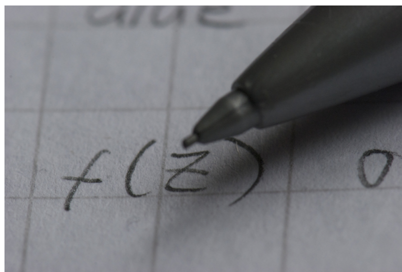
Python for Scientists
Part 7-1 – Deeper into Functions

Cyril Desjouy

June, 2016
Updated : November 7, 2018

Functions



Photography by Vestman

Syntaxe : `def ...return ...`

```
def mult(x, y=2):   # x :  positional argument, y :  keyword argument
    return x*y
```

```
In  : mult(3)
Out : 6
```

```
In  : mult(3, 4)
Out : 12
```

- L'instruction `def` permet de définir une fonction
- Une fonction est un objet qui retourne un objet
- Par défaut, une fonction retourne `None` s'il n'y a pas d'instruction `return`
- Une fonction peut être utilisée comme tout autre objet. Elle peut être :
  - un argument ou un `return` d'une autre fonction
  - référencée par une variable
  - un élément d'une structure (`tuple`, `list`, `dict`, ...)
- Une définition de fonction accepte des paramètres :
  - obligatoires, appelés ***positional arguments***
  - optionnels, appelés ***keyword arguments*** spécifiant des valeurs par défaut[1]

[1] Ces valeurs sont évaluées au moment où la fonction est définie, pas lorsqu'elle est appelée

Nombre d'arguments variable

```
def myfct(*args, **kwargs):
    """Function printing positional and keyword arguments.

    Arguments :
    args -- my positional arguments
    kwargs -- my keyword arguments
     """

    print("args are {}".format(args))
    print("kwargs are {}".format(kwargs))
```

```
In  : myfct("abc", [1, 2], x="Opt", y = 12)
Out : args are ("abc", [1,2])              # args :  tuple
      kwargs are {x:  "Opt", y:  12}       # kwargs :  dictionnaire
```

- La *"docstring"* commence et se termine par les caractères `"""`
- La *"docstring"* peut contenir plusieurs paragraphes
- Documenter chaque fonction facilite le partage et la réutilisation
- Opérateur *splat* `"*"` pour l'unpacking des *positional arguments* (`*args`)
- Opérateur double *splat* `"**"` pour l'unpacking des *keyword arguments* (`**kwargs`)

Fonctions et muabilité

```
def myfct(x, y):
    x = 1           # Redéfinition locale de x !
    y.append(22)
    print("x is {}, and y is {}".format(x, y))
```

```
In  : a = 10                              # Objet immuable
In  : b = [2, 3]                          # Objet muable

In  : myfct(a, b)                         # Appel de la fonction
Out : x is 1, and y is [2, 3, 22]         # Résultat attendu ...

In  : print("a is {}, and b is {}".format(a, b))
Out : a is 10, and b is [2, 3, 22]        # ...mais b a aussi changé !
```

- Les arguments sont passés par valeurs : python passe l'objet référencé par la variable, pas la variable elle même :
    - si objet immuable : la fonction ne modifie pas la variable passée en argument
    - si objet muable : la fonction modifie la variable passée en argument
- Les fonctions ont une table locale des variables appelée *"local namespace"*
- Un objet défini dans une fonction existe seulement localement dans cette fonction

Syntaxe : `def ...yield ...`

```
def gen():                  # Déclarée comme une fonction classique
    for i in range(3):      # mais yield remplace return
        yield i*i
```

```
In  : type(gen)         # gen est bien une function...
In  : type(gen())       # ...mais l'appel de gen produit un generator
In  : g = gen()         # Le type generator est défini par yield
In  : for i in g:       # On itère sur le generator g
          print(i)      # Retourne 0, puis 1, puis 4
```

