Chapitre 5

Les fonctions

I Structurer son code avec des fonctions : diviser pour régner

Les *fonctions* sont des fragments de code réutilisables réalisant une tâche donnée pouvant dépendre d'un certain nombre de paramètres. Un fonction est définie par un *nom*, a généralement des *paramètres d'entrée* et retourne *une ou plusieurs valeurs*. Elles deviennent très utiles lorsqu'une action est réalisé à plusieurs endroit d'un programme.

I.1 Définir une fonction

Une fonction associe une séquence d'instruction à un nom. Voici l'exemple d'une fonction qui permet de dessiner un angle droit de longueur 100 à l'aide de l'outil turtle (pour plus de précisions sur l'outil turtle, se référer à la description du module plus bas dans le document).

```
def angle_droit():
        forward(100)
        left(90)
        forward(100)
```

La définition de la fonction commence par le mot-clé *def*, suivi par le nom de la fonction *angle_droit*, puis une paire de parenthèse et un deux-points. Tous les éléments indentés (avec une tabluation) appartiennent à la fonction, c'est le *corps de la fonction*. Par la suite, cette fonction sera appelé dans le code principal par la syntaxe **angle_droit()**. L'appel de cette fonction produit le même effet que si l'on avait exécuté les trois instructions présentes dans la fonction **angle_droit()**.

I.2 Fonction avec paramètres

Dans notre fonction *angle_droit*, la longueur est fixée à 100. Pour dessiner des angles droits de longueur quelconque, on peut ajouter un *paramètre* à notre fonction.

```
def angle_droit(x):
        forward(x)
        left(90)
        forward(x)
```

Le paramètre est désigné par un nom, ici **x**, ajouté entre les parenthèses. Il est ensuite utilisé dans le corps de la fonction, comme une variable. Le paramètre désigne une valeur qui pourra être différente à chaque appel à la fonction *angle_droit*. Il est possible de renseigner plusieurs paramètres à la définition de notre fonction. Dans ce cas, chaque paramètre est séparé d'une *virgule*, et toujours renseigné à l'intérieur des parenthèses. Par exemple, précisons l'angle que réalisera la fonction.

```
def angle_droit(x, angle):
        forward(x)
        left(angle)
        forward(x)
```

Lors de l'appel de la fonction, on renseignera autant de paramètre que la fonction en nécessite, dans le cas actuel, pour une longueur de 20 et un angle de 60 :

```
angle_droit(20, 60)
```

Toutes les instructions que nous avons utilisés jusqu'à présent, comme **print()** ou encore **range()** sont des fonctions natives et prédéfinies par le langage.

Exercice 38

A l'aide de la fonction angle_droit(), dessiner un escalier de 10 marches. Modifier ensuite votre programme pour que l'utilisateur puisse renseigner le nombre de marche et leur longueur.

Exercice 39

Écrire une fonction qui permet de dessiner un rectangle dont les longueurs seront renseigner par l'utilisateur. Modifier ensuite votre programme afin de demander à l'utilisateur si il souhaite que le rectangle soit colorié à l'intérieur et l'exécuter. Le remplissage de la forme géométrique sera réalisé à l'appel de la fonction *begin_fill()*, et ce jusqu'à l'appel de la fonction *end_fill()*.

I.3 Renvoyer un résultat

Dans la partie précédente, notre fonction angle_droit() permettait d'obtenir un résultat sous forme d'affichage graphique via la bibliothèque turtle. Néanmoins, les fonction Python peuvent aussi représenter des fonctions mathématiques qui calculent des valeurs. Par exemple, le code :

```
def f(x):
        return 2*x + 4
```

définit la fonction mathématique qui associé au nombre x, associe le nombre $2x+4$. Le mot-clé **return** spécifie la valeur *renvoyée* comme le résultat de la fonction.

```
print("Lorsque 'x' vaut 5, f(x) = ", f(5))
```

```
Lorsque "x" vaut 5, f(x) = 14
```

Il ne faut pas hésiter à donner à la fonction un nom bien explicite, même si il est un peu long. Le jours de la définition de la fonction, un nom simple peu sembler suffisant, mais c'est rarement le cas plusieurs jours ou semaines plus tard. Ainsi, une fonction qui définit les caractéristiques principales d'un personnage secondaire devrait s'appeler *definition_principale_personnage_secondaire* plutôt que *def_car_sec*. On peut aussi utiliser la syntaxe **camelCase** comme ceci : *definitionPrincipalePersonnageSecondaire*.

