Cours : Valeurs et types

Le langage Python manipule des valeurs de différents *types*. Nous rencontrerons d'abord le type booléen, puis les types numériques : les *entiers*, les *nombres flottants* et les *nombres complexes*. En associant différentes valeurs à l'aide d'*opérateurs* pour former des *expressions*, on crée de nouvelles valeurs.

1 Booléens

Le type le plus élémentaire en Python est le type booléen. Il ne possède que deux valeurs distinctes : `True` et `False`. Cette année, nous allons essentiellement utiliser Python dans ce que l'on appelle la boucle interactive. Ce mode est aussi appelé en anglais, mode « REPL » pour : Read, Evaluate, Print, Loop. Autrement dit, lorsque l'on tape une expression, Python lit ce que l'on a écrit, l'évalue, l'affiche et est prêt pour l'interaction suivante. Lorsque l'on écrit une valeur, Python se contente de l'afficher.

```
>>> True
 True
```

On obtient le type d'une valeur grâce à la fonction `type`.

```
>>> type(True)
 bool
```

Les opérateurs logiques usuels « et », « ou » et « non » sont disponibles. On remarquera que le « ou » est bien le ou inclusif, comme en mathématiques.

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

2 Nombres entiers

On écrit les entiers en Python de manière naturelle. Le type d'un entier est le type `int`.

```
>>> 42
 42
>>> type(42)
 int
```

Les opérateurs usuels d'addition « `+` », de soustraction « `-` », de multiplication « `*` » et d'exponentiation « `**` » sont disponibles pour manipuler les entiers.

```
>>> 2 + 3 * 5
 17
>>> 2**8 - 1
 255
```

Ces opérateurs possèdent différents niveaux de priorité. L'exponentation est effectuée en premier, puis la multiplication et enfin l'addition et la soustraction. Les parenthèses sont utilisées pour préciser dans quel ordre les calculs doivent être effectués.

```
>>> (2 + 3) * 5
 25
```

Ces opérateurs sont des opérateurs *binaires* : ils permettent de créer une valeur à partir de deux valeurs. Le symbole « `-` », utilisé pour la soustraction, est aussi utilisé pour la négation. Dans ce cas, c'est un opérateur *unaire* qui crée une valeur à partir d'une unique valeur.

```
>>> - 2 * 3
 -6
```

Sa priorité est plus basse que celle de l'exponentiation, mais plus haute que celle de la multiplication. Les différents niveaux de priorité sont parfois subtils et peuvent différer d'un langage à l'autre. Il est donc souhaitable, pour des raisons de lisibilité, d'ajouter des parenthèses dès lors que l'évaluation de notre expression repose sur leur connaissance fine. N'oubliez jamais qu'un programme est écrit pour être lu non seulement par un ordinateur mais aussi par des hommes, que ce soient des correcteurs de concours ou d'autres programmeurs.

Contrairement à ce qui se passe dans la plupart des autres langages comme OCaml, Python peut représenter des nombres aussi grands que l'on souhaite. Seule la capacité mémoire de l'ordinateur impose une limite théorique à la taille des entiers. Cependant, en pratique, on peut considérer qu'il n'y a pas de limite. On appelle n-ième nombre de Mersenne l'entier $M_n = 2^n - 1$. Il est courant de chercher des nombres premiers parmi ces entiers. Le dixième nombre de Mersenne premier est M_{89} et son calcul ne pose aucun problème à Python.

```
>>> 2**89 - 1
 618970019642690137449562111
```

Python offre deux types de division. Commençons par la division entière, qui produit des entiers. Rappelons le théorème de la division euclienne sur $\mathbb{Z}$ qui affirme que si $a \in \mathbb{Z}$ et $b \in \mathbb{N}^*$, il existe un unique couple (q, r) avec $q \in \mathbb{Z}$ et $r \in \{0, 1, \dots, b-1\}$ tel que $a = bq + r$. On dit que q est le reste de la division euclidienne de a par b et que r est son reste. Par exemple $7 = 3 \times 2 + 1$, donc 2 est le quotient de la division euclidienne de 7 par 3 et son reste est 1. De même $-7 = 3 \times (-3) + 2$, donc -3 est le quotient de la division euclidienne de -7 par 3 et son reste est 2. En Python, si $a \in \mathbb{Z}$ et $b \in \mathbb{N}^*$, on obtient le quotient de la division euclidienne de a par b avec l'expression `a // b` et son reste avec `a % b`.

```
>>> 7 // 3
 2
>>> 7 % 3
 1
```

La division par 0 est considérée comme une erreur en Python et lève ce que l'on appelle une *exception*. Même s'il est possible de rattraper les exceptions, nous ne le ferons pas en classes préparatoires et en pratique une division par 0 aura pour effet d'arrêter le programme en cours. Notons que si tous les langages sont d'accord pour définir la division entière de la même manière lorsque $a \in \mathbb{N}$ et $b \in \mathbb{N}^*$, ce n'est pas le cas dès que $a < 0$. Par exemple, en OCaml, ce n'est pas la division euclidienne qui est effectuée lorsque l'on fait une division entière de -7 par 3 car ce dernier renvoie -2. Ne parlons même pas de la division par des entiers strictement négatifs qui est bien définie en Python, mais que l'on ne cherchera pas à utiliser.

