Basics of Python

With NumPy and Pandas

Paulo S. A. Sousa

Why to learn a programming language?

- A computer can do a lot of calculations and processing for us.
- Consequently, computers can make our lives easier and more pleasant.
- Unfortunately, the computer cannot yet understand the human language.

Why to learn a programming language?

- Thus, we cannot give it orders and tasks by using our own voice.
- Great progress has recently been made.
- But we are still far from having computers understanding, without errors, human language.

Why to learn a programming language?

- A computer programming language is a language that computers can fully understand.
- There are dozens and dozens of computer programming languages.
- We will use Python and very occasionally R.

Why Python?

- Popularity, specially in machine learning
- Easy to learn and use
- Powerful
- Used in real-world companies
- Large community

Why Python?

- We do not have time to deepen the study of Python, but:
  - You are *very* encouraged to do it.
  - That is a great investment in human capital.
  - It will greatly help you to be more creative and more efficient.

An extremely simplified description of computer functioning

- As human beings, computers have memory.
- To simplify: Two types of memory: RAM and hard disk.
- RAM memory is very much faster than hard disks.

An extremely simplified description of computer functioning

- Therefore, the processor, the device that does the calculations, is always:
  - Exchanging information with the RAM.
  - This information corresponds to the inputs and outputs for the operations the processor performs.

Basic structure of a computer program

- A program is a text file:
  - Where each line is a command given to the computer.
  - The computer reads the text file and executes the commands, line after line.
- There are variables, to store information in the RAM.

Types of variables

The three fundamental types of variables in Python are:

- **Numeric**: these variables can store `numerical` values. As sub-types, we have `integer` and `float` variables.
- **String**: these variables can store `textual` values.
- **Boolean**: these variables can store `logical` values.

Lists

Typically, we need to store not only a single value, but multiple values under the same variable name:

- The `list` is the type of object to accomplish that.
- A list can contain elements of different types (numerical, string and boolean).
- The index of the first element is 0.
- (In general, in Python, indexing starts at 0.)

Indexing

It is through indexes that we can access to the elements of a list. Consider the list `l = [4, 8, 25]`.

- The first element of `l`, the number `4`, is at the position of index `0`.
- The second element of `l`, the number `8`, is at the position of index `1`.
- And so on.
- As a general rule, in Python, indexes start by `0`.

Slicing

Slicing is a compact way of indexing:

- Obtaining *all* elements:

```
1  l = [10, 20, 3, 5, 25]
2  l[:]
```

```
[10, 20, 3, 5, 25]
```

- Obtaining only the *last* element:

```
1  l = [10, 20, 3, 5, 25]
2  l[:-1]
```

```
[10, 20, 3, 5]
```

- Obtaining all elements up to index 2 (not inclusive):

```
1  l = [10, 20, 3, 5, 25]
2  l[:2]
```

```
[10, 20]
```

- Obtaining only the *last* element:

```
1  l = [10, 20, 3, 5, 25]
2  l[:-1]
```

```
[10, 20, 3, 5]
```

Slicing

More ways of indexing with slicing:

- Obtaining all elements but the last 3 ones:

```
l = [10, 20, 3, 5, 25]
l[:-3]
```

```
[10, 20]
```

- Obtaining only the *last* element:

```
l = [10, 20, 3, 5, 25]
l[0:4:2]
```

```
[10, 3]
```

- Obtaining all elements up to index 2 (not inclusive):

```
l = [10, 20, 3, 5, 25]
```

```
2  l[4:1:-2]
```

```
[25, 3]
```

- Obtaining only the *last* element:

```
1  l = [10, 20, 3, 5, 25]
2  l[-3:-1]
```

```
[3, 5]
```

Tuples

- Tuples are also used to store multiple items in a single variable.
- A tuple is a collection which is ordered and *unchangeable*.
- Unchangeability means that we *cannot* change, add or remove items after the tuple has been created.
- Tuples are written with round brackets.

Tuples

- Example:

```
1  a = tuple([5, 8, 6])
2  a
```

```
(5, 8, 6)
```

- Tuples also work with the usual indexing:

```
1  #a = tuple([5, 8, 6])
2  print(a[2], a[0], a[1])
```

```
6 5 8
```

- We cannot change a tuple, as we can confirm by trying to run the code:

```
1  #a = tuple([5, 8, 6])
2  a[1] = 4
```

Zip function

The `zip` function is useful to bind lists:

```
names = ['Mary', 'John', 'Catherine']
ages = [19, 21, 24]

zip(names, ages)
```

```
<zip at 0x7fb16d826980>
```

- As it is, we cannot see the content of the result of the `zip` function.

