) conditions can be chained together using conditional "and" and "or". A statement containing more than one condition MUST be encapsulated in an outer set parentheses "()" as you can see above. Inner statements may be encapsulated in further parentheses for neatness, and order of precedence, this is very important when it comes to negation, so be careful. Negation can be performed by placing a logical "not" statement outside of the enclosed statements, or before the single statement that you wish to negate. Do be mindful of De Morgan's Law when you do this as I wish reiterate through example.

```
1  hot = True
2  humid = False
3  if not(hot and humid):
4      print("The weather is cool or dry")
5
6  # this is logically equivalent to
7
8  if (not hot or not humid):
9      print("The weather is cool or dry")
```

   - **else**

     The *else* statement is the if block termination point, it represents what happens if no conditions within the if block are met, as as such takes no arguments. It extends if by the logic if x then y, otherwise z. As you have seen above, an else statement is not absolutely necessary, but is very useful since you usually want something to happen in the absence of the if conditions being met. Some examples reflecting on the above examples are as follows:

```
1  is_it_raining = True
2
3  if is_it_raining is True:
```

```
    print("Use your umbrella")
else:
    print("It is not raining")

is_it_raining = True
brought_an_umbrella = False

if (is_it_raining is True and brought_an_umbrella is False):
    print("You got rained on")
else:
    print("You are dry")

24hr_time = 900
am_I_hungry = True

if (24hr_time < 1200 and am_I_hungry is True):
    print("Time to have breakfast")
else:
    print("It is after morning or you aren't hungry")
```

When using multiple sequential if statements, you may wish to terminate them with an else statement. When using multiple if statements in a row, each if statement will be checked regardless of the outcome of the previous one. If statements can also be nested.

```

# it is assumed variable values come from elsewhere

if day == "monday":
     paid = True
     money += 800
     print("You have been paid")

if (days_since_rent_paid >= 14):
    if (money >= 300):
        days_since_rent_paid = 0
        print("Rent has been automatically debited")
    else:
        days_since_rent_paid = 0
        print("Rent has been automatically debited")
        print("Your account is overdrawn")

else:
    print("Rent is not due yet")
```

Notice that the second if statement is executed regardless of the outcome of the first if statement. They are independent events, and since they have no causal relationship, using sequential if statements will ensure that each if statement will be executed. There is however a causal relationship between the second if statement, and the final else statement, so make sure that your logic takes this into account.

- **elif**

The *elif* (else-if) statement is like the if statement except it establishes a causal link between all if style statements in the chain. It symbolises the logic if x then y, otherwise if z then p, otherwise not(y or p) (where the final otherwise represents the else). Let's write two programs that both test properties of line segments to try to find the highest dimensional shape (between 1 and 3 dimensional). Let's assume that all of the line segments start at the same point $(0, 0, 0)$, are positive, and are labeled x, y, z to indicate that they are all perpendicular to each other.

```
x = 1
y = 1
z = 1

if (x * y * z) > 0:
```

```
    print("Three bases form a cuboid with volume: ", x*y*z)

if (x*y + y*z + z*x) > 0:
    print("At least two bases form a plane:")
    print("x and y form a plane with area", x*y)
    print("y and z form a plane with area", y*z)
    print("z and x form a plane with area", z*x)

if (x + y + z) > 0:
    print("A basis forms a line segment")
    print("x forms a line segment with length: ", x)
    print("y forms a line segment with length: ", y)
    print("z forms a line segment with length: ", z)
else:
    print("x, y, z have no magnitude")
    print("x length: ", x, "y length: ", y, "z length: ", z)
```

In this situation, if one of these if statements evaluates to true it will just move onto the next one, even if the first if statement evaluating to true provides the condition upon which the code should terminate.

```
Three bases form a cuboid with volume:  1
At least two bases form a plane:
x and y form a plane with area 1
y and z form a plane with area 1
z and x form a plane with area 1
A basis forms a line segment
x forms a line segment with length:  1
y forms a line segment with length:  1
z forms a line segment with length:  1
```