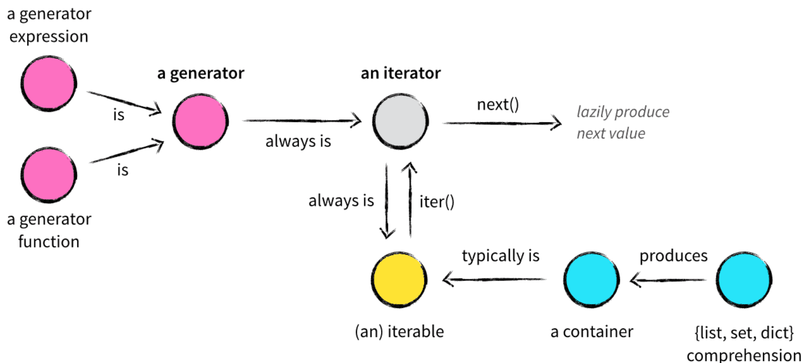
1. Appel de la fonction génératrice : `generator` retourné[2] par la fonction
2. Appel de la méthode spéciale `__next__()` : exécution de la fonction jusqu'au `yield`
3. Une valeur est alors retournée
4. Mise en pause de la fonction jusqu'au prochain appel de `__next__()` : *back to step 1*
5. S'il n'y a plus de valeurs : exception **StopIteration** levée

Avantages du `generator`:

- Génération de valeurs à la demande
- Génération de séquences énormes (même infinies) sans surcharger la mémoire

[2] sans même avoir commencé l'exécution de la fonction

Bilan sur les objets iterables et les itérateurs



Extrait de www.nvie.com

Syntaxe : @decorator

```
def mydecorator(function):
  def wrapper():
     print("Some actions before calling function())"
     function()
     print("Some actions after calling function())"
  return wrapper

@mydecorator                  # Ajoute un décorateur plutôt que...
def myfct():                  #
  print("Wheee!")             #
                              # ...de passer myfct() à mydecorator() :
                              # » myfct = mydecorator(myfct)
In  : myfct()                 # » myfct()
Out : Some actions before calling function()
      Wheee!
      Some actions after calling function()
```

Un decorator, c'est :

- comme une fonction qui modifie le comportement d'une autre fonction
- généralement utilisés pour ajouter du code à des fonctions existantes

Un exemple concret

```
import time

def timing(function):
    """ Outputs running time """
    def wrapper(*args, **kwargs):
        t1 = time.clock()
        function(*args, **kwargs)
        t2 = time.clock()
        return "Running time is  s.".format(round(t2 - t1, 6))
    return wrapper


@timing
def myfct(N):
    for i in range(N):
        do_some_time_consuming_stuff
```

```
In  : myfct(1000)
Out : Running time is 12.34657 s.
```

Les fonctions anonymes

From pixabay

Définition

- Syntaxe : `lambda arg1, arg2, ...: instruction du return`
- Syntaxe alternative aux définitions de fonction destinée aux fonctions "jetables"
- Fonctions `lambda` limitées à une seule instruction

Définition de fonction

```
def cri(n):
    return f"A{n*'a'}h"

» cri(3)
'Aaaah'
```

Fonction lambda

```
» cri = lambda n: f"A{n*'a'}h"
» cri(3)
'Aaaah'
```

```
def s(x, y):
    return x + y

» s(3, 5)
8
```

```
» s = lambda x, y: x + y
» s(3, 5)
8
```

Pourquoi utiliser `lambda` ?

- Pour rien, pour le style !
- Parce qu'elles s'intègrent au corps du code
- Parce qu'elles ont une écriture plus compacte que les `def`

La fonction filter

- `filter(function, iterable)`
- `function` doit retourner `True` ou `False`
- `filter` filtre `iterable` suivant `function`
- `filter` retourne un **itérateur**

Fonction filter

```
» a = filter(lambda x:x>3, range(6))
» list(a)
[4, 5]
```

```
» a = (x for x in range(6) if x>3)
```

La fonction map

- `map(function, iterable)`
- `function` appliquée aux éléments de `iterable`
- `map` retourne un **itérateur**

Fonction map

```
» a = map(lambda x:x**2, range(6))
» list(a)
[0, 1, 4, 9, 16, 25]
```

```
» a = (x**2 for i in range(6))
```

Pourquoi utiliser `map` et `filter` plutôt que des expressions génératrices ?

- Pour rien, pour le style !
- Parce que `map` est overridée dans les modules d'exécution concurrente
- Parce que `lambda` à son propre *namespace*, ce qui peut éviter certains conflits
- Parce qu'elles sont plus rapides que les expressions génératrice si exécutées sans fonction `lambda`.
  Par exemple `map(bin, range(4))` *retourne* `['0b0', '0b1', '0b10', '0b11']`