Exercice 40

Écrire une fonction qui permet d'appliquer une TVA à 19,5% à un montant renseigné par l'utilisateur et l'afficher.

Exercice 41

Écrire une fonction qui divise n fois par x un nombre renseigné par l'utilisateur et l'afficher.

II Variable globale, variable locale, portée des variables

D'après l'emplacement de la définition, une variable peu être *locale* ou *globale*. Une variable locale ne sera utilisé que dans la fonction où elle est définie, et supprimée lorsque l'on sort de la fonction. Une variable globale est définie dans le programme principale et est accessible à tout moment par l'ensemble du programme (et des fonctions donc).

```
Def addition_de_deux_entiers(x, y):
        return x + y

A = 10
B = 15
print (addition_de_deux_entiers(A, B))
```

```
25
```

Dans l'exemple ci-dessus, A et B sont définies comme des variables globales, puisqu'elles sont situées dans le programme principal. A contrario, les variables x et y sont des variables locales puisqu'elles sont sont définies dans la fonction addition_de_deux_entiers(). Lorsque la fonction addition_de_deux_entiers(A, B) est appelée, la fonction associe A à x et B à y. les deux variables locales x et y sont utilisées pour effectuer l'addition, puis sont détruites.

Au supermarché. On cherche à simuler la distribution de tickets dans une fil d'attente, comme au supermarché.

```
numero_ticket = 0

def prend_ticket():
    numero_ticket += 1
    return numero_ticket
```

```
for i in range(5):
    print(prend_ticket())
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-3-57e7268bb6a5> in <module>()
      1 for i in range(5):
----> 2     print(prend_ticket())

<ipython-input-2-edc590c61d7d> in prend_ticket()
      2
      3 def prend_ticket():
----> 4     numero_ticket += 1
      5     return numero_ticket

UnboundLocalError: local variable 'numero_ticket' referenced before assignment
```

Le problème c'est que Python interprète la variable `numero_ticket` comme une *variable locale* alors qu'on aimerai qu'elle soit *globale*. Pour cela, il faut lui indiquer avec le mot clef `global`.

```
numero_ticket = 0

def prend_ticket():
    global numero_ticket
    numero_ticket += 1
    return numero_ticket
```

```
for i in range(5):
    print(prend_ticket())
```

```
1
2
3
4
5
```

Exercice 42

Écrire une fonction qui tire au hasard un nombre entre 1 et 100 tout en comptant le nombre de fois qu'elle est appelée. On utilisera pour cela la fonction `randint` du module `random` dont la documentation se trouve ici : `https://docs.python.org/3/library/random.html#random.randint`.

III Passage par valeur ou par référence

Intéressons nous à la fonction suivante.

```
def swap(a, b):
    tmp = a
    a = b
    b = tmp
```

Qu'affiche alors le code ci-dessous ?

```
a = 7
b = 12
swap(a, b)
print(a, b)
```

Étonnant non ? En effet, ce ne sont pas les variables `a` et `b` qui sont passées à la fonction `swap` mais simplement leur *valeur*. On parle alors de passage par valeur. Il est impossible de modifier le contenu des variables globales `a` et `b` en les passant à la fonction `swap`[1]. On peut en revanche procéder comme ceci.

```
def swap(a, b):
    return (b, a)
```

```
a = 7
b = 12
a, b = swap(a, b)
print(a, b)
```

```
12 7
```

