Chapitre 6

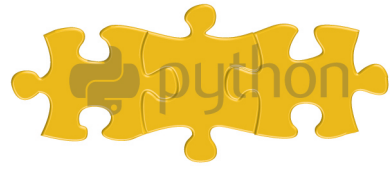
Pas à pas vers la modularité (1/2)

Difficulté :

En programmation, on est souvent amené à utiliser plusieurs fois des groupes d'instructions dans un but très précis. Attention, je ne parle pas ici de boucles. Simplement, vous pourrez vous rendre compte que la plupart de nos tests pourront être regroupés dans des blocs plus vastes, fonctions ou modules. Je vais détailler tranquillement ces deux concepts.

Les fonctions permettent de regrouper plusieurs instructions dans un bloc qui sera appelé grâce à un nom. D'ailleurs, vous avez déjà vu des fonctions : `print` et `input` en font partie par exemple.

Les modules permettent de regrouper plusieurs fonctions selon le même principe. Toutes les fonctions mathématiques, par exemple, peuvent être placées dans un module dédié aux mathématiques.



Les fonctions : à vous de jouer

Nous avons utilisé pas mal de fonctions depuis le début de ce tutoriel. On citera pour mémoire `print`, `type` et `input`, sans compter quelques autres. Mais vous devez bien vous rendre compte qu'il existe un nombre incalculable de fonctions déjà construites en Python. Toutefois, vous vous apercevrez aussi que, très souvent, un programmeur crée ses propres fonctions. C'est le premier pas que vous ferez, dans ce chapitre, vers la **modularité**. Ce terme un peu barbare signifie que nous allons nous habituer à regrouper dans des fonctions des parties de notre code que nous serons amenés à réutiliser. Au prochain chapitre, nous apprendrons à regrouper nos fonctions ayant un rapport entre elles dans un fichier, pour constituer un module, mais n'anticipons pas.

La création de fonctions

Nous allons, pour illustrer cet exemple, reprendre le code de la table de multiplication, que nous avons vu au chapitre précédent et qui, décidément, n'en finit pas de vous poursuivre.

Nous allons emprisonner notre code calculant la table de multiplication par 7 dans une fonction que nous appellerons `table_par_7`.

On crée une fonction selon le schéma suivant :

```
1 def nom_de_la_fonction(parametre1, parametre2, parametre3,
      parametreN):
2     # Bloc d'instructions
```

Les blocs d'instructions nous courent après aussi, quel enfer. Si l'on décortique la ligne de définition de la fonction, on trouve dans l'ordre :

- `def`, mot-clé qui est l'abréviation de « define » (définir, en anglais) et qui constitue le prélude à toute construction de fonction.
- Le nom de la fonction, qui se nomme exactement comme une variable (nous verrons par la suite que ce n'est pas par hasard). N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
- La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses ouvrante et fermante (là encore, les espaces sont optionnels mais améliorent la lisibilité).
- Les deux points, encore et toujours, qui clôturent la ligne.

Les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.

Le code pour mettre notre table de multiplication par 7 dans une fonction serait donc :

```
1 def table_par_7():
2     nb = 7
3     i = 0 # Notre compteur ! L'auriez-vous oublié ?
```

```
4       while i < 10: # Tant que i est strictement inférieure à 10,
5           print(i + 1, "*", nb, "=", (i + 1) * nb)
6           i += 1 # On incrémente i de 1 à chaque tour de boucle.
```

Quand vous exécutez ce code à l'écran, il ne se passe rien. Une fois que vous avez retrouvé les trois chevrons, essayez d'appeler la fonction :

```
1  >>> table_par_7()
2  1 * 7 = 7
3  2 * 7 = 14
4  3 * 7 = 21
5  4 * 7 = 28
6  5 * 7 = 35
7  6 * 7 = 42
8  7 * 7 = 49
9  8 * 7 = 56
10 9 * 7 = 63
11 10 * 7 = 70
12 >>>
```

Bien, c'est, euh, exactement ce qu'on avait réussi à faire au chapitre précédent et l'intérêt ne saute pas encore aux yeux. L'avantage est que l'on peut appeler facilement la fonction et réafficher toute la table sans avoir besoin de tout réécrire !

Mais, si on saisit des paramètres pour pouvoir afficher la table de 5 ou de 8... ?

