THÈME 5: FONCTIONS (BASES)

Notion du thème:

- Fonctions:
  - Principe
  - Syntaxe
  - Portée des variables

Fonctions : 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**

Un exemple

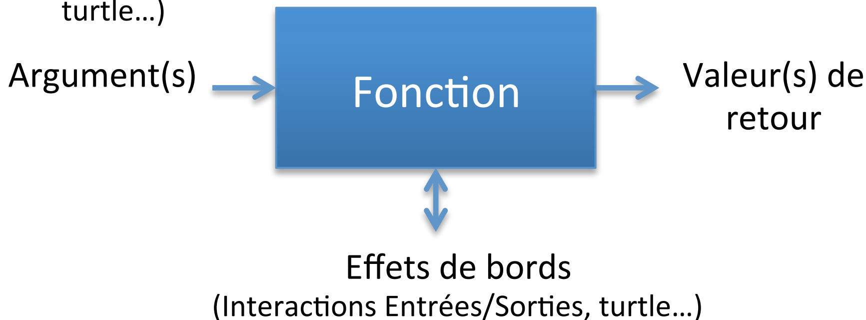
```
import turtle

def carre(cote) :
    # trace un carre de taille egale a cote
    i = 1 # compteur du nb de cotes
    while i <= 4 :
        turtle.forward(cote)
        turtle.right(90)
        i=i+1
```

Un exemple

```
# programme principal

carre(100)
turtle.up()
turtle.forward(130)
turtle.down()
carre(50)
```

Principe

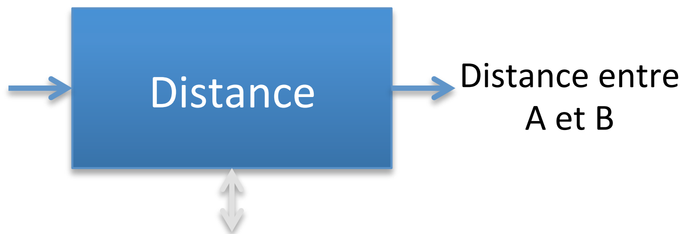
- Une suite d'instructions encapsulées dans une « boîte »
- Qui prend zéro, un ou des **arguments**
- Qui retourne zéro, une ou plusieurs **valeurs de retour**
- Et qui contient éventuellement des **"effets de bord"** qui modifient l'environnement (interactions entrées/sorties, turtle…)



Exemple

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

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



Exemple de la distance

Fonction à définir au dessus de votre programme principal:

```
def distance(absA, ordA, absB, ordB ) :
    d=(absB-absA)**2 + (ordB-ordA)**2
    d=d**(1/2)
    return d
```

4 **arguments**: absA, ordA, absB, ordB
1 **valeur de retour** de type float (d)
Pas d'effets de bord

Exemple de la distance

```
if __name__=="__main__": # prog. principal

    print(distance(1, 2, 1, 5))
    xA=2
    yA=3
    z=distance(xA, yA, 0, 0)
    print("Distance de (0,0) à A :", z)
```

Ou directement dans l'interpréteur:

```
>>> distance(0, 1, 3, 5)
5.0
```

Python Tutor

Start shared session
What are shared sessions?

Python 3.6

```
 1  def distance(absA, ordA, absB, ordB ) :
 2      d=(ordB-ordA)**2+ (absB-absA)**2
 3      d=d**(1/2)
 4      return d
 5
 6  # prog. principal
 7  if __name__=="__main__":
 8      xA=2
 9      yA=3
10      z=distance(xA, yA, 0, 0)
11      print("Distance de (0,0) à A :", z)
```

Edit code | Live programming

line that has just executed
next line to execute

NEW! Click on a line of code to set a breakpoint. Then use the Forward and Back buttons to jump there.

<< First | < Back | Step 10 of 11 | Forward > | Last >>

Print output (drag lower right corner to resize)

Frames | Objects

Global frame

distance	
xA	2
yA	3

function
distance(absA, ordA, absB, ordB)

distance

absA	2
ordA	3
absB	0
ordB	0
d	3.6056
Return value	3.6056

Syntaxe d'une nouvelle fonction

```
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!)

Programme principal

À placer en-dessous de la définition des fonctions.

