VINCENT LE GOFF

APPRENEZ À PROGRAMMER EN PYTHON

DÉVELOPPER EN PYTHON N'A JAMAIS ÉTÉ AUSSI FACILE !

le Livre du Zéro
www.siteduzero.com

Sauf mention contraire, le contenu de cet ouvrage est publié sous la licence :
Creative Commons BY-NC-SA 2.0
La copie de cet ouvrage est autorisée sous réserve du respect des conditions de la licence
Texte complet de la licence disponible sur : http://creativecommons.org/licenses/by-nc-sa/2.0/fr/

Simple IT 2011 - ISBN : 979-10-90085-03-9

Chapitre 15

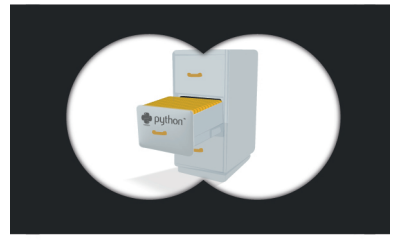
Portée des variables et références

Difficulté : 

Dans ce chapitre, je vais m'attarder sur la portée des variables et sur les références. Je ne vais pas vous faire une visite guidée de la mémoire de votre ordinateur (Python est assez haut niveau pour, justement, ne pas avoir à descendre aussi bas), je vais simplement souligner quelques cas intéressants que vous pourriez rencontrer dans vos programmes.

Ce chapitre n'est pas indispensable, mais je ne l'écris naturellement pas pour le plaisir : vous pouvez très bien continuer à apprendre le Python sans connaître précisément comment Python joue avec les références, mais il peut être utile de le savoir.

N'hésitez pas à relire ce chapitre si vous avez un peu de mal, les concepts présentés ne sont pas évidents.



La portée des variables

En Python, comme dans la plupart des langages, on trouve des règles qui définissent la **portée des variables**. La portée utilisée dans ce sens c'est « quand et comment les variables sont-elles accessibles ? ». Quand vous définissez une fonction, quelles variables sont utilisables dans son corps ? Uniquement les paramètres ? Est-ce qu'on peut créer dans notre corps de fonction des variables utilisables en dehors ? Si vous ne vous êtes jamais posé ces questions, c'est normal. Mais je vais tout de même y répondre car elles ne sont pas dénuées d'intérêt.

Dans nos fonctions, quelles variables sont accessibles ?

On ne change pas une équipe qui gagne : passons aux exemples dès à présent.

```
>>> a = 5
>>> def print_a():
...     """Fonction chargée d'afficher la variable a.
...     Cette variable a n'est pas passée en paramètre de la fonction.
...     On suppose qu'elle a été créée en dehors de la fonction, on veut voir
...     si elle est accessible depuis le corps de la fonction
    """
...
...     print("La variable a = {0}.".format(a))
...
>>> print_a()
La variable a = 5.
>>> a = 8
>>> print_a()
La variable a = 8.
>>>
```

Surprise ! Ou peut-être pas...

La variable `a` n'est pas passée en paramètre de la fonction `print_a`. Et pourtant, Python la trouve, tant qu'elle a été définie avant **l'appel** de la fonction.

C'est là qu'interviennent les différents espaces.

L'espace local

Dans votre fonction, quand vous faites référence à une variable `a`, Python vérifie dans l'**espace local** de la fonction. Cet espace contient les paramètres qui sont passés à la fonction et les variables définies dans son corps. Python apprend ainsi que la variable `a` n'existe pas dans l'espace local de la fonction. Dans ce cas, il va regarder dans l'espace local dans lequel la fonction a été appelée. Et là, il trouve bien la variable `a` et peut donc l'afficher.

D'une façon générale, je vous conseille d'éviter d'appeler des variables qui ne sont pas dans l'espace local, sauf si c'est nécessaire. Ce n'est pas très clair à la lecture ; dans l'absolu, préférez travailler sur des variables globales, cela reste plus propre (nous verrons cela plus bas). Pour l'instant, on ne s'intéresse qu'aux mécanismes, on cherche juste à savoir quelles variables sont accessibles depuis le corps d'une fonction et de quelle façon.

La portée de nos variables

Voyons quelques cas concrets. Je vais les expliquer au fur et à mesure, ne vous en faites pas.