As you can see this is not the behaviour we area looking for. The objective was to find the shape of greatest dimension, and yet the program has continued on after its discovery. While this might not seem like an issue, and in some cases it is desirable, if each of those following *if* statements were to execute a lot of resource intensive code, or if there were a lot of conditions, it could have a significant impact upon the efficiency of the program. Alternative we can use *elif* to short circuit the code to completion, once it has met the specified condition.

```
x = 1
y = 1
z = 1

if (x * y * z) > 0:
    print("Three bases form a cuboid with volume: ", x*y*z)

elif (x*y + y*z + z*x) > 0:
    print("At least two bases form a plane:")
    print("x and y form a plane with area", x*y)
    print("y and z form a plane with area", y*z)
    print("z and x form a plane with area", z*x)

elif (x + y + z) > 0:
    print("A basis forms a line segment")
    print("x forms a line segment with length: ", x)
    print("y forms a line segment with length: ", y)
    print("z forms a line segment with length: ", z)
else:
    print("x, y, z have no magnitude")
    print("x length: ", x, "y length: ", y, "z length: ", z)
```

This code will evaluate to:

```
Three bases form a cuboid with volume:  1
```

The above revised version of this block of code now produces the result we want in the least amount of time. As soon as the condition is met, the chain of *elif* statements "short circuits" to the point following the *else* statement and allows the code to continue executing on from there.

2. Looping conditional operations:

- **for loop**

The purpose of the *for* loop is to iterate over the elements of some sequence or "iterable" object. To summarise:

- An iterable object is an object that has the property of being a collection of separately identifiable elements, like how a list is defined as a collection, but each item on the list is separate
- A for loop will access each element of the collection, one at a time, and in sequence. Each time an element is accessed, control is passed to the body of the for loop allowing an operation to occur.
- The value of the element is assigned to the variable that is iterating over the object, in sequence. Hence, for each element in the object, the value of that element is made available to the iterating variable and can be operated upon in the body of the for loop. A basic for loop has the following form:

```
1  # define, or return from a function some iterable object
2
3  iterable_object = [1, 2, 3]
4
5  # invoke the for loop as such where element is the variable that
       will hold each indexed value of the iterable object for each
       step of the for loop
6
7  for element in iterable_object:
8      # perform some operation on element in the body of the for
           loop - here we have chosen the example of the print
           function as the operation performed in the body of the
           loop
9      print(element)
10
11 # will output the following:
12 1
13 2
14 3
```

Examples of iterables are:

- Strings
- Lists
- Sets
- Tuples
- Dictionaries
- The returned value of the *range()* function
- The elements and indices returned by the *enumerate()* function

The basic structure of the for loop is as follows:

```
1  # the reinforce the above example with a change of variable names,
       we shall present another with a list to demonstrate one of the
       most simple iterations over an object
2
3  days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "
       Friday", "Saturday"]
4
5  for day in days:
6      print(day)
7
8  # which outputs
9  Sunday
10 Monday
11 Tuesday
```

```
Wednesday
Thursday
Friday
Saturday

# Our next example will be a dictionary to demonstrate that there may be additional requirements when iterating over an iterable object. We will use our trusty my_computer dictionary from the Data Structures section since it is as good as any other

my_computer = {"CPU":"i7 4790k", "GPU":"NVIDIA GeForce GTX 980 Ti", "number_of_GPUs":1, "PSU": "Cooler Master 750 watt", "Motherboard": "Asus Maximus VI Hero"}

for key, value in my_computer.items():
    print(key, ":", value)

# will output in Python 2:
('Motherboard', ':', 'Asus Maximus VI Hero')
('GPU', ':', 'NVIDIA GeForce GTX 980 Ti')
('PSU', ':', 'Cooler Master 750 watt')
('CPU', ':', 'i7 4790k')
('number_of_GPUs', ':', 1)

# will output in Python 3:
CPU : i7 4790k
GPU : NVIDIA GeForce GTX 980 Ti
number_of_GPUs : 1
PSU : Cooler Master 750 watt
Motherboard : Asus Maximus VI Hero
```

Take note of three things here with this dictionary:

(a) The dictionary iterable object must call its method dictionary.items(), as seen in my_computer.items(). This is because if one simply iterates over the dictionary, only the keys will be made available to the variable that is iterating over it. A bit strange, but this is just how the dictionary object works. So keep in mind that certain seemingly similar objects may have to be treated differently when performing similar tasks on them.

