A Beginners Guide to Python and Getting Started

Contents

About Python

Python is a high-level, object-oriented programming language comparable to Perl, Ruby, and Java. It was first released in 1991 and is intended to be easier to read/ write than some other, similar languages.

Python ships with a standard library of modules to support many of the most common tasks, and its interactive mode makes testing small snippets quick and easy.

As Python code is typically interpreted, it can run on almost any platform including MacOS X, Windows, Linux, and UNIX

Python is free to download and to use.

Python Documentation

The Python docs are accessible at docs.python.org. This address will direct you to the documentation for the latest version of Python (3.7.2 at the time of writing). If you're going to be coding with an older release of the software, then you should include the version number at the end of the URL, e.g. docs.python.org/2.7.

The Python docs include, among other things, a detailed tutorial, language reference, and library reference. It is, in the author's opinion, an excellent resource for those for whom Python is not his/her first language.

In addition to this manual, the tutorial at w3schools.com/python is, perhaps, more accessible for those for whom Python is his/her first language.

Installing Python

You can either install Python to your local machine or alternatively you can use an online editor to save any installation ebing required. There are various online editors available online however one you could use for these challenges is

Online Editor

Online Python Compiler (Interpreter) (tutorialspoint.com)

Otherwise you can install Python by following the guides below.

Windows

To install Python on Windows:

1. Download the installer version of your choice from python.org/downloads.
2. Run the installer, making sure to add Python to the PATH.

NOTE
The 32-bit version of Python will typically work on a 64-bit machine. The 32-bit version consumes less memory but may be less performant than the 64-bit version.

Linux

To install Python on **Debian** derivatives, use apt:

```
$ sudo apt-get install python3
```

To install Python on **Red Hat** derivatives, use yum:

```
$ sudo yum install python?
```

To install Python on **SUSE** derivatives, use zypper:

```
$ sudo zypper install python3
```

Mac OS X

A version of Python usually ships with OS X but it is likely to be out-of-date. Before installing Python you'll have to install GCC (the GNU Compiler Collection) which can I obtained by downloading Code from https://developer.apple.com/xcode/.

Python is typically installed using a package manager like Homebrew. Instructions for installing Homebrew may be found here: https://brew.sh/ #install.

To install Python on Mac OS X, use Homebrew:

```
$ brew install python3
```

Using the Interpreter

Assuming Python was added to the PATH during installation, the interpreter may be invoked on the command line as follows (assuming version 3.7):

```
$ python3.7
```

To execute a Python script, add the script path as an argument.

```
$ python3.7 my_script.py
```

To execute a Python module, add the -m option following by the module name.

```
$ python3.7 -m my_script
```

To execute a Python command or suite of commands, add the -c option followed by the command (s) surrounded in quotes.

```
$ python 3.7 -c 'print("Hello world"); print("This is Python") '
```

Interactive mode

When the interpreter is invoked without any options or arguments, it is said to be in interactive mode. Interactive mode is useful for testing small snippets of code.

```
$ python3.7
Python 3.7 (default, February 5 2019, 09:00:00)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information
$ >>>
```

The interpreter waits for a command to be input from standard in (usually the keyboard), evaluates said command, writes the result to standard out (usually the screen), and then resumes waiting for input. This cycle is referred to as REPI. (read, evaluate, print, loop).

```
$>>> print ("Hello world")
Hello world
$ >>>
```

NOTE
Where the command is composed of more than one line, pressing the Enter key will result in a secondary prompt (...).

To exit interactive mode, type an end-of-file character (Control + D on UNIX/QS X, Control + Z on Windows) or execute the quit function.

```
$>>> quit()
```

Python Editors

There are many editors available with which to write Python code. Some are written specifically for Python, and some are generic. Python typically ships with an editor named Idle, which, assuming Python was added to the PATH during installation, may be invoked on the command line as follows (assuming version 3.7):

```
$ idle3.7
```

Tabled below are some of the more popular Python-capable editors.