```
if __name__=='__main__': # programme principal
     instructions à exécuter
```

Note : cette ligne est facultative dans les TD/TP mais **obligatoire dans Caseine**.

Appel d'une fonction

`nom_fonction(argument1, argument2, …)`

➔ vaut la valeur de retour de la fonction (s'il y en a)

Exemple, en supposant que les fonctions `distance` et `carre` ont été définies au-dessus.

```
z=distance(2, 3, 4, 5)
print(z)
carre(50)   # pas de valeur de retour,
            # mais des effets de bord (turtle)
```

Note: un appel de fonction peut se faire **dans le programme principal** mais aussi à l'intérieur **d'une autre fonction**.

Appel d'une fonction depuis une autre fonction

En supposant que la fonction `carre` a été définie au-dessus.

```
def deplace_sans_tracer(distance):
    up()
    forward(distance)
    down()

def ligne_carres(nb_carres, cote):
    i=0 # compte le nb de carres déjà tracés
    while i<nb_carres:
        carre(cote) # appel a la fonction carre
        deplace_sans_tracer(cote+10) # appel
        i=i+1
```

Fonction sans argument

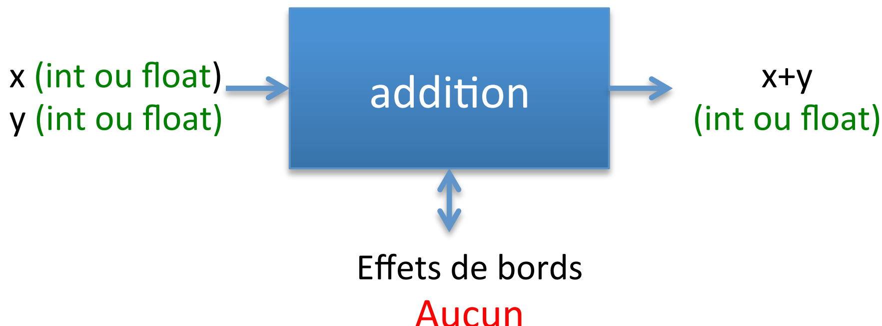
```
import turtle
def carre_standard():
      i = 1 # compteur du nb de cotes
      while i <= 4 :
            turtle.forward(100)
            turtle.right(90)
            i=i+1

def demander_nom():
     nom=input("Quel est ton nom?")
     return nom
```

Exemple d'appels:

```
name=demander_nom() # pas d'argument
carre_standard() # pas d'argument ni valeur de retour
```

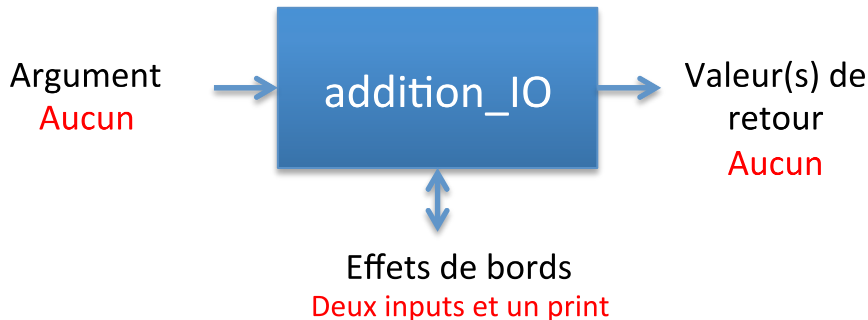
Différence
Effets de bord vs. Valeur de retour

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



Différence
Effets de bord vs. Valeur de retour

```
def addition_IO():
    x=float(input("x ?"))
    y=float(input("y ?"))
    print(x+y)
```



Mise en situation

Trois équipes:

- Une équipe fonction `moyenne`
- Une équipe fonction `ecart_plus_grand_que`
- Une équipe programme principal
- Communication entre les équipes par
  - Appel de fonction (avec les arguments)
  - Valeur de retour

  (modélisée par des papiers)
- Le tableau sert uniquement pour les `print` et `input`

Fonctions utilisées

```
def moyenne(x,y):
    # renvoie la moyenne de x et y
    resultat=(x+y)/2
    return resultat

def ecart_plus_grand_que(x,y,seuil):
    # calcule l'ecart entre x et y
    # et renvoie True si l'ecart est plu grand que seuil, False
      sinon
    if x>y:
        ecart=x-y
    else:
        ecart=y-x
    # ecart contient la valeur absolue de x-y
    # on aurait pu faire: ecart=abs(x-y)
    if ecart>=seuil:
        return True
    else:
        return False
```

Prog. Principal utilisé