Qu'advient-il des variables définies dans un corps de fonction ?

Voyons un nouvel exemple :

```
1  def set_var(nouvelle_valeur):
2      """Fonction nous permettant de tester la portée des
           variables
3      définies dans notre corps de fonction"""
4
5      # On essaye d'afficher la variable var, si elle existe
6      try:
7          print("Avant l'affectation, notre variable var vaut {0}
               .".format(var))
8      except NameError:
9          print("La variable var n'existe pas encore.")
10     var = nouvelle_valeur
11     print("Après l'affectation, notre variable var vaut {0}.".
           format(var))
```

Et maintenant, utilisons notre fonction :

```
1  >>> set_var(5)
2  La variable var n'existe pas encore.
3  Après l'affectation, notre variable var vaut 5.
4  >>> var
5  Traceback (most recent call last):
6    File "<stdin>", line 1, in <module>
7  NameError: name 'var' is not defined
8  >>>
```

Je sens que quelques explications s'imposent :

– Lors de notre appel à `set_var`, notre variable `var` n'a pu être trouvée par Python : c'est normal, nous ne l'avons pas encore définie, ni dans notre corps de fonction, ni dans le corps de notre programme. Python affecte la valeur `5` à la variable `var`, l'affiche et s'arrête.

– Au sortir de la fonction, on essaye d'afficher la variable `var`... mais Python ne la trouve pas ! En effet : elle a été définie dans le corps de la fonction (donc dans son espace local) et, à la fin de l'exécution de la fonction, l'espace est détruit... donc la variable `var`, définie dans le corps de la fonction, n'existe que dans ce corps et est détruite ensuite.

Python a une règle d'accès spécifique aux variables extérieures à l'espace local : on peut les lire, mais pas les modifier. C'est pourquoi, dans notre fonction `print_a`, on arrivait à afficher une variable qui n'était pas comprise dans l'espace local de la fonction. En revanche, on ne peut modifier la valeur d'une variable extérieure à l'espace local, par affectation du moins. Si dans votre corps de fonction vous faites `var = nouvelle_valeur`, vous n'allez *en aucun cas* modifier une variable extérieure au corps.

En fait, quand Python trouve une instruction d'affectation, comme par exemple `var = nouvelle_valeur`, il va changer la valeur de la variable dans l'espace local de la fonction. Et rappelez-vous que cet espace local est détruit après l'appel à la fonction.

Pour résumer, et c'est ce qu'il faut retenir, *une fonction ne peut modifier, par affectation, la valeur d'une variable extérieure à son espace local.*

Cela paraît plutôt stupide au premier abord... mais pas d'impatience. Je vais relativiser cela assez rapidement.

Une fonction modifiant des objets

J'espère que vous vous en souvenez, *en Python, tout est objet.* Quand vous passez des paramètres à votre fonction, ce sont des objets qui sont transmis. Et pas les valeurs des objets, mais bien les objets eux-mêmes, ceci est très important.

Bon. On ne peut affecter une nouvelle valeur à un paramètre dans le corps de la fonction. Je ne reviens pas là-dessus. En revanche, on pourrait essayer d'appeler une méthode de l'objet qui le modifie... Voyons cela :

```
>>> def ajouter(liste, valeur_a_ajouter):
...     """Cette fonction insère à la fin de la liste la valeur
    que l'on veut ajouter"""
...     liste.append(valeur_a_ajouter)
...
>>> ma_liste=['a', 'e', 'i']
>>> ajouter(ma_liste, 'o')
>>> ma_liste
['a', 'e', 'i', 'o']
>>>
```

Cela marche ! On passe en paramètres notre objet de type `list` avec la valeur à ajouter. Et la fonction appelle la méthode `append` de l'objet. Cette fois, au sortir de la fonction, notre objet a bel et bien été modifié.

Je vois pas pourquoi. Tu as dit qu'une fonction ne pouvait pas affecter de nouvelles valeurs aux paramètres ?

