Partie II - Manipuler les variables

par Yves Bailly

Date de publication : 01/07/2005

Dernière mise à jour : 20/11/2005

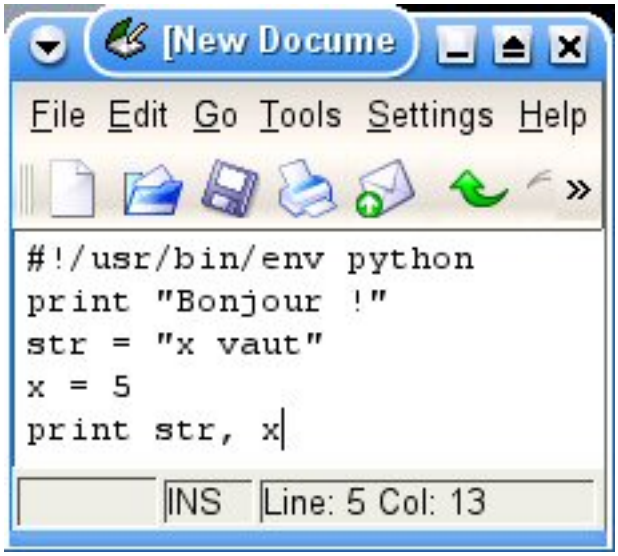
La dernière fois, nous avons découvert la notion de variable, une zone mémoire pour contenir des données que l'on souhaite manipuler. Aujourd'hui nous allons voir quelques opérations fondamentales impliquant les variables.

I - Avant propos

Mais avant de commencer, une petite mise au point. J'ai reçu un message d'un lecteur un peu désespéré, parce qu'il n'arrivait pas à faire fonctionner le petit programme présenté dans le numéro précédent. Ce programme était présenté ainsi :

```
0 #!/usr/bin/env python
1 print "Bonjour !"
2 str = "x vaut"
3 x = 5
4 print str, x
```

Les numéros de 1 à 5 en début de ligne ne sont là qu'à titre indicatif ! Si certains langages anciens, comme le Basic, exigeaient que les lignes du programme soient numérotées, ce n'est pas le cas de la grande majorité des langages modernes. La numérotation n'est affichée ici que pour faciliter la description du programme, il ne faut pas l'inscrire lorsque vous tapez le programme. Donc, dans votre éditeur de texte, le programme doit ressembler à ceci, avec éventuellement de la couleur :



Remarquez qu'il n'y a pas de numéros de lignes !

Cette mise au point étant effectuée, passons aux thèmes du jour.

II - Généralité sur les variables

II-A - Le type d'une variable

Nous l'avions évoqué, une variable possède un type, qui précise la nature de l'information qu'elle peut contenir. Lancez un interpréteur Python, et expérimentez :

```
>>> x = 1
>>> print type(x)
<type 'int'>
```

La première ligne vous est maintenant familière, nous stockons la valeur numérique entière 1 dans une variable nommée **x**. Sur la deuxième ligne, vous reconnaissez l'instruction **print**, dont le rôle est d'afficher quelque chose à l'écran. Par contre, **type** est nouveau.

Il s'agit d'une *fonction*. Nous reviendrons sur cette importante notion plus tard. Pour l'instant, disons simplement qu'une fonction est un regroupement d'instructions, qui donnent un résultat dépendant de ce qui se trouve entre les parenthèses qui la suivent. C'est pourquoi, lorsque je voudrai parler de « la fonction **f** », j'écrirai « **f()** » : la présence des parenthèses signal qu'il s'agit d'une fonction.

