1 Déclaration de fonctions

On peut définir ses propres fonctions. Pour cela la syntaxe est :

def *nom_de_la_fonction*(*paramètres*)
..... Bloc d'instructions
..... **return** *expressions*

meme espace — bloc d'instructions

Pour cela :

- La première ligne (entête) commence par **def** suivi du nom de la fonction, et entre parenthèses, de ses paramètres. La ligne finit par deux points `:`.
- Les noms de fonctions suivent les mêmes règles que les noms de variables.
- Le bloc de définition de la fonction est décalé d'un même espace.
- L'instruction **return** *expression* renvoie à celui qui appelle la fonction, le résultat expression.

Exercice 1. Le script suivant déclare une fonction `somme(n)` prenant en paramètre un entier `n`, et qui calcule la somme des entiers de 1 à `n`. L'exécuter et appeler la fonction avec différentes valeurs de `n` pour tester son fonctionnement.

```
def somme(n):
    S = 0
    for k in range(1,n+1):
        S += k
    return S
```

Exercice 2.

Écrire les deux fonctions `puissance1(x,y)` et `puissance2(x,y)` suivantes :

```
from math import exp, log

def puissance1(x,y):
    return exp(y*log(x))

def puissance2(x,y):
    print(exp(y*log(x))
```

a) Les interpréter puis saisir dans la console :

```
>>> puissance1(2,0.5)
>>> puissance2(2,0.5)
```

Expliquer les résultats obtenus.

b) Saisir dans la console :

```
>>> 1 + puissance1(2,0.5)
>>> 1 + puissance2(2,0.5)
```

Expliquer les résultats obtenus.

Exercice 3.

1. Écrire une fonction `factorielle(n)` prenant en paramètre un entier `n` et qui renvoie $n!$.
2. Écrire un script qui utilise cette fonction pour afficher dans la console toutes les valeurs de $n!$ pour n variant de 0 à 10.

Exercice 4.

1. Écrire une fonction `fibonacci(n)` prenant en paramètre un entier `n` non nul et qui renvoie la valeur du terme F_n de rang n de la suite de Fibonacci :
$$F_0 = 0, \quad F_1 = 1 \quad \forall\, n \in \mathbb{N},\ F_{n+2} = F_n + F_{n+1}$$
2. Écrire un script qui affiche les 20 premiers termes de la suite de Fibonacci.
3. Écrire une fonction `sommeFibonacci(n)` qui renvoie la somme $\sum_{k=1}^{n} F_k$.

2 Expression booléenne ou condition

Une *condition*, ou *expression booléenne*, est comme une proposition mathématique : elle est soit vraie soit fausse.
En python une condition est vraie lorsqu'elle vaut la valeur `True`, sinon elle vaut la valeur `False`.
Pour écrire une condition, on peut utiliser les **opérateurs de comparaison**.

2.1 Les opérateurs == et !=

Ils permettent de comparer les valeurs de deux expressions :
• L'opérateur == est l'égalité mathématique. La condition vaut `True` dès que les deux expressions en opérande ont même valeur.

```
>>> 1 == 1
True

>>> 1 == 2
False

>>> 1 == 1.0
True

>>> 1 == 1+0j
True

>>> 'Bonjour' == "Bonjour"
True

>>> 'Bonjour' == 'bonjour'
False
```

Les deux opérandes n'ont pas besoin d'être de même type pour être égales. Par exemple l'entier `1`, le flottant `1.0` et le complexe `1+0j` ont tous trois même valeur.

Attention à ne pas confondre l'affectation : $=$ et l'égalité de valeur : $==$.