Absolument. Mais c'est cela la petite subtilité dans l'histoire : on ne change pas du tout la valeur du paramètre, on appelle juste une méthode de l'objet. Et cela change tout. Si vous vous embrouillez, retenez que, dans le corps de fonction, si vous faites `parametre = nouvelle_valeur`, le paramètre ne sera modifié que dans le corps de la fonction. Alors que si vous faites `parametre.methode_pour_modifier(...)`, l'objet derrière le paramètre sera bel et bien modifié.

On peut aussi modifier les attributs d'un objet, par exemple changer une case de la liste ou d'un dictionnaire : ces changements aussi seront effectifs au-delà de l'appel de la fonction.

Et les références, dans tout cela ?

J'ai parlé des références, et vous ai promis d'y consacrer une section ; c'est maintenant qu'on en parle !

Je vais schématiser volontairement : les variables que nous utilisons depuis le début de ce cours cachent en fait des références vers des objets.

Concrètement, j'ai présenté les variables comme ceci : un nom identifiant pointant vers une valeur. Par exemple, notre variable nommée `a` possède une valeur (disons 0).

En fait, une variable est un nom identifiant, pointant vers une référence d'un objet. La référence, c'est un peu sa position en mémoire. Cela reste plus haut niveau que les pointeurs en C par exemple, ce n'est pas vraiment la mémoire de votre ordinateur. Et on ne manipule pas ces références directement.

Cela signifie que deux variables peuvent pointer sur le même objet.

Bah... bien sûr, rien n'empêche de faire deux variables avec la même valeur.

Non non, je ne parle pas de valeurs ici mais d'objets. Voyons un exemple, vous allez comprendre :

```
>>> ma_liste1 = [1, 2, 3]
>>> ma_liste2 = ma_liste1
>>> ma_liste2.append(4)
>>> print(ma_liste2)
[1, 2, 3, 4]
>>> print(ma_liste1)
[1, 2, 3, 4]
>>>
```

Nous créons une liste dans la variable `ma_liste1`. À la ligne 2, nous affectons `ma_liste1` à la variable `ma_liste2`. On pourrait croire que `ma_liste2` est une copie de `ma_liste1`. Toutefois, quand on ajoute 4 à `ma_liste2`, `ma_liste1` est aussi modifiée.

On dit que `ma_liste1` et `ma_liste2` contiennent une référence vers le même objet : si on modifie l'objet depuis une des deux variables, le changement sera visible depuis les deux variables.

Euh... j'essaye de faire la même chose avec des variables contenant des entiers et cela ne marche pas.

C'est normal. Les entiers, les flottants, les chaînes de caractères, n'ont aucune méthode travaillant sur l'objet lui-même. Les chaînes de caractères, comme nous l'avons vu, ne modifient pas l'objet appelant mais renvoient un nouvel objet modifié. Et comme nous venons de le voir, le processus d'affectation n'est pas du tout identique à un appel de méthode.

Et si je veux modifier une liste sans toucher à l'autre ?

Eh bien c'est impossible, vu comment nous avons défini nos listes. Les deux variables pointent sur le même objet par jeu de références et donc, inévitablement, si vous modifiez l'objet, vous allez voir le changement depuis les deux variables. Toutefois, il existe un moyen pour créer un nouvel objet depuis un autre :

```
1 >>> ma_liste1 = [1, 2, 3]
2 >>> ma_liste2 = list(ma_liste1) # Cela revient à copier le
      contenu de ma_liste1
3 >>> ma_liste2.append(4)
4 >>> print(ma_liste2)
5 [1, 2, 3, 4]
6 >>> print(ma_liste1)
7 [1, 2, 3]
8 >>>
```

À la ligne 2, nous avons demandé à Python de créer un nouvel objet basé sur `ma_liste1`. Du coup, les deux variables ne contiennent plus la même référence : elles modifient des objets différents. Vous pouvez utiliser la plupart des constructeurs (c'est le nom qu'on donne à `list` pour créer une liste par exemple) dans ce but. Pour des dictionnaires, utilisez le constructeur `dict` en lui passant en paramètre un dictionnaire déjà construit et vous aurez en retour un dictionnaire, semblable à celui passé en paramètre, mais seulement semblable par le contenu. En fait, il s'agit d'une copie de l'objet, ni plus ni moins.

