The Python Pilgrim Part One: The Theoretical Minimum

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

Welcome to a new path.

Perhaps you're at the beginning, or possibly already on a similar path. There are many starting points and many ways that lead to the end, and as such there are vast opportunities to learn the merit of trying new approaches to solving problems. Python is not only another way to solve problems, it is an excellent starting point for those wanting to understand programming and formal logic. 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. You are the culmination of the human capacity to reason and use logic, and over 3000 years of recorded knowledge that has been distilled into your education. 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 you were always capable of.

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 $4 \cdot 7 = \mathbf{14}$ and $4 \cdot 8 = \mathbf{15}$ ...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 $4n$ is two digits, 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? 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 dilligent 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.

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 quotient

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

!= : Non-equivalence comparator. For objects a, b, that are not equal to each other if a != b (not equals) 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

++, −− : For integers and floats a++ means $a = a + 1$, same for subtraction

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

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 Functions section

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

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.

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.

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.**

The content 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

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. Perhaps to your surprise you are already a programmer. Our brains are like computers that know a spoken language and know how to transform the language of our thoughts into a spoken language, and that language back into thoughts, so you are in a pretty good position to get started as a sentient, cognitive, language-user. A programming language is just like a spoken human language, however the subject you are communicating with, a computer, is not very intelligent or flexible (the subject of artificial intelligence aside). This means that a programming language only has a fairly small vocabulary, or set of rules, not because there is only a small amount of ideas that can be communicated through them, but to completely eliminate the ambiguity of spoken language and narrow down the many way that rules can be expressed into just one meaning per rule. In essence, programming is using this very refined vocabulary in order to define instructions, instructions that explain to the computer what you want it to do. Your instructions will run in a certain order determined by other instructions and the rules of the language, and all of these instructions run based upon the rules that govern their logic. Each rule has a well defined functionality and will always work the same way. Despite the rules always working the same way, it is still possible to accomplish the same thing in different ways in a programming language based upon your goals or creativity, but the basic rules that dictate those implementations will always have the same meaning every time they are used. 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. These flavours include things like:

- statements that tell the computer what to do if the code has a certain property (the *if* statement)
- arithmetic rules on numbers
- order of operations
- what code gets run when based on the order in which it is written
- how and when code should be indented in order to correctly define a block

All of the critical rules in Python will be covered in later section. 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 lengths of the sides of a right angle triangle" is 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. 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 rules can be built on top of axioms via proofs is a lot like the way that you will define how things work while programming. You will be building instructions that operate logically in accordance to axioms. Instructions are where you define your own rules that follow this hierarchy. 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 will happen when you are learning, or even later on (even experienced programmers make mistakes), so you're going to have to put your ego aside and not get too upset if something gives you an error or does something strange, because the computer's logic is never wrong, you have just made a mistake. 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. If you're imagining that an instruction is just an implementation of rules, based on the previous section, you have read correctly. 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:

```
x = 2
x = x + 2
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:

```
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:

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

or

```
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?

because it's cool desu

How do I start?

install python, install editor or IDE, preferably install linux

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.

Document structure

Each Python script follows a simple template. I will begin with presenting a very simple script that incorporates all essential elements you will encouter before learning about classes.

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

```
import module
```

```
def function ():
''' function prints the string "Hello , world!" '''
        print ("Hello , world!")
```

```
function ()
```

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.

Import

```
import module
```

The import statement is used to make the features of external modules available to your script. "Module" is not the name of a module, just a placeholder for a module's name.

Function definition

```
def function ():
''' function prints the string "Hello , world!" '''
        print ("Hello , world!")
```

This is called a function definition. For now let's just highlight that it consists of the def statement, followed by the function's name with parentheses at the end in which arguments in the form of objects can be passed for the purpose of being made available to the code it contains. Inside of the function definition is the code that will run if the function is *called*. This function object will be discussed in much greater length in the Function section.

Docstring

```
'''function prints the string "Hello , world!"'''
```

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!"
```

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

```
from __future__ import print_function
```

Function call

```
function ()
```

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 parameters inside of the parentheses, and the output of the function can be assigned to a variable on the same line that it is called, but don't worry about it for now if you don't understand what these things mean, that will be explained later. For now just remember that this is how you run a function.

Variables

In very high level terms a variable is like a container that you can store data inside of. A variable can only store one piece of data (in the case of python an "object") at a time, and 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. I will go on to explain this shortly. If you are feeling adventurous consider the following (the following is **OPTIONAL** and not necessary to understanding Python). More accurately a variable is a section of your computer's volatile memory (RAM) (actually with paged memory variables may be allocated onto your hard disk before being swapped back into RAM for execution but this is *way* beyond the scope of this guide) that has a size in bits allocated based upon the size of the type of data it will contain. When you declare a variable you tell the computer to allocate that memory, and to associate the variable with it such that the variable "points" to those memory addresses. This means that if you have a data structure like a "list" inside of a variable, those memory addresses will contain contiguous variables itself (pointers) that point to that data structure's elements in other memory addresses. This is how Python lists can have multiple data types as their elements. A list's memory addresses are known as a *dynamic* array which means more memory addresses are added to the variable if the list grows.

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 language variables do not have to be declared before they can be used. In fact the concepts of variable declaration and initialisation do not really exist in Python, instead the term *assignment* is used. Also unlike many other programming languages you do not need to assign a type to the variable, a variable does not have a type despite its value having a type. These properties allows you to assign variables on the fly as well as assign different types to the same variable, but do be cautious of scope. It is implemented as follows:

```
#not real code, just an illustration of assignment
variable <- data

#real code assigning the integer 1 to the variable x
x = 1

#this can be proven with the print() function
x = 1
print(x)

1          #the output to the terminal

#it is exactly the same for different object types
list = [1, 2, 3]

true_or_false = True

state_capital_dictionary = {QLD:Brisbane, NSW:Sydney}

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

one thousand seven hundred and twenty nine
```

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

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

Another very important variable assignment is assigning the return value of a function to a variable. The return statement, followed by the object to be returned sets the value of the function to that object being returned. For example

```
def return_a_number():
        return 2

x = return_a_number()

print(x)

2
```

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 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 its 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
    * 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
    * 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 open a collection of statements enclosed in parentheses that is not immediately obvious. This is as follows:
    - 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

Boolean logic

all non-none type objects are true by default, short circuiting, the operators don't do what you think they do, bitwise op bools return only lsb result. Empty data structures evaluate to false.

Control flow

The dot operator

Statements

Functions

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)
def function():                    #(a)
        print("Hello, world!")     #(b)
```

```
#(2)
if  x:                                          #(a)
        print("Hello ,  world!"                  #(b)


#(3)
while  True:                                     #(a)
        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:

```
def function(): #block 1
        y = 2 #block 1
        x = 0 #block 1
        if  x = 0: #block 2
                y = 0 #block 2
                while  y < 2: #block 3
                        print("Hello ,  world!") #block 3
                        y += 1 #block 3
                        day = "monday" #block 3
                print(day) #block 2


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.

```
def function():
        y = 2
        x = 0
```

```
        if  x = 0:
                y = 0
                while  y < 2:
                        print("Hello,  world!")
                        y += 1
                        day = "monday"
                print(day)

function()
```

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

```
Hello,  world!
Hello,  world!
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.

```
def function():
    y = 2
    x = 0
    if  x == 1:
        y = 0
        while  y < 2:
            print("Hello,  world!")
            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.

Modules

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
```