

Chapitre 2

Fonctions et portée de variables

2.1 Fonctions

La notion de fonction en informatique est primordiale. On les utilise quotidiennement.

Les fonctions en Python sont un peu comme les fonctions en mathématiques : elles ont besoin de paramètres et donnent un résultat.

Le grand intérêt des fonctions est de factoriser le code : au lieu d'écrire plusieurs fois la même chose, on ne l'écrit qu'une seule fois et on l'utilise autant de fois qu'on veut. En cas d'erreur, cela permet aussi de ne corriger le problème qu'à un seul endroit ; la correction se répercutera partout ailleurs.

Enfin, un autre intérêt est de rendre le code plus clair en utilisant un nom parlant pour une fonction. Ceci permet d'indiquer les étapes de traitement en masquant les détails techniques.

2.1.1 Définition

Le code générique d'une fonction est le suivant :

```
1 def function_name(params):  
2     instruction1  
3     instruction2  
4     ...  
5     return output
```

Quelques remarques :

- ⊗ Une fonction est définie dans un bloc de code.
- ⊗ `function_name` est le nom de la fonction. On peut utiliser ce qu'on veut sauf les mots clé de Python, les symboles d'espace et de tabulation et ne pas commencer par un chiffre. Il ne faut pas hésiter à donner un nom parlant, même long.
- ⊗ `params` est la liste des paramètres de la fonction, séparés par des virgules. Ces paramètres sont les entrées de la fonction. De la même manière, il ne faut pas hésiter à leur donner un nom parlant. La valeur de ces paramètres n'est pas définie dans la fonction mais lors de l'appel de cette dernière. S'il n'y a pas de paramètre, il faut mettre des parenthèses vides `()`.
- ⊗ `output` est la sortie de la fonction. C'est le résultat de son travail. Tous les autres calculs seront perdus.
- ⊗ Il y a une différence très importante entre `return` et `print` : la deuxième instruction permet d'afficher quelque chose mais ce n'est pas récupérable par le programme. En revanche, un `return` n'affiche rien mais permet de garder en mémoire un résultat. Il est donc largement préférable d'utiliser cette dernière instruction dans une fonction. Si on souhaite

l'afficher, on utilisera `print` en dehors de la fonction.

Voici un exemple de fonction Python qui prend deux paramètres et calcule une puissance :

```
1 def puissance( base , exposant ) :
2     resultat = base ** exposant
3     return resultat
```

Nous verrons un peu plus tard dans ce cours des spécificités de python pour la définition de fonction. Pour l'instant, voyons déjà comment les utiliser.

Définir une fonction ne la fait pas s'exécuter. Pour cela, il faut l'appeler : `puissance(4,3)` . On peut bien évidemment aussi l'appeler avec des variables en paramètres : `puissance(2, valeur)` .

Attention, l'ordre des paramètres est important : `puissance(2,3) != puissance(3,2)` .

Comme dans certains parcs d'attraction, pour une fonction, toute sortie est définitive : quel que soit l'état de l'exécution, une fonction s'arrête quand elle tombe sur une instruction `return` . On peut donc utiliser cette caractéristique :

```
1 def est_pair( nombre ) :
2     if nombre % 2 == 0 :
3         return True
4     return False
```

Une fonction peut bien évidemment servir pour une affectation ou une condition :

```
1 nombre = puissance( 51, 6 )
2
3 if est_pair(nombre):
4     print("51^6 est pair")
```

2.1.2 Spécificités de python

Valeurs par défaut

On peut spécifier des valeurs par défaut :

```
1 def puissance( base , exposant=1 ) :
2     resultat = base ** exposant
3     return resultat
```

On peut alors utiliser la fonction avec un seul paramètre : `puissance(5)` est équivalent à `puissance(5,1)` .

Attention, les paramètres par défaut ne peuvent être qu'à la fin de la liste des paramètres (lors de la définition).

Paramètres nommés

Lors de l'appel, on peut décider de nommer des paramètres, ceci améliore considérablement la lisibilité. Les paramètres n'ont alors plus besoin d'être dans le bon ordre.

```
1 valeur_approchee(f, 0, 1, 0.0001)
2
3 valeur_approchee(fonction=f, valeur_min_de_recherche=0,
4                 valeur_max_de_recherche=1, precision=0.0001)
```

Ceci renforce l'idée qu'il faut nommer correctement les paramètres lors de la définition des fonctions.

On n'est pas obligé de nommer tous les paramètres. Les paramètres non-nommés devront alors être en premier et traités dans l'ordre.

Attention, les paramètres nommés doivent se trouver à la fin de la liste des paramètres (lors de l'appel).

