Rappels : Variables

Chaddaï Fouché

2023

I] Programmes et mémoire

Dans un sens purement matériel et pragmatique, un programme informatique est simplement une façon de manipuler le contenu de la mémoire d'un ordinateur. En examinant le langage machine réellement exécuté par un processeur classique moderne, on voit qu'il est possible de :

- lire le contenu d'une adresse mémoire,
- modifier le contenu d'une adresse mémoire,
- copier le contenu d'une adresse à une autre,
- faire des calculs et stocker le résultat à une certaine adresse,
- choisir l'adresse mémoire où lire la prochaine instruction à exécuter pour le processeur.

Ainsi que quelques autres instructions pour interagir avec les périphériques.

En réalité, le terme d'adresse mémoire couvre ici plusieurs niveaux, en ordre ascendant de latence (temps nécessaire pour que le contenu de la mémoire soit accessible au processeur, ici donnée en nombre de cycles de l'horloge interne soit pour un processeur à 5 GHz, 0,2 ns) :

1) les registres du processeur, extrêmement rapide (latence : aucune, l'accès est immédiat) ;
2) la mémoire cache du processeur, très rapide car montée directement sur le processeur (latence : 1 à 5 cycles d'horloge) ;
3) la mémoire vive, la RAM (latence : 400 à 1000 cycles d'horloge pour remonter la valeur dans le cache avant de l'utiliser).

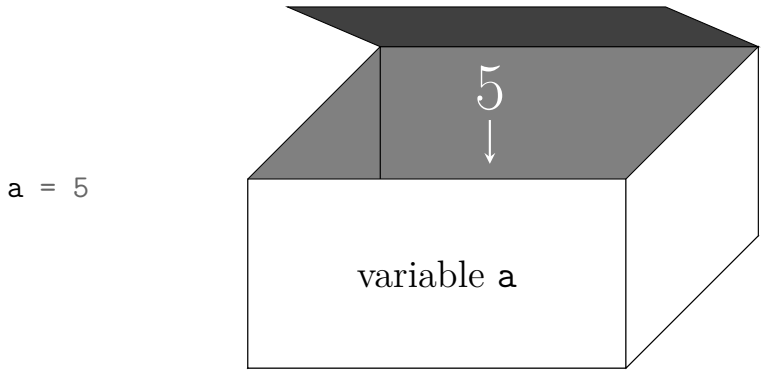
Au-delà de la mémoire vive, on trouve le disque dur, lequel peut-être plus ou moins rapide mais est de toute façon plusieurs ordres de magnitude plus lent que la mémoire vive (un SSD moderne est à peu près 1000 fois plus lent que la mémoire vive), pour cette raison le disque dur est généralement traité comme un périphérique et il n'est pas possible d'y accéder directement via une adresse mémoire.

Ainsi les instructions machines manipulent directement la mémoire et sont préoccupés d'où se trouve chaque valeur et des détails de leurs contenus binaires... Ceci ne facilite pas la pensée abstraite et donc la conception de programmes, même de taille raisonnable et encore moins de grands projets. Il n'est généralement pas pertinent de comprendre le détail des manipulations de mémoire effectuées pour programmer un site météo qui résume et présente des informations acquises auprès d'une API web.

Pour résoudre ce problème, on introduit une abstraction par laquelle on *nomme* les valeurs qu'on va manipuler, ou plus exactement on nomme les emplacements mémoires où l'on va entreposer les valeurs à manipuler : les **variables**.

II] Variables

Une *variable* est donc un nom donné à un emplacement mémoire, on peut la visualiser comme une boîte dans laquelle on peut mettre une valeur.



En Python en particulier, les noms de variables autorisés doivent commencer par une lettre (au sens Unicode) ou un soulignement `_`, puis se poursuivre par des lettres, chiffres ou soulignement (pas d'espaces !). Des noms valables seraient donc : `a`, `i`, `résultat`, `x1`, `x_max`, `__name__`, `_`, etc.