(b) Instead of one variable iterating over the object, there are two, "key" and "value". This is because the method dictionary.items() returns tuples containing the key, and the value, in that order. As this is so, it is necessary to iterate over it with a number of variables equal to the number of objects the my_computer.items() method has returned for each element. To reiterate, for a key and value being returned, we need a key and value variable to hold both of them for each step.

(c) As mentioned, dictionaries are not ordered in Python 2. Also note that due to the fact that 'print' is a statement in Python 2, it has enclosed each print operation with the brackets that the arguments are normally enclosed in, within Python 3's print *function*.

If you encounter a data structure containing another data structure, as such as a list as an element of a list, or a list as a value of a dictionary, you will want to be able to iterate over that inner iterable object. In order to do this we can nest a for loop within another for loop. The act of doing so allows us to treat the variable that iterates over the outside iterable object, like an iterable object itself. This follows on with the same logic as the outer for loop. Another variable will be defined to iterate over that object in the outer loop itself. As always, be careful and consistent about the kinds of things you nest inside of object, because as seen above, some iterable objects require special treatment in order to be read from correctly when looping over them.

```
# Example 1: a very basic nested for loop, looping over nested elements of a list

x = [[1,2], [3, 4, 5], [6, 7]]
for i in x:
    for j in i:
        print(j)
```

```
7
8  # should output
9  1
10 2
11 3
12 4
13 5
14 6
15 7
```

As you can see here there is no requirement to perform some specific action within the body of a for loop. It is perfectly fine to nest another for loop in there, another flow control statement, or anything that you may desire.

```
1
2  # Example 2: an exaggerated case of nesting to show what can be
       achieved, and what might happen unexpectedly
3
4  x = ["a", ["b", "c", [1, 2]]]
5
6  for i in x:
7      print(i)
8      for j in i:
9          print(j)
10         for k in j:
11             print(k)
12
13 # should result in
14 a
15 a
16 a
17 ['b', 'c', [1, 2]]
18 b
19 b
20 c
21 c
22 [1, 2]
23 1
24 2
```

Now, while it is nice that Python hasn't thrown any errors when trying to iterate over a string of length 1, it has consequently printed that single length string as many times are there are for loops from the point at which that element is reached. This result should tell us to take care when constructing data structures with a variable amount of nesting per element, as python will exhibit this behaviour.

```
1  # Example 3: a dictionary nested inside of a list to demonstrate
       possible special requirements
2
3  pets = [{"species":"cat", "name":"Peter", "age":6}, {"species":"
       dog", "name":"Jeff", "age":5}]
4
5  for pet in pets:
6      for key, value in pet.items():
7          print(key, value)
8
9  # which will output with Python 2:
10 ('age', 6)
11 ('species', 'cat')
12 ('name', 'Peter')
13 ('age', 5)
14 ('species', 'dog')
```

```
('name', 'Jeff')

# which will output with Python 3:
species cat
name Peter
age 6
species dog
name Jeff
age 5
```

Sometimes you will want to retrieve both the index, and the value at that index when iterating over an object with a for loop. This can be accomplished by passing your object as an argument to the enumerate() function that is built into python. Similar to the above dictionary example, it too returns a tuple of values corresponding to "index" and "value" and as such requires two variables to iterate over the object. This is demonstrated as follows:

```
days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "
    Friday"]
for index, day in enumerate(days):
        print(index, day)

# which will output:

0 Sunday
1 Monday
2 Tuesday
3 Wednesday
4 Thursday
5 Friday
6 Saturday
```

Finally, we can emulate the function of a for loop that you may more commonly see in other programming languages where the number of loops is specified explicitly by the programmer rather than being specified by the number of iterable elements of an object. This behaviour can be accessed via the range(start, stop, step) function where start, stop, step must be integers. Some notes:

- By default range starts at zero, and ranges to stop - 1. In our interval notation, the range will be [0, stop-1]. Hence range(5) will be the range 0, 1, 2, 3, 4
- When specifying a start point, range will consist of the interval [start, stop-1] having length stop minus start.
- Start and step are options, hence stop is necessary.
- In order to supply an argument for step, a start argument must be supplied as well, as range() expects three arguments in order to take step as an argument.
- to use the length of an interable object the len() function may be used. For object x, len(x) will return the number of elements in that iterable object. An example will be given for this.