Editor	Description
PvCharm	Full-featured and dedicated for Python; paid and community eds.
Spyder	Optimised for data science; open source
Thonny	Designed for beginners; bundled with Python
Eclipse + PyDev	For those developers already familiar with Eclipse
Sublime Text	Good generic editor; not free but may be used in evaluation mode
Visual Studio Code	Good generic editor; open source

Script Naming

According to PEP (Python Enhancement Proposal) 0008.

> *Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability.*
>
> **NOTE**
> A module is just a Python source code file that exposes classes/ functions/ variables for use in other scripts/modules.

The following example script names adhere to this convention:

- my_script.py
- myscript.py

Comments

In a Python source code file, any line or part thereof that is prefixed with the hash symbol (#) is a comment and will be ignored by the interpreter.
According the PEP 0008:

- Comments should be complete sentences
- The first word of a comment should be capitalised (unless it's an identifier)
- Each sentence of a comment should end with a period
- Two spaces should be inserted between comment sentences
- Block comments should be indented to the same level as that of the code.
- Inline comments should be separated from the code by at least two spaces

Statements

A single-line statement is terminated with a newline character.

```
name = "David"
age = 30
```

Two or more statements may be written on one line using the semicolon as a delimiter.

```
Name = "David";  age = 30
```

Names

Most of the variables, functions, and classes you create will be named.

```
my_number = 1

def my_function():
    pass

class my_class:
    pass
```

In this case, my_number, my_function, and my_class are all names. The names that yo choose are subject to rules and conventions.

The rules

- A name must start with a letter or an underscore
- The remainder of the name may be composed of letters, digits, and underscores
- Names are case sensitive

Console IO

To write simple Python scripts that do something useful, we need to be able to read input from standard in (the keyboard) and write output to standard out (the screen). To do this, we exploit two built-in functions - input and print.

The input function

The built-in input function is passed an optional prompt and returns the value input by the user (as a string).

```
$>>> value = input ('Enter a value: ')
$ Enter a value: Hello world
$>>> value
$ 'Hello world'
```

The user types Hello world and presses Enter in response to the prompt

NOTE
In this example the lines prefixed with $ >>> represent the code to be evaluated, while the lines prefixed with $ represent the result of the evaluation.

The print function

The built-in print function is passed one or more values to be written to standard out. It does not return anything.

```
$>>> print("Hello world")
$ Hello world
```

When the print function is passed more than one value, each is written to standard out delimited by a space.

```
$>>> print("Hello world", "this is Python")
$ Hello world this is Python
```

The print function might be used to print the value input by the user (or a variant of it).

```
$ >>> value = input ('Enter a value: ')
$ Enter a value: Hello world
$>>> print ("you entered", value)
$ You entered Hello world
```

Script Execution

To execute a Python script, invoke the interpreter with the script path as an argument.

```
$ python3.7 my_script.py
```

To execute a Python script as a module, invoke the interpreter with the -m option following by the module name (excluding the extension).

```
$ python3.7 -m  my_script
```

Variables and Data Types

Variable

In Python, a variable is a named reference/pointer to an object (some data). Unlike some other languages, there are no primitive-type variables in Python. Everything is an object, which means that every variable is a reference-type variable. Consider the following:

```
x = 1
```

The variable x is assigned a reference to the number 1. Or to put it another way, the variable x points to the segment of memory containing the number 1.

> **NOTE**
> The assignment operator (=) dictates that a reference/pointer to the thing on the right is assigned to the variable on the left.

Assigning one variable to another has the effect of copying the reference from the variable on the right to the variable on the left.

```
y = x
```

The variable y is assigned a reference to the number 1, just like x.

Data Types

Every object has a type. Consider the following:

```
name = "David"
email = "david@mail.com"
```

The objects "David" and "david@mail.com" are str (string) type objects. Python has a number of built-in data types that provide for the storage of simple data - numbers, strings of characters, lists etc. You can create your own data types too.

