Programming with Python - Refresher course II

Sonia Martinot

Master in Data Science & Business Analytics

CentraleSupelec & ESSEC

September 8, 2024

Contents

Current Section

Modules: Definition

Definition: A **module** is a collection of functions that are already implemented and available for use. Also called a **library.**

- A huge number of modules are already installed with Python.
- You can install modules using Anaconda or `pip`:
  `pip install module_name`
- You need to import a module at the beginning of the python script to use it: `import module_name`

Modules: Loading

Load the entire module:

```
# Import the module
import module1
# Call a function of the module
output = module1.function1()
```

Load and rename the module:

```
import module1 as m1
output = m1.function1()
```

Load only specific functions:

```
# Import only functions f1 and f2
from module1 import f1, f2
output1 = f1()
output2 = f2()
```

Load all the functions of the module (not recommended):

```
from module1 import *
output = function1()
```

Modules: the classics

Famous modules:

- **Math**: Mathematical functions.
- **Numpy**: Computations on multidimensional matrix.
- **Matplotlib**: Visualizations.
- **Scipy**: Optimization, integration, interpolation.
- **Pandas**: Data structures and data analysis.
- **Scikit-Learn**: Machine Learning.
- **Pytorch**: Deep Learning.
- **Time**: Time measurement.
- **Random**: Random number generator & probabilistic distributions.
- **Os**: Manage files and folders.

Current Section

Functions

- Create a function using the keyword `def`.
- Get the function to give an output using `return`.
- If you do not use `return`, the function will yield `None`.
- Functions with multiple outputs return a **tuple**.
- Everything variable made **inside** are **local**: they do not exist outside of the function.

Best practice:

- Give explicit name to functions (**not** *f1*, *function2*...)
- Every time you write something more than 3 times: code a function.

Functions: example

```python
def power_of_2(x):
    x_squared = x ** 2
    return x_squared
```

More direct implementation, one less line of code:

```python
def power_of_2(x):
    return x ** 2
```

Functions: Arguments

- You can define a function with any number of parameters, also called **arguments**:

  ```
  def my_function(x1, x2, x3)
  ```

- You can define the function with **default values** for some arguments:

  ```
  def my_function(x1, x2=value2, x3=value3):
  ...
  ```

Handling multiple parameters - Positional vs Keyboard arguments:

- `my_function(1, 2, 3)` is **different** from `my_function(2, 3, 1)`
- `my_function(x1=1, x2=2, x3=3)` is the **same** as `my_function(x2=2, x3=3, x1=1)`

Functions: Example 1

```python
# Function with a default argument
def exponent(x, n=2):
    return x ** n

# Call the function
x_exponent = exponent(x=5)
# Change argument
x_exponent = exponent(x=5, n=3)
```

```python
# Function with no argument and return
def salute_world():
    print("Hello world")

output = salute_world()
# output is None
```

Functions: Example 2

```
In [2]: # Create a function
   ...: def my_first_functions():
   ...:     print('Hello world')
   ...:

In [3]: # Create a function
   ...: def my_first_functions():
   ...:     print('Hello world')
   ...:
   ...: # Out of the functions
   ...:
   ...: # Execture the function :
   ...: my_first_functions()
Hello world

In [4]: # Default Parameter Value
   ...: def presentation(name, country = 'Planet Earth'):
   ...:     print('Hello my name is {}. I come from {}'.format(name, country))
   ...:
   ...: presentation('Theo', 'France')
Hello my name is Theo. I come from France

In [5]: presentation('Nicolas')
Hello my name is Nicolas. I come from Planet Earth

In [6]: presentation('Lea', country='Germany')
   ...:
Hello my name is Lea. I come from Germany

In [7]: |
```

Functions: Example 3

```
In [7]: def function_without_return():
   ...:     print('I am a function')
   ...:

In [8]: a = function_without_return()
I am a function

In [9]: print(a)
None

In [10]: print(type(a))
<class 'NoneType'>

In [11]: def function_with_return(a, b):
    ...:     return a + b, a * b
    ...:

In [12]: a, b = function_with_return(2, 3)

In [13]: c = function_with_return(1, 4)

In [14]: print(type(c))
<class 'tuple'>

In [15]: print(c)
(5, 4)

In [16]: print(a)
5

In [17]: print(b)
6
```

Functions

- Functions can be stored as variables.
- Functions can be passed as arguments to other functions.

```python
def square(x):
    return x * x
a = square
print("The type of variable a is:", type(a))
print(a(5))

def print_function(f, x):
    a = f(x)
    print("F ({}) == {}".format(x, a))

print_function(square, 5)
```

Recursive Functions

Definition: A recursive function is a function that calls itself.