L'emploi d'une *variable* est donc double : en tant que cible d'une **affectation** (à gauche d'un `=`, opérateur d'affectation en Python) ou en tant que source d'une valeur dans une expression (calcul, appel de fonction, test, etc).

1) Affectation

Une *affectation* range une valeur, ou une référence à une valeur, à l'emplacement mémoire désigné par une variable :

Pseudo-code : $a \leftarrow 5$
Python : `a = 5`
stocke l'entier 5 dans la variable `a`.

Par abus de langage, on parle de *la valeur de la variable* pour la valeur stockée dans l'emplacement mémoire désigné par la variable.

Il est possible d'affecter plusieurs fois la même variable et sa valeur dépend alors de la dernière affectation exécutée :

```
1 x = 3
2 # ici x vaut 3
3 print(x) # affiche "3"
4 x = 10
5 # ici x vaut 10
6 print(x * 5) # affiche 50
```

Ainsi un exercice classique consiste à échanger les valeurs de deux variables, la première solution naïve est erronée :

```
1 # on commence avec a qui vaut 1 et b qui vaut 7
2 a = b
3 # ici a vaut 7
4 b = a
5 # et donc ici b vaut aussi 7 puisque a vaut 7 à la ligne 4
6 # a et b valent tous les deux 7
```

Donc la valeur de la première variable affectée à la ligne 2 est *écrasée* par la seconde et donc perdue : elle n'est plus présente nulle part en mémoire à partir de la ligne 2.

La solution correcte, sans astuce, nécessite une variable auxiliaire, utilisée uniquement pour stocker temporairement la valeur de la variable écrasée :

```
# on commence avec a qui vaut 1 et b qui vaut 7
temp = a # on met la valeur de a de côté dans temp
a = b    # on écrase a par la valeur de b
# ici a vaut 7
b = temp # on récupère la valeur mise de côté dans b
# et donc ici a vaut 7 et b vaut 1
```

2) Utilisation de variables

En dehors du contexte de l'affectation (à gauche du `=`), chaque utilisation d'une variable correspond à une lecture du contenu de l'emplacement mémoire. Conceptuellement, chaque expression contenant une variable est équivalente à écrire l'expression en remplaçant la variable par sa valeur :

Pseudo-code : Afficher$(a + 2)$ $\implies$ Afficher$(5 + 2)$.
Python : `print(a + 2)` $\implies$ `print(5 + 2)`.

3) Ambiguïté de l'opérateur d'affectation

Notez que le choix de l'opérateur `=` en Python est trompeur : il ne s'agit pas d'une déclaration d'égalité entre les deux membres, mais bien d'un transfert d'une valeur calculée à droite vers un emplacement mémoire nommé à gauche. En particulier, on peut tout à fait utiliser la variable à laquelle on va affecter (à gauche du `=`) dans le membre de droite à affecter :

Pseudo-code : $a \leftarrow a - 1$ $\implies$ $a \leftarrow 5 - 1$ $\implies$ $a \leftarrow 4$.
Python : `a = a - 1` $\implies$ `a = 5 - 1` $\implies$ `a = 4`.
Calcule la valeur de $a - 1$ avec l'ancienne valeur de a (ici 5), puis l'affecte à la variable a qui vaut donc 4 dans la suite du programme. Ici $a \leftarrow a - 1$ revient donc à retirer 1 à la valeur stockée dans a.

Cette possibilité est particulièrement utile dans des boucles telles que les mêmes instructions ne produisent pas le même résultat à chaque tour de boucle :

```
n = 0
while n < 10:
    print(n)
    n = n + 1
```

Affiche les entiers de 0 à 9.

