Les fonctions

Les **fonctions** permettent de donner un nom à une partie de votre code dans le but de pouvoir les réutiliser plus tard, plusieurs fois et à différents endroits.
Pour créer une fonction - on dit définir une fonction - on utilise le mot clé **`def`** suivi du nom de la fonction et de parenthèses. Les instructions incluses dans la fonction sont indentées par rapport à sa définition.

```
def triangle():                      # Définition de la fonction triangle
    print('*')                       # -- Instructions
    for i in range(5):               #    contenues
        print('*' +  ' '*i + '*' )   #    dans la
    print('*******')                 #    fonction. --
```

Créer une fonction n'exécute pas ses instructions. Pour exécuter les instructions contenues dans la fonction, il faut l'appeler. Pour faire appel à une fonction, on la nomme avec des parenthèses.

```
triangle()        # Appel à la fonction, ses instructions sont exécutées
```

Il est possible de transmettre des informations à une fonction lors d'un appel. Pour cela on définit des variables spécifiques à la fonction, appelées **paramètres de la fonction** et les valeurs que prendront ces paramètres seront appelées **arguments**.

```
def somme(a, b):    # Deux paramètres, a et b
    print(a + b)    # On les utilise ici

somme(1, 2)         # appel à la fonction somme avec a = 1 et b = 2
```

Pour les types de bases (`int`, `float`, `str`, `bool`), passer une variable en argument d'une fonction copie sa valeur. Ainsi, modifier la valeur d'un paramètre dans une fonction ne modifie pas la variable d'origine.

```
a = 0
def foo(n) :
    n = n + 1
    print(n)
foo(a)              # la valeur de a est copiée, affiche 1
print(a)            # affiche 0
```

Les variables créées dans une fonction ou les paramètres de la fonction ne sont définis que dans celle-ci, pas dans le reste du code. On dit d'une variable qu'elle est **locale** à la fonction. Une variable placée dans le bloc principal est dite **globale**, elle est accessible partout dans le code.
Pour éviter les bugs, on ne nomme pas une variable local avec le même nom qu'une variable globale.

```
m = 0          # Variable global
def foo(n) :   # n est un parametre, elle est locale à foo
    print(n)   # --OK-- car n est locale a foo
    print(m)   # --OK-- car m est globale
foo(1)
print(m)       # --OK-- car m est globale
print(n)       # --erreur-- car n n'est locale qu'à foo
```

Le but d'une fonction est le plus souvent de **renvoyer une valeur**. Cette action est faîte grâce au mot clé **return**.

```
def somme(a, b) :
    return a + b         # renvoie le résultat de a + b
resultat = somme(5, 2)   # resultat contient 7
print(resultat)          # affiche 7
```

Une fonction pour qui ne possède pas d'instructions `return` retourne par défaut la valeur `None`.

L'exécution d'une instruction `return` arrête l'exécution de la fonction et revient au code appelant. Il en découle que du code de fonction placé après un `return` n'est jamais exécuté et qu'une fonction ne peut retourner qu'une valeur.

Lorsque des fonctions s’appellent entre elles, l’appel à une nouvelle fonction entraîne la mémorisation du contexte de la fonction appelante. Le contexte de la fonction comprend la valeurs de ses paramètres, de ses variables, l’instruction courante. Il permet de reprendre l’exécution de la fonction là où on l’avait laissé et dans les mêmes conditions.
Ce contexte est stocké sur une **pile d'exécution**. On parle de pile, car les exécutions successives "s'empilent" les unes sur les autres.

```
# Le code suivant ...
def a():
     b(1)
     c(2)
     b(3)

def b(n) :
     c(2*n)

def c(n):
     print(n)

a()
```

```
# ... appelle successivement les fonctions
# a()
#     b(1)
#         c(2)
#             print()    affiche 2
#     c(2)
#         print()    affiche 2
#     b(3)
#         c(6)
#             print()    affiche 6
```

Dans le code ci-dessus, les paramètres des fonctions `a`, `b` et `c` s’appellent tous `n`. Chaque fonction a sa propre version de `n` avec sa valeur associée, l’appel a une autre fonction utilisant un `n` n’écrase pas la version courante, elle fait partie du contexte stocké sur la pile.

`print`, `input`, `int`, `float`, `range`, `randint`, `sleep` sont des fonctions fournies par Python.
La liste complète des fonctions est disponible ici https://docs.python.org/fr/3.13/library/index.html.

Exercice 1 : Predict, run, investigate.

Sans exécuter, qu'affichent les codes suivants ? Vérifier en exécutant le code.

```
# Code 1

def mystere1():
    print("bonjour")
```

```
# Code 2

def mystere2():
    print("bonjour")

mystere2()
```

```
# Code 3

def mystere3(a):
    print(a*3)

print(mystere3(10))
```

```
# Code 4

def mystere4(a, b):
    return a == b

print(mystere4(2,3))
print(mystere4(4,4))
```

```
# Code 5

def mystere5(a, b):
    return a // (b - 1)

print(mystere5(6,3))
print(mystere5(9,4))
```

```
# Code 6

def mys6(a, b):
    s = 0
    for _ in range(b):
        s = s + a
    return s

print(mys6(4,3))
```

Exercice 2 : Predict, run, investigate.

