Chapitre 1.3 : Fonctions

I Création de fonctions

Une fonction est une suite d'instructions qui dépend de paramètres : on peut alors exécuter un bloc d'instructions en modifiant les valeurs des paramètres sans avoir à recopier tout le bloc. On peut en trouver dans les modules à importer, par exemple dans le module `math`, mais on peut aussi en programmer. Par exemple, étant définie la fonction suivante :

```
def f(x):  
    return x**2 + x + 1
```

les instructions `f(1)`, `f(-4.5)`, etc... permettent de calculer les valeurs $1^2 + 1 + 1$, $(-4.5)^2 - 4.5 + 1$, etc... sans avoir à réécrire les expressions à chaque fois.

La syntaxe de base est la suivante :

```
def nom(arg1, arg2, ...):  
    instruction1  
    instruction2  
    ...  
    return valeur
```

L'instruction `def` permet de définir une fonction en précisant :

- le nom de la fonction qui servira à l'appeler ;
- les arguments (0, 1 ou plus) : on met des parenthèses même s'il n'y a aucun argument ;
- une suite d'instructions ;
- éventuellement une valeur de retour en utilisant l'instruction `return`.

Et bien entendu, l'indentation pour le **corps** de la fonction est très importante !

II Appel et retour

Le code suivant :

```
def ma_fonction(x, y):  
    if x < y:  
        return x  
    else:  
        return y
```

ne fait rien de particulier : l'interpréteur lit la définition de la fonction, et crée une variable contenant cette fonction, mais ne fait rien avec tant que la fonction n'est pas appelée. Pour appeler une fonction une fois qu'elle a été définie, c'est comme en maths : on donne des valeurs aux arguments ; par exemple, `ma_fonction(1, 2)`.

Le mot-clé `return` indique que l'interpréteur renvoie une valeur à l'utilisateur : libre à lui de s'en servir en la stockant par exemple dans une variable, ou bien en l'utilisant directement dans une autre instruction. Par exemple,

- `min = ma_fonction(1, 2)` affecte la valeur 1 à la variable `min`
- `1 + ma_fonction(1, 2)` est évalué à 2.

L'instruction `return` n'est pas indispensable : la fonction ne renvoie alors rien à l'utilisateur. On dit que c'est une **procédure**. Par exemple :

```
def ma_fonction2(x, y):
    if x < y:
        print(x)
    else:
        print(y)
```

est une procédure : l'exécution de `ma_fonction2(1, 2)` affiche 1, mais l'utilisateur ne donne pas de valeur à l'utilisateur.

Attention, les variables qui sont déclarées à l'intérieur d'une fonction sont **locales** : on ne peut pas y accéder en dehors de la fonction. Par exemple, avec

```
def ma_fonction3(x):
    a = 2*x
    return a
```

l'exécution de

```
ma_fonction3(1)
print(a)
```

renvoie une erreur ! Les arguments d'une fonction sont aussi des variables locales. Par exemple :

```
def ajoute(x):
    x = x + 1
    return x
```

puis dans la console

```
>>> a = 1
>>> ajoute(a)
2
>>> a
```

Les arguments d'une fonction peuvent être du type qu'on veut. Par exemple, une fonction peut avoir une fonction comme argument :

```
def pos(f, x):
    return max(f(x), -f(x))
```

renvoie

On peut aussi définir une fonction dans une fonction, mais attention, elle reste une variable locale.

III Commentaires

Il est important de commenter ses programmes : il faut pouvoir se relire, ou bien que d'autres utilisateurs puissent relire notre code et comprendre ce qu'il fait. On écrit donc un docstring de la façon suivante :

```
def nom(arg1:type, arg2:type, ...) -> type:
    """Spécification
    """
    corps
```

On précise les types des arguments et des retours et on écrit une description brève de ce que fait la fonction. Pour écrire un commentaire sur plusieurs lignes, on utilise trois guillemets pour ouvrir le commentaire et trois pour le fermer.

On pourra alors utiliser la fonction `help` pour afficher le docstring de la fonction définie.

IV Exercices

Exercice 1 — Des spécifications. Écrire les spécifications des trois fonctions suivantes :

```
def hypotenuse(x:      , y:      ) ->      :
    '''
    '''
    c = math.sqrt(x**2 + y**2)
    return c
def hypotenuse(x:      , y:      ) ->      :
    '''
    '''
    c = math.sqrt(x**2 + y**2)
    print(c)
def hypotenuse(x:      , y:      ) ->      :
    '''
    '''
    c = math.sqrt(x**2 + y**2)
def compose(f:      , g:      ) ->      :
    '''
    '''
    def h(x):
        return f(g(x))
    return h
```

Exercice 2 Corriger la fonction suivante pour qu'elle respecte la spécification donnée :

```
def somme(n:int) -> int:
    '''Renvoie la somme des entiers de 0 à n
    '''
    for i in range(n):
        s = s + i
    return s
```

Exercice 3 — Termes d'une suite récurrente. On considère la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = 1$, $u_1 = 1$ et $u_{n+2} = 3u_{n+1} - 5u_n$. Écrire une fonction `u(n)` renvoyant la valeur de u_n .

Exercice 4 — Recherche de seuil. Écrire une fonction d'entête `def seuil(M:int) -> int:` qui renvoie le plus petit entier naturel n tel que $\sqrt{1} + \sqrt{2} + \dots + \sqrt{n} \geq M$.
On utilisera la fonction `sqrt` du module `math` pour la racine carrée. ■

Exercice 5 — Recherche de minimum. Écrire une fonction `minimum(a, b, c)` qui renvoie le minimum entre a , b et c . ■

Exercice 6

1. Écrire une fonction `somme_div(n)` qui étant donné un entier naturel non nul n renvoie la somme de ses diviseurs positifs.
2. Un entier naturel non nul n est parfait si la somme de ses diviseurs positifs vaut $2n$. Écrire une fonction `est_parfait(n)` qui renvoie `True` si n est parfait et `False` sinon.
3. Écrire une procédure `nb_parfaits(N)` qui affiche tous les nombres parfaits entre 1 et N . ■