• L'opérateur $!=$: c'est le contraire. La condition vaut True dès que les deux expressions en opérande ont des valeurs différentes.

```
>>> 1 != 2
True

>>> 1 != 1.0
False
```

2.2 Les opérateurs de comparaison $<$, $<=$, etc.

Ils permettent de comparer des valeurs de type entier ou flottant, pour l'ordre sur les réels :

```
>>> 1 < 2
True

>>> 3 >= 3.0
True
```

2.3 Tableau des opérateurs de comparaison

En résumé :

Opérateurs de comparaison	
==	égalité (de valeur).
!=	non égalité (de valeur).
<	Strictement inférieur (entiers ou flottants).
>	Strictement supérieur (entiers ou flottants).
<=	Inférieur ou égal (entiers ou flottants).
>=	Supérieur ou égal (entiers ou flottants).

2.4 Le type booléen (`bool`)

Une condition a une valeur `True` ou `False`. Cette valeur est une donnée de type booléen (**`bool`**). Une donnée de type booléen ne peut prendre que deux valeurs : `True` et `False`.

• La fonction **`type()`** renvoie le type de l'expression passée en opérande :

```
>>> type (1<2)
bool
```

2.5 Opérations logiques

On peut appliquer aux données de type booléen, les trois opérations logiques suivantes ; le résultat sera un booléen.

Opérations logiques	
or	OU logique.
and	ET logique.
not	NON logique.

• **Exemple.**

```
>>> a = 1
>>> a < 1e3 and a > 1e-3
True

>>> 1e-3 < a < 1e3
True
```

☞ **Astuce.** python accepte la condition `a < b < c` pour `a < b and b < c` (etc).

3 Structures de contrôle.

3.1 Boucle While

Une boucle **while** permet de répéter un bloc d'instruction, en boucle, tant qu'une *condition* est vérifiée.

```
while condition :
.......  Instruction1
.......  ⋮
.......  Dernière instruction
```

même espace — bloc d'instructions

• **Exemple 1.** Le script suivant demande à l'utilisateur de répondre par ¡¡oui¿¿ ou ¡¡non¿¿, et se répète tant que la réponse n'est pas ¡¡oui¿¿ ou ¡¡non¿¿.

```
reponse = ""
while reponse != "oui" and reponse != "non":
    reponse = input('Répondre par oui/non : ')
print('Merci.')
```

Exercice 5.
La fonction **float()** convertit son argument, si c'est possible, en nombre flottant. Par exemple avec en argument la chaine de caractères "1", l'appel de **float**("1") renvoie le flottant 1.0.
On considère le script suivant qui définit une fonction mystere(). Déterminer ce qui s'exécute à l'appel de mystere(7). Plus généralement à quoi sert la fonction mystere() ?

```
def mystere(n):
    for k in range(11):
        rep = "-1"
        while float(rep) != k*n:
            print(k,'x',n,'= ',end="")
            rep = input()
```

Exercice 6. Algorithme d'Euclide
L'algorithme d'Euclide date du IVe siècle avant J.C.. Il permet de calculer le plus grand diviseur commun, ou PGCD, de deux nombres entiers.
Soit a et b deux nombres entiers ; pour calculer leur pgcd, répéter :

Tant que $b \neq 0$:
    Soit r le reste dans la division euclidienne de a par b.
    changer a en b et b en r.
À la fin (lorsque $b = 0$), a contient le PGCD des valeurs de a et b initiales.

1. Écrire le script d'une fonction pgcd(a,b) qui renvoie le PGCD des entiers a et b calculé à l'aide de l'algorithme d'Euclide.
2. Le tester pour trouver le PGCD des nombres 26221 et 42357
3. En déduire le script d'une fonction ppcm(a,b) qui renvoie le PPCM des entiers a et b.

4 Complément

Exercice 7.
Écrire une fonction monSapin(n) prenant en paramètre un entier n non-nul et qui affiche dans la console une figure semblable à la suivante, mais avec n niveaux (ex : monSapin(4) affichera le motif suivant :).

```
    o
   ooo
    o
   ooo
  ooooo
    o
   ooo
  ooooo
 ooooooo
    o
   ooo
  ooooo
 ooooooo
ooooooooo
```