ENTWANNE

Zeste de savoir

Variables, scopes et closures en Python (billet)

31 juillet 2019

Table des matières

Après plusieurs années à errer sur des forums d'entraide spécialisés en Python, il est courant de retomber régulièrement sur les mêmes problèmes.

Et s'il est un sujet qui concentre beaucoup d'incompréhension et d'interrogations, c'est celui de la gestion des variables. J'ai en tête de nombreux sujets demandant pourquoi telle valeur est modifiée alors qu'elle ne devrait pas, d'utilisation du mot-clé `global` à tire-larigot, etc. C'est suite à ce sujet qui rassemble quelques unes de ces erreurs que je me suis décidé à rédiger ce billet.

Les variables sont l'un des premiers mécanismes que l'on rencontre en Python, mais on peut facilement constater que leur fonctionnement est souvent mal compris (car malheureusement souvent mal enseigné aussi). Elles forment un point d'incompréhension majeur du langage dont vont découler beaucoup d'autres, notamment en raison de la gestion des scopes et des variables mutables.

1. Une histoire de variables

1.1. Représentation des variables

Lorsqu'on exécute des calculs, appelle des fonctions ou effectue divers traitements, il est souvent intéressant de pouvoir en conserver le résultat. C'est tout le principe des variables : garder la trace d'une valeur.

Le concept de variables existe dans la majorité des langages de programmation, mais il peut prendre différentes formes. Les deux plus courantes vont être les variables sous forme de boîtes et les étiquettes.

Les boîtes sont la représentation que l'on rencontre dans des langages tels que le C, où une variable correspond à une case mémoire. Il s'agit donc d'un nom associé à un emplacement mémoire, et assigner une valeur à la variable permet de stocker cette valeur à cet emplacement précis. Deux variables distinctes correspondent à deux emplacements différents.

Les étiquettes sont la représentation utilisée par Python, elles peuvent sembler similaires aux boîtes à l'utilisation, mais diffèrent en bien des points. En Python une variable est une étiquette – juste un nom – posée sur une valeur. C'est-à-dire que la valeur existe quelque part en mémoire et qu'on vient lui attacher une étiquette. On peut aisément placer plusieurs étiquettes sur une même valeur, mais aussi retirer une étiquette pour la placer sur une autre valeur.

La représentation des variables en Python est expliquée dans cet article qui décrit très bien les choses.

1.2. Déclaration et définition

Certains langages distinguent la déclaration et la définition d'une variable, ce n'est pas le cas en Python où les deux sont confondues. Mais il s'agit pourtant bien de deux concepts différents.

- Déclarer une variable, c'est indiquer au compilateur que tel nom existe dans tel contexte pour lui permettre de résoudre les utilisations de ce nom.
- Définir une variable, c'est lui assigner une valeur (soit en Python poser l'étiquette sur cette valeur).

`foo = 'bar'` en Python revient à déclarer une variable `foo` puis à la définir sur `'bar'`. Par la suite, un `foo = 'rab'` dans le même contexte revient à redéfinir une variable déjà déclarée. C'est-à-dire déplacer l'étiquette sur une autre valeur.

Pour être utilisée, une variable a besoin d'avoir été déclarée et définie, sans quoi vous rencontreriez une erreur de type `NameError` ou `UnboundLocalError` selon les cas.

```
>>> def func():
...     print(x)
...
>>> func()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func
NameError: name 'x' is not defined
```

```
>>> def func():
...     print(x)
...     x = 0
```

```
...
>>> func()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func
UnboundLocalError: local variable 'x' referenced before assignment
```

Vous pourriez vous demander pourquoi je tiens à marquer cette distinction si Python la masque, mais c'est parce que cela a de l'importance sur d'autres notions expliquées dans la suite du billet.

1.3. Plusieurs étiquettes sur une même valeur (références multiples)

Comme je le disais, il est parfaitement envisageable d'avoir plusieurs étiquettes posées sur une même valeur, c'est même quelque chose de très courant. Ça se produit même chaque fois que l'on assigne une variable à une autre.

```
a = []
b = a
```

Ici nous créons une nouvelle liste à laquelle nous ajoutons une première étiquette `a`. Puis nous ajoutons une seconde étiquette `b` sur cette même liste. `a` et `b` référencent la même valeur, la même liste.