1. Il faudrait utiliser le mot clef `global`.

Intéressons nous maintenant à la fonction suivante.

```
def modifie_tableau(tab):
    tab[0] = 13
```

Qu'affiche alors le code ci-dessous ?

```
mon_tableau = [1, 2, 3, 4, 5]
modifie_tableau(mon_tableau)
print(mon_tableau)
```

Mais alors, on peut en fait modifier une variable passée en paramètre d'une fonction ?! Si la variable est un tableau, elle n'est en fait pas passée par valeur mais par *référence* à la fonction. C'est à dire que ce n'est pas le tableau qui est passé à la fonction mais, en quelque sorte, son emplacement dans la mémoire. On peut alors accéder au tableau et en modifier les valeurs !

Exercice 43

Écrire une fonction `swap` qui prend en paramètre un tableau à deux éléments et les échange. On ne demande pas de retourner le résultat.

Exercice 44

Comment peut-on modifier la fonction de l'exercice précédent pour qu'elle vérifie que le tableau passé en paramètre est bien de taille 2.

IV Conditions sur les arguments et la valeur de retour, assertions

Il arrive souvent que les fonctions ne soient valables que pour certaines valeurs des paramètres. Voici par exemple une fonction qui calcule le PGCD de deux entiers.

```
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Cette fonction n'est valable que pour `a` et `b` des entiers positifs, on parle alors de *préconditions* sur les arguments de la fonction. On souhaiterai que l'utilisateur soit averti de cette limitation. On peut alors ajouter un commentaire au code de la fonction.

```
# Retourne le PGCD de a et b
# Conditions : a et b sont des entiers positifs
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

C'est déjà mieux mais l'utilisateur n'a pas toujours le code d'une fonction sous les yeux. C'est pour cela qu'au lieu d'utiliser des commentaires, on utilise des *docstring* : ce sont des chaînes de caractères délimitées par trois guillemets « """ » et placées juste après les deux points qui définissent la fonction.

```
def pgcd(a, b):
    """
    Retourne le PGCD de a et b
    Conditions : a et b sont des entiers positifs.
    """
    assert a > 0 and b > 0, "a et b doivent être positifs."
    while b != 0:
        a, b = b, a % b
    return a
```

L'utilisateur qui n'a pas le code de la fonction sous les yeux peut alors consulter la documentation de la fonction avec `help(pgcd)`.

```
Help on function pgcd in module __main__:

pgcd(a, b)
    Retourne le PGCD de a et b
    Conditions : a et b sont des entiers positifs.
```

C'est déjà très clair mais on peut faire encore plus robuste ! On voudrait que l'utilisateur reçoive un message d'erreur lorsqu'il utilise mal la fonction. On peut alors utiliser le mot clef **assert**, comme ceci.

```
def pgcd(a, b):
    """
    Retourne le PGCD de a et b
    Conditions : a et b sont des entiers positifs.
    """
    assert a > 0 and b > 0, "a et b doivent être strictement positifs."
    while b != 0:
        a, b = b, a % b
    return a
```

```
pgcd(96, -20)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-13-e79328fdcd81> in <module>()
----> 1 pgcd(96, -20)

<ipython-input-11-8ed2b06e45dc> in pgcd(a, b)
      2 # Conditions : a et b sont des entiers positifs
      3 def pgcd(a, b):
----> 4     assert a > 0 and b > 0, "a et b doivent être strictement positifs."
      5     while b != 0:
      6         a, b = b, a % b

AssertionError: a et b doivent être strictement positifs.
```

Astuce

On peut aussi utiliser **assert** pour faire des tests. Soit des tests en dehors de la fonction comme ici,

```
assert pgcd(96, 20) == 4, "mauvaise valeur retournée par la fonction pgcd."
```

on peut aussi s'assurer qu'une condition est vraie avant de retourner le résultat de la fonction.

```python
def pgcd(a, b):
    """
    Retourne le PGCD de a et b
    Conditions : a et b sont des entiers positifs.
    """
    assert a > 0 and b > 0, "a et b doivent être strictement positifs."
    while b != 0:
        a, b = b, a % b
    assert b <= a, "erreur interne dans la fonction pgcd."
    return a
```

Exercice

Exercice 45

Modifier la fonction `pgcd` pour qu'elle vérifie que les arguments sont de type `int`.