This behaviour is as follows:

```
# Example 1: range(stop)

x = []

for i in range(6):
        x.insert(i, i)

print(x)

# which results in:
[0, 1, 2, 3, 4, 5]

# Example 2: range(start, stop)

x = []
y = 0
```

```
17
18 for number in range(1, 6):
19     y += number
20     x.append(number)
21
22 print(x)
23 print(y)
24
25 # which should result in
26 [1, 2, 3, 4, 5]
27 15
28
29 # Example 3: range(start, stop, step)
30
31 x = []
32 y = 0
33
34 for number in range(1, 6, 2):
35     y += number
36     x.append(number)
37
38 print(x)
39 print(y)
40
41 # which should result in
42 [1, 3, 5]
43 9
```

- **While loop**

  As you may guess from its name, a while loop is a loop that will continue looping while some condition remains a constant value, and terminates when that value changes. This constant value may be a single value, or a whole boolean expression that has to completely evaluate to True or False. At its core, all of these values represent either True or False boolean values, but may be abstracted.

  For example, a while loop may run while the value of an incrementing variable is less than some value, or variable's value. Compared to a for loop which iterates over a particular structure, the flavour of a while loop is more that of a loop that continues until some "task" is completed. Because this is true, you must be careful to ensure that the loop does actually terminate (unless explicitly desired otherwise), otherwise it is not hard to create a scenario in which the loop may run indefinitely. This possibility has caused a fair amount of undue caution and skepticism towards their usefulness, but if used carefully, they are a valuable tool. To break it down, the while loop's characteristics are:

  - a while loop consists of the while comparator "head" that is checked each loop to see if the specified condition is met, and the body where the code is executed, much like a for loop
  - the head can contain an arbitrary amount of comparators connected with boolean operators "and" and "or"
  - like all control flow operations, while loops can be nested inside of other loops and branching conditionals

```
1  # A quick rundown
2
3  # This while loop will run while the day of week is less than or
       equal to 7, and increment the week day variable in the body
     with each iteration of the loop. When it is does control will
       exit from the loop and print the text that follows.
4
5  end_of_week = 7
6  day_of_week = 1
7
8  while day_of_week <= end_of_week:
9      print(day_of_week)
10     day_of_week += 1
11
12 print("The week has ended")
13
14 # Will output:
15 1
16 2
17 3
18 4
19 5
20 6
21 7
22 The week has ended
```

  Obviously there are much more useful operations you can perform with a while loop. As previously stated one may create a condition out of an arbitrary amount of boolean statements chained together, but with growing complexity comes increased potential for errors resulting in a non terminating loop.

Keywords

In Python there is a collection of syntax referred to generally as "keywords" that are reserved names that perform special functions. Name reservation means they cannot be used for variable names. Already you will have encountered keywords, as such as *for*, *if*, and *while*, as well as *and*, and *or*. As you can see these keywords require different formatting for them to function, so it is not enough to say that a keyword is simply an instruction that precedes some variable - they have differing rules and syntax. As these have already been covered in detail, I will turn the reader's attention to some of the other common and useful keywords that are applicable to the scope of this volume.

- del: del precedes the name of an object and will delete that object. del can also be used to delete elements of objects as such as elements of dictionaries or lists. To do so place the del keyword before the expression for that data structure which references as particular index, or member of that data structure. For example:

```
1 list = ["a", "b", "c"]
2 del list[0]
3 print(list)
4
5 # Which outputs
6 ["b", "c"]
```

- in: used outside of a for loop, in checks whether a specified variable is a member of a given data structure and returns a boolean of True or False depending on whether or not that variable is "in" that data structure. When used as part of a for loop, it checks for membership of a "next" element, which is assigned to the variable before in. For example:

```
1  states_of_australia = ["Western Australia", "South Australia", "
       Victoria", "Tasmania", "New South Wales", "Queensland"]
2
3  states_and_territories = ["Western Australia", "South Australia", "
       Victoria", "Tasmania", "New South Wales", "Queensland", "Northern
       Territory", "Australia Capital Territory"]
4
5  for item in states_and_territories: # for loop context
6      if item in states_of_australia: # purely boolean evaluation
           context
7          print(item, "is a state")
8      else:
9          print(item, "is a territory")
10
11 # which in Python 3 outputs:
12
13 Western Australia is a state
14 South Australia is a state
15 Victoria is a state
16 Tasmania is a state
17 New South Wales is a state
18 Queensland is a state
19 Northern Territory is a territory
20 Australia Capital Territory is a territory
```