Cela peut être mis en évidence à l'aide de l'opérateur `is` :

```
>>> a is b
True
```

Le résultat aurait été `False` si les deux listes avaient été distinctes :

```
>>> c = []
>>> a is c
False
```

Notez que j'utilise ici des listes (objets mutables) afin de ne pas être être embêté dans mes exemples par les optimisations du compilateur, mais le fait que `b = a` ajoute une étiquette supplémentaire à la valeur existante est vrai pour tout type de valeur. En d'autres termes, `a is b` sera toujours vrai après un `b = a`, quelle que soit la valeur initiale de `a`.

La conséquence de cela est que modifier `b` revient à modifier `a` (et inversement), puisque les deux étiquettes sont posées sur la même valeur :

```
>>> b.append('value')
>>> a
['value']
>>> a += ['foobar']
>>> b
['value', 'foobar']
```

Dans ce dernier exemple, faites bien attention à l'opérateur `+=` qui n'est pas simplement un opérateur d'assignation mais peut aussi modifier la valeur existante.

Des cas de références multiples, on en rencontre très régulièrement en Python, en voici deux un peu plus insidieux.

1.3.0.1. Multiplication de liste

```
>>> table = [[0] * 4] * 3
>>> table[0][0] = 1
>>> table
[[1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0]]
```

`table` est une liste contenant 3 fois la même valeur, modifier l'une d'elle revient à modifier les autres.

(Techniquement, `table[0]` est aussi une liste composée de 4 fois la même valeur, mais cette valeur étant immuable et donc redéfinie à chaque changement, il n'y a pas d'effet de bord.)

Pour pallier à ce problème, on utilisera plutôt une liste en intension avec un `range`.

```
>>> table = [[0] * 4 for _ in range(3)]
>>> table[0][0] = 1
>>> table
[[1, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

1.3.0.2. Valeur par défaut d'un paramètre de fonction

```
>>> def append_to_list(value, dest=[]):
...     dest.append(value)
...     return dest
...
>>> append_to_list(10)
[10]
>>> append_to_list(15)
[10, 15]
```

Le problème ici est que les valeurs par défaut des paramètres sont définies une bonne fois pour toutes lors de la définition de la fonction, et conservées pour tous les appels. Donc chaque appel à `append_to_list` utilisant la valeur par défaut référencera la même liste.

Pour contourner ce soucis, il est conseillé d'éviter les valeurs par défaut mutables, ou d'utiliser des sentinelles (`None` par exemple).

```
>>> def append_to_list(value, dest=None):
...     if dest is None:
...         dest = []
...     dest.append(value)
...     return dest
...
>>> append_to_list(10)
[10]
>>> append_to_list(15)
[15]
```

2. Référencement et déréférencement

2.1. Étiquettes = références

Plutôt que d'étiquettes, on parle plus couramment de références, mais l'idée est exactement la même : une variable est une référence vers une valeur. Deux variables distinctes peuvent référencer la même valeur. Une variable peut être réassignée pour référencer une valeur différente.

Chaque définition d'une variable en Python crée une nouvelle référence vers la valeur assignée. Cela est vrai pour toute variable, incluant au passages les paramètres d'une fonction, qui deviennent lors de l'appel de nouvelles références vers les valeurs passées en arguments. Il en est de même pour les valeurs insérées dans un conteneur (liste, *tuple*, dictionnaire, etc.) : c'est une référence vers la valeur qui est stockée dans le conteneur.

Tant qu'il existe au moins une référence vers une valeur, on dit que cette valeur est référencée. Une valeur référencée ne peut jamais être supprimée de la mémoire (cela poserait des problèmes pour les utilisations futures de la valeur via d'autres variables).

Comment alors supprimer une valeur ?

2.2. Supprimer une valeur

Dans un premier temps il faut bien faire la distinction entre supprimer une variable et supprimer une valeur, les deux n'étant pas du tout équivalents.

Supprimer une variable, c'est faire en sorte que son nom n'existe plus. Ça se produit naturellement lorsque l'on sort du contexte dans lequel la variable est déclarée, c'est ce qu'il se passe pour les variables locales après l'exécution d'une fonction.

Cela peut aussi être déclenché manuellement à l'aide du mot-clé `del`. `del foo` a pour but de supprimer la variable `foo`, de faire en sorte que le nom `foo` ne corresponde plus à rien.

Quand une variable est supprimée, la référence vers la valeur est rompue. On dit que l'on déréférence la valeur. Il y a d'autres moyens que la suppression de variable pour déréférencer une valeur : la réassignation est aussi très courante, comme dans le code suivant.

