Chapitre 3

Le monde merveilleux des variables

Difficulté : 

Au chapitre précédent, vous avez saisi vos premières instructions en langage Python, bien que vous ne vous en soyez peut-être pas rendu compte. Il est également vrai que les instructions saisies auraient fonctionné dans la plupart des langages. Ici, cependant, nous allons commencer à approfondir un petit peu la syntaxe du langage, tout en découvrant un concept important de la programmation : les variables.

Ce concept est essentiel et vous ne pouvez absolument pas faire l'impasse dessus. Mais je vous rassure, il n'y a rien de compliqué, que de l'utile et de l'agréable.

C'est quoi, une variable ? Et à quoi cela sert-il ?

Les variables sont l'un des concepts qui se retrouvent dans la majorité (et même, en l'occurrence, la totalité) des langages de programmation. Autant dire que sans variable, on ne peut pas programmer, et ce n'est pas une exagération.

C'est quoi, une variable ?

Une variable est une donnée de votre programme, stockée dans votre ordinateur. C'est un code alpha-numérique que vous allez lier à une donnée de votre programme, afin de pouvoir l'utiliser à plusieurs reprises et faire des calculs un peu plus intéressants avec. C'est bien joli de savoir faire des opérations mais, si on ne peut pas stocker le résultat quelque part, cela devient très vite ennuyeux.

Voyez la mémoire de votre ordinateur comme une grosse armoire avec plein de tiroirs. Chaque tiroir peut contenir une donnée ; certaines de ces données seront des variables de votre programme.

Comment cela fonctionne-t-il ?

Le plus simplement du monde. Vous allez dire à Python : « je veux que, dans une variable que je nomme `age`, tu stockes mon âge, pour que je puisse le retenir (si j'ai la mémoire très courte), l'augmenter (à mon anniversaire) et l'afficher si besoin est ».

Comme je vous l'ai dit, on ne peut pas passer à côté des variables. Vous ne voyez peut-être pas encore tout l'intérêt de stocker des informations de votre programme et pourtant, si vous ne stockez rien, vous ne pouvez pratiquement rien faire.

En Python, pour donner une valeur à une variable, il suffit d'écrire `nom_de_la_variable = valeur`.

Une variable doit respecter quelques règles de syntaxe incontournables :

1. Le nom de la variable ne peut être composé que de lettres, majuscules ou minuscules, de chiffres et du symbole souligné « _ » (appelé *underscore* en anglais).
2. Le nom de la variable ne peut pas commencer par un chiffre.
3. Le langage Python est sensible à la casse, ce qui signifie que des lettres majuscules et minuscules ne constituent pas la même variable (la variable `AGE` est différente de `aGe`, elle-même différente de `age`).

Au-delà de ces règles de syntaxe incontournables, il existe des conventions définies par les programmeurs eux-mêmes. L'une d'elle, que j'ai tendance à utiliser assez souvent, consiste à écrire la variable en minuscules et à remplacer les espaces éventuels par un espace souligné « _ ». Si je dois créer une variable contenant mon âge, elle se nommera donc `mon_age`. Une autre convention utilisée consiste à passer en majuscule le premier caractère de chaque mot, à l'exception du premier mot constituant la variable. La variable contenant mon âge se nommerait alors `monAge`.

Vous pouvez utiliser la convention qui vous plaît, ou même en créer une bien à vous, mais essayez de rester cohérent et de n'utiliser qu'une seule convention d'écriture. En effet, il est essentiel de pouvoir vous repérer dans vos variables dès que vous commencez à travailler sur des programmes volumineux.

Ainsi, si je veux associer mon âge à une variable, la syntaxe sera :

```
1 mon_age = 21
```

L'interpréteur vous affiche aussitôt trois chevrons sans aucun message. Cela signifie qu'il a bien compris et qu'il n'y a eu aucune erreur.

Sachez qu'on appelle cette étape *l'affectation de valeur à une variable* (parfois raccourci en « affectation de variable »). On dit en effet qu'on a affecté la valeur 21 à la variable `mon_age`.

On peut afficher la valeur de cette variable en la saisissant simplement dans l'interpréteur de commandes.

```
1 >>> mon_age
2 21
3 >>>
```

Les espaces séparant « = » du nom et de la valeur de la variable sont facultatifs. Je les mets pour des raisons de lisibilité.

