A brief introduction to Python

*By Nicola Romanò - last updated on **27 August 2021***

This document is a brief introduction to the Python programming language. It is by no means complete, but it should be sufficient to kick-start your Python programming skills.

0. First steps

This guide assumes you have installed Python 3.7 or higher. You can download Python from the Python website.

You can use the Python interpreter to test your code. To do so, open a command prompt (Windows) or terminal (Linux/Mac) and type: `python`. You will see a prompt (`>>>`), where you can enter your first Python command.

Try writing the following and press `Enter`:

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

The result should look like this:

```
Hello, world!
```

Ok, not the most exciting thing, but it's a start!

A better way to code in Python is to write a **script**. A script is a file containing Python commands, generally saved with the `.py` extension.

There are many different editors that can be used to write Python code. Some, such as Spyder, are more lightweight, while others such as PyCharm or VSCode are more powerful (but might be more confusing at first). It's up to you which editor you use, it's a matter of personal preference; you can use any text editor, really.

Once you have a script, you can run it by typing its name in the command prompt (Windows) or terminal (Linux/Mac). For example, if you have a script called `hello.py`, you can run it by typing:

```
python hello.py
```

Some editors also have a "Run" button or function that will run the script, without having to open the command prompt/terminal.

I will use the Python command line for this guide, but feel free to write the commands in a script and run them from there.

1. Variables

Like in most programming languages, Python uses variables to store values. For example, if you want to store the number `42` in a variable called `x` you can simply write

```
>>> x = 42
```

You can use the basic arithmetical operators to perform calculations on variables. For example, if you want to add `5` to `x`, you can write

```
>>> x = x + 5
```

You can use `print` to print the value of a variable. For example, if you want to print the value of `x`, you can write

```
>>> print(x)

47
```

You can perform arithmetic operations on variables. For example:

```
>>> x = 35
>>> y = x + 5
>>> print(x, y)

35 40
```

Note how I passed multiple values to `print` by separating them with a comma.

You can store decimal numbers (called *floating point* numbers) in variables. For example, if you want to store the number `3.14` in a variable called `pi`, you can write

```
>>> pi = 3.14
```

Similarly, you can store strings (i.e. series of characters) in variables. For example, if you want to store the string `"Hello, world!"` in a variable called `hello`, you can write

```
>>> hello = "Hello, world!"
```

Strings can be enclosed in single or double quotes. The command above is equivalent to

```
>>> hello = 'Hello, world!'
```

If you are dealing with long stretches of text on multiple lines you can use triple quotes to enclose the text in a single string. Just start typing the string and you can press `Enter` to add a new line. The prompt will changes to `...`, and you can type the rest of the string. For example:

```
>>> hello = """
... Hello, world!
... This is a very long string.
... """
```

Python 3.6 has introduced a new syntax, called the *f-string* syntax. This is extremely useful when mixing strings and variables. For example, try to run the following:

```
>>> a = 15
>>> b = a ** 2 # Square of a
>>> c = a ** 3 # Cube of a
>>> print(f"The square of {a} is {b} and the cube is {c}.")

The square of 15 is 225 and the cube is 3375.
```

There are a few new concepts in the code above:

- `f"..."` indicates a string with variables, or **f-string**. Variable names are enclosed in curly braces `{}`.
- `#` indicates a comment. The Python interpreter ignores comments.
- I used `**` to exponentiate.

2. Basic data types

In part 1 we learned that Python uses variables to store values. In this part we will learn about the different data types that Python supports.

You have already seen a few examples of different data types. Listed below are the most important ones, with some examples:

- Integers: `42`, `0`, `1`, `-1`
- Floating point numbers: `3.14`, `0.0`, `-1.0`, `-15.45563`
- Strings: `"Hello, world!"`, `'Python is cool!'`
- Booleans: `True`, `False`

The Python interpreter automatically understands the data types of the variables you assign to them. For example, if you assign the number `42` to a variable called `x`, the interpreter will understand that `x` is an integer.

