Les fonctions

Une fonction est un sous-programme qui peut être utilisé plusieurs fois dans le programme principal. Ce sous-programme effectue un traitement précis (une suite d'instructions) sur des données en paramètres.

Définitions

Une fonction est une structure informatique composée d'une qui sont exécutées par un mécanisme d' à la fonction.

Une fonction est déclarée par le mot-clef `def`, et les instructions qui la composent (le corps de la fonction) sont .

Une fonction peut comporter des , qui sont des valeurs passées en **argument** lors des appels à la fonction.

Dans un programme, le nom d'une fonction est (presque) toujours suivi de **parenthèses** :

- soit pour préciser les noms des paramètres dans sa définition ;
- soit pour indiquer les valeurs des paramètres lors d'un appel.

Les paramètres sont utilisés exclusivement dans la fonction (Ils n'ont pas d'existence à l'extérieur, même si d'autres variables portent le même nom).

Les variables définies dans une fonction ne peuvent être utilisées que dans cette fonction : on parle de variables **locales**.

Dans une fonction, on veille à n'utiliser que les paramètres et les variables locales.

Le **nom d'une fonction** doit commencer par une lettre minuscule et ne comporter que des caractères alphanumériques et éventuellement des tirets bas '_'.

Même si cela est permis, on ne doit pas écraser la valeur d'un paramètre dans une fonction.

Dans un programme bien écrit, les fonctions sont définies **avant la partie script**.

Valeurs renvoyées

Une fonction **renvoie toujours une valeur** lors de son appel.

Par défaut, cette valeur est l'objet `None`, qui désigne l'absence de valeur particulière. Le renvoi de valeur est effectué grâce au mot-clé `return`.

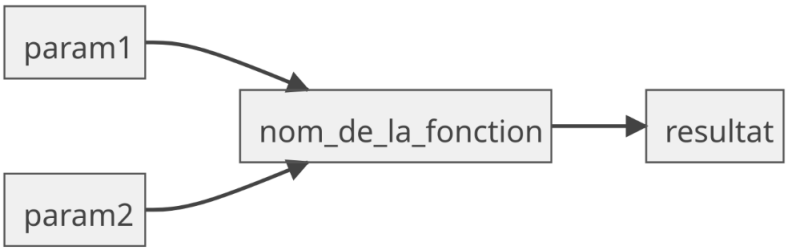
Lors d'un renvoi de valeur,

Il est d'usage de **documenter** une fonction en indiquant ce qu'elle renvoie dans un commentaire appelé *docstring*, délimité par trois doubles guillements juste après l'entête de la fonction.

Une fonction peut être définie ainsi en Python :

```
def nom_de_la_fonction(param1, param2, ...)
    instruction1
    instruction2
    ... # le nombre de paramètres dépend de la fonction
    return resultat # le renvoi explicite d'un résultat est facultatif
```

On peut représenter son fonctionnement schématiquement de la façon suivante.



Elle peut avoir de 0 à un nombre n de paramètres, de même pour les valeurs de résultat. Au lieu de paramètres et résultats, on parle aussi d'**entrées** (les données qu'on a avant de faire un traitement) et de sorties (ce qu'on obtient après le traitement).

L'appel de la fonction se fait de la manière suivante.

```
val1 = ...
val2 = ...
nom_de_la_fonction(val1, val2)
```

> **Indication**
> La **définition** d'une fonction peut être comparée à une **recette de gâteau** écrite sur un morceau de papier. Elle permet de connaître la marche à suivre pour faire quelque chose (dans le cas de la recette, comment faire un gâteau au chocolat par exemple).
> Quand on a la recette entre les mains, on n'a pas effectivement de gâteau au chocolat. Pour y remédier, on doit mettre en œuvre la recette. C'est pareil pour une fonction : la définition de la fonction indique le **procédé** pour faire quelque chose, et ,pour faire effectivement cette chose, il faut **appeler** la fonction avec les paramètres attendus.

