# Cours – Les fonctions et les conditions

Objectifs :

- ⇨ Reprendre les notions de fonction et d'exécution conditionnelle
- ⇨ Comprendre le passage d'arguments à une fonction
- ⇨ Comprendre la notion de portée de variable
- ⇨ Voir la notion d'évaluation paresseuse dans l'évaluation des expressions logiques

## I – Les fonctions et les procédures

### 1) Intérêt des fonctions

Les fonctions permettent :

- de simplifier l'écriture et la lecture du code source du programme en .....
- de ..... les parties du programme d'une façon plus claire pour un esprit humain, ce qui facilite son analyse et la recherche d'éventuelles erreurs par des tests séparés.
- de ..... certains sous-programmes utiles pour plusieurs autres (exemple : la fonction « `print` » n'a pas besoin d'être réécrite par chaque programmeur et son code source a été « masqué »)
- de ..... de programmation lors de l'écriture d'un programme complexe.
- de faire de la programmation récursive (détaillé dans un prochain cours)

### 2) Arguments de la fonction

On a déjà vu pendant le TP qu'une fonction pouvait prendre un nombre quelconque d'arguments séparés par des virgules. En réalité une même fonction peut même prendre un nombre d'argument variables suivant l'usage que l'on en fait. C'est ce que l'on appelle le ***polymorphisme*** en programmation.
La fonction `color` du module `turtle` par exemple peut prendre :
0 argument : dans ce cas elle renvoie juste les couleurs actuellement utilisées pour le trait et le remplissage
1 argument : change les deux couleurs (trait et remplissage) à la valeur donnée en argument
2 arguments : change la couleur du trait à celle du 1er argument et celle de remplissage à celle du 2ème
3 arguments : comme avec 1 argument sauf qu'ici les 3 valeurs représentent les composantes rouge, vert, bleu de la couleur

On ne détaillera pas ici le mécanisme permettant de gérer un nombre d'argument variable.

A l'intérieur de la parenthèse de la ligne de déclaration (`def`) de la fonction, on indique les arguments de la fonction avec le nom de la variable qui y sera associée. L'ordre dans lequel on indique les arguments est important et devra être respecté.

Exemple :

```
def f(a, b, z):
    y = a*a + 2*b - z
    return y
print(f(2,3,5))
```

```
  2  3  5
f(a, b, z):
    y = 2*2 + 2*3 - 5
    return y
```

Lorsque l'interpréteur python évalue l'expression `f(2,3,5)`, il va mettre la première valeur (`2`) dans une variable `a`, la deuxième (`3`) dans une variable `b` et la troisième (`5`) dans une variable `z`, puis il va exécuter le programme de la fonction f.

Il y a donc des variables (`a`, `b` et `z`) qui sont créées au moment de l'appel de la fonction f et un bout de programme (le code de la fonction) qui est exécuté. Cela se passe comme lorsqu'on importe un module ou que l'on exécute un nouveau programme.

Il est possible de spécifier des ..... à la fonction en respectant certaines conditions :

- Ces paramètres doivent avoir une valeur par défaut qui sera utilisée si l'argument n'est pas fourni.
- Ces paramètres doivent être indiqués en dernier (après les paramètres obligatoires).

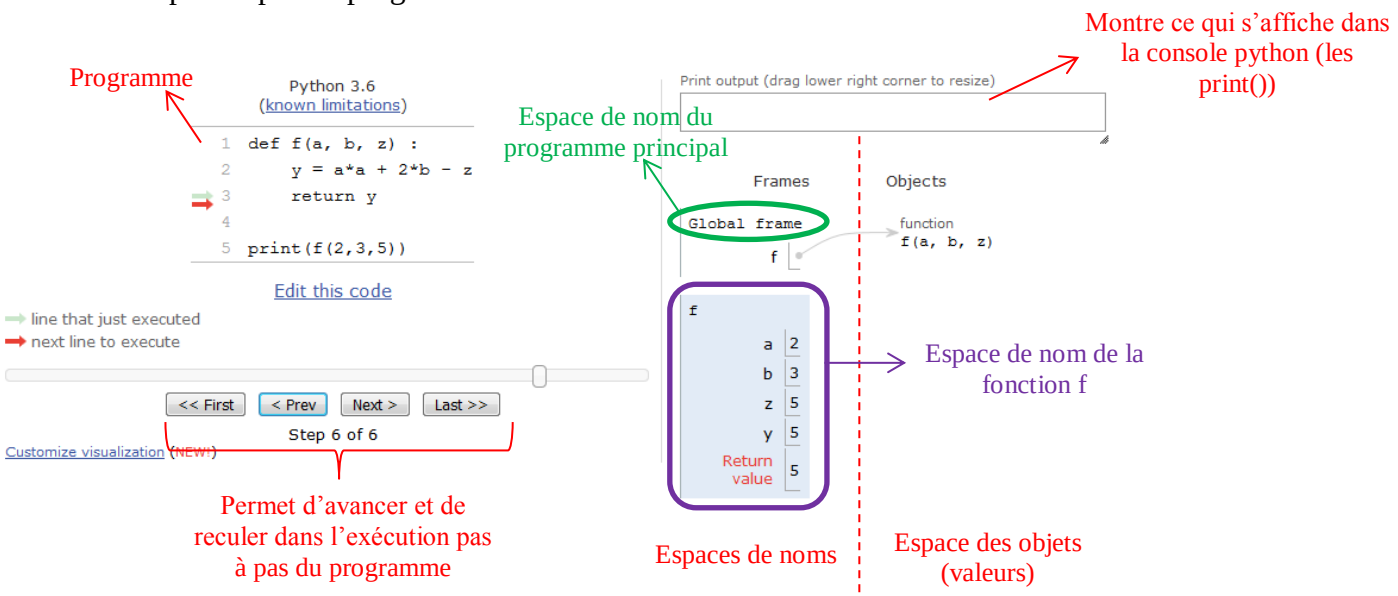
Lors de l'appel de la fonction :
Les paramètres obligatoires doivent être donnés en premier dans le bon ordre, puis les paramètres optionnels peuvent être précisés dans n'importe quel ordre mais en les faisant toujours précéder de leur nom suivi de '='.