Zip function

- But we can use list comprehension to see the such a content:

```
1  [x for x in zip(names, ages)]
```

```
[('Mary', 19), ('John', 21), ('Catherine', 24)]
```

- Each element of the result of the `zip` function is a tuple with two elements: the name and the respective age.

Dictionaries

- A dictionary allows us to store information, which can be called by a key.
- Thus, each element of a dictionary is constituted by two components:
  - The key, with which we can call the needed information.
  - The information we need to retrieve.

Dictionaries

- An example:

```
1  wages = {'Peter': 1000, 'Mary': 1500, 'Bob': 1200}
2  wages
```

```
{'Peter': 1000, 'Mary': 1500, 'Bob': 1200}
```

- Getting the wage of Mary:

```
1  wages.get('Mary')
```

```
1500
```

- Another way of getting Mary's wage is:

```
1  wages['Mary']
```

```
1500
```

Dictionaries

- Getting the keys:

```
1  wages.keys()
```

```
dict_keys(['Peter', 'Mary', 'Bob'])
```

- Creating a dictionary by using `zip` function:

```
1  names = ['Peter', 'Mary', 'Bob']
2  salaries = [1000, 1500, 1200]
3
4  wages = dict(zip(names, salaries))
5  wages
```

```
{'Peter': 1000, 'Mary': 1500, 'Bob': 1200}
```

Dictionaries

- Creating a dictionary by using dictionary comprehension:

```
1  {x[0]:x[1] for x in zip(names, salaries)}
```

```
{'Peter': 1000, 'Mary': 1500, 'Bob': 1200}
```

- Notice that `x[0]` and `x[1]` are, respectively, the first and the second elements of tuple `x`.

Numpy arrays

- The homogeneous multidimensional array is the core component of `NumPy`.
- It is a table of identically typed entries (often numbers), each of which is indexed by a tuple of positive integers.

Numpy arrays

- Axes are what NumPy refers to as dimensions.

Numpy arrays

- To create a 1D numpy array, we can use the following code:

```
1  import numpy as np # do this only once
2
3  a = np.array([1,2,3,4,5])
4  a
```

```
array([1, 2, 3, 4, 5])
```

Numpy arrays

- To create a 2D numpy array, we can use the following code:

```
1  b = np.array([[1, 2], [4, 8]])
2  b
```

```
array([[1, 2],
       [4, 8]])
```

Numpy arrays

- To create a 3D numpy array, we can use the following code:

```
c = np.array([[[1, 2], [4, 8]],
[[8, 2], [5, 7]],
[[10, 20], [1, 5]]])
c
```

```
array([[[ 1,  2],
        [ 4,  8]],

       [[ 8,  2],
        [ 5,  7]],

       [[10, 20],
        [ 1,  5]]])
```

Numpy arrays

Numpy has a very great variety of functions to operate on arrays.

```
1 b
```

```
array([[1, 2],
       [4, 8]])
```

```
1 np.sum(b)
```

```
15
```

```
1 np.sum(b, axis=0)
```

```
array([ 5, 10])
```

```
1  np.sum(b, axis=1)
```

```
array([ 3, 12])
```

Numpy arrays

Numpy has a very great variety of functions to operate on arrays.

```
1 c
```

```
array([[[ 1,  2],
        [ 4,  8]],

       [[ 8,  2],
        [ 5,  7]],

       [[10, 20],
        [ 1,  5]]])
```

```
1 np.sum(c)
```

```
73
```

```
1 np.sum(c, axis=0)
```

```
array([[19, 24],
       [10, 20]])
```

```
1 np.sum(c, axis=1)
```

```
array([[ 5, 10],
       [13,  9],
       [11, 25]])
```

```
1 np.sum(c, axis=2)
```

```
array([[ 3, 12],
       [10, 12],
       [30,  6]])
```

Pandas dataframes

- Data are usually in tabular format.
- Dataframe is the adequate object to contain tabular data of multiple types (numerical, logical, string, etc.).
- Library `pandas` makes dataframes available for our use and a vast number of functions to operate with dataframes.