Bon, c'est bien joli tout cela, mais qu'est-ce qu'on fait avec cette variable ?

Eh bien, tout ce que vous avez déjà fait au chapitre précédent, mais cette fois en utilisant la variable comme un nombre à part entière. Vous pouvez même affecter à d'autres variables des valeurs obtenues en effectuant des calculs sur la première et c'est là toute la puissance de ce mécanisme.

Essayons par exemple d'augmenter de 2 la variable `mon_age`.

```
1 >>> mon_age = mon_age + 2
2 >>> mon_age
3 23
4 >>>
```

Encore une fois, lors de l'affectation de la valeur, rien ne s'affiche, ce qui est parfaitement normal.

Maintenant, essayons d'affecter une valeur à une autre variable d'après la valeur de `mon_age`.

```
1 >>> mon_age_x2 = mon_age * 2
```

```
2 >>> mon_age_x2
3 46
4 >>>
```

Encore une fois, je vous invite à tester en long, en large et en travers cette possibilité. Le concept n'est pas compliqué mais extrêmement puissant. De plus, comparé à certains langages, affecter une valeur à une variable est extrêmement simple. Si la variable n'est pas créée, Python s'en charge automatiquement. Si la variable existe déjà, l'ancienne valeur est supprimée et remplacée par la nouvelle. Quoi de plus simple ?

Certains mots-clés de Python sont **réservés**, c'est-à-dire que vous ne pouvez pas créer des variables portant ce nom.

En voici la liste pour Python 3 :

and	del	from	none	true
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	false	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Ces mots-clés sont utilisés par Python, vous ne pouvez pas construire de variables portant ces noms. Vous allez découvrir dans la suite de ce cours la majorité de ces mots-clés et comment ils s'utilisent.

Les types de données en Python

Là se trouve un concept très important, que l'on retrouve dans beaucoup de langages de programmation. Ouvrez grand vos oreilles, ou plutôt vos yeux, car vous devrez être parfaitement à l'aise avec ce concept pour continuer la lecture de ce livre. Rassurez-vous toutefois, du moment que vous êtes attentifs, il n'y a rien de compliqué à comprendre.

Qu'entend-on par « type de donnée » ?

Jusqu'ici, vous n'avez travaillé qu'avec des nombres. Et, s'il faut bien avouer qu'on ne fera que très rarement un programme sans aucun nombre, c'est loin d'être la seule donnée que l'on peut utiliser en Python. À terme, vous serez même capables de créer vos propres types de données, mais n'anticipons pas.

Python a besoin de connaître quels types de données sont utilisés pour savoir quelles opérations il peut effectuer avec. Dans ce chapitre, vous allez apprendre à travailler avec

des chaînes de caractères, et multiplier une chaîne de caractères ne se fait pas du tout comme la multiplication d'un nombre. Pour certains types de données, la multiplication n'a d'ailleurs aucun sens. Python associe donc à chaque donnée un type, qui va définir les opérations autorisées sur cette donnée en particulier.

Les différents types de données

Nous n'allons voir ici que les incontournables et les plus faciles à manier. Des chapitres entiers seront consacrés aux types plus complexes.

Les nombres entiers

Et oui, Python différencie les entiers des nombres à virgule flottante !

Pourquoi cela ?

Initialement, c'est surtout pour une question de place en mémoire mais, pour un ordinateur, les opérations que l'on effectue sur des nombres à virgule ne sont pas les mêmes que celles sur les entiers, et cette distinction reste encore d'actualité de nos jours.

Le type entier se nomme `int` en Python (qui correspond à l'anglais « integer », c'est-à-dire entier). La forme d'un entier est un nombre sans virgule.

```
1 | 3
```

Nous avons vu au chapitre précédent les opérations que l'on pouvait effectuer sur ce type de données et, même si vous ne vous en souvenez pas, les deviner est assez élémentaire.