If you try to combine different variables, Python will automatically decide how to combine them. For instance:

```
>>> a = 15    # Integer
>>> b = 13.2  # Float
>>> c = a + b
>>> print(c)

28.2

>>> print(f"{a} is a {type(a)}, {b} is a {type(b)} and {c} is a
{type(c)}.")

15 is a <class 'int'>, 13.2 is a <class 'float'>, and 28.2 is a <class
'float'>
```

c has been assigned the value 28.2 and therefore has been automatically turned into a float. Note how you can use `type(variable)` to check out the type of any variable.

What happens when you try to add a string and an integer?

3. Advanced data types - lists, tuples and dictionaries

Lists

Sometimes you need to store more than one value in a variable. For example, if you want to store the coordinates of a point in a 2D space you can write

```
>>> coordinates = [3.5, 6.4]
```

This is called a **list**.

A list can contain different data types.

```
>>> my_list = [3.5, 6.4, "Hello, world!", 8, 3.14]
```

You can access elements of a list using the `[]` operator. Note that in Python the index starts at **0**, not 1.

So, if you want to print the first element of the list, you can write

```
>>> print(my_list[0])

3.5
```

You can also access the last element of a list using the `-1` index (or the one before using `-2` and so on).

```
>>> print(my_list[-1])

3.14
```

You can use the `:` operator to define a slice of a list. The full syntax is `[start:stop:step]`. The step is 1 by default.

```
>>> print(my_list[1:3]) # Prints the elements at index 1 and 2
>>> print(my_list[0:3]) # Prints the elements at index 0, 1 and 2
>>> print(my_list[:3])  # Equivalent to the above
>>> print(my_list[1:])  # Prints the elements at index 1 and after
>>> print(my_list[:-3]) # Prints all the elements except the last 3
>>> print(my_list[::2]) # Prints every second element
```

Experiment with the `:` operator and try to print different parts of a list

You can append elements to a list using the `append` method. For example:

```
>>> my_list = [1, 2, 3, 4]
>>> my_list.append(5)
>>> print(my_list)
[1, 2, 3, 4, 5]
```

Similarly, you can remove elements from a list using the `pop` method. For example:

```
>>> my_list = [1, 2, 3, 4]
>>> my_list.pop() # By default, it removes the last element
4
>>> print(my_list)
[1, 2, 3]
>>> my_list.pop(0) # Removes the element at index 0
1
>>> print(my_list)
[2, 3]
```

Tuples

Tuples are similar to lists, but they are **immutable**. That means that once you create a tuple, you cannot change its elements.

```
>>> my_tuple = (3.5, 6.4, "Hello, world!", 2)
>>> print(my_tuple[0])
3.5
>>> my_tuple[0] = 5 # This will raise an error
```

Dictionaries

Dictionaries are used to map keys to values. For example:

```
>>> favourite_colours = {"Alice": "red", "Bob": "blue", "Carol": "green"}
>>> print(favourite_colours["Bob"])
blue
```

Dictionaries are **mutable** so you can do:

```
>>> favourite_colours["Bob"] = "yellow"
>>> favourite_colours["Dave"] = "orange"
>>> print(favourite_colours)
{'Alice': 'red', 'Bob': 'yellow', 'Carol': 'green', 'Dave': 'orange'}
```

However, contrary to lists and tuples, dictionaries are **unordered**, so you cannot access the elements of a dictionary by index.

```
>>> print(favourite_colours[1]) # This will raise an error
```

4. Conditional statements

Python allows you to write code that is executed depending on the value of a variable. For example, you can write:

```
>>> x = 5
>>> if x > 0:
...     print("x is positive")
... elif x < 0:
...     print("x is negative")
... else:
...     print("x is zero")
```

```
x is positive
```

Note that the `if` statement is followed by a colon `:` and that the `elif` and `else` statement are optional.

Importantly, Python is sensitive to indentation (i.e. you need to have some space before the statement following the `if`). For example, the following code will not work:

```
>>> if x > 0:
... print("x is positive")
```

You can nest conditional statements, for example:

```
>>> x = 5
>>> y = 8
>>> if x > 0:
...     if y > 0:
...         print("Both x and y are positive")
...         print("Something else...")
...         print("...indented at the same level")
...     print("This is less indented, so it is printed")
...     print("independently from the value of y")

Both x and y are positive
Something else...
...indented at the same level
This is less indented, so it is printed
independently from the value of y
```

Experiment changing the value of x and y and see what happens.