Sans exécuter, donner la pile d'appels des programmes suivants et les valeurs affichées.
Vous pourrez vérifier votre prédiction en exécuter votre code instruction par instruction à l'adresse suivante :
https://pythontutor.com/render.html#mode=edit

```
# Code 1

def a(x):
    b(x+1)
    c(x+2)
    b(x+3)

def b(x):
    c(x * 2)

def c(x):
    print(x-4)

a(1)
```

```
# Code 2

def a(x):
    return x + 1

def b(x):
    return x * 2

def c(x):
    return x / 4

print(a(b(c(a(1)))))
```

Exercice 3 : Modify.

1) En modifiant le code 4 de l'exercice 1, écrire une fonction `div10AndNot3(n)` qui reçoit un nombre entier en paramètre et qui renvoie `True` si cet entier est divisible par 10 mais pas par 3 et `False` sinon.

2) En modifiant le code 4 de l'exercice 1, écrire une fonction `diff3(n, m, o)` qui reçoit trois nombres entiers en paramètre et qui renvoie `True` si au moins deux de ces nombres sont égaux et `False` sinon.

3) En modifiant le code 5 de l'exercice 1, écrire une fonction `moyenne(a,b)` qui renvoie la moyenne des valeurs `a` et `b` passées en paramètres.

4) En modifiant le code 6 de l'exercice 1, écrire une fonction `puissance(x, k)` qui renvoie `x` a la puissance `k`. On utilisera une boucle `for` pour faire le calcul. On suppose `k` est un entier positif ou nul et on rappelle que `x**0 = 1`.

Exercice 4 : Make.

1) Écrire une fonction `table(n)` qui affiche la table de multiplication de `n` avec une boucle `for`.

2) Écrire une fonction `fact(n)` qui retourne le n!. On utilisera une boucle `for` pour faire le calcul. On suppose `n` est un entier positif et on rappelle que 1! = 1 et n! = n × (n -1) × (n -2) × ... × 2 × 1. Ainsi 5! = 5×4×3×2×1 = 120.

3) Écrire une fonction `max2(a, b)` qui renvoie le plus grand des deux entiers `a` et `b`.

4) En se servant de la fonction `max2` de l'exercice précédent, compléter la fonction `max3(a, b, c)` qui renvoie le plus grand des trois entiers `a`,`b` et `c`.

```
def max3(a, b, c):
    return ...
```

5) Écrire une fonction `convertToSec(h, m, s)` qui prend en paramètre un temps donné en heures minutes, secondes et qui retourne ce temps converti en secondes.

6) Écrire une fonction `bissextile(a)` qui renvoie `True` si l'année `a` est une année bissextile, `False` sinon. On rappelle qu'une année bissextile est une année multiple de 4 mais pas de 100, ou multiple de 400.

7) Écrire une fonction `nbjoursannee(a)` qui renvoie le nombre de jours de l'année `a`, appellant la fonction `bissextile(a)` de l'exercice précédent pour savoir si l'année `a` est bissextile.

8) Écrire une fonction `nbjoursmois(a, m)` qui renvoie le nombre de jours dans le mois `m` de l'année a, appellant la fonction `bissextile(a)` pour savoir si l'année `a` est bissextile. On suppose que `m` est un entier compris entre 1 et 12.

9) En utilisant les fonctions des exercices précédents, écrire une fonction `nbjours(jn, mn, an, j, m, a)` qui renvoie le nombre de jours compris entre deux dates données (par exemple votre date de naissance et la date d'aujourd'hui). Par exemple, `print(nbjours(20, 11, 1989, 25, 12, 2025))` doit renvoyer 13184.

Aide : C'est un exercice est très difficile si on ne s'y prend pas méthodiquement. Une méthode de développement possible pour le résoudre est le développement piloté par les tests qui consiste a écrire des cas de tests pour notre fonction avant de coder la fonction elle-même. Les cas de test ici sont des appels à la fonction `nbjours` pour lesquels on sait calculer à la main les résultats attendus. En voici quelque uns, vous pouvez en ajouter davantage :

```
print(nbjours(25, 12, 2025, 25, 12, 2025)) # doit afficher 0, c'est le même jour.
print(nbjours(24, 12, 2025, 25, 12, 2025)) # doit afficher 1, c'est la veille.
print(nbjours(25, 11, 2025, 25, 12, 2025)) # doit afficher 30
print(nbjours(25, 12, 2025, 25, 1, 2026)) # doit afficher 31
print(nbjours(1, 1, 2024, 1, 1, 2025)) # doit afficher 366
print(nbjours(1, 1, 2025, 1, 1, 2026)) # doit afficher 365
```

10) On dispose des quatre fonctions suivantes :

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

```
def m(x, y):
    return x * y
```

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

```
def d(x, y):
    return x / y
```

Sans utiliser les opérateurs *, -, +, /, ** mais en utilisant uniquement les quatre fonctions précédentes, des variables et des boucles, écrire les fonctions suivantes :

a) La fonction `carre(x)` qui calcule et renvoie `x**2`.

b) La fonction `cube(x)` qui calcule et renvoie `x**3`.

c) La fonction `f(x)` qui calcule et renvoie `2*x**3 + 4/3*x**2 - 9`.

d) La fonction `puis(x, p)` qui calcule et renvoie `x**p`, avec `p` un entier positif ou nul.

e) La fonction `puis(x, p)` qui calcule et renvoie `x**p`, avec `p` un entier relatif.

f) La fonction `fact(n)` qui calcule et renvoie la factoriel de `n` : n!, avec `n` un entier positif ou nul.