4) Affectation combinée à un opérateur : Mise à jour de variable

L'emploi d'une affectation effectuant un calcul sur une variable pour affecter à cette même variable, autrement dit une affectation *mettant à jour* la valeur de la variable, est si fréquent que des opérateurs d'affectation spécialisés sont disponibles :

`n += 1` $\implies$ `n = n + 1`. Ainsi `var += valeur` augmente le contenu de la variable `var` de `valeur`. On dit qu'on *incrémente* `n` de 1.

De même il existe les opérateurs `-=` (*décrémente*) ; `*=` ; `/=` ; `**=` ; `//=` ; `%=` ; etc.

5) Affectations simultanées

En Python, il est possible d'affecter des valeurs à plusieurs variables à la fois avec la syntaxe `a, b = valeur_a, valeur_b`. En dehors de la compacité du code, l'intérêt réside en le fait que les deux valeurs sont calculées avant les deux affectations, ce qui permet par exemple d'utiliser les

anciennes valeurs de `a` et `b` afin de calculer leurs nouvelles valeurs. On peut donc échanger les valeurs de deux variables en une seule ligne : `a, b = b, a`.

Plus généralement, la syntaxe peut s'utiliser avec n'importe quelle séquence qui contient au moins deux éléments : `a, b = (1,2,3,4)` affecte 1 à `a` et 2 à `b`. Il est également possible de faire plus de deux affectations simultanées : `a, b, c = [1,10,100,1000,10000]` affecte 1 à `a`, 10 à `b` et 100 à `c`.

III] Variables et fonctions : portées de variables

1) Paramètres

Les paramètres d'une fonction sont des variables auxquelles les valeurs des arguments d'un appel à la fonction sont automatiquement affectées lors de cet appel :

```
def f(x, y):
    print(x * y)

f(2,7)      # affiche 14
# est équivalent à exécuter le code suivant :
x = 2
y = 7
print(x * y) # affiche 14
```

2) Variables locales

Le code qui précède est légèrement mensonger, dans le sens où `x` et `y`, tout autre paramètre de fonction et toute autre variable définie dans la fonction sont des **variables locales** à la fonction. Cela signifie que ces variables n'ont rien à voir avec une autre variable du même nom définie en dehors de la fonction ou dans une autre fonction. Ainsi :

```
x = 5
# ici x vaut 5
f(2, 7)  # affiche 14 # "x" vaut 2 dans l'appel à la fonction
# ici x vaut toujours 5
print(x) # affiche 5
```

On rappelle que les variables sont des noms pour des emplacements mémoires, une variable locale ne désigne pas le même emplacement mémoire qu'une variable externe à la fonction portant le même nom. En pratique chaque appel à la fonction a ses propres variables locales, même si plusieurs appels à une même fonction ont lieu simultanément (par exemple pour une fonction récursive) chaque appel aura ses propres variables :

```
def f(n):
    if n == 0:
        return 5
    else:
        return n * f(n-1)

print(f(1)) # affiche 5
```

L'appel à `f(1)` et son appel à `f(0)` ont chacun leur propre version de `n` qui ont des valeurs différentes mais n'interfèrent pas l'un avec l'autre (parce qu'elles correspondent à des emplacements mémoires différents).

Ce comportement est absolument indispensable à l'écriture de programmes conséquents puisqu'il permet de concevoir une fonction sans avoir à se préoccuper du nom de toutes les autres variables définies dans ce programme...