5. Loops and iteration

Python makes it super easy to do loops. For example, you can write a loop that prints the numbers from 1 to 5:

```
>>> for i in range(1, 6):
...     print(i)

1
2
3
4
5
```

The `range` function returns a list of integers from `start` (inclusive) to `stop` (excluded). You can also specify a step size. For example:

```
>>> for i in range(1, 10, 2):
...     print(i)
1
3
5
7
9
```

If you only specify a single value, Python will start at 0. For example:

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

You can loop over a list using the `for` loop. For example:

```
>>> my_list = [1, 2, 3, 4, 5]
>>> for i in my_list:
...     print(i**2)
1
4
9
16
25
```

You can even loop through different lists at the same time. For example:

```
>>> my_list = [1, 2, 3, 4, 5]
>>> other_list = ["a", "b", "c", "d", "e"]
>>> for i, c in zip(my_list, other_list):
...     print(i, c)

1 a
2 b
3 c
4 d
5 e
```

This works because the `zip` function returns a list of tuples, where each tuple contains the elements of the lists at the same index.

```
>>> print(zip(my_list, other_list))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

6. Comprehensions

Comprehensions are a very "pythonic" way of creating lists. For example, you can create a list of squares of the numbers 1 to 5 using the following code:

```
>>> squares = [i**2 for i in range(1, 6)]
>>> print(squares)
[1, 4, 9, 16, 25]
```

You can also have conditions in the comprehension:

```
>>> # Create a list of squares of the numbers
>>> # 1 to 10, but only if the number is even
>>> squares = [i**2 for i in range(1, 10) if i % 2 == 0]
>>> print(squares)
[4, 16, 36, 64]
```

Comprehensions support multiple variables. **What is the output of the following code?**

```
>>> odd = [1, 3, 5, 7, 9]
>>> even = [2, 4, 6, 8, 10]
>>> print([(o, e) for o in odd for e in even])
```

While you can obtain the same result using a loop, the loop is more verbose and comprehensions are generally considered to be more concise and elegant, at least for simple situations.

7. Functions

Functions are defined using the `def` keyword. Functions can accept arguments and return values. For example:

```
>>> def square(x):
...     return x**2

>>> square(5)
```

```
25
```

Other functions might not have any arguments:

```
>>> def say_hello():
...     print("Hello!")

>>> say_hello()

Hello!
```

You can return multiple values from a function:

```
>>> def get_min_max(numbers):
...     return min(numbers), max(numbers)

>>> min, max = get_min_max([1, 2, 3, 4, 5])
>>> print(f"The minimum value is: {min}, the maximum is: {max}")

The minimum value is: 1, the maximum is: 5
```

It is good practice to include docstrings in your functions. This is a string that describes what the function does. For example:

```
>>> def square(x):
...     """Return the square of x.
...     Parameters
...     x : int or float - the number to square
...     Returns
...     int or float - the square of x
...     """
...     return x**2
```

7. Importing external libraries

Python has a large ecosystem of libraries that can be used in your code. You can import a library by using the `import` keyword. For example to import the `math` library you can use:

```
>>> import math
```

All of the functions and variables in the `math` library are now available to you. For example:

```
>>> print(math.sqrt(16)) # Square root of 16
4.0
>>> print(math.pi)
3.141592653589793
```

Sometimes the name of the library is long or not descriptive enough. In this case you can use an alias to refer to the library:

```
>>> import math as m
>>> print(m.sqrt(16))
4.0
```

Certain libraries are very complex and contain many functions. If you just need one or two of the functions, you can import just those. You can then use the function names without the library name.

```
>>> from math import sqrt, pi
>>> print(sqrt(16))
4.0
>>> print(pi)
3.141592653589793
```

8. Conclusion

You made it to the end! Well done! This was just a short introduction to Python, and I hope it has been useful to get you started.

There are many more topics to cover and the Python documentation is very comprehensive. There are also many documents on the web that can help you learn Python.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.