```
>>> l = ['foo']
>>> l = ['bar'] # L'ancienne valeur de l est déréférencée
```

Le modèle mémoire de Python fonctionne à l'aide d'un compteur de références : chaque assignation d'une valeur à une variable incrémente ce compteur, et chaque suppression le décrémente. Quand ce compteur atteint 0 (ce qui veut dire que la valeur n'est plus référencée par aucune variable, et donc plus accessible de nulle part dans le code), la valeur peut alors être supprimée en toute sécurité, et c'est le travail réalisé par le ramasse-miettes pour libérer la mémoire.

Ainsi, pour supprimer une valeur et libérer l'espace mémoire qu'elle occupe, il est nécessaire de la déréférencer totalement, de supprimer toutes les références vers cette valeur. Cela concerne les variables aussi bien que les références stockées dans les conteneurs. Donc un `del` sur une variable ne suffit pas si la valeur est toujours référencée par une autre variable, il faut aussi s'occuper des autres.

`del` peut d'ailleurs être utilisé pour supprimer tout type de référence et pas seulement les variables.

```
>>> value = object() # value est une référence vers la valeur
>>> l = [value] # on crée une seconde référence depuis la liste
>>> d = {'key': value} # puis une troisième dans le dictionnaire
>>>
>>> del value # plus que deux références
>>> del l[0] # plus qu'une
>>> del d['key'] # plus du tout, la valeur est déréférencée
```

Quand Python supprime une valeur, il en appelle la méthode spéciale `__del__` (si elle en possède une). Cette méthode permet d'intervenir juste avant la suppression de l'objet pour finaliser des traitements dessus. Pour reprendre l'exemple précédent :

```
>>> class Obj:
...     def __del__(self):
...         print('deleting', self)
...
```

```
>>> value = Obj()
>>> l = [value]
>>> d = {'key': value}
>>>
>>> del value
>>> del l[0]
>>> del d['key']
deleting <__main__.Obj object at 0x7f33ade3fba8>
```

On constate bien que ce n'est pas l'utilisation d'un `del` qui déclenche l'appel à `__del__`, mais bien le déréférencement total de notre valeur. La perte de toutes les références vers cette dernière.

Je n'ai utilisé ici que des déréférencements explicites, mais ils peuvent aussi être provoqués par des réassignations ou par un déréférencement parent (supprimer une liste revient à en déréférencer toutes les valeurs).

```
>>> value = Obj()
>>> l = [value]
>>> d = {'key': value}
>>>
>>> value = None
>>> l[:] = []
>>> d['key'] = 20
deleting <__main__.Obj object at 0x7f33ade3fb38>
```

2.3. Références cycliques

Dans tout cela, un cas qui peut être problématique est celui des références cycliques. Que faire par exemple si un objet `obj1` contient une référence vers un objet `obj2`, qui contient lui-même une référence vers `obj1` ?

```
>>> obj1 = Obj()
>>> obj2 = Obj()
>>> obj1.ref = obj2
>>> obj2.ref = obj1
>>>
>>> del obj1
>>> del obj2
```

Comme vous le voyez, il ne se passe rien, les deux valeurs étant toujours référencées. Mais à la sortie du programme, les références cycliques seront résolues et les valeurs supprimées.

```
>>> ^D
deleting <__main__.Obj object at 0x7f8fcf561b70>
deleting <__main__.Obj object at 0x7f8fcf561ba8>
```

Il peut néanmoins être gênant de devoir attendre la fin du programme pour collecter ces valeurs (elles occupent inutilement de l'espace mémoire obligeant ainsi le programme à en allouer toujours plus), et dans ce genre de cas il est utile de pouvoir invoquer manuellement le ramasse-miettes. Cela se fait à l'aide la méthode `collect` du module `gc`, qui renvoie le nombre de valeurs non atteignables trouvées et supprimées.

