4 Fonction

FONCTION

Fonction : pourquoi ?

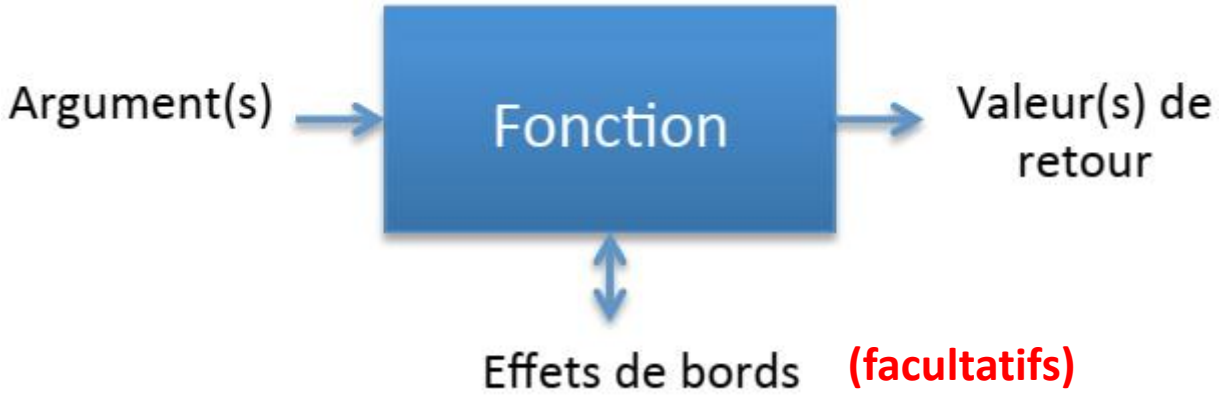
- **But:** **structurer** son code lorsque l'on fait plusieurs fois la même chose (ou presque)

  - Pour qu'il soit plus **lisible** (plusieurs morceaux)
  - Pour qu'il soit plus **facilement modifiable**
  - Pour qu'il soit plus **facile à tester**

FONCTION

Fonction : Principe

- Une suite d'instructions encapsulées dans une « boîte »



- Qui prend **zéro**, **un** ou **des arguments**
- Qui retourne **zéro**, une ou **plusieurs valeurs de retour**
- Et qui contient éventuellement des "**effets de bord**" qui modifient l'environnement.

FONCTION

Exemple

- Dans un exercice de géométrie, on doit souvent calculer la distance entre deux points.

$$\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

- **Arguments : xa, ya, xb, yb**
- **Valeur de retour : distance AB**
- **Effet de bord : afficher les points A et B**

FONCTION

Fonction : Syntaxe

```
def nom_fonction(argument1,…, argumentN) :
    instructions à exécuter
    return valeur de retour
```

- **Note :** le **return** est facultatif, ainsi que les arguments (mais pas les parenthèses!)

Fonction : Appel d'une fonction

- L'appel de la fonction prend la forme :

nomdefonction(expression$_1$, expression$_2$, ... expression$_k$)

- **Exemple 1:** La fonction ***sommeCarre*** suivante retourne la somme des carrées de deux réels x et y :

```
def  sommeCarre ( x, y) :
        z = x**2 + y**2
        return z
```

L'appel de la fonction sommeCarre peut se faire :

```
>>> print(sommeCarre(2,3))
13
```

Note: un appel de fonction peut se faire à l'intérieur **d'une autre fonction**.

FONCTION

- **Exercice 12:** Ecrire une fonction **table(n)** qui permet d'afficher la table de multiplication du nombre n:

L'appel de la fonction *table* peut se faire :

```
>>> table(7)
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x7 = 70
```

```
def  table(n) :
        for i in range(1,11):
                print(i, " x ", n , " = " , i*n)
```

FONCTIONS

- **<u>Execice 13</u>:** Ecrire une fonction ***fact*** qui retourne le factoriel d'un entier passé en paramètre.

```
def  fact(n) :
        p=1
        for i in range(1,n+1):
                p=p*i
        return p
```

- Servez vous de cette fonction est écrire une autre nommée ***comb*** qui permet de calculer coefficient binomial

$$C_n^p = \frac{n!}{p!\,(n-p)!}.$$

```
def  comb(n,p) :
        if  p==n or p==0:
                return 1
        elif p==1:
                return n
        else :
                return  fact(n)//(fact(p)*fact(n-p))
```

FONCTION

Plusieurs valeurs de retour : Un exemple

```
def division(a,b) :
    # renvoie le quotient et le reste
    # de la division de a par b
    quotient=a//b
    reste= a%b
    return quotient, reste


# programme principal
q,r = division(22,5)
print("q=", q, "et r=", r)
```

FONCTION

Docstring: Un exemple

```
def division(a,b) :
      """ Renvoie le quotient et le reste
       de la division de a par b """
      quotient=a//b
      reste= a%b
      return quotient, reste
```

Dans l'interpréteur (ou dans un programme):

```
>>> help(division)
Help on function division in module __main__:

division(a, b)
    Renvoie le quotient et le reste
    de la division de a par b
```

FONCTION

Définir une fonction avec des arguments optionnels: exemple

```
def affiche_pizza(saveur, taille="normale"):
        """ Affiche saveur, taille et prix de la pizza
        """
        print("Pizza", saveur, "taille:", taille)
        if taille=="normale":
          prix=9
        elif taille=="maxi":
          prix=12
        print("Prix", prix, "euros.")
```

➔taille est un argument optionnel ayant comme **valeur par défaut** "normale".