Les nombres flottants

Les flottants sont les nombres à virgule. Ils se nomment `float` en Python (ce qui signifie « flottant » en anglais). La syntaxe d'un nombre flottant est celle d'un nombre à virgule (n'oubliez pas de remplacer la virgule par un point). Si ce nombre n'a pas de partie flottante mais que vous voulez qu'il soit considéré par le système comme un flottant, vous pouvez lui ajouter une partie flottante de 0 (exemple **52.0**).

```
1 | 3.152
```

Les nombres après la virgule ne sont pas infinis, puisque rien n'est infini en informatique. Mais la précision est assez importante pour travailler sur des données très fines.

Les chaînes de caractères

Heureusement, les types de données disponibles en Python ne sont pas limités aux seuls nombres, bien loin de là. Le dernier type « simple » que nous verrons dans ce chapitre

est la chaîne de caractères. Ce type de donnée permet de stocker une série de lettres, pourquoi pas une phrase.

On peut écrire une chaîne de caractères de différentes façons :

– entre guillemets (`"ceci est une chaîne de caractères"`) ;
– entre apostrophes (`'ceci est une chaîne de caractères'`) ;
– entre triples guillemets (`"""ceci est une chaîne de caractères"""`).

On peut, à l'instar des nombres (et de tous les types de données) stocker une chaîne de caractères dans une variable (`ma_chaine = "Bonjour, la foule !"`)

Si vous utilisez les délimiteurs simples (le guillemet ou l'apostrophe) pour encadrer une chaîne de caractères, il se pose le problème des guillemets ou apostrophes que peut contenir ladite chaîne. Par exemple, si vous tapez `chaine = 'J'aime le Python!'`, vous obtenez le message suivant :

```
File "<stdin>", line 1
chaine = 'J'aime le Python!'
^
SyntaxError: invalid syntax
```

Ceci est dû au fait que l'apostrophe de « J'aime » est considérée par Python comme la fin de la chaîne et qu'il ne sait pas quoi faire de tout ce qui se trouve au-delà. Pour pallier ce problème, il faut **échapper** les apostrophes se trouvant au cœur de la chaîne. On insère ainsi un caractère anti-slash « \ » avant les apostrophes contenues dans le message.

```
chaine = 'J\'aime le Python!'
```

On doit également échapper les guillemets si on utilise les guillemets comme délimiteurs.

```
chaine2 = "\"Le seul individu formé, c'est celui qui a appris
    comment apprendre (...)\" (Karl Rogers, 1976)"
```

Le caractère d'échappement « \ » est utilisé pour créer d'autres signes très utiles. Ainsi, « \n » symbolise un saut de ligne (`"essai\nsur\nplusieurs\nlignes"`). Pour écrire un véritable anti-slash dans une chaîne, il faut l'échapper lui-même (et donc écrire « \\ »).

L'interpréteur affiche les sauts de lignes comme on les saisit, c'est-à-dire sous forme de « \n ». Nous verrons dans la partie suivante comment afficher réellement ces chaînes de caractères et pourquoi l'interpréteur ne les affiche pas comme il le devrait.

Utiliser les triples guillemets pour encadrer une chaîne de caractères dispense d'échapper les guillemets et apostrophes, et permet d'écrire plusieurs lignes sans symboliser les retours à la ligne au moyen de « \n ».

```
>>> chaine3 = """Ceci est un nouvel
... essai sur plusieurs
... lignes"""
>>>
```

Notez que les trois chevrons sont remplacés par trois points : cela signifie que l'interpréteur considère que vous n'avez pas fini d'écrire cette instruction. En effet, celle-ci ne s'achève qu'une fois la chaîne refermée avec trois nouveaux guillemets. Les sauts de lignes seront automatiquement remplacés, dans la chaîne, par des « \n ».

Vous pouvez utiliser, à la place des trois guillemets, trois apostrophes qui jouent exactement le même rôle. Je n'utilise personnellement pas ces délimiteurs, mais sachez qu'ils existent et ne soyez pas surpris si vous les voyez un jour dans un code source.

Voilà, nous avons bouclé le rapide tour d'horizon des types simples. Qualifier les chaînes de caractères de type simple n'est pas strictement vrai mais nous n'allons pas, dans ce chapitre, entrer dans le détail des opérations que l'on peut effectuer sur ces chaînes. C'est inutile pour l'instant et ce serait hors sujet. Cependant, rien ne vous empêche de tester vous mêmes quelques opérations comme l'addition et la multiplication (dans le pire des cas, Python vous dira qu'il ne peut pas faire ce que vous lui demandez et, comme nous l'avons vu, il est peu rancunier).