Le rôle de **type()** est de donner le type de ce qui se trouve entre les parenthèses (on dit aussi, son *paramètre*). Dans l'exemple, nous avons donné la valeur entière **1** à **x**. Il semble donc normal que **type()** nous réponde que **x** est de type '**int**', qui est l'abbréviation de *integer*, *entier* en anglais. Poursuivons l'expérience :

```
>>> x = "coucou"
>>> print type(x)
<type 'str'>
```

Nous réutilisons **x**, en lui affectant une nouvelle valeur, ici une chaîne de caractères. Ce faisant, nous avons changer la nature de ce qui était contenu dans x (et incidemment perdu l'ancien contenu, qui était une valeur numérique). Tout les langages ne permettent pas de réutiliser une variable en changeant la nature du contenu. Par exemple, c'est impossible dans des langages tels que C, C++, Pascal... Mais Python, comme Perl et d'autres, permet cette manipulation. Maintenant **x** contient une chaîne de caractères, ce que nous confirme la fonction **type()** : le mot '**str**' est l'abbréviation de l'anglais *string*, que l'on peut traduire par *chaîne*. Continuons :

```
>>> x = 3.14
>>> print type(x)
<type 'float'>
```

Encore une fois, nous avons changer la nature de **x**, cette fois pour contenir une valeur numérique décimale. Notez bien que dans l'opération, le contenu précédent de **x** (la chaîne de caractères) est définitivement perdu ! Là encore, **type()** nous confirme le changement de nature. '**float**' est l'abbréviation de l'anglais *floating point*, désignation d'un nombre à *virgule flottante*, c'est-à-dire un nombre décimal. Les nombres décimaux posent de nombreux problèmes en informatique, nous les aborderons plus tard.

II-B - Conversions entre types

Effectuer une conversion de type consiste à consulter le contenu d'une variable comme si elle était d'un type différent. Par exemple, créons une variable chaîne de caractères :

```
>>> s = "10"
```

Cette chaîne représente une valeur numérique, mais pour l'instant ce n'est qu'une chaîne. Que ce passe-t-il si nous essayons, par exemple, de lui ajouter une valeur ?

```
>>> print s + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

La sanction est immédiate : il est absurde de vouloir ajouter une valeur numérique à quelque chose qui n'est, finalement, que du texte. C'est en substance le sens des injures émises par l'interpréteur Python. Pourtant, nous savons que notre texte représente bien une valeur numérique ! La solution consiste à effectuer une *conversion*, dans notre cas à transformer la chaîne de caractères en une valeur numérique correspondante :

```
>>> print int(s) + 5
15
```

C'est beaucoup mieux ! **int()** est une fonction (encore une !) qui tente de fabriquer un entier à partir de ce qu'on lui donne entre parenthèses. Ici, nous lui donnons une chaîne, que **int()** « reconnaît » comme représentant une valeur entière correcte. La conversion est donc possible. Mais ce n'est pas toujours le cas :

```
>>> s = "erreur !"
>>> print int(s) + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): erreur !
```

Ici **int()** n'a pas réussi à convertir la chaîne de caractères en entier, le résultat est encore une protestation courroucée de l'interpréteur.

On peut naturellement construire un entier à partir d'un nombre décimal :

```
>>> print int(10.8) + 5
15
```

Remarquez que l'entier fabriqué par **int()** n'est pas le plus proche (qui est 11), mais plutôt celui obtenu simplement en faisant disparaître la partie décimale (ici, le **.8**).

Il existe d'autres fonctions de conversion. Pour l'heure, citons simplement **float()** qui permet de fabriquer un nombre décimal, et **str()** qui permet de fabriquer une chaîne de caractères. Par exemple :

```
>>> s = "10.7"
>>> x = float(s) + 1.1
>>> print x
11.8
>>> print str(x) + "0"
11.80
```

La dernière commande peut vous surprendre... tentons de l'expliquer.

III - Les chaînes de caractères

Ce type de données mérite un traitement spécial. En premier lieu, il est nécessaire de bien comprendre la nature réelle d'une chaîne de caractères. Du point de vue de l'ordinateur, il s'agit en fait d'une suite de caractères en mémoire, que l'on peut représenter sous la forme d'un tableau. Par exemple, exécutez :