Literal

The term literal is used to refer to a fixed value.

```
name = "David"
age = 30
hobbies = ["leading", "walking", "eating out"]
```

In this case, "David" is a string literal, 30 is an integer literal, and ["reading", "walking", "eating out"] is a list literal. Of course, in Python, each of these literals is also an object. In fact, every literal is an object.

The type function

The built-in type function is used to determine the data type of a literal or the object referenced by a variable.

```
$ >>> type("David")
$ <class 'str'>

$ >>> name = "David"
$ >>> type(name)
$ <class 'str'>
```

Dynamic typing

A variable need not be declared before it is assigned. Indeed, doing so will yield an error:

```
$>>> age
$ NameError: name 'age' is not defined
```

Python is a dynamically-typed language. That means that a variable is assigned a type only when it is assigned a value (at runtime). It also means that the data type of a variable may change over the course of its life.

```
$ >>> y = 1
$ >>> type(x)
$ <class 'int'>

$ >>> x = "one"
$ >>> type(x)
$ <class 'str'>
```

Strong typing

Python is a strongly-typed language. That means that the type of an object cannot change. Consider the following example:

```
$ >>> “Age: “ + 30
$ TypeError: must be str, not int
```

An integer literal cannot be added to a string literal. That is, an integer cannot be coerced into being a string.

NOTE
The difference between dynamic and strong typing is subtle, but important. The type of a variable is dynamic (can change) whilst the type of an object cannot.

Numbers

There are four built-in number types with which we ought to be familiar: bool, int, float, and complex. The standard library includes other number types too: the fractions module and the decimal module provides support for user-defined precision.

bool

A Boolean object is one whose value is either True or False. Since Python version 3, True and False are keywords.

```
$>>> my_bool = True
$ >>> type (my_bool)
$ <class 'bool'>
```

bool is a number type because the values True and False are equivalent to the numbers 0 (zero) and 1 (one) respectively.

```
$ >>> True == 1
$ True
$>>> False == 0
$ True
```

NOTE
The == symbol is used to test for equality; it is the equality operator.

Int

An integer object is one whose value is a whole number.

```
$>>> my_int = 3
$>>> type(my_int)
$ <class 'int'>
```

Note
In Python 2, there are two whole number types: int and long. Values of type int occupy 32 bits of memory, while values of type long can occupy any amount of memory. In Python 3, values of type int can occupy any amount of memory.

Float

A floating point object is one whose value is a fractional number.

```
$ >>> my_float = 19.95
$ >>> type(my_float)
$ <class 'float'>
```

Collections

This section provides an introduction only to the various built-in collection types.
Each collection type is covered in detail later in the course.

str

A string is a collection of characters. In Python, a string literal may be surrounded by either single or double quotes.

```
$ >>> my_string = 'Hello'
$ >>> type (my_string)
$ <class "str'>

$ >>> my_string = "Hello"
$ >>> type (my_string)
$ <class 'str'>
```

List

A list is a collection of values, that is, a collection of anything. In Python, a list literal comprises zero or more comma separated values surrounded by square brackets.

```
$ >>> my_list = [1, 2, 3]
$ >>> type (my_list)
$ <class 'list'>
```

You might code a list of numbers, a list of strings, a list of lists, or a list of variously typed things (although we don't recommend that).

```
odd_numbers = [1, 3, 5, 7, 9]
names = ["David", "Sarah", "Tom", "Jane"]
scores = [
          [3, 1],
          [2, 5]
   ]
```

Each element of a list is indexed, with the first element having an index of zero.
A list element is accessible by its index.

```
$ >>> second_name = names [1]
$ >>> second name
$ ‘Sarah’

$ >>> names[2] = "Thomas"
$ >>> names[2]
$ ‘Thomas’
```

Operators & Expressions

Introduction

Operators don't vary much between modern languages, and Python is no exception.
In this chapter we table the various operators for reference and provide detailed information only where we think it's needed.
For completeness - an operator is a special symbol that can be applied to one or more values, referred to as operands, and that either changes the operand (unary operator) or evaluates to a value (binary operator). An expression is a statement that yields a new value. That value is often assigned to a variable.

```
new_value = num + 1
```

In this case, the + (plus symbol) is the operator, the variable num and the integer literal 1 are the operands, and the part of the statement on the right of the assignment operator is the expression.

