Valeurs, expressions et variables

Dans ce chapitre, nous décrivons les opérations de bases supportées par le langage Python (ce que nous avons appelé précédemment les « capacités » de l'ordinateur). Fondamentalement, un ordinateur sert à effectuer des *calculs*, mais on va voir que ce terme est à prendre au sens large. Après avoir présenté les **types de valeur** supportés par Python et les calculs qu'il peut effectuer avec ces valeurs, on verra comment les résultats de ces calculs sont stockés dans la mémoire de l'ordinateur.

Rappelons encore que ce cours n'est pas un cours de Python. La présentation ci-dessous ne présente pas le langage Python de manière exhaustive. Elle vise seulement à fournir les bases qui nous permettront d'écrire des algorithmes.

Valeurs et expressions

Dans cette section, nous présentons les valeurs que nous allons utiliser dans nos algorithmes, et les calculs possibles sur ces valeurs. Nous regroupons sous le terme **expression** les valeurs simples et les calculs plus ou moins complexes. Tous les exemples d'expressions de cette section peuvent être tapés directement dans un interpréteur Python qui affichera le même résultat.

Nombres entiers (`int`)

Python supporte, comme tous les langages de programmation classiques, le calcul sur les nombres entiers (en anglais : *integer*, abrégé par `int`). Un nombre entier s'écrit comme une série de chiffres, éventuellement précédée par un signe (`+` ou `-`). Par exemple : `0`, `42`, `+123` et `-987654` sont des entiers.

Python supporte les opérations arithmétiques usuelles sur les entiers :

```
>>> 10+3   # addition
13
>>> 10-3   # soustraction
7
>>> 10*3   # multiplication
30
>>> 10//3  # division entière
3
>>> 10%3   # modulo
1
>>> 10**3  # puissance
1000
```

Notez que le modulo est l'opération qui permet de calculer le reste de la division euclidienne (ou division entière). Par exemple, `9%2=1` car `9 = 2*4 + 1`, `4` étant la valeur obtenue par la division entière `9//2`.

Ces opérateurs peuvent bien sûr être combinés. Python respecte les règles de priorité habituelles entre opérateurs, et autorise l'utilisation des parenthèses pour les surcharger. Les espaces peuvent aussi être utilisés pour améliorer la lisibilité :

```
>>> 5+4*3**2
41
>>> 5  +  4 * 3**2
41
>>> ((5+4)*3)**2
729
>>> ((5+4) * 3)**2
729
```

Pour mémoire, il existe un ordre de priorité entre les opérateurs arithmétiques : les parenthèses sont les plus prioritaires, viennent ensuite les exposants, la multiplication et la division, puis l'addition et la soustraction. N'hésitez pas à utiliser les parenthèses pour forcer les priorités et rendre vos algorithmes plus lisibles !

Notons quelques spécificités de Python qui le distinguent d'autres langages de programmation (comme C ou Java) :

- L'opérateur de division entière `//` est différent de l'opérateur de division réelle `/` (qu'on présentera dans la section suivante). Dans d'autres langages[1], il n'existe souvent que l'opérateur `/` et c'est le contexte qui détermine le type de division qu'il représente. Le fait d'avoir deux opérateurs distincts améliore la lisibilité des programmes.
- Alors que dans la plupart des langages de programmation, les entiers sont limités en taille, Python peut faire des calculs sur des entiers arbitrairement grands (pour s'en convaincre, on pourra taper `123**456`). Si cela entraîne de moins bonnes performances, cela permet en revanche de se focaliser sur les aspects algorithmiques et calculatoires, en s'affranchissant des limites imposées par d'autres langages.

Nombres flottants (`float`)

Au delà des entiers, les ordinateurs sont capables de représenter des nombres à virgule. Ils utilisent pour cela un système nommé *nombre à virgule flottante* ou encore *nombre flottant*, ou tout simplement *flottant* (en anglais : *floating point number*, abrégé par `float`).