Remarque : C'est le cas de la fonction « `print` » que l'on a déjà vu qui comporte plusieurs arguments optionnels, notamment `sep` et `end`.

```
def affiche_carte(x, y, zoom=1, color=True):
    ...

affiche_carte(0,0)    # Affichage initial
c = int(input("Numéro carte ? "))
if c==1:
    affiche_carte(25, 0, zoom=3)
if c==2:
    affiche_carte(-5, 10, color=False)
if c==3:
    affiche_carte(80, 2, zoom=4, color=False)
if c==4:
    affiche_carte(7, -20, color=False, zoom=8)
```

L'appel de la fonction va créer un nouvel espace de nom propre à la fonction. Cet espace de nom va exister pendant tout le temps d'exécution de la fonction et sera détruit lorsque l'on quittera la fonction. Seule la valeur de retour sera transmise au programme appelant.
Pour mieux comprendre ce qui se passe, on va utiliser le site www.pythontutor.com qui permet de visualiser les variables manipulées par un programme.




## 3) Portée des variables

Examinons ce qui se passe lorsque l'on appelle plusieurs fois la même fonction ou lorsqu'on utilise le même nom de variable dans le programme et dans la fonction ou dans deux fonctions différentes.

Application 1 :
Copier-coller le code ci-contre dans python tutor et exécutez-le pas à pas. Notez ensuite vos conclusions sur le fonctionnement des variables ci-dessous.

```
def f(a, b, z):
    y = a*a + 2*b - z
    return y

def g(x, y):
    print("A l'intérieur de g, b=",b)
    a = x*y + 1
    print("et a=",a)
    return a - 2

a = f(2,3,5)
b = f(-1, 0, 2)
t = g(a, 4)
print("a=", a, "b=", b, "t=", t)
```

Conclusion :

Les variables qui sont définies dans le programme principal (en dehors de toute fonction) sont appelées ..... , celles définies dans les fonctions sont les ..... .

Définition : La **portée d'une variable** correspond aux endroits du programme où on peut utiliser cette variable.

Lorsque l'interpréteur python tombe sur un nom de variable ou de fonction, il essaye de trouver la variable correspondante dans les espaces de noms en suivant la règle **LEGB** :

Il cherche d'abord dans l'espace de nom **L**ocal (celui de la fonction dans laquelle il se trouve), puis s'il ne trouve pas, il essaye de trouver dans les espaces de noms **E**nglobant la fonction dans laquelle il se trouve : celui de la fonction qui a appelé la fonction, puis la fonction qui a appelé celle-ci, etc… Si la variable n'y est toujours pas, il cherche dans les variables **G**lobales. S'il n'a toujours rien trouvé, il cherche dans le module **B**uiltins (Celui-ci contient les variables `False`, `True` et `None`). Enfin si rien n'a fonctionné, python déclenche l'erreur « `NameError` ».

Application 2 :

1) Dans le code ci-contre, quel est le type (Locale, Englobante ou Globale) de chaque variable (`w`, `c`, `a` et `d`) au moment où la ligne 5 s'exécute ?

2) Quelles seront les valeurs des variables a et d à la fin du programme ?

```
1  def f(x, a, b):
2      c = 2*b
3
4      def g(x, a):
5          w = c + a -d
6          return 2*w
7
8      y = a*g(2*x,-c) + b
9      return y
10
11 d = 1
12 a = f(2,3,4)
13 print("a=", a, "d=", d)
```

Copier-coller le code dans python tutor et exécutez-le pas à pas pour contrôler vos réponses.

