Portée des variables et notion de référence

Table des matières

I. Utilisation des portées en Python

Durée : 1 h

Environnement de travail : REPL.IT

Contexte

La variable fait partie des fondamentaux de la programmation. En effet, elle va permettre de contenir une ou plusieurs données non figées, données qui seront alors traitées par le programme. En réalité, on peut classer les variables en deux catégories au sein d'un programme : les variables dites « *locales* » et les variables dites « *globales* ». Une variable locale est utilisable uniquement au sein de la fonction dans laquelle elle est définie. Une variable globale, quant à elle, est utilisable partout dans le programme.

On appelle « *portée d'une variable* » l'espace de code dans lequel cette variable est utilisable. On parle donc soit de portée globale, soit de portée locale. L'utilisation de portées locales, c'est-à-dire d'espaces dans le code qui délimitent le champ d'utilisation d'une variable, réduit le nombre de lignes de code qui peuvent être à l'origine de bugs. Si votre programme ne contenait que des variables globales, il serait plus difficile de retrouver les lignes qui posent un problème. En fait, vous sauriez rapidement que l'erreur se trouve dans une portée locale, dont vous connaissez l'emplacement exact ! C'est vrai, l'utilisation seule de variables globales, donc utilisables dans la portée globale, c'est-à-dire dans tout le programme, est relativement efficace dans un court programme. Mais imaginons un programme de 1 000 lignes. Il sera beaucoup plus difficile de cibler l'origine d'un bug si on ne crée pas de variables locales ayant un champ d'utilisation limité dans une portée locale.

L'objectif de ce cours est donc de comprendre concrètement comment utiliser les différentes portées dans un programme Python. Nous parlerons aussi du passage par référence en seconde partie.

Une portée locale est créée à chaque fois qu'une fonction est appelée. Toutes les variables attribuées dans cette fonction existent dans la portée locale. Lorsque la fonction renvoie les données, la portée locale est détruite et ces variables sont oubliées. La prochaine fois que vous appellerez cette fonction, les variables locales ne conserveront pas les valeurs qui y étaient précédemment stockées pendant le dernier appel de la fonction.

Quelles sont les caractéristiques des portées locales ?

- Le code dans le contexte global ne peut pas utiliser de variables appartenant à une portée locale.
- Une portée locale peut accéder aux variables globales.
- Le code dans la portée locale d'une fonction ne peut pas utiliser de variables d'une autre portée locale.
- Vous pouvez utiliser le même nom pour différentes variables si elles sont de portées différentes, cependant c'est une pratique déconseillée.

Complément

Considérez la portée comme un conteneur pour les variables. Lorsqu'une portée est détruite, toutes les valeurs stockées dans les variables de la portée sont oubliées. Il n'y a qu'une seule portée globale et elle est créée lorsque votre programme commence. Quand votre programme se termine, le contexte global est détruit et toutes ses variables effacées. Si ce n'était pas le cas, les variables conserveraient leur valeur à chaque nouvelle exécution du programme.

Considérons quelques exemples pratiques :

Les variables locales ne peuvent pas être utilisées dans la portée globale

```
1 def spam():
2     eggs = 3131
3 spam()
4 print(eggs)
```

Le terminal renvoie alors l'erreur suivante :

```
Traceback (most recent call last):
  File "doc.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

L'erreur se produit car la variable eggs n'existe que dans la portée locale lors de l'appel de `spam()`. La fonction `spam()` renvoie alors ses données, sa portée locale est donc détruite, et il n'y a plus de variable nommée eggs dans le programme (sa portée étant détruite). Ainsi, lorsque votre programme essaie d'exécuter `print(eggs)`, Python indique que la variable eggs n'est pas définie puisqu'elle n'existe plus.

Attention

Seules les variables globales peuvent être utilisées dans la portée globale.

Une portée locale ne peut pas utiliser une variable appartenant à une autre portée locale