```
>>> obj1 = Obj()
>>> obj2 = Obj()
>>> obj1.ref = obj2
>>> obj2.ref = obj1
>>>
>>> del obj1
>>> del obj2
>>>
>>> import gc
>>> gc.collect()
deleting <__main__.Obj object at 0x7f4077157b70>
deleting <__main__.Obj object at 0x7f4077157be0>
4
```

Quelques liens pour aller plus loin sur le sujet :

— https ://rushter.com/blog/python-garbage-collector/
— https ://docs.python.org/3/library/gc.html

Ainsi que le module `weakref`, qui permet d'avoir une « référence faible » sur une valeur (référence qui n'existe qu'à l'utilisation et qui n'empêche donc pas la valeur d'être supprimée) :

— https ://docs.python.org/3/library/weakref.html

3. Portée des variables

3.1. Scope

Une variable n'est certes qu'un nom posé sur une valeur, mais plusieurs variables portant le même nom peuvent coexister au sein d'un programme. Un nom de variable est pourtant unique, mais seulement dans le contexte où il est déclaré, dans son scope.

Une variable peut en effet être déclarée au niveau global d'un module et ainsi être accessible depuis n'importe où dans ce module (on dit que sa portée est globale), mais elle peut aussi être

déclarée dans le scope d'une fonction et accessible depuis cette fonction uniquement (portée locale).

Il est donc tout à fait possible d'avoir une variable locale à une fonction portant le même nom qu'une variable du module, comme dans l'exemple suivant pour la fonction `g`. La variable locale prendra la priorité sur la globale lors de la résolution du nom par le compilateur.

```
>>> a = 0
>>> def f():
...     print('f: a =', a)
...
>>> def g():
...     a = 1
...     print('g: a =', a)
...
>>> print('global: a =', a)
global: a = 0
>>> f()
f: a = 0
>>> g()
g: a = 1
>>> print('global: a =', a)
global: a = 0
```

Le scope de déclaration définit la portée dans laquelle est accessible une variable. Cette portée inclut le contexte courant, mais aussi tous les scopes enfants. L'exemple précédent ne présentait que deux niveaux, global et local, mais il y en a en réalité une multitude, les scopes de fonctions pouvant s'imbriquer.

```
>>> def outer():
...     def inner():
...         print(var)
...     var = 10
...     inner()
...
>>> outer()
10
```

Ici, une variable déclarée dans le scope de la fonction `outer` est accessible dans `inner`. Et il en serait de même si `inner` définissait encore une sous-fonction, celle-ci aurait accès à `var` déclarée dans un scope parent.

L'inverse n'est pas vrai, une variable déclarée dans un scope enfant n'est pas accessible depuis le parent.

```
>>> def outer():
...     def inner():
...         var = 10
...     print(var)
...
>>> outer()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in outer
NameError: name 'var' is not defined
```

Lorsqu'on quitte un scope, le scope ainsi que les variables qu'il contient sont détruit · e · s, ce qui permet de supprimer les valeurs si elles ne sont plus référencées.

```
>>> class Obj:
...     def __del__(self):
...         print('deleting', self)
...
>>> def func():
...     obj1 = Obj()
...
>>> func()
deleting <__main__.Obj object at 0x7f1f6eeedcc0>
```

Bien sûr, ça ne s'applique pas au cas où la fonction renverrait une valeur qui serait assignée à une variable, puisqu'on l'on en conserverait alors une référence.