```
>>> s = "abcde"
```

En mémoire, on a alors une situation qui peut se schématiser ainsi

Autrement dit, **s** est un nom donné à une zone de mémoire, laquelle zone est en fait constituée d'éléments individuels, un par caractère dans la chaîne. Il est possible d'obtenir un élément individuel de cette zone, c'est-à-dire un caractère individuel :

```
>>> print s[2]
c
```

L'utilisation des crochets nous permet d'indiquer quel élément du tableau nous voulons obtenir. Le nombre donné entre les crochets est appelé l'indice de l'élément voulu. Ici, nous demandons l'élément d'indice 2. Peut-être vous attendiez-vous, alors, à obtenir la lettre '**b**', qui est le deuxième caractère dans la chaîne ? Par convention, les indices commencent à 0 (zéro). Donc le premier caractère, le '**a**', a pour indice 0 ; le deuxième, le '**b**', a pour indice 1 ; et enfin, le troisième caractère a pour indice 2 : c'est pourquoi nous obtenons le '**c**'. Il s'agit là d'une convention très répandue dans le monde de la programmation, qui dépasse le cadre des chaînes de caractères. Elle ne connaît que quelques exceptions, dont deux exemples notables sont les langages Pascal (où les indices commencent à 1) et Ada (qui permet de choisir librement le début des indices).

Dans le cas précis de Python, il est possible de donner des indices négatifs :

```
>>> print s[-2]
d
```

L'utilisation de nombres négatifs permet de compter « à rebours », c'est-à-dire de droite à gauche, le premier caractère rencontré se voyant attribué l'indice -1. Donc, l'écriture **s[-2]** signifie « *le deuxième élément de* **s** *en partant de la fin* », c'est bien l'élément '**d**'.

Découpage...

Les chaînes de caractères jouent un rôle tellement important pour Python que des écritures très particulières ont été inventées. Par exemple, il est possible d'obtenir un morceau d'une chaîne (ou encore une *tranche*, traduction de l'anglais *slice* qui désigne ce principe) :

```
>>> print s[1:4]
bcd
```

Cette écriture, qui utilise les deux-points dans les crochets, se lit : « *la tranche qui commence à l'élément d'indice 1 et va jusqu'à l'élément d'indice 4, ce dernier exclu* ». Le dernier étant exclu, on demande ici la tranche des indices 1, 2 et 3. Ce sont bien les caractères '**b**', '**c**' et '**d**'. La règle importante à retenir, c'est que la deuxième partie de la tranche (après les deux-points) désigne un élément qui ne sera pas dans le résultat. Par exemple, nous avons vu plus haut que l'indice -2 donne le caractère '**d**'. Utilisons cet indice pour construire une tranche :

```
>>> print s[1:-2]
bc
```

Cette fois nous n'avons pas le caractère '**d**'.

Fait intéressant, aucun des deux membres de la tranches n'est obligatoire. Si le premier manque, cela signifie « *depuis le début* ». Si le deuxième manque, cela signifie « *jusqu'à la fin* ». Par exemple :

```
>>> print s[2:]
cde
```

Nous demandons la tranche commençant à l'indice 2 et allant jusqu'à la fin. À l'inverse :

```
>>> print s[:2]
ab
```

Nous demandons la tranche allant du début jusqu'à l'indice 2... ce dernier étant, rappelez-vous, exclu ! C'est pourquoi nous n'avons pas le caractère '**c**' dans le résultat. Dernier exemple, si vous n'y prenez garde vous pouvez obtenir un résultat vide :

```
>>> print s[4:2]
```

L'indice de départ de la tranche fait référence à un caractère qui se trouve après celui auquel fait référence l'indice d'arrivée : dans ce cas, aucun caractère n'est renvoyé. Le même phénomène surviendrait si les deux indices étaient égaux.

...et assemblage

Nous venons de voir comment découper une chaîne, mais il est également possible de coller plusieurs chaînes ensembles pour en créer une nouvelle. Par exemple :