```
def spam():
    eggs = 99
    bacon()
    print(eggs)
def bacon():
    ham = 101
    eggs = 0
spam()
```

Lorsque le programme démarre, la fonction `spam()` est définie. Dans cette fonction, une portée locale est créée. La variable locale eggs est définie sur 99. Puis la fonction `bacon()` (définie plus bas) est appelée et une deuxième portée locale est créée au sein de cette seconde fonction.

Plusieurs portées locales peuvent exister en même temps. Dans cette nouvelle portée locale, la variable locale ham est définie sur 101 et la variable locale eggs (variable différente de celle appartenant à la portée locale de la fonction `spam()`) est également créée et définie sur « *0* ».

Lorsque `bacon()` retourne sa valeur, la portée locale de cet appel est détruite (cela veut donc dire que la variable eggs appartenant à la fonction `spam()` est elle aussi détruite). Le programme poursuit l'exécution dans la fonction `spam()` pour imprimer la valeur de eggs. Comme la portée locale de l'appel `spam()` existe toujours ici, la variable eggs est toujours définie sur « *99* ».

Le programme imprime donc « *99* ».

Pour faire simple, on peut dire que dans ce programme, les deux variables eggs sont deux variables différentes étant chacune dans une portée locale différente. Leurs valeurs sont indépendantes l'une de l'autre. Donc, lorsque la fonction `bacon()` est appelée et que sa portée est détruite, alors les variables ham et eggs appartenant à `bacon()` sont détruites. Mais à ce moment-là, la variable eggs créée dans la fonction `spam()` n'est pas détruite, puisque ce n'est pas la même variable car elle n'appartient pas à la même portée.

Les variables globales peuvent être lues et utilisées dans la portée locale

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

Dans cet exemple, puisqu'il n'y a aucun paramètre nommé eggs (dans les parenthèses lors de la définition de la fonction `spam()`), ni aucune ligne de code qui crée la variable locale eggs dans la fonction `spam()`, alors, lorsque eggs est utilisée dans `spam()`, Python considère qu'il s'agit d'une référence à la variable globale eggs. C'est pourquoi « *42* » est imprimé lorsque le programme est exécuté.

Variables locales et globales avec le même nom

Pour vous simplifier la vie, évitez d'utiliser des variables locales ayant le même nom que des variables globales ou que d'autres variables locales (même si techniquement, c'est parfaitement possible de le faire en Python comme le montre cet exemple).

```
def spam():
    eggs = "spam local" # 1.
    print(eggs) # Imprime "spam local"
def bacon():
    eggs = "bacon local" # 2.
    print(eggs) # Imprime "bacon local"
    spam()
    print(eggs) # Imprime "bacon local"
eggs = "global" # 3.
bacon()
print(eggs) # Imprime "global"
```

Il existe en fait trois variables différentes dans ce programme, mais elles sont toutes nommées eggs. Les variables sont les suivantes :

1. Une variable nommée eggs qui existe dans une portée locale lorsque `spam()` est appelée.
2. Une variable nommée eggs qui existe dans une portée locale lorsque `bacon()` est appelée.
3. Une variable nommée eggs qui existe dans la portée globale.

Étant donné que ces trois variables distinctes ont toutes le même nom, il peut être déroutant de savoir laquelle est utilisée à un moment donné. C'est pourquoi il vaut mieux éviter d'utiliser le même nom de variable dans différentes portées.

La déclaration « global »

```
def spam():
    global eggs
    eggs = "spam"
eggs = "global"
spam()
print(eggs)
```

Si vous devez modifier une variable globale à partir d'une fonction, utilisez la déclaration globale. Ici, la ligne de code, qui dans la fonction `spam()` indique global eggs, dit à Python :

« *Dans cette fonction, eggs se réfère à la variable globale, donc ne crée pas une variable locale avec ce nom.* »

