01_python_intro

February 20, 2019

1 Introduction to Python

Guillaume Lemaitre

1.1 Basic usage

1.1.1 An interprated language

Python is an interpreted language as your are used to with Matlab. Unlike C++, whatever code your are writing is actually directly executed without the need to compile it first.

```
In [ ]: print('Hello world')
```

The previous cell call the funcion `print` which will return the parameter which pass to it. This function is directly evaluated by the Python interpreter without the need of an extra step.

1.1.2 An untyped language

You surely programmed in Matlab or C++ in the past. You already notice that C++ is a typed language, meaning that you are required to declare the type of a variable before to assign a value. This info will be used by the C++ compiler to create the executable.

```
In [ ]: // This is an example of C++ declaration.
        // Note that it will fail during the execution of the cell.
        // We are programming in Python!!!
        int a = 10;
```

Python is an untype language similar to Matlab. Then, there is no need to specify the type of a variable. Python will infer the most appropriate type when running the instruction.

```
In [ ]: x = 2
```

```
In [ ]: type(x)
```

We can first try to check which built-in types Python offers.

```
In [ ]: x = 2
        type(x)
```

```
In [ ]: x = 2.0
        type(x)

In [ ]: x = 'two'
        type(x)

In [ ]: x = True
        type(x)

In [ ]: x = False
        type(x)
```

`True` and `False` are booleans. In addition, these types can be obtained when making some comparison.

```
In [ ]: x = (3 < 4)
        type(x)

In [ ]: x
```

1.1.3 Python as a calculator

Python provides some built-in operators as in other languages.

```
In [ ]: 2 * 3

In [ ]: 2 / 3

In [ ]: 2 + 3

In [ ]: 2 - 3
```

As previously mentioned, Python will infer the most appropriate data type when doing the operation.

```
In [ ]: type(2 * 3)

In [ ]: type(2 * 3.0)

In [ ]: type(2 / 3)
```

Other useful operators are availble and differ from other languages.

```
In [ ]: 3 % 2

In [ ]: 3 // 2

In [ ]: 3 ** 2
```

1.1.4 Python to make some logic operations

Python provides some common logic oprators `and`, `or`, `not`.

```
In [ ]: True or False
```

Let's look at the Carnaugh table of the `and` operator.

```
In [ ]: for x, y in [(False, False), (False, True),
                    (True, False), (True, True)]:
            print(
                '{} and {} => {}'
                .format(x, y, x and y)
            )
```

```
In [ ]: True and False
```

Let's look at the Carnaugh table of the `or` operator.

```
In [ ]: for x, y in [(False, False), (False, True),
                    (True, False), (True, True)]:
            print(
                '{} and {} => {}'
                .format(x, y, x or y)
            )
```

```
In [ ]: not False
```

Be aware that `not` with other type then boolean.
An empty string `''`, `0`, `False`, will be interpretated as `False` when doing some boolean operation. We will see later what are lists, but an empty list `[]` will also be interpretated as `False`.

```
In [ ]: not ''
```

```
In [ ]: not 0
```

```
In [ ]: not []
```

In the same manner, non-zero numbers, non-empty list or string will be interpreted as `True` in logical operations.

```
In [ ]: not 'xxx'
```

```
In [ ]: not [1, 2, 3]
```

```
In [ ]: not 50
```

1.2 The standard library

1.2.1 The example of the `math` module

Up to now, we saw that Python allows to make some simple operation. What if you want to make some advance operations, e.g. compute a cosine.

```
In [ ]: cos(2 * pi)
```

Unlike Matlab, these functionalities are organised into differnt **modules** from which you have to first import them before to use them.

```
In [ ]: import math
```

```
In [ ]: math.cos(2 * math.pi)
```

The main question is how to we find out which module to use and which function to use. The answer is the Python documentation:

- The Python Language Reference: http://docs.python.org/3/reference/index.html
- The Python Standard Library: http://docs.python.org/3/library/

Never try to reinvente the wheel by coding your own sorting algorithm (apart of of didactic reason). Most of what you need are already efficiently implemented. If you don't know where to search in the Python documentation, Google it, Bing it, Yahoo it (this will not work).