- not: not is the boolean negation keyword. It can be used to semi-directly check whether something does not evaluate to a certain boolean value. This might sound a little bit abstract, so for example, considering what you have learned above about the *in* keyword, *not* can be placed before *in* to evaluate to true if a certain element is not a member of some collection of objects. To explain how this relates to boolean negation, the *in* keyword will evaluate to false for all members of a collection if something is not in it, but you want this expression to evaluate to True because you want to know if something is not in that collection. Therefore you place *not* before *in* to negate that False return value to True, thus making the expression evaluate to True.
- return: return is placed within a function definition for the purpose of passing a value out of the function's scope, back to where the function was called from. A value follows the *return* keyword on the same line, and that is the valued that is "returned" back to where it was called from. This is performed under the condition that when the function is called, it is called by assigning it to a variable with the "=" assignment operator. Thus when the code inside of the function

reaches the *return* keyword, that value that follows it will be assigned to that variable where it was called from. For instance:

```
def weekday_name(day_number):
    days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "
       Friday", "Saturday"]
    return days[day_number] # returns the value of the variable here

weekday = weekday_name(5) # calls the function, then when the
    function reaches the return keyword, assigns the returned variable
     to the variable "weekday"

print(weekday)

# Will output (remembering that lists are indexed from 0)

Friday
```

- is: *is* serves as a sibling to the "==" comparitor, but functions somewhat differently. Where "==" checks whether two objects have equal representations in the computer's language (look into hash functions if you're interested), and therefore equal values, *is* checks for equality by seeing if two objects reside in the same place in the computer's memory. For example:

```
# Example 1: Mutable list with same elements at different memory
    addresses
colours = ["red", "blue", "green"]
same_colours = ["red", "blue", "green"]

print(colours == same_colours)

# Will output:
True

print(colours is same_colours)
# Will output
False

# Example 2: when things don't go as we expect with immutable types
alices_age = 21
bobs_age = 21

print(alices_age == bobs_age)

# Will output
True

print(alices_age is bobs_ages)

# Will output
True
```

As you can see in the second example, Python may regard the value of an immutable variable as belonging to multiple, separately defined variables, hence Python sees the values for those variables stored at the same memory address.

- break: *break* is used to promptly exit from the inner most loop in which the break statement is placed. Even if it is in the body of an *if* statement inside of a loop, the execution of the *break* keyword will result in the code exiting the loop that the *if* statement is contained in. This is very useful if, for example, you are interating over a collection like a list in order to perform some action on one, or a number of elements less than the number of elements in the list, and you don't want the for loop to continue once those elements have been found. This will speed up the runtime of your code if you are dealing with large data structures. Remember that break will not exit out of all of the loops, if there is nested looping, just the inner most one that contains the *break* keyword

- global: if a variable is defined outside of a function, but one wishes to access it inside of a function, a variable with the SAME name must be stated within the function with *global* keyword preceding it. Further explanation is given in the following Scope section, but I will give a quick example:

```
1  bird = "Pigeon"
2
3  def bird_name():
4      global bird
5      return bird
6
7  get_bird = bird_name()
8  print(get_bird)
9
10 # Will output
11 Pigeon
```

The dot operator

Although this volume will not cover the concept of classes, if you use a module, you will invariable encounter the concept of accessing a class member. In order to do so you place a dot at the end of the module name, class name, or class instance name, followed by the member of the class you wish to access. Class members are referred to as methods. They are basically the same as functions, but they reside inside of classes and have a couple of **special properties** that aren't really relevant to your usage of them now. Using the "math" module, I will give a couple of examples of the syntax of the *dot* operator.

```
1 import math
2
3 theta = math.pi()/4
4
5 sine_of_angle = math.sin(theta)
6 cosine_of_angle = math.cos(theta)
7 radians_to_degrees = math.degrees(theta)
```

As you can see, invoking a class member method is pretty much exactly the same as calling a function, with the addition of using the *dot* operator to denote that the method you wish to access belongs to the given class, and hence to class that method from said class.