On remarque qu'il n'est pas possible pour une fonction de modifier la valeur d'une variable globale ou englobante car l'affectation d'une valeur a une variable crée automatiquement cette variable dans l'espace de nom de la fonction et masque ainsi la variable le plus haut niveau.
Il est cependant possible d'indiquer à l'interpréteur que l'on souhaite modifier et manipuler une variable globale ou englobante à l'intérieur d'une fonction. On doit pour cela, avant d'utiliser la variable, la déclarer à l'aide du mot clé `global` (pour les variables globales) ou `nonlocal` (pour les variables englobantes).

```
def f(x, y):
    global a
    a = 2
    return x*y

a = 1
b = f(2,3)
print("a=", a, "b=", b)
```

→ On obtient a = 2 et b = 6

**Ces mécanismes (portée des variables, masquage, variables globales et locales) sont importants à bien comprendre car ils peuvent être à l'origine de nombreuses erreur de programmation qui sont souvent difficile à repérer.**

### 4) Cas des procédures

On a vu dans le TP qu'une **procédure** est tout simplement le nom que l'on donne à une **fonction qui ne renvoie rien**. Si elle ne renvoie rien, cela ne signifie pas pour autant qu'elle ne fait rien, mais qu'elle **a des effets de bords** qui nous intéressent. C'est le cas de la fonction `print()` que nous avons déjà vu et qui ne renvoie rien, mais qui a pour effet de bord d'afficher quelque chose sur la console.
Lorsque python arrive à la fin du bloc de code d'une fonction sans rencontrer le mot-clé « `return` », il sort tout de même de la fonction (ou plutôt de la procédure), et renvoie « `None` ». En d'autres termes, l'instruction « `return` » est optionnelle pour les procédures et il est équivalent de ne rien mettre et de mettre « `return None` » ou « `return` » à la fin d'une procédure. L'intérêt de l'instruction return est alors de permettre de quitter la procédure, même si la fin du code n'est pas atteinte (on parle alors de **sortie anticipée**).

```
def f(a):
    if a == 0:
        return    # sort si nul
    print("Valeur non-nulle")
    return        # optionnel
```

# II - L'exécution conditionnelle

On a déjà vu lors du TP la syntaxe et l'intérêt des instructions d'exécution conditionnelle. Reprenons un exemple et essayons d'aller voir d'un peu plus près le fonctionnement.

Application 3 :
Ecrire un programme qui demande à l'utilisateur comment il va. Si la réponse est « Bien » ou « bien », il répond « C'est la réponse que j'attendais !». Si la réponse est « Bof », « bof », « Mal » ou « mal », le programme répond « Soignez-vous bien. ». Si la réponse est n'importe quoi d'autre il répond « Je ne comprends pas votre réponse. ».

### 1) Optimisation

Dans l'application précédente, si on teste si la réponse est « Bien » ou « bien », python doit à la base exécuter 2 tests : `reponse == "Bien"` et `reponse == "bien"`. Mais puisqu'il s'agit d'un OU logique entre les deux, il suffit qu'une des deux soit égale à True pour que l'expression complète vaille True (voir tableau de vérité de la fonction OU ci-contre).
Du coup, si le premier test donne `True`, il est inutile de faire le deuxième puisqu'on sait déjà que le résultat global sera `True`.

Tableau de vérité de la fonction OU (**or**)

| a | b | a **or** b |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

C'est ce que fait python et qui porte le nom d' ..... : dès que l'un des termes d'une série de OU logique est vrai, on s'arrête et le reste des tests n'est pas fait.
Ceci peut avoir son importance, notamment si le test a des effets de bords.

Application 4 :
1) Qu'affiche le programme ci-contre ?

```
def test(a):
    print(a)
    if a == 0:
        return True
    return False

if test(2) or test(0) or test(5):
    print("Au moins un de nul")
```

2) Qu'afficherait-il si python évaluait tous les termes ?

3) D'après vous, qu'en est-il de l'évaluation d'une série de ET logiques ?

Tableau de vérité de la fonction ET (**and**)

| a | b | a **or** b |
|---|---|---|
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

```
if a == 0 or (b != 0 and a % b == 0):
    print("divisible")
else:
    print("non divisible")
```

4) Pourquoi le programme ci-contre ne provoque-t-il pas d'erreur lorsque `b` vaut `0` ?