Pour approcher de plus près les références, vous avez la fonction `id` qui prend en paramètre un objet. Elle renvoie la position de l'objet dans la mémoire Python sous la

forme d'un entier (plutôt grand). Je vous invite à faire quelques tests en passant divers objets en paramètre à cette fonction. Sachez au passage que `is` compare les ID des objets de part et d'autre et c'est pour cette raison que je vous ais mis en garde quant à son utilisation.

```
>>> ma_liste1 = [1, 2]
>>> ma_liste2 = [1, 2]
>>> ma_liste1 == ma_liste2 # On compare le contenu des listes
True
>>> ma_liste1 is ma_liste2 # On compare leur référence
False
>>>
```

Je ne peux que vous encourager à faire des tests avec différents objets. Un petit tour du côté des variables globales ?

Les variables globales

Il existe un moyen de modifier, dans une fonction, des variables extérieures à celle-ci. On utilise pour cela des **variables globales**.

Cette distinction entre variables locales et variables globales se retrouve dans d'autres langages et on recommande souvent d'éviter de trop les utiliser. Elles peuvent avoir leur utilité, toutefois, puisque le mécanisme existe. D'un point de vue strictement personnel, tant que c'est possible, je ne travaille qu'avec des variables locales (comme nous l'avons fait depuis le début de ce cours) mais il m'arrive de faire appel à des variables globales quand c'est nécessaire ou bien plus pratique. Mais ne tombez pas dans l'extrême non plus, ni dans un sens ni dans l'autre.

Le principe des variables globales

On ne peut faire plus simple. On déclare dans le corps de notre programme, donc en dehors de tout corps de fonction, une variable, tout ce qu'il y a de plus normal. Dans le corps d'une fonction qui doit modifier cette variable (changer sa valeur par affectation), on déclare à Python que la variable qui doit être utilisée dans ce corps est globale.

Python va regarder dans les différents espaces : celui de la fonction, celui dans lequel la fonction a été appelée... ainsi de suite jusqu'à mettre la main sur notre variable. S'il la trouve, il va nous donner le plein accès à cette variable dans le corps de la fonction.

Cela signifie que nous pouvons y accéder en lecture (comme c'est le cas sans avoir besoin de la définir comme variable globale) mais aussi en écriture. Une fonction peut donc ainsi changer la valeur d'une variable directement.

Mais assez de théorie, voyons un exemple.

Utiliser concrètement les variables globales

Pour déclarer à Python, dans le corps d'une fonction, que la variable qui sera utilisée doit être considérée comme globale, on utilise le mot-clé `global`. On le place généralement après la définition de la fonction, juste en-dessous de la `docstring`, cela permet de retrouver rapidement les variables globales sans parcourir tout le code (c'est une simple convention). On précise derrière ce mot-clé le nom de la variable à considérer comme globale :

```
1  >>> i = 4 # Une variable, nommée i, contenant un entier
2  >>> def inc_i():
3  ...     """Fonction chargée d'incrémenter i de 1"""
4  ...     global i # Python recherche i en dehors de l'espace
       local de la fonction
5  ...     i += 1
6  ...
7  >>> i
8  4
9  >>> inc_i()
10 >>> i
11 5
12 >>>
```

Si vous ne précisez pas à Python que `i` doit être considérée comme globale, vous ne pourrez pas modifier réellement sa valeur, comme nous l'avons vu plus haut. En précisant `global i`, Python permet l'accès en lecture et en écriture à cette variable, ce qui signifie que vous pouvez changer sa valeur par affectation.

J'utilise ce mécanisme quand je travaille sur plusieurs classes et fonctions qui doivent s'échanger des informations d'état par exemple. Il existe d'autres moyens mais vous connaissez celui-ci et, tant que vous maîtrisez bien votre code, il n'est pas plus mauvais qu'un autre.

En résumé

- Les variables locales définies avant l'appel d'une fonction seront accessibles, depuis le corps de la fonction, en lecture seule.
- Une variable locale définie dans une fonction sera supprimée après l'exécution de cette fonction.
- On peut cependant appeler les attributs et méthodes d'un objet pour le modifier durablement.
- Les variables globales se définissent à l'aide du mot-clé `global` suivi du nom de la variable préalablement créée.
- Les variables globales peuvent être modifiées depuis le corps d'une fonction (à utiliser avec prudence).