Sous leur forme la plus simple, les flottants sont composés d'une partie entière composée de chiffres, suivie du point décimal `.`, suivi d'une partie décimale composée de chiffres, le tout éventuellement précédé d'un signe `+` ou `-`. Par exemple : `0.0`, `4.2`, `+12.3`, ou `-987.654` sont des flottants.

Mais ils peuvent également être suivis d'un exposant, formé par la lettre `e` suivi d'un nombre entier. Cette notation correspond à la notation scientifique (le `e` signifiant « multiplié par 10 puissance... »). Par exemple, `1e2` équivaut à `100.0`, `-3.4e+5` équivaut à `-340000.0`, et `6.7e-8` équivaut à `0.000000067`.

Là encore, Python supporte les opérations arithmétiques de base sur les nombres flottants :

```
>>> 2.5 + 1.5    # addition
4.0
>>> 2.5 - 1.5    # soustraction
1.0
>>> 2.5 * 1.5    # multiplication
3.75
>>> 2.5 / 1.5    # division réelle
1.6666666666666667
>>> 2.5 ** 1.5   # puissance
3.952847075210474
```

Notons que toutes ces opérations (y compris la division réelle) peuvent combiner entiers et flottants, mais le résultat sera alors toujours un flottant, *même* si sa partie décimale est 0.

Notons enfin que, contrairement au calcul sur les entiers, le calcul sur les flottants est un calcul *approché* : certains nombres ne peuvent pas être représentés de manière exacte par des flottants, comme *1/3* (qui n'a d'ailleurs pas non plus d'écriture décimale exacte) mais aussi *1/10* ou *1/5* (ce qui peut sembler plus surprenant) :

```
>>> 0.1 + 0.2
0.30000000000000004
```

Booléens (`bool`)

On est souvent amené dans les algorithmes à manipuler des valeurs de vérité, nommées booléens (en anglais *boolean*, abrégé par `bool`). Il n'existe que deux valeurs booléennes : `True` et `False`. Les opérateurs sur les booléens sont les opérateurs logiques `and`, `or` et `not` :

```
>>> False and True
False
>>> False or True
True
>>> not False
True
```

L'intérêt des valeurs booléennes réside notamment dans la possibilité de les produire en comparant des valeurs d'autres types, puis de combiner éventuellement ces résultats avec les opérateurs logiques :

```
>>> 1+1 == 2    # égalité
True
>>> 1+1 != 2    # différence
False
>>> 3.3 < 10/3  # inférieur strict
True
>>> 3.3 > 10/3  # supérieur strict
False
>>> 1+1 <= 2    # inférieur ou égal
True
>>> 1+1 >= 2    # supérieur ou égal
True
>>> (1+1 == 2) and (3.3 > 10/3)
False
```

Note

On remarque ci-dessus que le test d'égalité utilise l'opérateur `==`, et non l'opérateur usuel `=`. On verra ci-dessous dans la section Variables que ce dernier a un *autre* sens en Python (ainsi que dans la quasi-totalité des langages de programmation).

Comme les opérateurs numériques, les opérateurs booléens et les opérateurs de comparaison ont des priorités. Pour une liste exhaustive des priorités en python, consultez la documentation.

Chaînes de caractères (`str`)

Python ne se limite pas aux valeurs purement mathématiques. Il peut également manipuler du texte, sous forme de *chaînes de caractères* ou tout simplement *chaînes* (en anglais : *character string*, abrégé par `str`). Une chaîne est une séquence de caractères de longueur arbitraire. Python supporte le codage UNICODE, ce qui permet d'utiliser un large éventail de caractères, incluant les caractères accentués du français.

Une chaîne de caractères est délimitée par les symboles `'` ou `"`. Par exemple `'hello'` ou `"bonjour le monde"` sont des chaînes de caractères.