```
>>> s1 = "bonjour "
>>> s2 = "monde !"
>>> s = s1 + s2
>>> print s
bonjour monde !
```

La chaîne s est le résultat de la *concaténation* des chaînes **s1** et **s2**. Le verbe *concaténer* est le mot savant utiliser pour dire « *coller ensemble* ». En Python, la concaténation est réalisée par l'opérateur d'addition +. Essayez maintenant ceci :

```
>>> s1 += s2
>>> print s1
bonjour monde !
```

Nous avons utiliser un opérateur un peu particuliers : +=. Il combine en une seule opération la concaténation puis l'affectation dans la chaîne donnée à sa gauche. La ligne **s1 += s2** est donc équivalente à **s1 = s1 + s2**, c'est-à-dire, « *coller les chaînes **s1** et **s2**, puis stocker le résultat dans **s1*** ».

Essayons un petit exercice. Nous voudrions insérer au milieu de s les mots "**tout le**", pour construire la chaîne "**bonjour tous le monde !**", en supposant (pour l'intérêt du jeu) que nous ne disposons pas des chaînes **s1** et **s2**. Comment faire ?

Pour réaliser cette opération, nous devons d'abord découper la chaîne **s**, pour obtenir les deux morceaux extrêmes. Puis nous construisons une chaîne en concaténant ces morceaux avec les mots à insérer, et enfin nous stockons le résultat dans **s**.

Le premier morceau, "**bonjour**", va du début jusqu'au caractère d'indice 6. On peut l'obtenir avec **s[:7]** (où 7 = 6 + 1, rappelez-vous que le dernier indice est exclu du résultat). Le deuxième morceau, " **monde !**", commence là où nous nous sommes arrêtés et va jusqu'à la fin : on l'obtient donc avec **s[7:]**. Il ne reste plus qu'à utiliser l'opérateur de concaténation, et on obtient finalement l'expression suivante :

```
>>> s = s[:7] + " tous le" + s[7:]
>>> print s
bonjour tous le monde !
```

Remarquez que l'on peut utiliser plusieurs fois l'opérateur +.

Normalement, les plus attentifs parmis vous auront noté que la chaîne que nous venons de construire présente un problème, ou plus précisément une faute de grammaire. Nous devons changer le "**s**" en "**t**" ! Avec certains langages, il serait possible d'écrire quelque chose dans le genre :

```
>>> s[11] = "t"
```

(car la lettre incorrecte est à l'indice 11). Mais ceci ne fonctionnera pas avec Python : dans ce langage, les chaînes sont ce que l'on appelle des *objets non-mutables*, c'est-à-dire qu'on ne peut pas les modifier directement. Cela peut paraître une contrainte ennuyeuse, mais cela permet des optimisations importantes et confère généralement une grande robustesse aux programmes manipulant des chaînes de caractères.

La solution à notre problème consiste, encore une fois, à découper la chaîne, puis la reconstruire par morceaux. Saurez-vous trouver l'expression nécessaire ? La voici :

```
>>> s = s[:11] + "t" + s[12:]
```

Tout se passe comme si nous avions modifié la chaîne **s**. En réalité, nous l'avons recréée.

Enfin, dernier mot (pour l'instant !) sur les chaînes de caractères, il est possible de les multiplier :

```
>>> s = "echo " * 5
>>> print s
echo echo echo echo echo
```

C'est un peu comme une multiplication « normale » : on utilise l'opérateur *****. La valeur numérique doit être un entier. Par ailleurs, l'opération est commutative : **s*5** et **5*s** donnent le même résultat.

IV - Retour sur les types numériques

Je voudrais terminer par quelques informations complémentaires sur les types numériques, les entiers et les nombres à virgule (Python propose également les nombres complexes, mais nous verrons cela plus tard).

IV-A - Les entiers

Les entiers de Python n'ont (presque) pas de limite, si ce n'est celles imposées par votre ordinateur, mémoire et puissance du processeur. Il est ainsi possible de calculer de grands nombres. Par exemple, savez-vous combien il existe de combinaisons possibles au Loto ? Il faut choisir 6 numéros parmis 49, sans tenir compte de l'ordre, la formule est :