Arithmetic Operators

Operator	**Description**
+	Addition
-	Subtraction
*****	Multiplication
/	Division
%	Modulus (returns the remainder)
******	Exponentiation (to the power)
//	Floor division

Division

A division operation always yields a float, even if the operands are of type int.

```
$ >>> 9 / 2
$ 4.5
```

The floor division operator yields an int.

```
$>>> 9 // 2
$ 4
```

Implicit type conversion (promotion)

If an arithmetic expression comprises one int operand and one float operand, then the int value is implicitly converted to a float (promoted).

```
$ >>> 9 + 2.0
$ 11.0
```

Non-numeric arithmetic

Sounds silly doesn't it. But in Python, arithmetic operators can be applied to anything

```
$ >>> "ha" * 3
$ ‘hahaha’

$ >>> [1, 2, 3] + [4, 5, 6]
$ [1, 2, 3, 4, 5, 6]
```

Assignment Operator’s

Operator	Description
=	Assignment
+=	Compound assignment (addition)
-=	Compound assignment (subtraction)
*=	Compound assignment (multiplication)
/=	Compound assignment (division)
%=	Compound assignment (modulus)
**=	Compound assignment (exponentiation)
//=	Compound assignment (floor divition)

Multiple assignment

Many variables may be assigned values on one line. The following..

```
$ >>> x = 1
$ >>> y = 2
$ >>> z = 3
```

...may be rewritten as:

```
$ >>>x, y, z = 1, 2, 3
```

Increment and decrement

If you're coming to Python from another language you may be surprised to learn that* Python does not have increment/ decrement operators. One of the main reasons for this is that, due to the way in which things like iteration is done in Python, increment/ decrement operators are rarely needed.

```
$ >>> x = 1
$ >>> x++
$ SyntaxError: invalid syntax

$ >>> x += 1
$ >>> x
$ 2
```

Comparison operators

Operator	**Description**
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
is	References the same object as
is not	Does not reference the same object as
in	Is a member of
not in	Is not a member of

Testing for equality (==)

Testing for equality seems like a simple thing.

```
$ >>> 2 == 2
$ True
```

but what about

```
$ >>> x == y
```

What are we comparing? The objects referenced by the variables x and y, or the references/ pointers themselves? The answer is that it should be the former, that is, testing for equality should compare the objects referenced by the variables x and y. And indeed this is the case fur all the data types we've discussed thus far - bool, int, float, complex, str, list, tuple, set, and dict, but it may not be the case for custom data types. See the chapter on Objects and Classes for more information.

```
$ >>> "Hello" == "Hello"
$ True
$ >>> [1, 2, 3] == [1, 2, 3]
$ True
$ >>> { 'a' : 1, 'b' : 2} = { 'a' : 1, 'b' : 2}
$ True
$ >>> [1, 2, 3] == [3, 2, 1]
$ False
$ >>> {'a' : 1, 'b' : 2} == ('b' : 2, 'a' : 1}
$ True
```

The last two of the examples above may seem a little odd. The two lists are not equivalent because lists are ordered, and so the elements at each index are compared in turn. The two dictionaries are equivalent because dictionaries are unordered, and so the test only serves to determine if the content of each dictionary is the same.