In matlab, you are used to have the function in the main namespace. You can have something similar in Python.

```
In [ ]: from math import cos, pi

        cos(2 * pi)
```

You could even do something which is actually a bad idea by importing all available function from the module.

```
In [ ]: from math import *

        tanh(10)
```

This is a bad idea since you can have different module implementing the same function with an identical name. Then, there is no way to know which implementation is used.

```
In [ ]: type(cos)
```

```
In [ ]: from numpy import cos

        cos(2 * pi)
```

```
In [ ]: type(cos)
```

```
In [ ]: type(math.cos)
```

So never use `from xxx import *`, please!!!

Python allows to use `alias` during import to avoid name collision.

```
In [ ]: from math import cos as std_cos
```

```
In [ ]: from numpy import cos as np_cos
```

```
In [ ]: type(std_cos)
```

```
In [ ]: type(np_cos)
```

What if you need to find the documentation and that Google is broken or you simply don't have internet. You can use the `help` function.

```
In [ ]: help(math)
```

This command will just give you the same documentation than the one you have on internet. The only issue is that it could be less readable. If you are using `ipython` or `jupyter notebook`, you can use the `?` or `??` magic functions.

```
In [ ]: math.log?
```

```
In [ ]: math.log??
```

1.2.2 Exercise:

- Write a small code computing the next power of 2 bigger than `n`. Let's `n` to be successively `7`, `13`, and `23`. You don't know yet what are `for` and `if` statement. Use only the `math` module.

```
In [ ]:
```

1.2.3 Other modules which are in the standard library

There is more than the `math` module. You can interact with the system, make regular expression, etc: `os`, `sys`, `math`, `shutil`, `re`, etc.

Refer to https://docs.python.org/3/library/ for a full list of the available tools.

1.3 Containers: strings, lists, tuples, set (let's skip it), dictionary

1.3.1 Strings

We already introduce the string but we give an example again.

```
In [ ]: s = 'Hello world!'
```

```
In [ ]: s
```

```
In [ ]: type(s)
```

A string can be seen as a table of characters. Therefore, we can actually get an element from the string. Let's take the first element.

```
In [ ]: s[0]
```

As in some other languages, the indexing start at 0 in Python. Unlike other language, you can easily iterate backward using negative indexing.

```
In [ ]: s[-1]
```

`slice` function If you come from Matlab you already are aware of the slicing function, e.g. `start:end:step`. Let see the full story how does it works in Python.

The idea of slicing is to take a part of the data, with a regular structure. This structure is defined by: (i) the start of the slice (the starting index), (ii) the end of the slice (the ending index), and (iii) the step to take to go from the start to the end. In Python, the function used is called `slice`.

```
In [ ]: help(slice)
```

So I can select a sub-string using this slice.

```
In [ ]: my_slice = slice(2, 7, 2)
        s[my_slice]
```

What if I don't want to mention the `step`. Then, you can they that the step should be `None`.

```
In [ ]: my_slice = slice(2, 7, None)
        s[my_slice]
```

Similar thing for the `start` or `end`.

```
In [ ]: my_slice = slice(None, 7, None)
        s[my_slice]
```

However, this syntax is a bit long and we can use the well-known `[start:end:step]` instead.

```
In [ ]: s[2:7:2]
```

Similarly, we can use `None`.

```
In [ ]: s[None:7:None]
```

Since `None` mean nothing, we can even remove it.

```
In [ ]: s[:7:]
```

And if the last `:` are followed by nothing, we can even skip them.

```
In [ ]: s[:7]
```

Now, you know why the slice has this syntax.
Be aware: Be aware that the `stop` index is not including within your data which sliced.

```
In [ ]: s[:2]
```

The third character (index 2) is discarded. Why so? Because:

```
In [ ]: start = 0
        end = 2

        assert (end - start) == len(s[start:end])
```

String manipulation We already saw that we can easily print anything using the `print` function.

```
In [ ]: print(10)
```

