TD4

February 17, 2020

1 Les fonctions en python

Les fonctions permettent de préparer un bloc d'instructions que l'on pourra appeler et reappeler plus tard grâce à un nom de fonction. Nous avons déjà vu des fonctions uselles `print()`, `input()`, `int()`...

1.1 Définir une fonction

Il est possible de créer ses propres fonctions. Lors de la création, il faut définir le nom de la fonction ainsi que les arguments qui seront nécessaires. Voici la syntaxe :

```
def nom_de_fonction(argument1,argument2) :
    instruction 1
    instruction 2
    instruction 3...
```

- Le mot clé `def` permet à python de savoir que vous allez définir une fonction.
- Nous nous servirons ensuite du `nom_de_fonction` pour appeler la fonction.
- Les arguments seront à fournir pour que la fonction puissent opérer.

Voici un petit exemple :

```
In [23]: def cube(x) :
             print(x*x*x)
```

Une fois définie, l'appel de la fonction se fait de la façon suivante :

```
In [24]: cube(4)
         cube(10.1)

64
1030.301
```

1.2 Valeurs par défaut des arguments

Il est souvent utile de préciser des valeurs par défaut à chaque paramètre. Pour cela nous allons à l'aide de = donner ces valeurs par défauts lors de la création de la fonction.

```
def nom_de_fonction(argument1 = valeur_defaut1, argument2 = valeur_defaut2) :
    instruction 1
    instruction 2
    instruction 3...
```

Reprenons l'exemple précédent en précisant une valeur par defaut.

```
In [25]: def cube(x = 2) :
             print(x*x*x)
```

La fonction donne les mêmes résultats que précédemment. Mais il est possible de l'appeler sans lui donner d'argument. Dans ce cas, nous obtiendrons toujours 2^3 :

```
In [26]: cube(4)
         cube(10.1)
         cube()          # Utilisation de la fonction avec les paramètres par défaut
```

```
64
1030.301
8
```

Voici un deuxième exemple. Cette fonction prend deux arguments `nombre` et `max`. Elle affiche la table de multiplication associée à `nombre` de 0 jusqu'à `max` :

```
In [4]: def TableMultiplication(nombre=2, max=10):
            for i in range(0,max+1):
                print(i,"*",nombre,"=",i*nombre)
```

```
In [5]: TableMultiplication(2,5) # on demande la table de 2 de 0 à 5
```

```
0 * 2 = 0
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
5 * 2 = 10
```

```
In [6]: TableMultiplication(5) # on demande la table de 5. On ne spéccifie pas max.
                               # La table ira jusqu'à 10, la valeur par défaut
```

```
0 * 5 = 0
1 * 5 = 5
2 * 5 = 10
```

```
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
10 * 5 = 50
```

```
In [7]: TableMultiplication(max=5) # on peut préciser quel argument on veut modifier
```

```
0 * 2 = 0
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
5 * 2 = 10
```

```
In [9]: TableMultiplication(max=5, nombre=3) # et ce dans l'ordre que l'on veut
```

```
0 * 3 = 0
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
4 * 3 = 12
5 * 3 = 15
```

1.3 Sortie d'une fonction (`return`)

Les fonctions précédentes ne font qu'afficher des choses et il n'est pas possible de ce servir d'un résultat obtenu en dehors de la fonction.

Afin de récupérer la *sortie* d'une fonction, on utilise la commande `return` puis on indique ce que l'on veut *sortir*. Reprenons la fonction `cube()`. On peut placer le résultat de cette fonction dans une variable :

```
In [1]: def cube(x = 2) :
            return x*x*x

        resultat = cube(10) # On affecte le resultat de cube(10) dans la variable resultat

        print("La variable résultat vaut : ", resultat)
```

```
La variable résultat vaut :  1000
```

Il est également possible de retourner plusieurs résultats en même temps. Le résultat est sous forme de `list`. Nous verrons dans le TD suivant ce qui nous pouvons en faire.

```
In [13]: def classer(x,y) :
             a,b =  x,y
             if a > b :
                 a,b = b,a
             return a,b

         classer(2,1)
```

```
Out[13]: (1, 2)
```

1.4 Les librairies de fonctions

Nous avons déjà chargé une librairie de fonctions : *numpy* Pour cela nous avions utilisé la commande :

```
from numpy import *
```

Pour que cela fonctionne, il faut naturellement que la librairie *numpy* soit installée. On peut alors utiliser les fonctions mathématiques de *numpy*, exemple :

```
sqrt(12)
tan(15)
```

Cet appel n'est en réalité pas très propre. La bonne façon d'appeller une librairie est la suivante :

```
import numpy as np
```

Pour utiliser la librairie *numpy*, il faut maintenant écrire :

```
np.sqrt(12)
np.tan(15)
```

Cela pourrait paraître plus fastidieux, mais maintenant lorsque nous appellons une fonction, nous savons dans quelle librairie nous allons la chercher. Lorsque l'on utilise plusieurs librairie, cela évite de se tromper et cela accélère *python*.

Voici un exemple d'erreur. Ici nous allons charger deux librairies mathématiques. *numpy* et *math*. Lorsque l'on appelle la fonction racine (`sqrt()`), nous voyons qu'il y a une différence.

```
In [7]: import numpy as np
        import math as mt

        print(mt.sqrt(12))
        print(np.sqrt(12))
```

```
3.4641016151377544
3.46410161514
```

Si maintenant, nous prenons la racine d'un complexe, on voit que *numpy* y arrive, mais pas *math*.