```
>>> def func():
...     obj1 = Obj()
...     obj2 = Obj()
...     return obj1
...
>>> ret = func()
deleting <__main__.Obj object at 0x7f1f6eeedcc0>
>>> del ret
deleting <__main__.Obj object at 0x7f1f6eeedc88>
```

Ce que cela nous montre aussi, c'est que bien qu'une variable soit associée à un scope, sa valeur peut elle transiter de scope en scope et ainsi remonter vers les parents à l'aide du `return`. La variable référençant la valeur change (on passe de `obj1` à `ret`), mais la valeur reste toujours la même (elle n'est pas copiée).

3.2. Variables locales et extérieures

Le corps d'une fonction a donc accès aux variables définies dans le scope courant (les variables locales), mais aussi à celles des scopes parents (les variables extérieures). Ces variables extérieures ne se limitent pas aux variables globales : dans le premier exemple donné plus haut sur `outer` et `inner`, `inner` accède à une variable `var` qui est définie dans `outer`.

Il ne s'agit donc pas d'une variable globale (elle n'existe pas dans le scope global du module), mais d'une variable locale d'`outer`, et donc une variable extérieure (non-locale) à `inner`.

Les variables extérieures peuvent être utilisée de la même manière que les variables locales. Et lors de la résolution du nom si elles sont plusieurs à porter le même nom, c'est la variable du scope parent le plus proche qui sera utilisée en priorité.

```
>>> var = 10
>>> def outer():
...     var = 20
...     def inner():
...         print(var)
...     inner()
...
>>> outer()
20
>>> var
10
```

Cet exemple nous amène à une petite subtilité des variables extérieures. Nous y voyons que nous pouvons définir dans une fonction une nouvelle variable portant le nom d'une variable extérieure (sans que cela n'affecte la variable extérieure). Comment alors redéfinir la valeur d'une variable extérieure ?

Il est dans ce cas nécessaire d'indiquer à Python qu'une variable de ce nom est déjà déclarée à l'extérieur. Pour cela, deux mots-clés sont à notre disposition : `global` et `nonlocal`.

Le premier permet d'indiquer qu'une variable est globale, et Python comprendra qu'elle devra être déclarée et définie dans le scope global. Il s'utilise suivi d'un ou plusieurs noms de variables : `global foo` ou `global foo, bar, baz`.

```
>>> x = 5
>>> def define_globals():
...     global x
...     x = 10
...
>>> x
5
>>> define_globals()
>>> x
10
```

On notera que la variable n'a pas besoin d'avoir déjà été définie dans le scope global pour être utilisée dans notre fonction.

```
>>> def define_globals():
...     global y
...     y = 2
...
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
>>> define_globals()
>>> y
2
```

Le mot-clé `nonlocal` est similaire mais pour indiquer qu'une variable existe dans un scope local parent. Au contraire de `global`, cette variable doit donc nécessaire avoir été définie dans un scope parent (autre que le scope global).

```
>>> def outer():
...     x = 0
...     def inner():
...         y = 0
...         def inception():
...             nonlocal y
...             y = 10
...         inception()
...         nonlocal x
...         x = y
...     inner()
...     return x
...
>>> outer()
10
```

L'omission de l'un de ces deux `nonlocal` aurait pour effet de redéclarer une variable locale (`x` ou `y`), et donc ne permettrait pas la remontée de la valeur `10`.

Si plusieurs scopes parents déclarent une variable du même nom, c'est la variable du scope le plus proche qui sera utilisée, comme c'est le cas pour tout accès à une variable extérieure.

Par ailleurs, il est souvent dit que les mots-clés `global` et `nonlocal` sont nécessaires pour modifier une variable extérieure, pour y accéder en écriture.

Cela est faux, ils ne sont nécessaires que pour la redéfinir. Il est par exemple parfaitement possible d'utiliser depuis une fonction la méthode `append` d'une liste définie au niveau global, sans avoir à utiliser le mot-clé `global`.

```
>>> values = []
>>> def append_value(value):
...     values.append(value)
...
>>> append_value(0)
>>> append_value(1)
>>> append_value(2)
>>> values
[0, 1, 2]
```

Ainsi, les utilisations de `global` ou `nonlocal` sont en réalité plutôt rares, et il est généralement déconseillé d'utiliser ces mots-clés (ils prêtent à confusion). Mais leur connaissance permet de résoudre certains cas problématiques.

3.3. Closure (fermeture)

Je disais quelques paragraphes plus haut que le contexte local à une fonction est détruit à la sortie de cette fonction. Je précisais ensuite que cela n'entraînait pas nécessairement la destruction de toutes les valeurs de ce contexte si certaines étaient toujours référencées, comme ça peut être le cas de la valeur de retour de la fonction.

Un autre cas intéressant est celui des *closures* (fermetures). Celles-ci vont permettre de capturer des variables locales pour qu'elles soient toujours accessibles dans un contexte particulier.

Je sais que ces explications ne sont pas très claires et je préfère alors vous fournir tout de suite un exemple :