This `print` function can even take care about converting into the string format some variables or values.

```
In [ ]: print("str", 10, 2.0)
```

Sometimes, we are interested to add the value of a variable in a string. There is several way to do that. Let's start with the old fashion way.

```
In [ ]: s = "val1 = %.2f, val2 = %d" % (3.1415, 1.5)
        s
```

```
In [ ]: s = "the number %s is equal to %s"
        print(s % ("pi", math.pi))
        print(s % ("e", math.exp(1.)))
```

But more recently, there is the `format` function to do such thing.

```
In [ ]: s = "Pi is equal to {} while e is equal to {}".format(math.pi, math.e)
        s
```

And in the future, you will use the format string.

```
In [ ]: s = f'Pi is equal to {math.pi} while e is equal to {math.e}'
        s
```

A previously mentioned, string is a container. Thus, it has some specific functions associated with it.

```
In [ ]: print("str1" + "str2" + "str2")
```

```
In [ ]: print("str1" * 3)
```

In addition, a string has is own methods. You can access them using the auto-completion using Tab after writing the name of the variable and a dot.

```
In [ ]: s = 'hello world'
```

```
In [ ]: s.capitalize()
```

But we will comeback on this later on.

Exercise

- Write the following code with the shortest way that you think is the best:

```
'Hello MAIA! Hello MAIA! Hello MAIA! Hello MAIA! Hello MAIA! GO GO GO!'
```

1.3.2 Lists

Lists are similar to strings. However, they can contain whatever types. The squared brackets are used to identified lists.

```
In [ ]: l = [1, 2, 3, 4]

In [ ]: l

In [ ]: type(l)

In [ ]: l = [1, '2', 3.0]

In [ ]: l

In [ ]: for elt in l:
            print(f'The element {elt} is of type {type(elt)}')

In [ ]: l = [1, 2, 3, 4, 5]
```

We can use the same syntax to index and slice the lists.

```
In [ ]: l[0]

In [ ]: l[-1]

In [ ]: l[2:5:2]
```

A list is also a container. Therefore, we would expect the same behavior for + and * operators.

```
In [ ]: l + l + l

In [ ]: l * 3
```

Append, insert, modify, and delete elements In addition, a list also have some specific methods. Let's use the auto-completion

```
In [ ]: l = []

In [ ]: len(l)

In [ ]: l.append("A")

In [ ]: l

In [ ]: len(l)
```

`append` is adding an element at the end of the list.

```
In [ ]: l.append("x")

In [ ]: l
```

```
In [ ]: l[-1]
```

`insert` will let you choose where to insert the element.

```
In [ ]: l.insert(1, 'c')
```

```
In [ ]: l
```

We did not try to modify an element from string before. We can check what would happen.

```
In [ ]: s
```

```
In [ ]: s[0] = "H"
```

So we call the `string` an immutable container since it cannot be changed.
What happens with a list?

```
In [ ]: l
```

```
In [ ]: l[1] = 2
```

```
In [ ]: l
```

A list is therefore mutable. We can change any element in the list. So we can also remove an element from it.

```
In [ ]: l.remove(2)
```

```
In [ ]: l
```

Or directly using an index.

```
In [ ]: del l[-1]
```

```
In [ ]: l
```

1.3.3 Built-in functions

Now that we introduced the `list` and `string`, we can check the so called built-in functions: https://docs.python.org/3/library/functions.html

These functions are a set of functions which are commonly used. For instance, we already presented the `slice` functions. From this list, we will present three functions: `in`, `range`, `enumerate`, and `sorted`. You can check the other functions later on.

`sorted` function The sorted function will allow us to introduce the difference between inplace and copy operation. Let's take the following list:

```
In [ ]: l = [1, 5, 3, 4, 2]
```

We can call the function `sorted` to sort the list.

```
In [ ]: sorted(l)
```

We can observe that a sorted list is returned by the function. We can also check that the original list is actually unchanged:

```
In [ ]: l
```

It means that the `sorted` function made a copy of `l`, sorted it, and return us the result. The operation was not made inplace. However, we saw that a list is mutable. Therefore, it should be possible to make the operation inplace without making a copy. We can check the method of the list and we will see a method `sort`.

```
In [ ]: l.sort()
```

```
In [ ]: l
```

We see that this `sort` method did not return anything and that the list was changed inplace.