```
>>> print (49*48*47*46*45*44) / (6*5*4*3*2)
13983816
```

Ne désespérez pas, certains parviennent à gagner ! Il est possible d'élever un nombre à une certaine puissance. Par exemple :

```
>>> print 2**128
340282366920938463463374607431768211456
```

Pour les curieux, il s'agit du nombre maximum théorique d'adresses disponibles pour la prochaine norme IPv6 qui devrait bientôt se répendre sur l'Internet. Pour mémoire, l'élévation d'un nombre à une puissance consiste à le multiplier par lui-même, ici 2 est multiplié par lui-même 128 fois. Si vous donnez un nombre trop grand, il est possible que le résultat prenne du temps pour arriver...

Le tableau suivant récapitule les principaux opérateurs disponibles sur les entiers :

Opérateurs	Exemples
+, addition	**1+2** donne **3**
-, soustraction	**1-2** donne **-1**
***, multiplication	**2*5** donne **10**
/, division	**5/2** donne **2**
%, reste	**5%2** donne **1**
****, puissance	**2**4** donne **16** (**2*2*2*2**)

Remarquez l'opérateur / : le résultat de la division entre deux entiers est un entier (on parle de division euclidienne), ceci même si ça ne "tombe pas juste". Dans ce cas, le reste est donné par l'opérateur %.

Tous ces opérateurs peuvent être « étendus » avec le signe =, par exemple :

```
>>> a = 5
>>> a += 2
>>> print a
7
```

L'ajout du signe = provoque le même effet que celui évoqué plus haut pour les chaînes de caractères : **a += 2** est équivalent à **a = a + 2**.

IV-B - Les décimaux et la notation scientifique

Les mêmes opérateurs peuvent s'appliquer aux nombres à virgule, avec la même signification. Il est important toutefois de noter qu'actuellement, les ordinateurs ne sont capables de stocker qu'une valeur approchée (approximative) d'un nombre décimal. S'il est possible d'obtenir tous les chiffres d'un entier, même très grand, ce n'est pas le cas pour un nombre décimal. Voyez cet exemple frappant :

```
>>> 2**128
340282366920938463463374607431768211456
>>> 2.0**128
3.402823669209384e+38
```

Dans la première ligne, le calcul s'effectue sur des entiers. Dans la deuxième, le premier nombre est décimal. Mathématiquement, « 2 » et « 2.0 » sont parfaitement équivalents, mais l'ordinateur ne les stocke pas du tout de la même manière. Python nous renvoi un résultat exacte dans le premier cas (39 chiffres), mais un résultat approché dans le deuxième (« seulement » 17 chiffres).

Remarquez la forme spéciale de ce deuxième résultat : à la fin de la partie décimale, vous voyez les caractères "**e+38**". C'est ce que l'on appelle la *notation scientifique*. Elle indique que le nombre qui se trouve à gauche doit être multiplié par 10 élevé à la puissance indiquée, ici 38, c'est-à-dire que le résultat est en fait égal à 3.4028.... multiplié par 100000000000000000000000000000000000000. Ou comment écrire un grand nombre en peu de caractères.

Ce qu'il est important de garder à l'esprit, c'est que les nombres décimaux ont une précision limitée. En gros, dès que l'écriture du nombre nécessite plus de 16 chiffres, l'ordinateur effectue un arrondi et ne manipule plus qu'une valeur approximative, donc inévitablement entachée d'erreur. La plupart du temps, ce n'est pas grave. Mais cela pose de délicats problèmes dans des domaines où une grande précision est exigée, comme l'usinage industriel par exemple. Dernier exemple de ce phénomène, qui parfois conduit à des catastrophes : quel est le résultat de **10.0**50 + 2 - 10.0**50** ? Mathématiquement, les deux grands nombres s'annulent, cette expression vaut donc 2. Mais pas pour Python :