Un petit bonus

Au chapitre précédent, nous avons vu les opérateurs « classiques » pour manipuler des nombres mais aussi, comme on le verra plus tard, d'autres types de données. D'autres opérateurs ont été créés afin de simplifier la manipulation des variables.

Vous serez amenés par la suite, et assez régulièrement, à incrémenter des variables. L'incrémentation désigne l'augmentation de la valeur d'une variable d'un certain nombre. Jusqu'ici, j'ai procédé comme ci-dessous pour augmenter une variable de 1 :

```
variable = variable + 1
```

Cette syntaxe est claire et intuitive mais assez longue, et les programmeurs, tout le monde le sait, sont des fainéants nés. On a donc trouvé plus court.

```
variable += 1
```

L'opérateur += revient à ajouter à la variable la valeur qui suit l'opérateur. Les opérateurs -=, *= et /= existent également, bien qu'ils soient moins utilisés.

Quelques trucs et astuces pour vous faciliter la vie

Python propose un moyen simple de permuter deux variables (échanger leur valeur). Dans d'autres langages, il est nécessaire de passer par une troisième variable qui retient l'une des deux valeurs... ici c'est bien plus simple :

```
>>> a = 5
```

```
>>> b = 32
>>> a,b = b,a # permutation
>>> a
32
>>> b
5
>>>
```

Comme vous le voyez, après l'exécution de la ligne 3, les variables `a` et `b` ont échangé leurs valeurs. On retrouvera cette distribution d'affectation bien plus loin.

On peut aussi affecter assez simplement une même valeur à plusieurs variables :

```
>>> x = y = 3
>>> x
3
>>> y
3
>>>
```

Enfin, ce n'est pas encore d'actualité pour vous mais sachez qu'on peut couper une instruction Python, pour l'écrire sur deux lignes ou plus.

```
>>> 1 + 4 - 3 * 19 + 33 - 45 * 2 + (8 - 3) \
... -6 + 23.5
-86.5
>>>
```

Comme vous le voyez, le symbole « \ » permet, avant un saut de ligne, d'indiquer à Python que « cette instruction se poursuit à la ligne suivante ». Vous pouvez ainsi morceler votre instruction sur plusieurs lignes.

Première utilisation des fonctions

Eh bien, tout cela avance gentiment. Je me permets donc d'introduire ici, dans ce chapitre sur les variables, l'utilisation des fonctions. Il s'agit finalement bien davantage d'une application concrète de ce que vous avez appris à l'instant. Un chapitre entier sera consacré aux fonctions, mais utiliser celles que je vais vous montrer n'est pas sorcier et pourra vous être utile.

Utiliser une fonction

À quoi servent les fonctions ?

Une fonction exécute un certain nombre d'instructions déjà enregistrées. En gros, c'est comme si vous enregistriez un groupe d'instructions pour faire une action précise et que vous lui donniez un nom. Vous n'avez plus ensuite qu'à appeler cette fonction par son nom autant de fois que nécessaire (cela évite bon nombre de répétitions). Mais nous verrons tout cela plus en détail par la suite.

La plupart des fonctions ont besoin d'au moins un paramètre pour travailler sur une donnée ; ces paramètres sont des informations que vous passez à la fonction afin qu'elle travaille dessus. Les fonctions que je vais vous montrer ne font pas exception. Ce concept vous semble peut-être un peu difficile à saisir dans son ensemble mais rassurez-vous, les exemples devraient tout rendre limpide.

Les fonctions s'utilisent en respectant la syntaxe suivante :
`nom_de_la_fonction(parametre_1,parametre_2,...,parametre_n)`.

- Vous commencez par écrire le nom de la fonction.
- Vous placez entre parenthèses les paramètres de la fonction. Si la fonction n'attend aucun paramètre, vous devrez quand même mettre les parenthèses, sans rien entre elles.

La fonction « type »

Dans la partie précédente, je vous ai présenté les types de données simples, du moins une partie d'entre eux. Une des grandes puissances de Python est qu'il comprend automatiquement de quel type est une variable et cela lors de son affectation. Mais il est pratique de pouvoir savoir de quel type est une variable.

