A brief introduction to Python - part 1, variables, data types and loops

*By Nicola Romanò - last updated on **25 July 2024***

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; at the moment of writing, the latest version is 3.12.4.

Details for the installation of Python vary depending on the operating system you are using, but the Python website provides detailed instructions for each operating system.

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. Note that some systems might require you to type `python3` instead of `python`.

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.

Finally, a very common way to write Python code is to use a **Jupyter notebook**. Jupyter notebooks are a great way to write code, as they allow you to write code in cells and run them individually. They are also great for writing reports, as you can include text and images in the same document. This is similar to what you previously saw using RMakrdown in R, however it is more interactive. We will use Jupyter notebooks in the workshops for the course, see the Introduction to Workshops document for more information.

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:

```
print(x)
```

You can perform arithmetic operations on variables. For example:

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

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.
    """
```

Exercise 1

1. Create a variable called `a` and assign it the value `15`.
2. Create a variable called `b` and assign it the value `a` squared (hint: use the `**` operator to raise a number to a power).
3. Finally divide `b` by `a` and store the result in a variable called `c`.

F-strings

Python 3.6 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}.")
```

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.

f-strings are extremely powerful and can be used to format strings in many different ways. For example, you can specify the number of decimal places to print for a floating point number:

```
pi = 3.141592653589793
print(f"The value of pi is {pi:.2f}")
```

In the code above, `:.2f` specifies that the number should be printed with 2 decimal places.

Exercise 2

1. Try running the following:

   ```
   a = 154
   b = 3
   print(f"{a:03d}, {b:03d}")
   ```

   What is the output? What do you think the `:.03d` specifier do?

2. Now try running the following:

   ```
   print(f"{a=}, {b=}")
   ```

What happens? What do you think the `=` specifier does?

---

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)}.")
```

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

Exercise 3

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, like in some other languages.

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

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

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])
```

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
```

Exercise 4

1. Create a list called `my_list` containing the numbers from 1 to 10.
2. Print the first 5 elements of the list.
3. Print the last 5 elements of the list.
4. Print the elements at index 2, 4, 6 and 8.
5. Print the elements at index 2, 5 and 8.

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)
```

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
print(my_list)

my_list.pop(0) # Removes the element at index 0
print(my_list)
```

Finally, you can modify elements of a list by assigning a new value to the element. For example:

```
my_list = [1, 2, 3, 4]
my_list[0] = 5
print(my_list)
```

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])
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", "Chan": "green", "Wei":
"green"}

print(favourite_colours["Bob"])
```

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

```
favourite_colours["Bob"] = "yellow"
favourite_colours["Wei"] = "orange"
print(favourite_colours)
```

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
```

Exercise 5

1. Create a dictionary called `favourite_numbers` with the following keys and values:
    - "Alice" -> 42
    - "Bob" -> 13
    - "Chan" -> 7
    - "Wei" -> 3.14
2. Print the value associated with the key "Bob".
3. Change the value associated with the key "Chan" to 15.
4. Add a new key-value pair to the dictionary: "Amina" -> 0.

---

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")
```

Note that each "branch" of 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 the same number of spaces before the statement following the `if`). For example, the following code will not work:

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

Similarly, the following code will not work:

```
if x > 15:
    x = x + 5
        print("x is greater than 15")
```

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")
```

Exercise 6

1. Experiment changing the value of x and y and see what happens.
2. Write a conditional statement that prints "x is even" if x is even and "x is odd" if x is odd. (Hint: use the modulo operator `%` to find the remainder of the division of x by 2).

---

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)
```

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)
```

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

```
for i in range(5):
    print(i)
```

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

```
my_list = [1, 2, 3, 4, 5]
for i in my_list:
    print(i)
```

Exercise 7

1. Write a loop that prints the squares of the numbers from 1 to 10.
2. Modify the loop to print the squares of the numbers from 1 to 10, but only if the number is even.

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)
```

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

You can check this by running the following code:

```
print(zip(my_list, other_list))
```

6. 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, for example the beginner's guide is also a good place to start.

Adapt — remix, transform, and build upon the material
for any purpose, even commercially.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

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