3) Portée de variables

Lorsqu'une variable est définie dans du code à l'extérieur de toute fonction, on l'appelle une **variable globale**, elle appartient au module la contenant et il est possible de l'importer depuis un autre module ou script. Cette variable globale peut être utilisée partout mais elle n'est modifiable, c'est-à-dire qu'on ne peut lui affecter une nouvelle valeur, que depuis du code externe à toute fonction ou en précisant que la fonction va manipuler cette variable globale par la directive `global var` :

```
compte = 2

def f(n):
    # on peut accéder à compte depuis n'importe quelle fonction
    print(compte * n)

def g(n):
    # si on essaie d'affecter une valeur à compte via =
    # on crée une nouvelle variable locale qui s'appelle compte
    # et on n'affecte donc pas la valeur de la variable globale
    compte = n
    print(compte)

def h(n):
    # si on essaie d'utiliser +=, -=, etc
    # le code plante car il n'existe pas de variable compte locale
    # qu'on puisse modifier avant cette ligne
    compte += n

def k(n):
    # si on veut éviter cette erreur on doit utiliser :
    global compte
    compte += n

f(4)   # affiche 8
g(10)  # affiche 10 # dans cet appel, la variable locale "compte" vaut 10
# ici compte, la variable globale, vaut toujours 2
print(compte)
# h(3) # plante avec UnboundLocalError
k(3)
# ici compte vaut 5
print(compte) # affiche 5
```

Dans `g` on dit que la variable locale `compte` vient *masquer* la variable globale `compte`, c'est un comportement subtil qui peut être source de bogues si l'on pensait être en train de manipuler la variable globale mais qui est également le bienvenu lorsqu'on a simplement défini une variable de nom courant dont on ne veut pas qu'elle interfère avec l'extérieur de la fonction. En anglais on parle de « variable shadowing » (la variable globale étant temporairement dans l'ombre de la variable locale).

La **portée** d'une variable est la portion de code dans laquelle elle existe et est lisible et parfois

modifiable. La portée d'une variable globale est l'ensemble du module dans lequel elle est définie, la portée d'une variable locale est seulement le code de la fonction dans laquelle elle est définie. Par exemple le paramètre `n` et la variable locale `compte` dans la fonction `g` ont une portée qui va de la ligne 7 à la ligne 12 (la définition de `g`).

4) Subtilité avancée (hors programme)

On peut définir une fonction `g` à l'intérieur d'une fonction `f`, elle est alors redéfinie à chaque appel de `f` et les variables locales de `f` lors de cet appel y sont visibles de façon analogue à des variables globales. Pour accéder en écriture à une telle variable locale à la fonction englobante, on utilise `nonlocal var` :

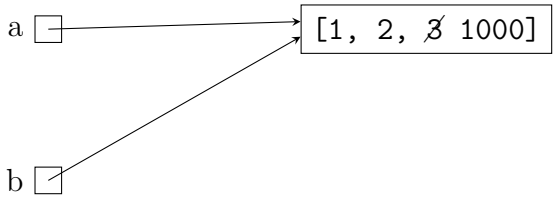
```
1  def f(x):
2      a = 5 + x
3      def g(y):
4          # g a accès à a en lecture
5          print(a + y)
6      g(2)
7
8  f(10) # affiche 17
9  f(2)  # affiche 9
10
11 def h():
12     a = 5
13     # notez que g est une variable locale à ces fonctions f et h
14     # ces deux g n'interfèrent pas l'une avec l'autre
15     def g(x):
16         # pour pouvoir modifier la valeur de a, on utilise :
17         nonlocal a
18         a = x
19     g(1)
20     print(a)
21
22 h()   # affiche 1
```

IV] Références et aliasing

1) Aliasing

Un « emplacement mémoire » seul est de taille limitée (32 ou 64 bits sur une architecture récente), il ne peut pas contenir un tableau d'une douzaine d'entier, ou un objet complexe. Lorsqu'on affecte une telle valeur à une variable, on n'y stocke en réalité qu'une *référence* à la valeur (autrement dit l'adresse mémoire où l'on trouvera la valeur, sur plusieurs emplacements mémoires consécutifs). Dans ce cas lorsqu'on affecte cette variable à une autre variable, ce qu'on copie est non pas la valeur mais la référence : il n'y a toujours qu'une seule valeur en mémoire, à laquelle plusieurs variables contiennent une référence. Si cette valeur est modifiable (mutable), ces transformations seront alors visibles depuis toutes les variables la référençant et pas seulement celle qu'on a utilisée pour la modification.

```
1  a = [1,2,3]
2  b = a
3  b[2] = 1000
4  print(a) # [1, 2, 1000]
```



Ce comportement est très fréquemment une source de bogues :

```
salariés = ["Jean", "Nicole"]
équipe = salariés
# on ajoute le patron
équipe.append("Chef")
print(salariés) # affiche ["Jean", "Nicole", "Chef"]
# soudainement le patron est devenu un salarié
```

On dit que toutes ces variables sont des *alias* pour une même valeur et on parle d'un phénomène d'*aliasing*.