Oui, ce serait déjà bien plus utile. Je ne pense pas que vous ayez trop de mal à trouver le code de la fonction :

```
1  def table(nb):
2      i = 0
3      while i < 10: # Tant que i est strictement inférieure à 10,
4          print(i + 1, "*", nb, "=", (i + 1) * nb)
5          i += 1 # On incrémente i de 1 à chaque tour de boucle.
```

Et là, vous pouvez passer en argument différents nombres, `table(8)` pour afficher la table de multiplication par 8 par exemple.

On peut aussi envisager de passer en paramètre le nombre de valeurs à afficher dans la table.

```
1  def table(nb, max):
2      i = 0
3      while i < max: # Tant que i est strictement inférieure à la
           variable max,
4          print(i + 1, "*", nb, "=", (i + 1) * nb)
5          i += 1
```

Si vous tapez à présent `table(11, 20)`, l'interpréteur vous affichera la table de 11, de 1*11 à 20*11. Magique non ?

Dans le cas où l'on utilise plusieurs paramètres sans les nommer, comme ici, il faut respecter l'ordre d'appel des paramètres, cela va de soi. Si vous commencez à mettre le nombre d'affichages en premier paramètre alors que, dans la définition, c'était le second, vous risquez d'avoir quelques surprises. Il est possible d'appeler les paramètres dans le désordre mais il faut, dans ce cas, préciser leur nom : nous verrons cela plus loin.

Si vous fournissez en second paramètre un nombre négatif, vous avez toutes les chances de créer une magnifique boucle infinie... vous pouvez l'empêcher en rajoutant des vérifications avant la boucle : par exemple, si le nombre est négatif ou nul, je le mets à 10. En Python, on préférera mettre un commentaire en tête de fonction ou une `docstring`, comme on le verra ultérieurement, pour indiquer que `max` doit être positif, plutôt que de faire des vérifications qui au final feront perdre du temps. Une des phrases reflétant la philosophie du langage et qui peut s'appliquer à ce type de situation est « *we're all consenting adults here* » [1] (sous-entendu, quelques avertissements en commentaires sont plus efficaces qu'une restriction au niveau du code). On aura l'occasion de retrouver cette phrase plus loin, surtout quand on parlera des objets.

Valeurs par défaut des paramètres

On peut également préciser une valeur par défaut pour les paramètres de la fonction. Vous pouvez par exemple indiquer que le nombre maximum d'affichages doit être de 10 par défaut (c'est-à-dire si l'utilisateur de votre fonction ne le précise pas). Cela se fait le plus simplement du monde :

```
1 def table(nb, max=10):
2     """Fonction affichant la table de multiplication par nb
3     de 1*nb à max*nb
4
5     (max >= 0)"""
6     i = 0
7     while i < max:
8         print(i + 1, "*", nb, "=", (i + 1) * nb)
9         i += 1
```

Il suffit de rajouter `=10` après `max`. À présent, vous pouvez appeler la fonction de deux façons : soit en précisant le numéro de la table et le nombre maximum d'affichages, soit en ne précisant que le numéro de la table (`table(7)`). Dans ce dernier cas, `max` vaudra 10 par défaut.

J'en ai profité pour ajouter quelques lignes d'explications que vous aurez sans doute remarquées. Nous avons placé une chaîne de caractères, sans la capturer dans une variable, juste en-dessous de la définition de la fonction. Cette chaîne est ce qu'on

1. « Nous sommes entre adultes consentants ».

appelle une `docstring` que l'on pourrait traduire par une chaîne d'aide. Si vous tapez `help(table)`, c'est ce message que vous verrez apparaître. Documenter vos fonctions est également une bonne habitude à prendre. Comme vous le voyez, on indente cette chaîne et on la met entre triple guillemets. Si la chaîne figure sur une seule ligne, on pourra mettre les trois guillemets fermants sur la même ligne ; sinon, on préférera sauter une ligne avant de fermer cette chaîne, pour des raisons de lisibilité. Tout le texte d'aide est indenté au même niveau que le code de la fonction.

Enfin, sachez que l'on peut appeler des paramètres par leur nom. Cela est utile pour une fonction comptant un certain nombre de paramètres qui ont tous une valeur par défaut. Vous pouvez aussi utiliser cette méthode sur une fonction sans paramètre par défaut, mais c'est moins courant.