```
>>> 10.0**50 + 2 - 10.0**50
0.0
```

Le calcul est effectué de gauche à droite, le premier nombre calculé est donc **10.0**50**, qui est un 1 suivi de 50 zéros : c'est un très grand nombre ! Puis on ajoute 2 : c'est tellement petit par rapport au premier nombre, que la machine ne va même pas effectuer le calcul, car le résultat est très au-delà de sa capacité de précision. Puis on retranche ce même grand nombre, et on obtient 0. Ce qui est mathématiquement faux.

J'insiste sur le fait que tous les langages connaissent ce problème, qui est lié à l'électronique actuelle. Ce n'est pas une faiblesse propre à Python.

IV-C - Ces virgules qui flottent

Peut-être vous demandez-vous pourquoi on désigne les nombres décimaux par le terme de *nombres à virgule flottante*. Cela est du au fait que dans la représentation de ces nombres, la virgule (qui est représentée par un point, selon la convention anglo-saxonne) n'a pas une position fixe, c'est-à-dire que le nombre de chiffres à gauche et à droite de la virgule peut varier selon la situation.

Il est important de comprendre que le nombre que nous voyons à l'écran n'est qu'une représentation du contenu d'une zone mémoire de l'ordinateur, qui finalement ne sait manipuler que des suites de 0 et de 1, le fameux codage binaire. Nous avons vu que dans le cas du type décimal de Python, environ 16 chiffres sont disponibles. La virgule flottante nous permet donc de contenir en mémoire les nombres 1.123456789012345 et 123456789012345.1 avec la même précision, sans arrondi (ou presque).

Cette représentation s'oppose à celle dite à virgule fixe. Celle-ci impose un nombre fixe de chiffres de part et d'autre de la virgule. Selon ce principe, pour pouvoir représenter avec la même précision les deux nombres précédents, il faudrait pouvoir disposer de 32 chiffres, ce qui impliquerait que la zone mémoire soit deux fois plus vaste.

Chaque représentation possède ses avantages et inconvénients. La représentation à virgule flottante permet d'obtenir une bonne précision dans peu d'espace, au prix d'une grande complexité pour effectuer des calculs. L'utilisation de la virgule fixe permet au contraire d'effectuer les calculs de façon assez simple, mais on tombe plus rapidement dans les problèmes d'arrondis.

De nos jours, les processeurs des ordinateurs utilisent (presque) tous la représentation en virgule flottante, car c'est celle qui offre le meilleur compromis entre précision, complexité de traitement et occupation mémoire.

V - Conclusion

J'espère que l'exposé des petits désagréments que vous pouvez rencontrer ne vous aura pas décourager de la programmation ! Des solutions existent à ces problèmes, mais elles ne sont généralement pas triviales. En attendant, amusez-vous à combiner des chaînes de caractères, peut-être parviendrez-vous à reproduire la « machine à écrire » qu'une auteur fort prolixe de romans à l'eau de rose, dit-on, utilisait pour produire en grande quantité...

La prochaine fois, nous commencerons à écrire de vrais programmes pour découvrir la programmation structurée, qui fut une petite révolution en son temps. Cela passera par l'utilisation d'une notion fondamentale que nous n'avons qu'évoquée aujourd'hui : la fonction.

VI - Téléchargement

Vous pouvez télécharger ce cours sous format **pdf**

Langue	Mise à jour	Pages	Taille	Mode FTP	Mode HTTP de secours
	2005.11.20	14	81 Ko	YB02.pdf	YB02.pdf