```
>>> def cached_addition():
...     cache = {}
...     def addition(x, y):
...         if (x, y) not in cache:
...             print(f'Computing {x} + {y}')
...             cache[x, y] = x + y
...         return cache[x, y]
...     return addition
...
>>> addition = cached_addition()
>>> addition(1, 2)
Computing 1 + 2
3
>>> addition(1, 2)
3
>>> addition(2, 3)
Computing 2 + 3
5
```

Ce n'est pas l'exemple le plus utile, mais on pourrait imaginer une fonction plus complexe à la place de l'addition, dont la mise en cache du résultat serait nécessaire.

L'idée ici est que notre fonction `cached_addition` retourne à chaque appel une nouvelle fonction `addition` créée dynamiquement, utilisant un cache particulier. Ce cache est une variable définie localement dans `cached_addition` et donc accessible depuis `addition`.

Cependant, une fois l'appel à `cached_addition` terminé, son scope local est détruit, ce qui doit impliquer la destruction des valeurs qu'il contient. Ici, on voit bien que `cache` lui survit puisqu'il continue à être utilisé sans problème par la fonction `addition`.

Ce qu'il se passe c'est que la fonction `addition` crée une *closure* qui emprisonne les variables locales des scopes parents qu'elle utilise. Cela permet à ces valeurs d'être toujours référencées. On peut d'ailleurs constater que notre fonction `addition` possède un attribut spécial `__closure__`.

```
>>> addition.__closure__
(<cell at 0x7f3a700a5d98: dict object at 0x7f3a70174fc0>,)
>>> addition.__closure__[0].cell_contents
{(1, 2): 3, (2, 3): 5}
```

L'intérêt des *closures*, c'est que plusieurs appels à `cached_addition` distincts renverront des fonctions utilisant un cache différent, parce qu'il s'agira à chaque fois d'une nouvelle variable locale.

```
>>> other_addition = cached_addition()
>>> other_addition(1, 2)
Computing 1 + 2
3
```

Le mécanisme de *closures* est souvent utilisé au sein de décorateurs puisqu'il permet de facilement attacher des variables à une fonction créée dynamiquement (quelques exemples peuvent être trouvés ici).

Nous avons vu que la *closure* permettait la persistance en mémoire de certaines valeurs en les capturant. Mais cette *closure* n'est pas éternelle est disparaît naturellement quand elle est elle-même déréférencée, c'est à dire quand la fonction qui emprisonne ces valeurs disparaît.

```
>>> class Obj:
...     def __del__(self):
...         print('deleting', self)
...
>>> def outer():
...     obj = Obj()
...     def inner():
...         print(obj)
```

```
...     return inner
...
>>> func1 = outer()
>>> func2 = outer()
>>> func1()
<__main__.Obj object at 0x7f58ba68dd68>
>>> func2()
<__main__.Obj object at 0x7f58ba68de10>
>>> del func1
deleting <__main__.Obj object at 0x7f58ba68dd68>
>>> del func2
deleting <__main__.Obj object at 0x7f58ba68de10>
```

Et pour terminer sur les *closures* voici un autre billet expliquant le concept avec des exemples en Python et en JS : http ://sametmax.com/closure-en-python-et-javascript/

3.4. Délimitation d'un scope

On pourrait croire, comme c'est le cas dans d'autres langages, qu'un nouveau scope est créé pour chaque bloc de code (identifié en Python par le niveau d'indentation). Mais le code suivant, plutôt courant, serait alors invalide :

```
>>> items = [1, 2, 3, ...]
>>> if items:
...     value = items[0]
... else:
...     value = None
...
>>> value
1
```

On remarque bien ici que notre variable `value` est définie dans le scope courant et non dans un scope fils créé spécialement pour le bloc conditionnel. Le niveau d'indentation n'a pas d'incidence sur le scope.

En fait, outre les modules, qui forment le scope global, seules les classes et les fonctions permettent de définir de nouveaux scopes. Ce n'est donc pas le cas des autres blocs de code comme les conditions, les boucles ni même les blocs `with`.

Cela peut être particulièrement trompeur pour les boucles `for`, et notamment pour leur variable d'itération qui n'est donc pas propre à la boucle. Elle est définie et existe à l'extérieur.

