Chapitre 5

Les fonctions

Quand on doit faire plusieurs fois un calcul similaire où il faut juste changer une valeur, quand on doit écrire à différents endroits d'un programme des instructions semblables, on utilise une **fonction**. Cela permet d'éviter les redondances : on n'écrit le code qu'une seule fois ; et si on doit le corriger, on n'a a le corriger qu'une seule fois.
Utiliser des fonctions rend le programme plus clair et plus simple à lire : pour comprendre le programme, il suffit de savoir ce que fait la fonction, il n'est pas nécessaire de savoir comment elle est programmée.
Utiliser des fonctions permet aussi d'organiser l'écriture d'un programme : on peut décider d'écrire la fonction un jour et le reste du programme le lendemain. On peut demander à un programmeur d'écrire la fonction pendant qu'on avance sur le programme.
Synonymes : procédures, méthodes

1 Définition

Définition

Une fonction est une suite d'instruction qui dépend de paramètres ou arguments.
Une fonction est définie par son **nom**, par ses **arguments** ou paramètres qui porteront les valeurs communiquées par le programme principal à la fonction au moment de son appel.
Eventuellement, une fonction renvoie une valeur de retour communiquée au programme en fin d'exécution.

```
1 def annoncer_train(direction, horaire) :
2     print("Le train en direction de", direction, "partira à", horaire)
3
4 annoncer_train("Brest", "9h00")
5 annoncer_train("Paris", "10h05")
```

La fonction est introduite par le mot-clé `def`, suivi par le nom de la fonction, de ses paramètres ou arguments en parenthèses, puis de : suivi d'un bloc d'instruction appelé *corps* de la fonction.

Les variables `direction` et `horaire` qui figurent comme arguments de la fonction sont appelés *arguments formels* car on ne sait pas qu'elle sera leur valeur au moment où on écrit la fonction.

Chaque instruction `annoncer_train(...)` est un `appel` à la fonction. On ne peut appeler une fonction qu'une fois qu'elle a été définie. Les lignes 4 et 5 ne peuvent pas arriver avant la ligne 1 !
Dans l'appel `annoncer_train("Brest", "9h00")`, les expressions `"Brest"` et `"9h00"` sont les `arguments effectifs` de l'appel : dans l'ordre, la fonction fait les liens suivants : `direction="Brest"` et `horaire="9h00"`.

2 Comprendre le return d'une fonction

Une fonction peut renvoyer au programme principal un ou plusieurs valeurs à l'aide du mot-clé `return`.
Une fonction s'arrête

- soit par qu'elle est arrivée à la fin de ces instructions
- soit par qu'elle a rencontré le mot-clé *return*

```
def message(nom) :
    return 4
    print('Bienvenue', nom)
```

renvoie 4 mais n'affiche aucun message de bienvenue car la fonction s'arrête sur le return.

```
def deux_valeurs(n) :
    return n, n+1
x, y = deux_valeurs(3) # x=3, y=4
z = deux_valeurs(3) # z = (3,4) tuple
x, y = z[0], z[1]
```

```
def parite(n) :
    if n%2 == 0 :
        return 'pair'
    return 'impair' # pas besoin du else
```

Ne pas confondre `return` et `print` :
- l'instruction `print` ne renvoie rien, elle affiche seulement à l'écran un message
- l'instruction `return` n'affiche rien, elle décide de ce que doit renvoyer la fonction et donc l'appel à la fonction

En python, une fonction qui n'utilise pas le mot-clé `return` renvoie quand même `None` !

3 Documentation d'une fonction

Pour comprendre ce que fait une fonction, il faut lire ou écrire sa documentation.
En python, la documentation d'une fonction est intégrée à la fonction via sa `docstring` (chaîne de documentation) qui doit commencer le corps de la fonction.

```
def nom_fonction(parametre) :
    "docstring"        |
    bloc d'instruction | # corps de la fonction
```

Si une fonction a été documentée, alors il suffit de taper `help(nom_fonction)` dans un interpréteur python pour avoir accès à sa `docstring`.

4 Variable locale ou globale, portée d'une variable

```
x=2
def modif(x) :
    x=3
    print(x)
modif(x) # appel à la fonction
print(x)
```

Que va afficher ce code ?
L'appel à la fonction `modif(x)` va afficher 3 alors que lorsqu'on revient au programme principal, `x` vaut 2 et le dernier `print(x)` va afficher 2.
La variable `x` dans la fonction est devenue **locale** à la fonction. Quand on sort de la fonction, le programme principal oublie tout ce qui concernant la fonction, il revient alors à `x` (variable **globale**) qui vaut 2.

```
1 x=2
2 def modif() :
3     y=x+3
4     print(y)
5 modif()
6 print(x)
```

Dans cet exemple, `x` est une variable globale (sa **portée** est l'ensemble du programme), alors que la variable `y` est locale à la fonction. En dehors de la fonction, elle n'existe pas : la portée de la variable `y` est réduite au corps de la fonction. Si après la ligne 6, on écrit `print(y)`, on aura le message d'erreur `NameError: name 'y' is not defined`.

Une variable globale doit être définie **avant** l'appel d'une fonction l'utilisant !

```
def volume (r) :
    return 4/3*pi*r**3 # pi doit être définie avant l'appel volume(...)
pi = 3.14159
print(volume(1))
```

5 Annotations de type dans les fonctions

```
def greeting(name : str) -> str :
    return 'Hello' + name
```

6 Bibliothèques ou Modules

Vous avez plein de fonctions prédéfinies à portée de main, comme les fonctions cos ou sin. Elles ont été organisées par groupe thématique appelé *bibliothèque* ou en python `module`.

```
from math import * # cos, sin, pi, ...
import math # math.cos, math.sin, math.pi
import math as m # m.cos, m.sin, m.pi
```

Les modules couramment utilisés sont les modules
- `math` pour toutes les fonctions mathématiques
- `random` pour tout ce qui est génération aléatoire de nombres
- `matplotlib` pour faire des graphiques

Pour savoir tout ce que contient un module :

```
import math
help(math)
```

Vous pouvez vous créer vos propres bibliothèques !