```
if __name__=="__main__": # prog. principal
    # demande deux nombres, affiche leur moyenne,
    # et recommence si les nombres étaient proches
    seuil=2
    continuer=True
    while continuer:
        a=float(input("a=?"))
        b=float(input("b=?"))
        m=moyenne(a,b) # appel de la fonction
        print("Moyenne :", m)
        if ecart_plus_grand_que(a,b,seuil): # appel de la
 fonction
            continuer=False
        else:
            print("Ecart plus petit que", seuil)
    print("Fin car l'écart était plus grand que", seuil)
```

Portée des variables

Chaque fonction a son propre "lot" de variables auquel elle a le droit d'accéder (cf Python Tutor).

Une variable

- **créée ou modifée** dans le corps d'une fonction ,
- ou qui contient un **argument** de la fonction

est dite **locale**, et ne sera pas accessible depuis le programme principal, ni depuis une autre fonction.

Variable locale: exemple

```
def moyenne(x,y):
    # renvoie la moyenne de x et y
    resultat=(x+y)/2
    return resultat

if __name__=="__main__": # prog. principal
    a=5
    b=6
    m=moyenne(a,b)
    print(m) # affiche 5.5
    print(resultat) # provoque une erreur
```

➔ NameError: name 'resultat' is not defined

Portée des variables

Une variable (de type int, float, bool ou str) définie dans le programme principal **ne peut pas être modifiée** par une instruction qui se trouve **à l'intérieur d'une fonction***.

Cela ne provoque pas d'erreur mais cela **crée une nouvelle variable locale** portant le **même identificateur (nom)** que l'autre variable.

*1. Sauf si le mot-clé `global` est utilisé, mais nous ne le ferons pas.
2. Il y aura des subtilités lorsque nous verrons les listes.

Portée des variables: exemple

```
def moyenne(x,y):
    # renvoie la moyenne de x et y
    resultat=(x+y)/2
    test=resultat # crée une nouvelle variable test
    return resultat

if __name__=="__main__": # prog. principal
    a=5
    b=6
    test=0
    m=moyenne(a,b)
    print("m =", m) # affiche "m=5.5"
    print("test =",test) # affiche "test=0"
# car test n'est pas modifié par l'appel de fonction
```

Portée des variables

Les variables définies dans le programme principal sont accessibles en lecture seule depuis l'intérieur d'une fonction mais ce comportement est ***dangeureux*** *car très subtil*.

On ne l'utilisera donc pas (sauf éventuellement pour des variables "constantes", initialisées une fois au début du programme et jamais modifiées ensuite.)

On préférera passer en arguments toutes les valeurs nécessaires.

À ne pas faire: exemple

```
def decalage(s):
    # renvoie la chaine s préfixee de n tirets
    espaces="-" * n
    resultat=espaces+s
    return resultat

if __name__=="__main__": # prog. principal
    n=5
    test=decalage("toto")
    print(test)
    n=10
    test=decalage("maison")
    print(test)
```

Il vaut mieux passer **n** en argument

Correction de l'exemple

```
def decalage(s, n):
    # renvoie la chaine s préfixee de n tirets
    espaces="-" * n
    resultat=espaces+s
    return resultat

if __name__=="__main__": # prog. principal
    n=5
    test=decalage("toto", n)
    print(test)
    n=10
    test=decalage("maison", n)
    print(test)
```

n est maintenant un argument (donc une variable locale)

Portée des variables

Résumé:

- Une variable créée ou modifiée **dans une fonction** est **locale** (= elle n'existe que dans la fonction).
- Une variable (de type int, float, str ou bool) du **programme principal** ne peut **pas être modifiée** à **l'intérieur d'une fonction**.
- On passera en **argument** de la fonction toutes les **valeurs nécessaires**.

- Ces slides ont été réalisés par:
  - Amir Charif
  - Lydie Du Bousquet
  - Aurélie Lagoutte
  - Julie Peyre
- Leur contenu est placé sous les termes de la licence **Creative Commons CC BY-NC-SA**