Les paramètres nommés peuvent bien évidemment se combiner avec les valeurs par défaut.

Sortie facultative

Une fonction n'a pas nécessairement besoin d'une valeur de retour (on préférera alors parler de procédure). L'instruction `return` n'est donc pas complètement nécessaire dans une fonction.

Lorsqu'on tente d'affecter le retour d'une procédure, la variable se voit attribuer la valeur `None`. C'est une valeur spéciale qui signifie simplement que la variable n'est pas initialisée (n'a pas de valeur).

2.2 Portée d'une variable

Prenons la formule mathématique $\int_0^1 (t^2 + 1)dt$. La variable t n'a pas de sens en dehors de l'intégrale. En informatique, c'est un peu pareil : les variables informatiques ont une **portée**. En dehors du bloc dans lequel elles ont été affectées, elles n'existent tout simplement (presque) pas.

```
1 mini = 1
2
3 def somme( maxi ):
4     resultat = 0
5     for n in range(mini, maxi+1):
6         resultat = resultat + n
7     return resultat
8
9 valeur = somme(5)
10 print(valeur)    ## Affichera 15
11
12 print(resultat)  ## message d'erreur: la variable resultat n'existe pas
13
14 mini=3
15 valeur = somme(5)
16 print(valeur)    ## Affichera 12
```

Dans ce script, la variable `mini` est définie dans le bloc principal, elle est donc disponible partout, notamment dans la fonction.

Par contre, la variable `resultat` n'a de sens que dans la fonction. Ceci explique le message d'erreur ligne 12. C'est la raison d'être de l'instruction `return` : cela permet de transmettre une valeur en dehors de la fonction.

Dans l'exemple précédent, la variable `mini` est accessible depuis la fonction mais si on essaye de modifier sa valeur dedans, on n'obtiendra pas ce qu'on souhaite. En effet, Python va créer une autre variable appelée également `mini` mais dont la portée sera restreinte à la fonction. On aura ainsi deux variables de même nom dont la portée est différente. Par défaut, Python utilisera la variable dont la portée est la plus petite c'est-à-dire celle définie dans la fonction.

Pour pallier à ce problème, on peut utiliser l'instruction `global` :

```

1 mini = 1
2
3 def somme( maxi ):
4     global mini  ## <-- maintenant, on peut modifier mini
5     mini = 3
6     resultat = 0
7     for n in range(mini, maxi+1):
8         resultat = resultat + n
9     return resultat
10
11 print(mini)  ## Afficher 1
12 valeur = somme(5)
13 print(valeur)  ## Affichera 12
14 print(mini)  ## Affichera 3

```

Attention, c'est en général une très mauvaise idée de modifier une variable globale dans une fonction : le code devient rapidement ingérable et difficile à maintenir.

Le mot clé `global` doit être utilisé avec parcimonie. C'est aussi le cas des variables globales en général. Il est préférable de passer ces variables en paramètre de la fonction.

2.3 Quelques exercices

Exercice 8.

1. Écrire une fonction `somme_puissance(n,p)` qui donne la somme des carrés des nombres entre `n` et `p` (bornes comprises) si `n <= p` et 0 sinon.
Par exemple `somme_puissance(5,7) == 110` car $25 + 36 + 49 = 110$.
2. Modifier cette fonction pour ajouter un paramètre `puissance` optionnel qui vaut 2 par défaut. La nouvelle fonction calculera la somme des puissances `puissance` entre `n` et `p`. Par exemple, `somme_puissance(2,5,puissance=3) == 224` car $2^3 + 3^3 + 4^3 + 5^3 = 8 + 27 + 64 + 125 = 224$.

Exercice 9.

1. Écrire une fonction `distance(xA,yA,xB,yB)` qui calcule la distance entre les points $A(x_A, y_A)$ et $B(x_B, y_B)$.
On utilisera l'instruction `from math import sqrt` en début de script et on pourra utiliser la fonction `sqrt` pour calculer une racine carrée.
2. Modifier cette fonction pour pouvoir calculer la distance dans l'espace entre les points $A(x_A, y_A, z_A)$ et $B(x_B, y_B, z_B)$. Votre fonction doit toujours être capable de calculer la distance dans le plan !

Exercice 10.

1. Écrire une fonction `nombre(num)` qui indique le nombre de zéros nécessaires pour écrire le nombre `num`.
Par exemple, `nombre(5001) == 2` car il faut 2 zéros pour écrire 5001 et `nombre(47) == 0`.
Indication : On peut utiliser la division euclidienne par 10...
2. Modifier cette fonction en `nombre(num, digit)` pour indiquer le nombre de n'importe quel chiffre dans l'écriture de `num`.