Formatting

Indentation blocks

In the Python programming language, most functionality can be broken down into a pair of entities: (a) one dictates the conditions of the code's execution, (b) and the other is the code to be executed. This grouping is called a *block*. Three examples are as follows:

```
1  #(1)
2  def function():                    #(a)
3      print("Hello, world!")         #(b)
4
5  #(2)
6  if x:                              #(a)
7      print("Hello, world!")         #(b)
8
9  #(3)
10 while True:                        #(a)
11     print("Hello, world!")         #(b)
```

The important thing to notice is that for any such operation, the inner code to be executed is *always* indented by one tab level more than the code that dictates the conditions of the code's execution (a level is the blank space created by a tab input). This isn't just to make your code look organised, it is a rule that the Python interpreter follows in order to know how to execute code. If you have ever seen another programming language like Java or C, it is analogous to enclosing the code to be executed in curly braces "{}" following its calling code. Code blocks can be "nested" inside of other code blocks using the same rule. An example is as follows:

```
1  def function(): #block 1
2      y = 2 #block 1
3      x = 0 #block 1
4      if x = 0: #block 2
5          y = 0 #block 2
6          while y < 2: #block 3
7              print("Hello, world!") #block 3
8              y += 1 #block 3
9              day = "monday" #block 3
10         print(day) #block 2
11
12 function()
```

Notice how the if statement is on the same layer as x = 0? Please be careful and don't confuse when I said "the inner code to be executed (b) is *always* indented by one tab level" to mean that "inner code can't also be (a) code that dictates the conditions of the code's execution ". Absolutely not. In this case the if statement becomes "the inner code to be executed (b)" because we want it to run after assignment x = 0 has occurred in the scope of the function definition.

Scope

So far variables and variable assignment has been covered, but we haven't discussed *when* a variable can be written to or read from. Scope is the set of rules that dictate this behaviour. In short, a scope rule defines a sort of variable "enclosure" in a way meaning that variables cannot be accessed across the borders of these enclosures without special conditions allowing them to. Examples of such "enclosures" are separate functions, and inside and outside of functions, and there are mechanisms for performing both of these functions. In fact these two scenarios deal with *local* and *global* scope. In Python there are in fact four scope types: local, enclosed, global and built-in. In this guide we shall only be dealing with local and global.

Local scope

Local scope is defined as the variables which are accessible in a single scope. The most common example of local scope you are going to encounter is within a function. In a function, *almost* any variable that has been assigned can be accessed below its definition no matter which block it belongs to. However I stress that just because a variable assigned occurs within your code, it does not mean that it is actually assigned when the code runs. There may be if statements within your code that have a variable assignments in one branch, but not the other, and if this variable is accessed later on without the code having taken that branch, the variable will not be assigned. More on this below.

Scope nesting and order

In Python, a block does not define a scope, unlike in many other programming languages. This means that you can assign a variable within a nested block and then call it outside of that block IF the interpreter reaches that variable assignment when the code runs.

```
1  def function():
2      y = 2
3      x = 0
4      if x = 0:
5          y = 0
6          while y < 2:
7              print("Hello, world!")
8              y += 1
9              day = "monday"
10         print(day)
11
12 function()
```

If you were to run this script the output would look like this:

```
1 Hello, world!
2 Hello, world!
3 monday
```

As you can see the print function is called from a different block to where the variable it is printing is assigned, but because it was assigned before it was accessed by the print function Python considers it to be properly assigned and available to be passed as an argument to a function. DO be careful of this particular behaviour of Python, especially if you have ever touched another language where blocks had scope, because if you accidentally use a variable name to perform some action within some block, it will come back to haunt you if you need to use or return another variable with the same name later.

Does the interpreter reach the assignment?