Exercice 1

Dans le programme ci-dessous, identifiez :

- les fonctions, leurs paramètres, les variables locales ;
- les appels aux fonctions et les valeurs passées en argument ;
- les affichages qui seront effectués.

```
# Définition des fonctions
def addition (a , b):
    somme = a + b
    return somme

def multiplication (a , b):
    produit = a * b
    return produit

# Script
a = 5
p = multiplication (a +3 , 7)
s = 3
m = multiplication (s , 6)
print (a ,p , s, m)
```

Exercice 2 – *(Ce qu'il ne faut pas faire ! Sur papier)*

Lister au moins six anomalies dans le code ci-dessous.

```
a = 10
def ma-fonction(x ,y):
    x = 0
    return a + x

ma-fonction(8)
y = 5
def Hello():
    print(x + y)
```

Exercice 3 – *(À faire sur ordinateur)*

1. Écrire une fonction `fois_deux(x)` qui prend en paramètre un entier ou un flottant `x` et **affiche** sa valeur multipliée par 2.
2. Tester son fonctionnement en l'appelant pour `x` valant 6.
3. Modifier cette fonction pour qu'elle **n'affiche pas** le résultat, mais le **renvoie**.
4. Tester son fonctionnement en l'appelant pour `x` valant 6. Comment récupérer le résultat ?
5. Écrire une fonction `multiplication(a, b)` affichant le produit de `a` par `b`.
6. Tester son fonctionnement en l'appelant avec les valeurs 3 et 4.
7. Tester son fonctionnement en l'appelant avec les valeurs 3.0 et 4.0. Que remarquez-vous ?
8. Modifier cette fonction pour qu'elle n'affiche pas le résultat, mais le renvoie.
9. Faire le même test que dans la question 6, et récupérer le résultat dans une variable.
10. De la même façon, écrire une fonction `addition(a, b)` renvoyant la somme de `a` et de `b`, et tester son fonctionnement en affectant 3 à `a` et 4 à `b`.

Exercice 4 – *(À faire sur papier puis vérifier sur PC)*

Déterminer les affichages produits par le script ci-dessous.

```
def carre(x):
    """ Renvoie le carré du nombre x """
    return x*x

resultat = carre(carre(5) - 15)
print(resultat)
```

Exercice 5

Écrire et tester une fonction `aire_trapeze(a, b, h)` qui renvoie l'aire d'un trapèze de bases `a` et `b` et de hauteur `h`.

→ L'aire d'un trapèze de bases a et b et de hauteur h est donnée par la formule $\dfrac{(a+b)h}{2}$.

Exercice 6

Écrire et tester une fonction `aire_triangle(a, b, c)` qui renvoie l'aire $\mathcal{A}$ d'un triangle de côtés `a`, `b` et `c`.

$\rightarrow$ On pourra utiliser la formule de Héron :

$$\mathcal{A} = \sqrt{p(p-a)(p-b)(p-c)} \quad \text{avec} \quad p = \frac{a+b+c}{2}.$$

Exercice 7

Considérons une fonction `est_rectangle(xa, ya, xb, yb, xc, yc)` qui, étant données les coordonnées de trois points A, B, C du plan, affiche `True` si le triangle ABC est rectangle, et `False` sinon. Une telle fonction va nécessairement contenir du code redondant, par exemple le calcul de AB^2, AC^2, et BC^2 si l'on utilise le théorème de Pythagore.

```
def est_rectangle(xa, ya, xb, yb, xc, yc):
    ab2 = (xb-xa)**2+(yb-ya)**2
    ac2 = (xc-xa)**2+(yc-ya)**2
    bc2 = (xc-xb)**2+(yc-yb)**2
```

Il devient alors intéressant d'utiliser une fonction auxiliaire `distance_au_carre` qui va contenir le code responsable du calcul des carrés des distances.

```
def distance_au_carre(x1, y1, x2, y2):
    return (x2-x1)**2+(y2-y1)**2

def est_rectangle(xa, ya, xb, yb, xc, yc):
    ab2 = distance_au_carre(xa, ya, xb, yb)
    ac2 = distance_au_carre(xa, ya, xc, yc)
```

1. Compléter la fonction `est_rectangle` pour qu'elle renvoie `True` si ABC est rectangle, et `False` sinon.
2. On considère les points $A(5\,;\,2)$, $B(2\,;\,2)$ et $C(2\,;\,6)$.
   Vérifier que l'appel `est_rectangle(5, 2, 2, 2, 2, 6)` renvoie bien la valeur `True` dans ce cas.
3. Proposer une autre solution au problème s'appuyant sur le produit scalaire (vu en maths).