Exercices de fin de chapitre

Exercice

Exercice 46 tiré du sujet 0 de NSI

Avec la définition de la fonction `f` suivante en Python, quelle est la valeur retournée par l'appel `f(42, 21)` ?

```python
def f(x, y):
    if x > y:
        return y, x
    else:
        return x, y
```

(a) `(21, 21)` (b) `(21, 42)` (c) `(42, 42)` (d) `(42, 21)`

Exercice

Exercice 47 tiré du sujet 0 de NSI

Dans la définition suivante de la fonction `somme` en Python, quelle est l'instruction à ajouter pour que la valeur retournée par l'appel `somme([10, 11, 12, 13, 14])` soit `60` ?

```python
def somme(tab):
    s = 0
    for i in range(len(tab)):
        ...
    return s
```

(a) `s = s + i`
(b) `tab[i] = tab[i] + s`
(c) `s = tab[i]`
(d) `s = s + tab[i]`

Exercice

Exercice 48 tiré du sujet 0 de NSI

Le programme Python suivant ne calcul pas toujours correctement le résultat de x^y. Parmi les tests suivant, lequel va permettre de détecter l'erreur ?

```python
def puissance(x, y):
    p = x
    for i in range(y - 1):
        p = p * x
    return p
```

(a) `puissance(2, 0)` (b) `puissance(2, 1)` (c) `puissance(2, 2)` (d) `puissance(2, 10)`

Exercice

Exercice 49 tiré du sujet 0 de NSI

La fonction suivante calcule la racine carré du double d'un nombre flottant.

```
from math import sqrt

def racine_du_double(x):
    return sqrt(2 * x)
```

Quelle est la précondition sur les arguments de cette fonction ?

(a) `2 * x > 0` (b) `x < 0` (c) `x >= 0` (d) `sqrt(x) >= 0`

Exercice

Exercice 50

Écrire une fonction `maximum` qui retourne la plus grande des deux valeurs qui lui sont passées en paramètres.
Mettre en place un jeu de tests avec des `assert`.

Exercice

Exercice 51

Modifier la fonction de l'exercice précédent pour quelle travaille non plus avec deux valeurs mais avec un tableau.
Mettre aussi en place un jeu de tests avec des `assert`.

Exercice

Exercice 52

Définir une fonction *testPythagore* qui prend 3 arguments entiers a, b, c et retourne un booléen si $a^2 + b^2 = c^2$.

Exercice

Exercice 53

Le cryptage des messages a été utilisé en premier par Jules César pour cacher les messages destinés, entre autre, à ses armées. Il utilisait une technique de décalage de l'alphabet, appelée *clé de cryptage*. Par exemple, avec un clé de 3 et la lettre C, on obtient la lettre F. Écrire une fonction qui permet de réaliser ce cryptage. *Rappel : on passe d'un caractère à un entier avec la fonction ord(), et l'inverse avec la fonction str(chr()).*

Exercice

Exercice 54

Réaliser un programme qui permet d'affecter la clé de cryptage à chaque caractère d'une chaîne de caractère. Pour cela, il sera nécessaire de modifier la fonction de l'exercice précédent afin d'appliquer la clé de cryptage à chaque élément du tableau.

Astuce

La séparation de la chaîne de caractère sera réalisé avec la fonction ci-dessous. Cette fonction associe un caractère à une case de notre tableau.

```
def separation(mot):
        return list(mot)
```

et le ré-assemblage (appelé *concaténation*) sera réalisé avec la fonction suivante. L'élément "" indique de n'associer aucun caractère à la concaténation.

```
def concatenation(mot):
        return "".join(mot)
```