## 2) Précisions sur la syntaxe

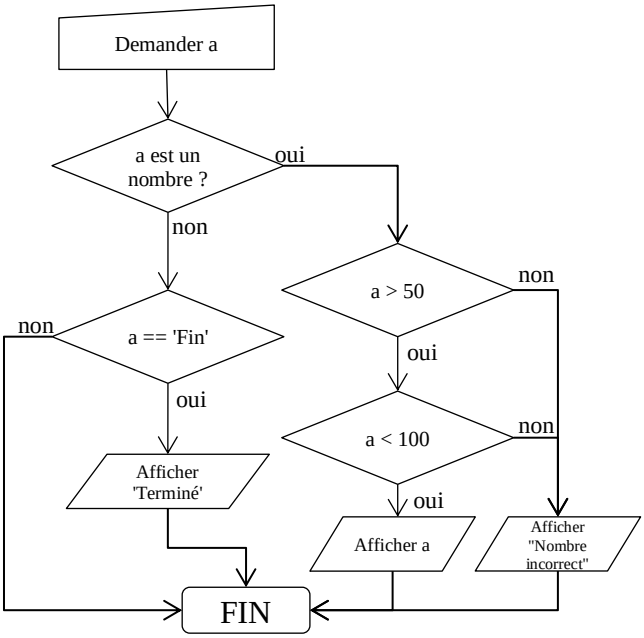
Lorsqu'on a des instructions `if`, `elif` et/ou `else`, python n'exécutera qu'un seul des blocs de code et dans tous les cas reprendra après la série des `elif` (et du `else`) éventuels à la ligne ou l'indentation reprend celle du `if` de départ (et des `elif` et `else`).

Ce point est appelé le ..... de la série de tests conditionnels. C'est ici que l'exécution du programme reprend une fois le bon embranchement pris.

```
1  reponse = input("Comment allez-vous ? ")
2  if reponse == "Bien" or reponse == "bien":
3      print("C'est la réponse que j'attendais !")
4  elif reponse == "Bof" or reponse == "bof" \
5    or reponse == "Mal" or reponse == "mal":
6      print("Soignez-vous bien.")
7  else:
8      print("Je ne comprends pas.")
9  print("Fin du programme")
```

Point de jonction (ligne 9)

Les instructions conditionnelles peuvent être imbriquées et mener ainsi à une situation parfois complexe. Dans ce cas, on peut essayer de schématiser la situation avec un organigramme (ou algorigramme) comme celui ci-dessous.




```
a = input("Que vaut a ?")
if a.isnumeric():
    if int(a) > 50:
        if int(a) < 100:
            print(a)
        else:
            print("Nombre incorrect")
    else:
        print("Nombre incorrect")
else:
    if a == 'Fin':
        print("Terminé")
```

Dans les cas où il n'y a qu'une seule instruction très courte qui suit le test, on peut la mettre directement à la suite des « : » sans avoir à passer à la ligne et à indenter. A utiliser avec parcimonie car le code peut devenir moins lisible.

Exemple :

```
if a > 10: print("a est trop grand !")
if b == 4 : a = 0
if c == "Fin": return
```

# III - Exercices

## Exercice 1

Définir une fonction `test_pythagore` qui prend trois entier a, b et c en arguments et renvoie un booléen indiquant si $a^2 + b^2 = c^2$.
*Pour vérifier : `test_pythagore(3,4,5)` doit renvoyer `True` ; `test_pythagore(1,2,3)` doit renvoyer `False`*

## Exercice 2

Définir une fonction `valeur_absolue` qui prend un nombre x en argument et renvoie sa valeur absolue.

## Exercice 3

Ecrire une fonction `max2` qui prend en arguments deux nombres et renvoie le plus grand des deux. Ex : `max(5,18)` renvoie `18`.

## Exercice 4

Ecrire une fonction `est_bissextile(a)` qui prend en argument une année `a` et renvoie `True` si l'année est bissextile et `False` sinon. On rappelle qu'une année bissextile est une année multiple de 4 mais pas de 100 ou multiple de 400.

## Exercice 5

Ecrire une fonction `nb_jours_annee(a)` qui prend en argument une année `a` et renvoie le nombre de jour dans l'année `a` en utilisant la fonction de l'exercice précédent.

## Exercice 6

Ecrire une fonction `nb_jours_mois(a, m)` qui prend en argument une année `a` et un mois `m` et renvoie le nombre de jour dans le mois `m` de l'année `a` en utilisant la fonction `est_bissextile`. On suppose que le mois `m` est un entier entre 1 (janvier) et 12 (décembre).

## Exercice 7

Ecrire une fonction `nb_jours(j1, m1, a1, j2, m2, a2)` qui prend en argument deux dates (`j1`, `m1` et `a1` pour la première, et `j2`, `m2` et `a2` pour la deuxième et renvoie le nombre de jours compris entre ces deux dates.
*Pour vérifier : Utiliser le site [https://www.ephemeride.com/calendrier/deux_dates/](https://www.ephemeride.com/calendrier/deux_dates/)*