NOTE
Further to the points made above, tuples are ordered, like lists, so...

```
$ >>> (1, 2, 3) == (3, 2, 1)
$ False
```

But sets are unordered, so..

```
$ >>> {1, 2, 3} == {3, 2, 1}
$ True
```

Testing for identity (is)

The is operator should not be confused with the equality operator. The is operator is used to compare references/ pointers. That is, do these two variables reference/ point to the same object or not?

```
$ >>> x = 1
$ >>> y = 1
$ >>> x == y
$ True
$ >>> x is y
$ True
```

The integers referenced by x and y are equivalent and, given that there can only ever be one integer literal with a value of 1 (one), the two variables reference the same object.

Testing for membership (in)

The in operator is used to test for the presence of a value in a collection.

```
$ >>> ‘e’ in ‘Hello'
$ True
$ >>> 2 in [1; 2, 3]
$ True
```

In the same way that the conversion of a dictionary to a list results in a list composed of the dictionary keys, and not its values, testing a value for membership in a dictionary means testing that the value is present in the dictionary's set of keys.

```
$ >>> ‘b’ in {'a' : 1, 'b' : 2}
$ True
$ >>> 1 in {'a' : 1, 'b' : 2}
$ False
```

Logical Operators

Operator	Description
and	Boolean AND: yields True if each expression is True
or	Boolean OR: yields True if either expression is True
not	Boolean NOT: inverts the value

Comparison operator chaining

Python allows for the chaining of comparison operators. Consider the following:

```
$>>> age >= 18 and age < 35
```

This is a pretty typical logical expression, but can be rewritten as.

```
$ >>> 18 <= age <= 35
```

Logical operators with non-Boolean values

We know that any value may be converted to a Boolean, and so we might assume that logical operators can be applied to any value.

```
$ >>> 1 and 0
$ 0
```

You might have expected the result to be False, given that 0 (zero) is False when converted to a Boolean, and 1 (one) is likewise True. But instead the result is 0 (zero).
The following rules are applied:

- A and B: if A is false, then the result is A, else the result is B
- A or B if A if true, then the result is A, else the result is B

To ensure the result of a compound expression is a Boolean, make sure that each constituent expression is a Boolean.

```
$ >>> bool (1) and bool (0)
$ False
```

Operator Precedence

The full table of operator precedence is provided below for reference, but most of the time it will suffice to know that any expression enclosed by the grouping operator (parentheses) takes precedence over everything else.

```
$ >>> 3 + 2 * 4 # multiplication takes precedence over addition
$ 11
$ >>> (3 + 2) * 4
$ 20
```

Operator Precedence (highest to lowest)	
(expressions…), [expressions..], {expressions..},	Grouping, tuple, list, set, dictionary
x(args...), x[index: index], x[index], x.attribute	Function call, slicing, subscription, attribute, reference
**	Eponentiation
+x, -x, ~x	Positive, negative, bitwise not
*, /, %, //	Multiplication, division, modulus, floor division
+. -	Addition, subtraction
<<, >>	Shift left, shift right
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
<, <=, ==, !=, =>, >, is, is not, in, not in	Less than, less than or equal to, equal to, not equal to, greater than or equal to, greater than, is, is not, in, not in
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
if else	Conditional expression
lamda	Lambda expression

Conditions and Loops

Introduction

Conditional and loop statements typically comprise one or more blocks of code.
As described in chapter 2, a Python block must be indented. The start of a block is indicated by the presence of a colon (;), and each statement within the block must be indented using the same amount of whitespace (the convention is to use four spaces). The block ends when the indenting ends.

Conditional Statements

Conditional statements are somewhat easier to learn in Python than they are in other languages, because there is only one type - the if statement.

The if statement

An if statement takes the following form:

```
if <expression>:
        <statement (s)>
```

If the expression evaluates to True, the statement(s) in the block that follow are executed.

```
$ >>> x = 2
$ >>> if x > 1:
$ ...     print("x is greater than 1")
$ ...
$ x is greater than 1
```

The expression need not be a Boolean expression. As any value can be converted to a Boolean, any expression that yields a value may be used.

```
$ >>> x = 2
$ >>> if x:
$ ... print("bool(x) is True")
$ ...
$ bool (x) is True
```

Note
You might recall from chapter 3 that any nonzero, non-empty value evaluates to True when passed to the built-in bool function.