```
def spam():
    global eggs
    eggs = "spam"
eggs = "global"
spam()
print(eggs)
```

Lorsque vous exécutez ce programme, que va afficher l'appel final `print()` ? On peut voir que dans la fonction `spam()`, la variable eggs n'est pas créée car la commande globale indique qu'on se réfère à la variable globale eggs qui est déclarée dans la portée globale (la valeur « *global* » lui est affectée).

Donc, dans la fonction `spam()`, on change la valeur de la variable globale eggs pour lui donner la valeur « *spam* » . Ce code s'exécute lorsque la fonction `spam()` est déclarée. La commande `print(eggs)` affiche donc la variable globale eggs qui est maintenant égale à « *spam* ».

Il existe quatre règles pour indiquer si une variable est dans une portée locale ou globale :

1. Si une variable est utilisée dans la portée globale (c'est-à-dire en dehors de toute fonction), alors c'est toujours une variable globale.
2. S'il existe dans une fonction une instruction globale pour cette variable, il s'agit d'une variable globale.
3. Si dans la fonction, la variable est utilisée dans une instruction d'affectation, c'est une variable locale.
4. Si la variable n'est pas utilisée dans une instruction d'affectation, c'est une variable globale.

Pour mieux comprendre ces règles, voici un exemple de programme :

Exemple **Quatre règles pour déterminer si une variable est globale ou locale**

```
1 def spam():
2     global eggs
3     eggs = "spam" # Variable globale
4 def bacon():
5     eggs = "bacon" # Variable locale
6 def ham():
7     print(eggs) # Variable globale
8 eggs = 42 # Variable globale
9 spam()
10 print(eggs)
```

- Dans la portée globale, la variable eggs est une variable globale. Elle est initialement définie sur 42.
- Dans la fonction `spam()`, eggs est la variable globale de eggs car il y a une déclaration « *global* » au début de la fonction.
- Dans `bacon()`, eggs est une variable locale car il y a une instruction d'affectation dans cette fonction.
- Dans la fonction `ham()`, eggs est la variable globale eggs car il n'y a pas d'instruction d'affectation dans cette fonction.

Dans une fonction, une variable sera toujours globale ou bien toujours locale.

Erreur dans l'ordre des instructions

Si comme dans le programme suivant, vous essayez d'utiliser une variable locale dans une fonction avant de lui attribuer une valeur, Python affichera une erreur.

```
1 def spam():
2     print(eggs) # Erreur
3     eggs = "spam local"
4 eggs = "global"
5 spam()
```

Le terminal renvoie :

```
1 Traceback (most recent call last):
2   File "doc.py", line 5, in <module>
3     spam()
4   File "doc.py", line 2, in spam
5     print(eggs) # Erreur
6 UnboundLocalError: local variable 'eggs' referenced before assignmentà retenir
```

Cette erreur se produit car Python voit qu'il y a une affectation pour eggs dans la fonction `spam()` et considère donc eggs comme locale. Puisque la commande `print(eggs)` est exécutée avant que eggs ne soit affectée à quoi que ce soit, la variable locale eggs n'existe pas. Python ne recourt pas à la variable globale eggs.

Exercice : Quiz

[solution n°1 p.13]

Question 1

Qu'est-ce que la portée d'une variable ?

- ○ Sa valeur
- ○ La plage d'instructions dans le code où elle est accessible
- ○ Son nom

Question 2

Quels sont les deux types de portées ?

- ☐ Portée générale
- ☐ Portée locale
- ☐ Portée globale

Question 3

Qu'arrive-t-il dans le contexte local lorsque l'appel de fonction est effectué ?

- ○ Le contexte local est détruit ainsi que les variables
- ○ Le contexte local et les variables sont utilisables en dehors de la fonction

Question 4

Comment pouvez-vous forcer une variable dans une fonction à se référer à la variable globale ?

- ○ Avec l'instruction « *global* »
- ○ Avec l'instruction « *return* »
- ○ Avec l'instruction « *def* »

Question 5

Quel sera le résultat du code Python suivant ?

```
1 def f1():
2     x=15
3     print(x)
4 x=12
5 f1()
```

- ○ Error
- ○ 12
- ○ 15
- ○ 1512

III. Passage par référence

Définition **Référence**

Le langage Python traite ce qu'on appelle des objets, c'est-à-dire que les données ont la forme d'objets. Une référence est donc un identifiant qui va permettre de cibler un objet. On peut donc dire que les variables sont des références nommées vers des objets.

On peut différencier deux types d'objets : les objets mutables et les objets immutables.

Les objets mutables peuvent changer de valeur (par exemple, une liste pourra changer de valeur mais sa référence restera la même).

Les objets immutables ne peuvent pas changer de valeur si on ne leur affecte pas une nouvelle valeur (les nombres entiers ou les chaînes de caractères par exemple).

Chaque objet a une référence unique. Nous allons par exemple dans ce script chercher avec la commande id la référence d'une variable du nom de eggs :

```
1 eggs = 1041
2 print (id(eggs))
```

Le terminal renvoie alors :

```
1 2253976033200
```

C'est la référence de l'objet auquel fait référence la variable eggs.

Par logique, on comprend qu'on peut se retrouver dans un cas de figure où plusieurs variables font référence au même objet.