Les principales opérations que nous utiliserons sur les chaînes de caractères consistent à calculer leur longueur, les concaténer, et en extraire des sous-chaînes :

```
>>> len("le monde")  # longueur
8
>>> "le" + "monde"   # concaténation
'lemonde'
>>> "le monde"[0]    # premier caractère
'l'
>>> "le monde"[:3]   # sous-chaîne contenant les 3 premiers caractères
'le '
>>> "le monde"[3:]   # sous-chaîne commençant au 4ème caractère
'monde'
```

Les exemples ci-dessus méritent quelques explications :

- La longueur d'une chaîne prend en compte tous les caractères, y compris les espaces.
- La concaténation n'insère pas d'espace ; pour obtenir le résultat `'le monde'` ci-dessus, il aurait fallu écrire (par exemple) `"le" + " monde"` (observez l'espace placé avant la lettre m).
- Contrairement à d'autres langages de programmation, Python n'a pas de type caractère distinct du type chaîne ; ainsi, la troisième valeur ci-dessus est simplement une *autre* chaîne, de longueur 1.
- Comme dans la plupart des langages de programmation, les chaînes de caractères sont indicées à partir de 0, donc le premier caractère a l'indice 0, le deuxième a l'indice 1, *etc*.
- En Python, lorsque l'on sélectionne un sous ensemble d'une chaîne à l'aide de bornes, la borne inférieure est incluse, et la borne supérieure est exclue. Par exemple, lorsque l'on écrit `"le monde"[3:5]`, on va sélectionner la sous-chaîne commençant à l'indice 3 et se terminant à l'indice 4, c'est-à-dire `"mo"`. Lorsque la borne inférieur n'est pas précisée, on commence au début de la chaîne, et lorsque la borne supérieure n'est pas précisée, on va jusqu'à la fin de la chaîne.

Les chaînes de caractères peuvent également être comparées avec les opérateurs de comparaison vus plus haut:

```
>>> "le monde" == "le " + "monde"
True
>>> "le monde" != "le monde "  # les espaces comptent
True
>>> "bonjour" < "bonsoir"
True
>>> "bon" < "bonjour"
True
>>> "Z" < "a"
True
>>> "é" >= "z"
True
```

On voit dans les exemples ci-dessus que l'ordre induit par les opérateurs de comparaison (`<`, `>`, `<=` et `>=`) sur les chaînes de caractères n'est pas totalement intuitif. Il se rapproche de l'ordre alphabétique, mais place les lettres majuscules avant les lettres minuscules. Par

ailleurs, il ne place pas correctement les lettres accentuées. Il est donc à utiliser avec précaution.

Enfin, notons que les chiffres sont des caractères valides ; il ne faut cependant pas confondre les chaînes de caractères composées de chiffres avec les nombres correspondants:

```
>>> "123" == 123
False
>>> "1" + "1"   # concaténation, et non addition
'11'
>>> "10" < "2"  # ordre alphabétique
True
```

Variables

Maintenant que nous avons vu les types de valeur que nos algorithmes allaient pouvoir manipuler, nous allons voir comment ces valeurs sont stockées en mémoire. La notion centrale est celle de *variable*.

variable

emplacement de la mémoire de l'ordinateur, muni d'un *nom* et contenant une valeur qui peut changer (varier) au fil du temps

affectation

opération consistant à fixer ou changer la valeur d'une variable

L'opérateur d'affectation en Python est l'opérateur `=` (comme dans la majorité des langages de programmation). À gauche de l'opérateur d'affectation, on indique le nom de la variable à affecter[2]. À droite, on donne une expression dont la valeur résultante est affectée à la variable :

```
>>> message = "bonjour le monde"
>>> message
'bonjour le monde'
```

Après sa première affectation, une variable peut être utilisée dans une expression. Son nom est simplement remplacé par sa valeur :

```
>>> a = 42
>>> a+1
43
>>> a = a+1
>>> a
43
```

Dans l'exemple ci-dessus, la variable *a* se voir d'abord affecter la valeur 42, donc `a+1` vaut 43. On affecte alors l'expression `a+1` à la variable *a*, dont la nouvelle valeur est donc 43.