Prenons un exemple de définition de fonction :

```
1 def fonc(a=1, b=2, c=3, d=4, e=5):
2     print("a =", a, "b =", b, "c =", c, "d =", d, "e =", e)
```

Simple, n'est-ce pas ? Eh bien, vous avez de nombreuses façons d'appeler cette fonction. En voici quelques exemples :

Instruction	**Résultat**
`fonc()`	a = 1 b = 2 c = 3 d = 4 e = 5
`fonc(4)`	a = 4 b = 2 c = 3 d = 4 e = 5
`fonc(b=8, d=5)`	a = 1 b = 8 c = 3 d = 5 e = 5
`fonc(b=35, c=48, a=4, e=9)`	a = 4 b = 35 c = 48 d = 4 e = 9

Je ne pense pas que des explications supplémentaires s'imposent. Si vous voulez changer la valeur d'un paramètre, vous tapez son nom, suivi d'un signe égal puis d'une valeur (qui peut être une variable bien entendu). Peu importent les paramètres que vous précisez (comme vous le voyez dans cet exemple où tous les paramètres ont une valeur par défaut, vous pouvez appeler la fonction sans paramètre), peu importe l'ordre d'appel des paramètres.

Signature d'une fonction

On entend par « signature de fonction » les éléments qui permettent au langage d'identifier ladite fonction. En C++, par exemple, la signature d'une fonction est constituée de son nom et du type de chacun de ses paramètres. Cela veut dire que l'on peut trouver plusieurs fonctions portant le même nom mais dont les paramètres diffèrent. Au moment de l'appel de fonction, le compilateur recherche la fonction qui s'applique à cette signature.

En Python comme vous avez pu le voir, on ne précise pas les types des paramètres. Dans ce langage, la signature d'une fonction est tout simplement son nom. Cela signifie que vous ne pouvez définir deux fonctions du même nom (si vous le faites, l'ancienne définition est écrasée par la nouvelle).

```
1 def exemple():
```

```
2       print("Un exemple d'une fonction sans paramètre")
3
4   exemple()
5
6   def exemple(): # On redéfinit la fonction exemple
7       print("Un autre exemple de fonction sans paramètre")
8
9   exemple()
```

A la ligne 1 on définit la fonction `exemple`. On l'appelle une première fois à la ligne 4. On redéfinit à la ligne 6 la fonction `exemple`. L'ancienne définition est écrasée et l'ancienne fonction ne pourra plus être appelée.

Retenez simplement que, comme pour les variables, un nom de fonction ne renvoie que vers une fonction unique, on ne peut surcharger de fonctions en Python.

L'instruction `return`

Ce que nous avons fait était intéressant, mais nous n'avons pas encore fait le tour des possibilités de la fonction. Et d'ailleurs, même à la fin de ce chapitre, il nous restera quelques petites fonctionnalités à voir. Si vous vous souvenez bien, il existe des fonctions comme `print` qui ne renvoient rien (attention, « renvoyer » et « afficher » sont deux choses différentes) et des fonctions telles que `input` ou `type` qui renvoient une valeur. Vous pouvez capturer cette valeur en plaçant une variable devant (exemple `variable2 = type(variable1)`). En effet, les fonctions travaillent en général sur des données et renvoient le résultat obtenu, suite à un calcul par exemple.

Prenons un exemple simple : une fonction chargée de mettre au carré une valeur passée en argument. Je vous signale au passage que Python en est parfaitement capable sans avoir à coder une nouvelle fonction, mais c'est pour l'exemple.

```
1   def carre(valeur):
2       return valeur * valeur
```

L'instruction `return` signifie qu'on va **renvoyer**[2] la valeur, pour pouvoir la récupérer ensuite et la stocker dans une variable par exemple. Cette instruction arrête le déroulement de la fonction, le code situé après le `return` ne s'exécutera pas.

```
1   variable = carre(5)
```

La variable `variable` contiendra, après exécution de cette instruction, 5 au carré, c'est-à-dire 25.

Sachez que l'on peut renvoyer plusieurs valeurs que l'on sépare par des virgules, et que l'on peut les capturer dans des variables également séparées par des virgules, mais je m'attarderai plus loin sur cette particularité. Retenez simplement la définition d'une fonction, les paramètres, les valeurs par défaut, l'instruction `return` et ce sera déjà bien.

2. Certains d'entre vous ont peut-être l'habitude d'employer le mot « retourner » ; il s'agit d'un anglicisme et je lui préfère l'expression « renvoyer ».