```
>>> for i in range(10):
...     pass
...
```

```
>>> i
9
```

Ce qui peut mener à des cas plus embêtants si l'on imaginait que cette variable serait capturée dans une *closure* propre à la boucle.

```
>>> functions = []
>>> for i in range(3):
...     def add_func(x):
...         return i + x
...     functions.append(add_func)
...
>>> functions
[<function add_func at 0x7ff229851268>, <function add_func at
    0x7ff2298512f0>, <function add_func at 0x7ff229851378>]
>>> [f(0) for f in functions]
[2, 2, 2]
```

Toutes les fonctions `f` renvoient la même valeur pour 0 car toutes accèdent à la même variable `i`, qui vaut 2 en sortie de boucle.

La solution dans ce genre de cas est donc de créer une fonction englobante et de l'appeler afin de tirer profit du mécanisme de *closures*. L'exemple précédent pourrait être corrigé comme suit.

```
>>> functions = []
>>> for i in range(3):
...     def get_add_func(i):
...         def add_func(x):
...             return i + x
...         return add_func
...     functions.append(get_add_func(i))
...
>>> [f(0) for f in functions]
[0, 1, 2]
```

La fonction `get_add_func` (qui pourrait tout aussi bien être placée en dehors de la boucle, ce qui serait même préférable) possède un paramètre `i` qui sera capturé dans la *closure* de la sous-fonction `add_func`. Ainsi, à chaque tour de boucle `get_add_func` est appelée avec une valeur différente, et c'est cette valeur qui est chaque fois emprisonnée.

3.5. Variables non déclarées ou non définies

Au tout début de ce billet j'évoquais les exceptions `NameError` et `UnboundLocalError` qui peuvent être levées lors de l'accès à une variable.

La première (`NameError`) est levée si le nom d'une variable n'existe pas, c'est-à-dire qu'elle n'est déclarée ni dans le scope courant, ni dans les parents.

```
>>> def func():
...     print(x)
...
>>> func()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func
NameError: name 'x' is not defined
```

La seconde (`UnboundLocalError`) est plus subtile et survient pour une variable déclarée mais non définie.

Pour rappel, l'instruction `x = 10` a pour effet de déclarer la variable `x` puis de la définir avec pour valeur 10. Ces deux opérations n'interviennent pas au même moment : la définition se fait au moment où cette instruction est rencontrée, mais la déclaration est valable pour tout le scope. C'est donc comme si la variable avait été déclarée au tout début du scope.

Ainsi, prenons l'exemple suivant :

```
>>> def func():
...     print(x)
...     x = 0
...
>>> func()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func
UnboundLocalError: local variable 'x' referenced before assignment
```

Nous essayons ici d'afficher la valeur de `x` mais celle-ci n'est pas encore définie. En revanche, le nom de notre variable existe bel et bien, car elle est déclarée dans le scope courant, à la ligne suivante.

Cette erreur est très courante quand on souhaite redéfinir une variable globale après l'avoir utilisée, mais en omettant le mot-clé `global`.

```
>>> var = 0
>>>
>>> def func1():
...     print(var)
...     var = 10
...
>>> func1()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func1
UnboundLocalError: local variable 'var' referenced before
    assignment
>>>
>>> def func2():
...     global var
...     print(var)
...     var = 10
...
>>> func2()
0
>>> var
10
```

Je pensais au départ que ce billet serait bien plus bref, mais force est de constater qu'il y a beaucoup de choses à dire sur ces concepts pouvant paraître très simples au premier abord, mais qui recèlent pas mal de pièges s'ils sont mal compris.

J'espère en tout cas que ce billet vous permettra d'en apprendre plus sur la gestion des variables en Python.