To demonstrate the importance of consistent indenting, consider the following:

```
$ >>> x = 2
$ >>> if x > 1:
$ ... print("Hello")
$ ... print("x is greater than 1")
$ ...
$ IndentationError: unindent does not match any outer indentation level
```

We expect both print statements to be executed. The problem is that the first print statement is indented using four spaces, while the second print statement is indented using three spaces.

Recall that the block ends when the indenting ends. There is no special symbol to indicate the end of the block.

```
$ >>> x = 2
$ >>> if x > 1:
$ ... print("x is greater than 1")
$ ...
$ >>> print ("Outside the block/the if statement")
```

NOTE
The example above can't be executed in one go in interactive mode. By its very nature a REPL shell can only evaluate one statement at a time (unless that statement involves the invoking of a function in which multiple statements are executed).

The else and elif clauses

A two.branch conditional statement is achieved with the addition of the else clause to an if statement. A two branch if statement takes the following form:

```
if <expression>:
        <statement (s)>
else:
        <statement (s)>
```

If the expression evaluates to True, the statement(s) in the first block are executed. If the expression evaluates to False, the first block is skipped and the statement(s) in the second block are executed.

```
$ >>> x = 2
$ >>> if x > 2:
$ ...       print("x is greater than 2")
$ ... else:
$ ...       print("x is not greater than 2")
$ ...
$ x is not greater than 2
```

A three or more branch conditional statement is achieved with the addition of the elif clause to an if else statement. A three branch if statement takes the following form:

```
if <expression>:
        <statement (s)>
elif    <expression>:
        <statement (s)>
else:
        <statement (s)>
```

Note that two expressions are required in this case.

```
$ >>> x = 2
$ >>> if x > 2:
$ ...       print("x is greater than 2")
$ ... elif x < 2:
$ ...       print("x is less than 2")
$ ... else:
$ ...       print("x is equal to 2")
$ ...
$ x is equal to 2
```

Only the first branch associated with an expression that evaluates to True is executed. That is, following the execution of a branch, the interpreter steps out of the if statement without evaluating any more expressions.

```
$ >>> X = 2
$ >>> if x > 1:
$ ...       print("x is greater than 1")
$ ... elif x != 1:
$ ...       print("x is not equal to 1")
$ ... else:
$ ...       print("None of the expressions evaluated to True")
$ ...
$  x is greater than 1
```

In this case, only the first branch is executed despite the second expression also evaluating to True.

One-liners

One branch if statements may be written on one line...

```
$ >>> if x > 1: print("x is greater than 1")
```

…even when the block comprises more than one statement:

```
$ >>> if x > 1: print("Hello"); print("x is greater than 1")
```

Conditional expression

A conditional expression is one in which either of two values is assigned to a variable depending on the result of a boolean expression. It takes the following form:

```
$ >>> x = <valuel> if <expression> else <value2>
```

Consider the following example:

```
$ >>> action = "Get up" if hour >= 7 else "Go back to sleep"
```

NOTE
For those of you familiar with one or more other languages, this may appear similar to what you know as the ternary operator. It is, indeed, the Python equivalent.

Iterative Statements

There are two types of iterative statement in Python - while and for.

The while statement

A while statement takes the following form:

```
while   <expression>:
        <<statement (s)>
```

If the expression evaluates to True, the statement (s) in the block that follow are executed. The cycle is repeated until the expression evaluates to False.

```
$ >>> x = 1
$ >>> while x <= 3:
$...     print(x)
$ ...    x+=1
$ ...
$ 1
$ 2
$ 3
```

The for statement

If you've written code in another language, you'll almost certainly be familiar with a for loop of the form:

```
for (<init_statement>; <expression>; <update_statement>) {
        ...
 }
```

There is no such statement in Python. Python for loops are used exclusively to iterate over collections (or iterables, to be precise).