La syntaxe de cette fonction est simple :

```
type(nom_de_la_variable)
```

La fonction renvoie le type de la variable passée en paramètre. Vu que nous sommes dans l'interpréteur de commandes, cette valeur sera affichée. Si vous saisissez dans l'interpréteur les lignes suivantes :

```
>>> a = 3
>>> type(a)
```

Vous obtenez :

```
<class 'int'>
```

Python vous indique donc que la variable `a` appartient à la classe des entiers. Cette notion de classe ne sera pas approfondie avant bien des chapitres mais sachez qu'on peut la rapprocher d'un type de donnée.

Vous pouvez faire le test sans passer par des variables :

```
>>> type(3.4)
<class 'float'>
```

```
>>> type("un essai")
<class 'str'>
>>>
```

str est l'abréviation de « string » qui signifie chaîne (sous-entendu, de caractères) en anglais.

La fonction `print`

La fonction `print` permet d'afficher la valeur d'une ou plusieurs variables.

Mais... on ne fait pas exactement la même chose en saisissant juste le nom de la variable dans l'interpréteur ?

Oui et non. L'interpréteur affiche bien la valeur de la variable car il affiche automatiquement tout ce qu'il peut, pour pouvoir suivre les étapes d'un programme. Cependant, quand vous ne travaillerez plus avec l'interpréteur, taper simplement le nom de la variable n'aura aucun effet. De plus, et vous l'aurez sans doute remarqué, l'interpréteur entoure les chaînes de caractères de délimiteurs et affiche les caractères d'échappement, tout ceci encore pour des raisons de clarté.

La fonction `print` est dédiée à l'affichage uniquement. Le nombre de ses paramètres est variable, c'est-à-dire que vous pouvez lui demander d'afficher une ou plusieurs variables. Considérez cet exemple :

```
>>> a = 3
>>> print(a)
>>> a = a + 3
>>> b = a - 2
>>> print("a =", a, "et b =", b)
```

Le premier *appel* à `print` se contente d'afficher la valeur de la variable `a`, c'est-à-dire « 3 ». Le second appel à `print` affiche :

```
a = 6 et b = 4
```

Ce deuxième appel à `print` est peut-être un peu plus dur à comprendre. En fait, on passe quatre paramètres à `print`, deux chaînes de caractères et les variables `a` et `b`. Quand Python interprète cet appel de fonction, il va afficher les paramètres dans l'ordre de passage, en les séparant par un espace.

Relisez bien cet exemple, il montre tout l'intérêt des fonctions. Si vous avez du mal à le comprendre dans son ensemble, décortiquez-le en prenant indépendamment chaque paramètre.

Testez l'utilisation de `print` avec d'autres types de données et en insérant des chaînes avec des sauts de lignes et des caractères échappés, pour bien vous rendre compte de la différence.

Un petit « Hello World ! » ?

Quand on fait un cours sur un langage, quel qu'il soit, il est d'usage de présenter le programme « Hello World ! », qui illustre assez rapidement la syntaxe superficielle d'un langage.

Le but du jeu est très simple : écrire un programme qui affiche « Hello World ! » à l'écran. Dans certains langages, notamment les langages compilés, vous pourrez nécessiter jusqu'à une dizaine de lignes pour obtenir ce résultat. En Python, comme nous venons de le voir, il suffit d'une seule ligne :

```
1 >>> print("Hello World !")
```

Pour plus d'informations, n'hésitez pas à consulter la page Wikipédia consacrée à « Hello World ! » ; vous avez même des codes rédigés en différents langages de programmation, cela peut être intéressant.

▷ Wikipédia - « Hello World ! »
Code web : 696192

En résumé

- Les variables permettent de conserver dans le temps des données de votre programme.
- Vous pouvez vous servir de ces variables pour différentes choses : les afficher, faire des calculs avec, etc.
- Pour affecter une valeur à une variable, on utilise la syntaxe `nom_de_variable = valeur`.
- Il existe différents types de variables, en fonction de l'information que vous désirez conserver : `int`, `float`, chaîne de caractères etc.
- Pour afficher une donnée, comme la valeur d'une variable par exemple, on utilise la fonction `print`.