It is very important to consider what code will be reached when the code is executed. I have set up the function below such that the initial if statement condition will not be met due to the value of the x variable. When control is passed to the else statement it will become clear that the variable "day" has not been assigned even though it appears in our code.

```
1  def function():
2      y = 2
3      x = 0
4      if x == 1:
5          y = 0
6          while y < 2:
```

```
            print("Hello, world!")AU

            y += 1
            day = "monday"
    else:
        print(day)

function()
```

When the code runs the output will look like this:

```
Traceback (most recent call last):
  File "scope.py", line 16, in <module>
    function()
  File "scope.py", line 14, in function
    print(day)
UnboundLocalError: local variable 'day' referenced before assignment
```

As anticipated the interpreter produces an error when it attempts to execute this code. As you might expect "local variable 'day' referenced before assignment" means that the code tried to access the variable before it was assigned, if it has even been included in the code at all. In this case it has been included in the code but because the if statement wasn't satisfied, it was never assigned in the consequential block of code that followed. So be careful when you're using control flow like if statements that you aren't relying on inaccessible code.

Accessing variables between functions

This isn't really a key topic for scopes, and there is no statement that will allow us to access a variable in another function like we will see for global scopes, but it is still an example of how local scopes can be isolated from each other. Ultimately the only way to access a variable in another function is if (a) you want to retrieve the variable, the function you are retrieving the variable from must *return* that variable or (b) you want to set a variable in that function, that function must be designed so that the variable you want to set is set by passing an argument to that function when it is called. I believe these mechanisms have been adequately described in the Functions section but here is a short example for each scenario:

```
#(a)
def returning_function():
    return "cat"

def getting_function():
    pet = returning_function() # variable pet will be set to string "cat
        "

getting_function()

#(b)

def receiving_function(age):
    person_age = age # variable person_age will be set to integer 20

def sending_function():
    receiving_function(20)

sending_function()
```

Global scope

Global scope refers to a location where a variable can be defined such that it is accessible everywhere within the script. To read and write to a global variable you first define your global variable in the global scope outside of any function. Then you define a variable inside of the function where you want to access the global variable prefaced by the statement "global". An example is as follows:

```
var = "global"

def print_global():
    global var
    print(var)

print_global()
```

Will output

```
global
```

As you can see by using the "global" statement we have granted access to variable "var" inside of the function definition's local scope access to the variable "var" in the global scope. If this "global" statement is not used, Python will only access variables in the local scope. For example:

```
var = "global"

def print_global():
    var = "local"
    print(var)

print_global()
```

Will output

```
local
```

It should be noted that the use of global variables is not a very common practice and should be avoided whenever possible.

Versions

Python is currently being maintained in two different versions: Python 2.7 and Python 3.7. Bear in mind that Python 2.7's lifecycle ends in 2020 and will not be maintained past then. I won't dwell too long on the differences as the important ones are well documented elsewhere, as such as **here** however I will mention a couple of key differences and important things that a beginning absolutely must know.

Syntax and operations

First of all in Python 2.x, 'print' is a statement, meaning that it is formatted like:

```
print 2
print "text"
print variable
```

Whereas in python 3.x 'print' is a function which means it takes an argument, and should look like this:

```
print(2)
print("text")
print(variable)
```

The other difference I will mention relates to the division operator, and whether it will return an integer or a float. The following operations in Python 2.x function like this:

```
3/2 = 1
3//2 = 1
3/2.0 = 1.5
3//2.0 = 1.0
```

Whereas in python 3.x the following operations function like this:

```
3/2 = 1.5
3//2 = 1
3/2.0 = 1.5
3//2.0 = 1.0
```

As you can see the concept of integer divison with rounding must be explicitly defined in Python 3.x.As you can see the concept of integer divison with rounding must be explicitly defined in Python 3.x.As you can see the concept of integer divison with rounding must be explicitly defined in Python 3.x.

Modules and running scripts

As you may expect different versions of python may have compatibility issues with different modules, although there are some that have been developed with the intention to run on both. This compatibility may be dependent on that module's version as well so ensure that you check the developer's documentation and download the correct version. When it comes to running your script you will have to tell your IDE which version of the Python interpreter you want to use if you are using something like Pycharm. If you are using a text editor you must specify which version of Python you want to use as the interpreter for your script. An example is as follows:

```
python my_script.py
python3 my_script.py
```