Thus, if the container is mutable, calling a method will try to do the operation inplace while calling the function will make a copy.

`range` function It is sometimes handy to be able to generate number with regular interval (e.g. start:end:step). The range function create such generator.

```
In [ ]: x = range(5, 10, 2)
```

```
In [ ]: type(x)
```

The result obtained from calling the `range` function is actually a generator. The main difference between a generator and a list is that the number a generated one after another instead of putting all of the number into the memory. However, we can easily convert this generator into a list.

```
In [ ]: list(x)
```

`enumerate` function The `enumerate` function allows to get the index associated with the element extracted from a container. Let see what we mean:

```
In [ ]: for idx, elt in enumerate(range(5, 10, 2)):
            print(f'The index of {elt} is {idx}')
```

`in` function The function `in` allows to know if a value is in the container.

```
In [ ]: l = [1, 2, 3, 4, 5]

In [ ]: 5 in l

In [ ]: 6 in l

In [ ]: s = 'Hello world'

In [ ]: 'h' in s

In [ ]: 'H' in s
```

1.3.4 Tuples

Tuple can be seen as an immutable list. The syntax used is `(values1, ...)`.

```
In [ ]: t = tuple([1, 2, 3])

In [ ]: t

In [ ]: type(t)

In [ ]: t[0] = 4
```

However, tuples are not only used as such. They are mainly used for unpacking variable. For instance they are usually returned by function when there is several values.

```
In [ ]: def func():
            return 1, 2, 3

In [ ]: x = func()

In [ ]: x

In [ ]: type(x)
```

We can easily unpack tuple with the associated number of variables.

```
In [ ]: x, y, z = func()

In [ ]: x

In [ ]: y

In [ ]: z
```

1.3.5 Dictionary

Dictionaries are used to map a key to a value. The syntax used is `{key1: value1, ...}`.

```
In [ ]: d = {
            'param1': 1.0,
            'param2': 2.0,
            'param3': 3.0
        }
```

```
In [ ]: d
```

```
In [ ]: type(d)
```

To access a value associated to a key, you index using the key:

```
In [ ]: d['param1']
```

Dictionaries are mutable. Thus, you can change the value associated to a key.

```
In [ ]: d['param1'] = 4.0
```

```
In [ ]: d
```

You can add a new key-value relationship in a dictionary.

```
In [ ]: d['param4'] = 5.0
```

```
In [ ]: d
```

And you can as well remove relationship.

```
In [ ]: del d['param4']
```

```
In [ ]: d
```

You can also know if a key is inside the dictionary.

```
In [ ]: 'param1' in d
```

You can also know about the key and values with the following methods:

```
In [ ]: d.keys()
```

```
In [ ]: d.values()
```

```
In [ ]: d.items()
```

It can allows to iterate:

```
In [ ]: for key, value in d.items():
            print(f'key {key} => {value}')
```

```
In [ ]: for key in d.keys():
            print(f'key {key} => {d[key]}')
```

1.4 Conditions and loop

1.4.1 `if`, `elif`, and `else` condtions

Python delimates code block using indentation.

```
In [ ]: a = 3
        b = 2

        if a < b:
            print('a is smaller than b')
        elif a > b:
            print('a is bigger than b')
        else:
            print('a is equal to b')
```

Be aware that if you do not indent properly your code, then you will get some nasty errors.

```
In [ ]: if True:
            print('whatever')
             print('wrong indentation')
```

1.4.2 `for` loop

You might be used to iterate using some indices.

```
In [ ]: l = range(5, 15, 2)

        for idx in range(len(l)):
            print(f'idx: {idx} => value: {l[idx]}')
```

However, in Python you can already get the element from a container.

```
In [ ]: l = range(5, 15, 2)

        for elt in l:
            print(f'value: {elt}')
```