A recursive function has **three requirements**:

- A base case.
- A modification to go from current state to base case.
- A call to itself.

Warning: Recursive functions can be dangerous as they can yield a huge number of calculations that crash your computer.

Recursive Functions: example

```
def recursive_sum(L):
    # Base case
    if len(L) == 1:
        return L[0]
    # Modification and call to itself
    else:
        return L[0] + recursive sum(L[1:])
```

```
def recursive_fact(n):
    # Base case
    if n == 0:
        return 1
    # Modification and call to itself
    else:
        return n * fact(n-1)
```

Recursive Functions: example

```python
def recursive_fibo(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Modification and call to itself
    else:
        return recursive_fibo(n-1) + recursive_fibo(n-2)
```

This definition is very inefficient as it will recompute values and have a very high number of operation.

Current Section

Algorithms

Definition: An algorithm is a function which takes an **input**, does a finite number of **non ambiguous instructions** and **returns** a result. Need to answer 3 questions:

- Does the algorithm stop ?
- Does the algorithm do what it is supposed to ?
- How much time does it last ?

Algorithms

- The term algorithm derives from the name of Muhammad ibn Musa al"Khwarizmi, a 9th-century Persian mathematician.
- Not originally related to computer science.
- Examples:
  - Cooking recipe
  - Sieve of Eratosthenes
  - Euclidean algorithm

Current Section

Complexity

Definition: Quantification of the performance of an algorithm in order to compare with other algorithms.

- **Time complexity**: Counting the number of elementary operations.
- **Space complexity**: Counting the RAM size needed. We focus

  only on time complexity and distinguish 3 cases:
  - **Best case complexity**: sorting an already sorted list.
  - **Worst case complexity**: sorting a list sorted in descending order.
  - **Average complexity**

Complexity

Idea: If the algorithm takes an input of size N, what is the **order of magnitude** of the number of operations on the input that will be required ? N ? N^2 ? $exp(N)$? $log(N)$?

Example:

- Do N times operation A: N operations -> $O(N)$ complexity
- Do N times operations B and C: 2N operations -> $O(N)$ complexity

Current Section

Sorting

We will see 3 easy ways to sort a list:

- Selection sort
- Insertion sort
- Bubble sort

More sophisticated sort algorithms exist:

- Merge sort
- Quicksort

Selection sort

Idea: Find the minimum element, put it in position 0 and repeat.

For i from 0 to n:

- Find minimal element in the list starting at i.
- Put it in position i.
- Repeat from position $i+1$.

Selection sort

Pseudo-code:

```
def selection_sort(T):
    for i from 0 to n-1:
        min = find_minimum(T[i:])
        exchange(t[i], t[min])
```

Complexity: $O(N^2)$. Why ?

- Selecting the minimum requires $n-1$ comparisons, then $n-2$...
- Therefore, the number of operations is:

$$(n-1)+(n-2)+...+1=\sum_{i=1}^{n-1} i=\frac{1}{2}n^2-n\sim O(n^2)$$

Insertion sort

Idea: We will take each element and put it at the right position.

For i from 0 to n:

- The list from 0 to i1 is sorted.
- Take the i^{th}element.
- Look for its position between 0 and i1.
- Exchange the element in i with its new position.
- The list is sorted from 0 to i.

Complexity: $O(n^2)$. Why ?

Insertion sort

Pseudo-code:

```
def insertion_sort(L):
    for i from 0 to n-1:
    # Save L[i]
        x = L[i]
        j = i
        # We look for the position of L[i] amongst the alre
        while j > 0 and L[j-1] > x:
            L[j] = L[j-1]
            j = j - 1
        L[j] = x
        # j = 0 if L[i] is the minimum
        # or if L[j-1] < x i.e. x is at the correct positio
```

Complexity: $O(N^2)$. Why ?

Bubble sort

Idea: We permute consecutive elements together and the biggest element will rise like bubble.

Let L be a list, for i from $n-1$ to 1: For j from 0 to $i-1$:

- Compare element $L[j]$ and $L[j+1]$.
- Exchange them if they are in the wrong order.

Pseudo-code:

```
def bubble_sort(L):
    for i from n-1 to 1:
        for j from 0 to i-1:
            if L[j+1] < L[i]:
                exchange(L[j+1], L[j])
```

Complexity: $O(n^2)$. Why ?

Thank you for your attention !
Now let's practice !