Ce comportement n'est pas visible lorsque la valeur n'est pas modifiable (elle est immuable) comme pour `float`, `int`, `bool`, `str` ou un `tuple` (de valeurs immuables). Ceci est l'une des raisons pour lesquelles l'usage de valeurs immuables entraîne souvent des programmes plus lisibles et ayant moins de bogues.

Pour notre part, on se méfiera particulièrement de l'aliasing avec les `list` et `dict`.

2) Copies

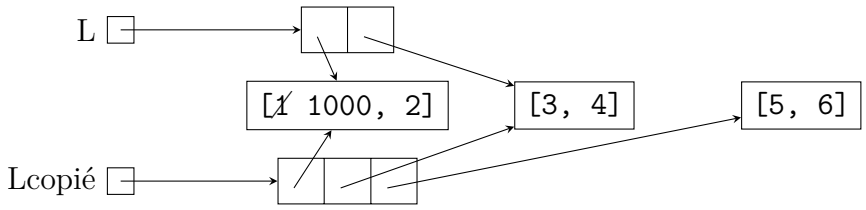
On peut créer une vraie copie avec la fonction `copy` du module `copy` mais attention, si votre valeur contenait elle-même des références à d'autres valeurs (par exemple une liste de listes), ces références continueront à désigner les mêmes valeurs, on dit que `copy` fait une copie superficielle (shallow copy en anglais) :

```
from copy import copy
L = [[1,2],[3,4]]
Lcopié = copy(L)

# Lcopié n'est plus un alias de L
# la modifier ne modifie pas L
Lcopié.append([5,6])
print(L)      # affiche [[1,2],[3,4]]

# mais Lcopié contient bien des références vers les mêmes
# listes que L, les modifier va aussi modifier L
Lcopié[0][0] = 1000
print(L)      # affiche [[1000,2],[3,4]]

print(Lcopié) # affiche [[1000,2],[3,4],[5,6]]
```



Pour éviter cela, on doit copier récursivement les valeurs contenues par notre valeur, on dit qu'on fait une copie profonde (deep copy en anglais) :

```
from copy import deepcopy
L = [[1,2],[3,4]]
Lcopié = deepcopy(L)

# plus aucune relation entre les deux variables
Lcopié[0][0] = 1000
print(L)      # affiche [[1,2],[3,4]]

print(Lcopié) # affiche [[1000,2],[3,4]]
```

3) Conséquences sur les fonctions

Comme appeler une fonction revient à affecter les arguments de l'appel aux paramètres, le phénomène d'aliasing se produit si l'on passe des valeurs mutables : les paramètres dans le code de la fonction sont des alias des arguments. Ceci permet à une fonction de modifier le contenu de ses arguments (si leur type est mutable), ce qui peut être souhaitable ou dangereux. On dit qu'on passe les arguments *par référence*, d'autres langages permettent de spécifier si l'on souhaite cela ou l'on préfère recevoir des vraies copies des arguments (passés *par valeur*).

L'avantage du passage par référence est qu'il est beaucoup plus rapide (pas besoin de copier les arguments) et qu'il est toujours possible d'obtenir l'équivalent d'arguments passés par valeurs, simplement en remplaçant les paramètres par des copies avant de les manipuler.

L'inconvénient est bien sûr le risque de bogues plus élevé et la nécessité de faire des copies manuellement au début de la fonction si l'on ne veut pas affecter les valeurs des paramètres pour l'appelant.