A for statement takes the following form:

```
for <element> in <collection>:
        <statement (s)>
```

The block is executed once for each element in the collection.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> for name in names:
$ ...            print (name)
$ ...
$ David
$ Sarah
$ Tom
$ Jane
```

The range function

What if you want to perform some task n times but you don't have a collection? You could use a while loop, but doing so is considerably slower than a for loop that uses the built-in range function.

The range function returns a range of numbers and can be invoked in a variety of ways.

```
$ >>> my_range = range (5)    # start = 0; stop = 5; step = 1
$ >>> list (my_range)
$ [0, 1, 2, 3, 4]

$ >>> my_range = range (1, 5)           # start = 1; stop = 5; step = 1
$ >>> list (my_range)
$ [1, 2, 3, 4]

$ >>> my_range = range(1, 10, 2)       # start = 1; stop = 10; step = 2
$ >>> list (my_range)
$ [1, 3, 5, 7, 9]
```

You'll have noticed that we're converting the returned value to a list. That's because the range function returns a range type object. You needn't convert it to a list when using it in a for loop though.

```
$>>> for number in range (5):
$ ...      print (number)
$ ...
$ 0
$ 1
$ 2
$ 3
$ 4
```

Break

Like other languages, the break statement is used to terminate a loop prematurely.

```
$ >>> while(True) :
$ ...     option = input ("More? (y/n): “)
$ ...     if option == ‘n’:
$ ...             print("Bye")
$ ...             break
$ ...     print("Some more...")
$ ...
$ More? (y/n): y
$ Some more…
$ More? (y/n): y
$ Some more….
$ More? (y/n): n
$ Bye
```

In this case, the infinite loop is terminated if the user enters the letter n at the prompt.

Continue

Also like other languages, the continue statement is used to jump to the next iteration, skipping one or more statements in the process.

```
$ >>> for number in range(5):
$ ...             if number == 3:
$ ...                     continue
$ ...             print (number)
$ ...
$ 1
$ 2
$ 4
$ 5
```

The optional else clause

The else clause may be added to an iterative statement where break is also present. The else block will only be executed if the loop exits cleanly, that is, without terminating prematurely.

```
$ >>> for line in file:
$ ...             if line. startswith("ERROR"):
$ ...                     print("The file has one or more errors")
$ ...                     break
$ ...             print(line)
$ ...     else:
$ ...             print ("File processed with no errors")
$ ...
```

The Challenges

First Factorial

Have the function take the number parameter being passed and return the factorial of it. For example: if num = 4, then your program should return (4 * 3 * 2 * 1) = 24.

For the test cases, the range will be between 1 and 18 and the input will always be an integer

Time Convert

Have the function take the number parameter being passed and return the number of hours and minutes the parameter converts to (ie. if num = 63 then the output should be 1:3). Separate the number of hours and minutes with a colon.

For the test cases, the numbers used will be 74 and 118

Consonant Count

Have the function take the string parameter being passed and return the number of consonants the string contains (ie. "All cows eat grass" would return 5).

For the test cases the two strings will be 'Today is a really hot day but will cool down later' and also 'Rabbits bounce whilst dogs bark'

H Distance (Hamming Disatnce)

Have the function take the array of strings, which will only contain two strings of equal length and return the number of characters at each position that are different between them. For example: ["house", "hours"] then your program should return 2. The string will always be of equal length and will only contain lowercase characters from the alphabet and numbers.

For the test cases, the first set of numbers will be [100101001 and 110110011] and [11001101 and 10110110]

Roman Numbers

Have the function take in an integer value of up to 4000 and then convert that number into the correct Roman numeral equivalent. For example if the function takes 1943 then the return should be MCMXLIII

For the test cases, the two numbers will be 3612 and 423

All code should be written and submitted regardless of whether the solution passes the test criteria or not, as points will be awarded for the approach as well as a working solution