FONCTION

Définir une fonction avec des arguments optionnels: exemple

```
>>> affiche_pizza("4 fromages")
Pizza 4 fromages taille: normale
Prix 9 euros.
>>> affiche_pizza("4 fromages", "maxi")
Pizza 4 fromages taille: maxi
Prix 12 euros.
>>> affiche_pizza("Reine", "normale")
Pizza Reine taille: normale
Prix 9 euros.
```

FONCTION

Les fonctions anonymes

Le mot-clé ***lambda*** en Python permet la création de **fonctions anonymes** (i.e. sans nom et donc non définie par **def**).

```
f = lambda x : x*3
```

```
>>> f ( 3 )
9
```

```
racine= lambda x : x**0.5 if x>=0 else False
```

```
>>> racine(9 )
3
```

FONCTION

Exercice 14: On peut approximer la dérivée d'une fonction **f** en un poit **x0** avec la formule suivante :

$$f'(x0) = (f(x0+h)-f(x0))/h$$

Avec 0<h<1 très petit.

Ecrire une fonction derive(f,x0) qui reçoit en paramètre une fonction f et un réel x0 et qui permet de retourner la valeur f'(x0).

```
def  derive(f,x0,h=1e-10):
            return  (f(x0+h)-f(x0))/h
```

```
>>> f=lambda x: x**2-2
>>> x0=2
>>> derive(f,x0)
4.000000330961484
```

```
fp=lambda f,x0,h: (f(x0+h)-f(x0))/h
```

```
>>> f=lambda x: x**2-2
>>> x0=2
>>> fp(f,x0,1e-5)
4.000010000027032
```

FONCTION

Le mot-clé **None**

- Il existe une valeur constante en Python qui s'appelle **None**. Cela correspond à "rien", "aucune".
- Lorsqu'une fonction n'a pas d'instruction return, elle renvoie la valeur ***None***.

```
def dit_bonjour():
    print("Bonjour!")
    print("Bienvenue")
    # pas de return

# prog. Principal
test=dit_bonjour()
print("Test vaut", test)
```

Affichage :
Bonjour!
Bienvenue.
Test vaut None

FONCTION

Déclaration d'une fonction sans connaître ses paramètres

```python
def f(*args, **kwargs):
    print(args)
    print(kwargs)

f(1, 3, 'b', j = 1)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Johri> & python c:/Users/Johri/Desktop/test.py
(1, 3, 'b')
{'j': 1}
PS C:\Users\Johri>
```

FONCTION

Passage des paramètres en Python :

Intéressons-nous au problème suivant :

Si on modifie une variable en paramètre d'une fonction à l'intérieure, est-elle réellement modifiée après l'appel ?

Exemple 1 :

Considérons la fonction **ajoute** suivante :

```
def   ajoute( a ) :
            a = a + 1
# appel
>>> b = 5
>>> ajoute(b)
>>> print(b)
5
```

a est un arguments de types **immuables**

FONCTION

Passage des paramètres en Python :

En Python, le passage des paramètres est comparable à une affectation. Le fil d'exécution ressemble donc à ceci :

```
>>> b = 5
# exécution d'ajout ( b )
a = b
a = a + 1
destruction de a
# retour au programme principal
print(b)
5
```

Exemple 1 :

```
def   ajoute( a ) :
            a = a + 1
# appel
>>> b = 5
>>> ajoute(b)
>>> print(b)
5
```

Tout se passe comme si le paramètre était ***passé par valeur*** : **une copie du contenu de la variable est recopiée de b vers a et c'est la copie qui est modifiée, pas l'original.**

Variables globales et locales

En Python, on distingue deux sortes de variables : les **globales** et les **locales**.

Par exemple, dans le programme suivant, **x** est une variable **globale** :

```
>>> x = 7
>>> print(x)
```

À l'inverse, la variable **y** dans la fonction **f** suivante est **locale** :

```
def  f( ):
        y = 8
        return  y
```

Après l'appel de la fonction f, la variable locale y disparaît.

En particulier, l'instruction suivante échoue en indiquant que la variable y n'est pas définie :

```
>>> print(y)
Error
```

Variables globales et locales

Remarque

- Si l'on veut accéder à une variable **globale** à l'intérieur d'une fonction, on utilise le mot-clé **global** en Python.

Exemple 1

Par exemple pour écrire une fonction qui réinitialise la variable globale **x** à 0, alors il ne faut pas écrire :

```
x = 7
def    reintialise(x) :
            global x
            x=0
```

Exemple 2

```
def  f( ):
      global a
      a = a + 1
      c = 2 * a
      return  a + c
```

FONCTION

Exercice 15: Le code d'une photocopieuse est un numéro N composé de 4 chiffres. Les codes corrects ont le chiffre le plus à droite égal au reste de la division par 7 de la somme des trois autres chiffres. Ainsi, le code 5733 est incorrect car 5+7+3 = 15 et 15 mod 7 = 1 ≠ 3 tandis que 5731 est correct. Le but de cet exercice est de créer une fonction qui prend en entrée le code et qui renvoie "VALIDE" ou "NON VALIDE".

```
1   def code(N):
2       s=0
3       for i in range(3,0,-1):
4           r=N//10**i
5           N=N%10**i
6           s+=r
7       if s%7==N:
8           return "VALIDE"
9       return "INVALIDE"
10
```