```
1 var1 = 2
2 var2 = 2
3 print(id(var1))
```

```
1 print(id(var2))
```

Le terminal renvoie alors :

```
1 2445138788624
2 2445138788624
```

On voit que la référence (l'id) de var1 est la même que var2 car leur valeur est le même objet (objet qui a pour valeur 2). Cet objet est immutable.

Complément

Maintenant, voyons comment ce que nous avons appris sur les concepts de portées et de variables globales et locales va nous aider. Étudions ensemble ce script :

```
1 def test (parametre):
2     print(id(parametre))
3     parametre.append(1)
4     parametre.append(2)
5     print (parametre)
6 liste=[]
7 print (id(liste))
8 test (parametre=liste)
9 print (liste)
```

Le terminal renvoie :

```
1 1803970750976
2 1803970750976
3 [1, 2]
4 [1, 2]
```

On cherche dans ce script à afficher deux variables, la variable paramètre et la variable liste. On comprend que la variable liste est globale puisqu'elle est déclarée dans la portée globale. La variable paramètre quant à elle est inscrite en paramètre de la fonction test. La fonction test va rajouter dans la variable inscrite en paramètre les valeurs 1 et 2, il s'agit donc d'une liste. L'objet qu'est cette liste est mutable. On voit dans les lignes que renvoie le terminal que liste et paramètre se réfèrent au même objet, ils ont la même référence et la même valeur [1, 2]. En fait, lorsqu'on appelle la fonction `test()`, on indique que paramètre sera égale à la variable globale liste. Ces deux variables portent sur le même objet, la référence étant identique.

On comprend donc qu'on ne crée pas deux objets, mais qu'un seul objet est créé et les deux variables portent sur ce même objet. Si on veut créer deux objets différents, il faut déclarer une variable locale qui ne fera pas référence au même objet que la variable globale test. Par exemple, si on fait :

```
def test (parametre):
    liste2=parametre.copy()
    print(id(liste2))
    liste2.append(3)
    liste2.append(4)
    print (liste2)
liste=[1, 2]
print (id(liste))
test (parametre=liste)
print (liste)
```

Le terminal renvoie :

```
3001762064896
3001762017984
[1, 2, 3, 4]
[1, 2]
```

Donc ici, on affecte à la variable liste : [1, 2]. Elle porte donc toujours sur un objet liste. Dans la fonction test, on crée une variable locale liste 2, puis on lui affecte une copie du paramètre qui au moment de la déclaration est égale à liste. « *liste 2* » est donc un autre objet. On ajoute à liste 2 (qui contient actuellement [1, 2]) les valeurs 3 et 4. On voit en imprimant liste et liste 2 que ces variables font bien référence à deux objets différents. Mais, sans même avoir besoin d'imprimer les valeurs des variables, le simple fait que les références des deux objets soient différentes prouve qu'il s'agit de deux objets différents.

Le passage par référence va vous permettre de comprendre et d'éviter de nombreuses erreurs qui pourraient bloquer votre programme. Vous pourrez identifier les objets sur lesquels portent les variables, et ne pas tomber dans un piège où plusieurs variables portent sur le même objet. Il est donc important de le maîtriser dans le but de clairement cibler les objets qui sont la base du Python.

Exercice : Quiz

[solution n°2 p.14]

Question 1

Une variable fait référence à un objet.

- ○ Vrai
- ○ Faux

Question 2

On connaît l'identifiant d'un objet avec :

- print()
- ide()
- id()

Question 3

Un objet mutable ne peut pas changer de valeur.

- Vrai
- Faux

Question 4

Une chaîne de caractères est immutable.

- Vrai
- Faux

Question 5

Dans le code suivant, « *a* » fait référence à un objet différent de « *b* ».