Langages statiquement et dynamiquement typés

On distingue deux familles de langages de programmation qui diffèrent au niveau du typage des variables. Python appartient à la famille des langages *dynamiquement typés*. Dans ces langages, seules les valeurs sont typées ; par exemple, 42 est de type entier, "toto" est de type chaîne de caractères. Les variables, en revanche, n'ont pas de type. Ainsi dans ces langages, une variable peut en théorie contenir à des moments différents des valeurs de types différents. En pratique, cependant, on évite souvent cela, car cela nuit à la lisibilité du programme.

L'autre famille, celle des langage *statiquement typés*, inclue des langages comme le C ou Java. Dans ces langages, toute variable est associée à un type, et ne peut contenir que des valeurs de ce type. En imposant au programmeur d'exprimer ces contraintes, ces langages sont plus rigides que leurs homologues dynamiquement typés, mais peuvent en contrepartie détecter plus facilement certaines erreurs. C'est notamment le cas pour les pré/post-conditions concernant le typage des paramètres, qui sont vérifiées à la compilation, et donc garanties au moment de l'exécution. Cependant, le respect des *autres* pré/post-conditions reste de la responsabilité du programmeur.

> **Note**
>
> Dans les langage statiquement typés, une conséquence du typage des variables est la nécessité de *déclarer* une variable avant de l'utiliser.
>
> Cette contrainte n'existe pas dans les langages dynamiquement typés comme Python; la première affectation fait office de déclaration. Toute utilisation d'une variable non affectée créera une erreur à l'exécution.

Communiquer avec le monde extérieur

Dans un programme interactif, il est nécessaire d'échanger des informations avec le monde extérieur, et en particulier l'utilisatrice ou l'utilisateur. On présente ici une manière simple de lui faire saisir une valeur, ou de lui afficher une valeur. Il existe bien sûr des manières plus sophistiquées et plus ergonomiques (utilisant toutes les fonctionalités des interfaces graphiques modernes) mais elles sortent du cadre de ce cours.

Saisie

Pour renseigner une variable avec une valeur saisie par l'utilisateur, on écrit une affectation avec à droite du signe `=` l'expression `input()` :

```
>>> val = input()
hello world        ← ce texte est saisi par l'utilisateur
>>> val
'hello world'
```

Une chaîne de caractères peut-être placée entre les parenthèses de `input()` pour afficher un message à l'utilisateur, indiquant ce qu'on attend de lui ou d'elle :

```
>>> nom = input("Quel est votre nom ? ")
Quel est votre nom ? Alice
>>> nom
'Alice'
>>> nom = input("Quel age avez vous ? ")
Quel age avez vous ? 42
>>> age
'42'
```

Comme on le voit ci-dessus, la valeur retournée par *input()* est toujours une *chaîne de caractères* (même si, par exemple, l'utilisateur ne saisit que des chiffres). Si on souhaite faire saisir une valeur d'un autre type (`int`, `float` ou `bool`), il faut le préciser comme dans l'exemple ci-dessous :

```
>>> age = int(input("Quel age avez-vous ? "))
Quel age avez-vous ? 42
>>> age
42
```

Affichage

En Python, toute information peut-être affichée avec l'inscruction *print()*. On place entre les parenthèses l'expression (ou les expressions, séparées par des virgules) à afficher :

```
>>> print("Merci")
Merci
>>> print("Alors comme ça", nom, ", vous avez", age, "ans ?")
Alors comme ça Alice , vous avez 42 ans ?
```

Notes de bas de page

1

Y compris les versions antérieures de Python. Ce n'est que depuis la version 3 que Python possède deux opérateurs distincts.

2

On verra plus tard que, dans certains cas, il peut y avoir plusieurs noms de variables séparés par des virgules.