Pandas dataframes

A pandas dataframe example:

```
    Name Gender  Age  Wage
0  Peter      M   20  1000
1   Mary      F   35  1500
2    Bob      M   28  1200
```

- There are several ways to create a dataframe.
- All of them require importing library pandas:

```
1 import pandas as pd
```

Pandas dataframes

Creating a dataframe from using a dictionary:

```
dict = {'Name': ['Peter', 'Mary', 'Bob'],
 'Gender': ['M', 'F', 'M'], 'Age': [20, 35, 28],
 'Wage': [1000, 1500, 1200]}

df = pd.DataFrame(dict)
print(df)
```

```
    Name Gender  Age  Wage
0  Peter      M   20  1000
1   Mary      F   35  1500
2    Bob      M   28  1200
```

Pandas dataframes

Creating a dataframe from using lists:

```
df = pd.DataFrame(
  [['Peter', 'M', 20, 1000],
  ['Mary', 'F', 35, 1500],
  ['Bob', 'M', 28, 1200]],
  columns=['Name', 'Gender', 'Age', 'Wage'])

print(df)
```

```
    Name Gender  Age  Wage
0  Peter      M   20  1000
1   Mary      F   35  1500
2    Bob      M   28  1200
```

Pandas dataframes

- When we import data into Python, we typically place the imported data in a dataframe.
- Pandas can *import* data from a variety of different types of files:
  - CSV files.
  - Excel files.
  - SQL database files.
  - …

Pandas dataframes

- Example of reading `csv` file to a pandas dataframe:

```
1  import pandas as pd
2
3  df = pd.read_csv('datasets/insurance.csv')
4  print(df)
```

```
      age     sex   bmi  children smoker     region  expenses
0      19  female  27.9         0    yes  southwest  16884.92
1      18    male  33.8         1     no  southeast   1725.55
2      28    male  33.0         3     no  southeast   4449.46
3      33    male  22.7         0     no  northwest  21984.47
4      32    male  28.9         0     no  northwest   3866.86
...   ...     ...   ...       ...    ...        ...       ...
1333   50    male  31.0         3     no  northwest  10600.55
1334   18  female  31.9         0     no  northeast   2205.98
1335   18  female  36.9         0     no  southeast   1629.83
1336   21  female  25.8         0     no  southwest   2007.95
1337   61  female  29.1         0    yes  northwest  29141.36

[1338 rows x 7 columns]
```

Pandas dataframes

- Likewise, Pandas can *export* data to a variety of different types of files:
  - CSV files.
  - Excel files.
  - SQL database files.
  - …

Pandas dataframes

- Example of writing a dataframe to a `csv` file:

```
1  import pandas as pd
2
3  df.to_csv('/tmp/insurance.csv')
```

FOR loop

- Sometimes, we need to repeat the same operations a known number of times.
- For instance, consider the following two lists:
- To print on the screen the name of each student and the respective marks, we will repeat the printing instruction for 3 times.

FOR loop

- The printing in code:

```
print(f'Student {names[0]} obtained {marks[0]} marks.')
print(f'Student {names[1]} obtained {marks[1]} marks.')
print(f'Student {names[2]} obtained {marks[2]} marks.')
```

```
Student Mary obtained 15 marks.
Student John obtained 18 marks.
Student Catherine obtained 19 marks.
```

- Can you imagine the needed number of lines of code if we had **1000** students?

FOR loop

The syntax of the for loop is the following:

```
1  for (x in object):
2    do something
```

FOR loop

- Let us print the student's marks through a `for` loop:

```
1  for i in range(len(names)):
2    print(f'Student {names[i]} obtained {marks[i]} marks.')
```

```
Student Mary obtained 15 marks.
Student John obtained 18 marks.
Student Catherine obtained 19 marks.
```

- Notice that we obtain the length of list `names` with:

```
1  len(names)
```

```
3
```

- Notice yet that we obtain a range of the indexes of `names` with:

```
1  range(len(names))
```

```
range(0, 3)
```

FOR loop

- Another way of printing the student's marks through a `for` loop:

```
1  for n, m in zip(names, marks):
2      print(f'Student {n} obtained {m} marks.')
```

```
Student Mary obtained 15 marks.
Student John obtained 18 marks.
Student Catherine obtained 19 marks.
```

- Take into consideration that the `zip` function takes iterables, aggregates them in a tuple, and returns it.

List comprehension

- That is a *compact* way of using a `for` loop to create a list.
- Its syntax is the following:

```
1  [element for x in object]
```

- The result is a list of all elements so created.
- Example:

```
1  [x**2 for x in range(15)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

Functions

- A function is a piece of code.
- Whenever we need to use the code of a function, we just call it by its name.
- Pieces of code that we need to repeat, can be placed in a function.

Functions

- By using functions, the code can become much:
  - shorter
  - easier to read
- Some examples of functions:
  - `print`
  - `sum`
  - `plot`
  - ...

Functions

In Python, functions can be created by using the following code structure:

```
def name_of_the_function(parameters):
  line of code
  line of code
  ...
  line of code
  return result
```