```
1 a = 10
2 b = 18
3 a = b
```

- Vrai
- Faux

V. Essentiel

Nous avons vu que dans un programme il est préférable, si ce n'est indispensable, d'utiliser des variables dites locales. Les variables globales sont utilisables dans l'ensemble des lignes du code, tandis que les variables locales sont utilisables seulement dans un champ délimité. On appelle « *portée* » l'ensemble des lignes de codes où peut être utilisée une variable. La portée globale correspond donc à l'ensemble du script tandis qu'une portée locale correspond à la fonction dans laquelle est déclarée une variable, qui délimite son champ d'utilisation.

Voici quelques règles à retenir : une variable globale peut être utilisée dans tout le script, à n'importe quel endroit. Le code global écrit dans une portée locale permet de faire référence à une variable globale. Une variable est locale si elle est utilisée dans une instruction d'affectation dans une fonction. Cependant, si on utilise une variable sans l'affecter dans une fonction, alors il s'agit d'une variable globale. L'intérêt d'utiliser des portées locales est que cela va permettre de cibler beaucoup plus rapidement l'origine d'erreurs potentielles.

Enfin, nous avons vu que Python traite des données sous forme d'objets et que chaque objet a une référence précise. Le passage par référence permet de cibler directement les objets. Si on passe uniquement par des variables, il est possible de commettre des erreurs car plusieurs variables peuvent cibler un seul et même objet. Si on cherche la référence des objets que nous utilisons dans un script, nous pourrons savoir si oui ou non, les variables font référence à des objets différents.

VI. Auto-évaluation

A. Exercice

Vous êtes un informaticien et vous cherchez à corriger un script pour que les erreurs soient plus faciles à détecter lorsque le code sera plus long et si un jour un bug se produisait. Voici le script en question :

```
1 nombre = 8
2 somme = nombre+nombre
3 produit = somme*somme
4 print (produit)
```

Question 1

[solution n°3 p.15]

Que va calculer ce programme ? Que va-t-il afficher ?

Question 2

[solution n°4 p.15]

Transformez le code suivant en tenant compte de ce qui suit :

- Créez une fonction avec des variables locales.
- C'est dans cette fonction que le calcul sera opéré et le résultat affiché.
- La variable nombre devra être globale.
- La fonction devra être déclarée.

B. Test

Exercice 1 : Quiz

[solution n°5 p.15]

Question 1

Quel est le résultat du code Python suivant ?

```
1 def f1():
2     b=100
3     print(b)
4 b=1
5 f1()
```

- ○ Error
- ○ 100
- ○ 101
- ○ 99

Question 2

Quel est le résultat du code Python suivant ?

```
1 def f1():
2     global b
3     b+=1
4     print(b)
5 b=12
6 print("b")
```

- ○ Error
- ○ 13
- ○ 42
- ○ b

Question 3

Toutes les données de Python ont la forme :

- D’objets
- De variables
- De texte

Question 4

Un objet qui ne peut pas changer de valeur est :

- Mutable
- Immutable
- Stable

Question 5

Le passage par référence permet :

- De cibler directement une variable
- De cibler directement un objet

Solutions des exercices

Exercice p. 7 **Solution n°1**

Question 1

Qu'est-ce que la portée d'une variable ?

- ○ Sa valeur
- ◉ La plage d'instructions dans le code où elle est accessible
- ○ Son nom

Q La portée d'une variable, c'est l'ensemble des lignes de codes où on pourra accéder à cette variable.

Question 2

Quels sont les deux types de portées ?

- ☐ Portée générale
- ☑ Portée locale
- ☑ Portée globale

Q La portée globale correspond à l'ensemble du programme tandis qu'une portée locale correspond à la fonction dans laquelle est déclarée une variable locale.

Question 3

Qu'arrive-t-il dans le contexte local lorsque l'appel de fonction est effectué ?

- ◉ Le contexte local est détruit ainsi que les variables
- ○ Le contexte local et les variables sont utilisables en dehors de la fonction

Q Lors de l'appel d'une fonction, sa portée est détruite, et les variables locales qu'elle contenait aussi. Elles n'existent plus.

Question 4

Comment pouvez-vous forcer une variable dans une fonction à se référer à la variable globale ?

- ◉ Avec l'instruction « *global* »
- ○ Avec l'instruction « *return* »
- ○ Avec l'instruction « *def* »

Q L'instruction « *global* » sert à déclarer et utiliser une variable globale au sein d'une fonction, et donc à ne pas se référer à une variable locale.

Question 5

Quel sera le résultat du code Python suivant ?