And if you wish to get the corresponding indices, you can always use `enumerate`.

```
In [ ]: l = range(5, 15, 2)

        for idx, elt in enumerate(l):
            print(f'idx: {idx} => value: {elt}')
```

You can have nested loop.

```
In [ ]: for word in ["calcul", "scientifique", "en", "python"]:
            for letter in word:
                print(letter)
```

Exercise

- Count the number of occurences of each character in the string `'HelLo WorLd!!'`. Return a dictionary associating a letter to its number of occurences.

```
In [ ]:
```

- Given the following encoding, encode the string `s`.
- Once the string encoded, decode it by inversing the dictionary.

```
In [ ]: code = {'e':'a', 'l':'m', 'o':'e', 'a': 'e'}
        s = 'Hello world!'
```

1.4.3 `while` loop

If your loop should stop at a condition rather than using a number of iterations, you can use the `while` loop.

```
In [ ]: i = 0

        while i < 5:
            print(i)
            i = i + 1

        print("OK")
```

Exercise

- Code the Wallis formula to compute π:

$$\pi = 2\prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}$$

1.5 Fonctions

We already used functions above. So we will give a formal introduction. Fonction in Python are using the keyword `def` and define a list of parameters.

```
In [ ]: def func(x, y):
            print(f'x={x}; y={y}')

In [ ]: func(1, 2)
```

These parameters can be positional or use a default values.

```
In [ ]: def func(x, y, z=0):
            print(f'x={x}; y={y};, z={z}')

In [ ]: func(1, 2)
```

```
In [ ]: func(1, 2, 3)

In [ ]: func(1)
```

Functions can return one or more values. The output is a tuple if there is several values.

```
In [ ]: def square(x):
            return x ** 2

In [ ]: square(2)

In [ ]: def square(x, y):
            return x ** 2, y ** 2

In [ ]: square(2, 3)

In [ ]: x_2, y_2 = square(2, 3)

In [ ]: x_2

In [ ]: y_2
```

How does the documentation is working in Python.

```
In [ ]: help(square)
```

We can easily define what should be our inputs and outputs such that people can use our function documentation.

```
In [ ]: def square(x, y):
            """Square a pair of numbers

            Parameters
            ----------
            x : real
                First number.
            y : real
                Second number.
            Returns
            -------
            squared_numbers : real
                The squared x and y.
            """
            return x ** 2, y ** 2

In [ ]: help(square)
```

1.6 Classes

This introduction is taken from the scipy lecture notes: https://scipy-lectures.org/intro/language/oop.html

Python supports object-oriented programming (OOP). The goals of OOP are:

- to organize the code, and
- to re-use code in similar contexts.

Here is a small example: we create a Student class, which is an object gathering several custom functions (methods) and variables (attributes), we will be able to use:

```
In [ ]: class Student(object):
            def __init__(self, name):
                self.name = name
            def set_age(self, age):
                self.age = age
            def set_major(self, major):
                self.major = major

        anna = Student('anna')
        anna.set_age(21)
        anna.set_major('physics')
```

In the previous example, the Student class has `__init__`, `set_age` and `set_major` methods. Its attributes are `name`, `age` and `major`. We can call these methods and attributes with the following notation: `classinstance.method` or `classinstance.attribute`. The `__init__` constructor is a special method we call with: `MyClass(init parameters if any)`.

Now, suppose we want to create a new class MasterStudent with the same methods and attributes as the previous one, but with an additional `internship` attribute. We won't copy the previous class, but **inherit** from it:

```
In [ ]: class MasterStudent(Student):
            internship = 'mandatory, from March to June'

        james = MasterStudent('james')
        james.internship

        james.set_age(23)
        james.age
```

The MasterStudent class inherited from the Student attributes and methods.

Thanks to classes and object-oriented programming, we can organize code with different classes corresponding to different objects we encounter (an Experiment class, an Image class, a Flow class, etc.), with their own methods and attributes. Then we can use inheritance to consider variations around a base class and **re-use** code. Ex : from a Flow base class, we can create derived StokesFlow, TurbulentFlow, PotentialFlow, etc.

```
In [ ]:
```