```
1 def f1():
2     x=15
3     print(x)
4 x=12
5 f1()
```

- ○ Error
- ○ 12
- ◉ 15
- ○ 1512

Q La variable x est une variable locale. Sa portée correspond à l'ensemble des lignes de code de la fonction f1(). Donc, l'instruction print(x), appartenant à cette même portée affichera 15 qui est la valeur affectée à la variable locale x.

Exercice p. 9 **Solution n°2**

Question 1

Une variable fait référence à un objet.

- ◉ Vrai
- ○ Faux

Q Une variable est une référence nommée vers un objet.

Question 2

On connaît l'identifiant d'un objet avec :

- ○ print()
- ○ ide()
- ◉ id()

Q « *id()* » permet de connaître l'identifiant de l'objet, c'est-à-dire sa référence unique.

Question 3

Un objet mutable ne peut pas changer de valeur.

- ○ Vrai
- ◉ Faux

Q Un objet mutable peut changer de valeur sans que sa référence ne change. Par exemple, une liste peut changer de valeur et pour autant, sa référence restera la même.

Question 4

Une chaîne de caractères est immutable.

- ◉ Vrai
- ○ Faux

Q On ne peut pas changer la valeur d'un objet de type chaîne de caractères. Par exemple, si une variable de type chaîne de caractères change de valeur, c'est qu'elle ne fait plus référence au même objet.

Question 5

Dans le code suivant, « *a* » fait référence à un objet différent de « *b* ».

```
a = 10
b = 18
a = b
```

- ○ Vrai
- ◉ Faux

Le programme affecte à « *a* » la valeur de « *b* », ces deux variables font donc référence au même objet.

p. 11 Solution n°3

Ce programme va calculer le carré de la somme d'un nombre et de lui-même. Ici, le nombre de départ est 8, donc le programme affichera 256.

p. 11 Solution n°4

```
def calcul (a):
    somme = a+a
    produit = somme*somme
    print (produit)
nombre=8
calcul (a=nombre)
```

Exercice p. 11 Solution n°5

Question 1

Quel est le résultat du code Python suivant ?

```
def f1():
    b=100
    print(b)
b=1
f1()
```

- ○ Error
- ◉ 100
- ○ 101
- ○ 99

Dans la fonction, la variable b est dans une instruction d'affectation, elle est donc locale. Étant donné que l'instruction print(b) est inscrite dans la portée locale de la variable locale b, alors la valeur affichée est celle de la variable locale b. Cette variable est différente de la variable globale b qui appartient à la portée globale.

Question 2

Quel est le résultat du code Python suivant ?

```python
def f1():
    global b
    b+=1
    print(b)
b=12
print("b")
```

- ○ Error
- ○ 13
- ○ 42
- ◉ b

Ici, le print doit imprimer une chaîne de caractères et non la valeur d'une variable, étant donné la présence de guillemets.

Question 3

Toutes les données de Python ont la forme :

- ◉ D'objets
- ○ De variables
- ○ De texte

En Python, les données ont la forme d'objets, et chaque objet a une référence précise qui lui est propre.

Question 4

Un objet qui ne peut pas changer de valeur est :

- ○ Mutable
- ◉ Immutable
- ○ Stable

Un objet qui ne peut pas changer de valeur est immutable. C'est le cas des nombres entiers, des chaînes de caractères, etc.

Question 5

Le passage par référence permet :

- ○ De cibler directement une variable
- ◉ De cibler directement un objet

Un objet a une référence unique. Le passage par référence va par exemple permettre d'éviter des erreurs causées par le fait que plusieurs variables se réfèrent au même objet.