Python offre aussi une division non entière, notée `/`. Elle produit une valeur qui a un type différent : un nombre flottant.

```
>>> 3 / 2
 1.5
>>> type(1.5)
 float
```

3 Nombres flottants

Les nombres flottants sont utilisés pour représenter les nombres réels. Comme tous les langages de programmation, Python utilise le « . » comme séparateur décimal. Pour calculer une valeur approchée de la circonférence d'un cercle de rayon 1, on calcule :

```
>>> 2.0 * 3.14
 6.28
```

Commençons par remarquer que les mêmes opérations arithmétiques sont disponibles avec les nombres flottants. On peut par ailleurs mélanger ces derniers avec les entiers. Tout se passe comme si les entiers étaient automatiquement convertis en flottants avant d'effectuer les calculs. Attention cependant, aucune conversion automatique n'a lieu des flottants vers les entiers. Comme pour les entiers, la division par 0.0 lève une exception et conduit à l'arrêt du programme.

Pour simplifier l'écriture de grands et de petits nombres, on peut utiliser la notation scientifique. Ainsi l'âge de l'univers étant estimé à 13.8 milliards d'années et la vitesse de le lumière étant de l'ordre de 3.0×10^8 mètres par seconde, il est impossible d'observer des endroits de l'univers à une distance supérieure à 4.14×10^{18} mètres de la terre.

```
>>> 13.8e9 * 3e8
 4.14e+18
```

Pour le calcul de la circonférence du cercle de rayon 1, on a approché π par 3.14. On pourrait bien sûr utiliser une approximation plus précise comme 3.14159.

```
>>> 2 * 3.14159
 6.28318
```

Mais la précision disponible n'est pas illimitée. En première approximation, on peut considérer que les nombres flottants ne peuvent représenter des nombres qu'avec une précision de 16 chiffres significatifs.

```
>>> 1234567890.12345678 - 1234567890.1234567
 0.0
```

Le premier nombre 1 234 567 890.123 456 78 possède 18 chiffres significatifs. Le second en possède 17. Avant d'effectuer la soustraction, les deux nombres sont donc arrondis au même nombre et le résultat final est nul. La situation est en fait plus complexe que cela, car les flottants ne sont pas stockés en base 10 mais en base 2. Ne soyez donc pas surpris si en faisant vos propres essais, vous avez l'impression que Python garde parfois 16 chiffres significatifs, parfois 17.

Pour les mêmes raisons, le résultat de chaque opération arithmétique est arrondi. Cela conduit à des résultats surprenants comme le calcul suivant qui n'est pas égal à 10^{-16} comme on pourrait s'y attendre.

```
>>> (1.0 + 1.0e-16) - 1.0
 0.0
```

En effet, $1.0 + 10^{-16}$ possède 17 chiffres significatifs. Il est donc arrondi à 1.0 avant d'effectuer la soustraction. On obtient donc un résultat nul. Comme les flottants sont stockés en base 2, même les nombres décimaux les plus simples ne sont pas représentables exactement. On peut donc avoir des résultats très surprenants.

```
>>> 0.1 + 0.2 - 0.3
 5.551115123125783e-17
```

Vous comprendrez pourquoi les logiciels de comptabilité ne travaillent pas en interne avec des nombres flottants mais plutôt avec des nombres entiers. Les nombres flottants ont cependant de nombreuses qualités et sont utilisés en simulation numérique et en intelligence artificielle. On n'oubliera pas cependant que des arrondis sont effectués à chaque opération. Nous verrons que les erreurs accumulées peuvent parfois devenir significatives et fausser complètement un résultat.

On a vu que les conversions des entiers vers les nombres flottants se font automatiquement. Cette conversion est plutôt légitime, car contrairement aux nombres décimaux, les nombres entiers de taille raisonnable (disons, ceux qui s'écrivent avec moins de 16 chiffres) sont tous représentables de manière exacte par des flottants. On parle dans ce cas de *promotion* de type.

La conversion réciproque est possible. Cependant, comme elle fait perdre de l'information, il faut la demander explicitement en utilisant la fonction `int`. Cette fonction arrondit le flottant à un entier en se rapprochant de 0, ce que l'on nomme aussi la troncature à l'unité.

```
>>> int(3.14)
 3
>>> int(-3.14)
 -3
```

De manière générale, chaque type numérique possède une fonction associée pour forcer une conversion.

Le fonction `abs` renvoie la valeur absolue d'un nombre. De nombreuses autres fonctions sont disponibles dans la bibliothèque `math`. On y trouve par exemple la fonction `floor` qui renvoie la partie entière d'un nombre. On peut calculer la racine carrée d'un nombre avec la fonction `sqrt`, abréviation de « square root ».

```
>>> abs(-1.0)
 1.0
>>> import math
>>> math.floor(3.14)
 3
>>> math.sqrt(2.0)
 1.41421356237
```

Les autres autres fonctions usuelles sont aussi disponibles.

```
>>> math.exp(1.0)
 2.71828182846
```

Le logarithme naturel (`log`), le logarithme en base 10 (`log10`) et le logarithme en base 2 (`log2`) sont aussi disponibles. Bien sûr, les fonctions trigonométriques circulaires cos, sin et tan sont présentes tout comme la constante π.

```
>>> math.cos(math.pi / 17)
 0.982973099684
```

La principale devise de Python est « batteries included ». Autrement dit, de nombreuses bibliothèques (« libraries » en anglais), sont disponibles. Nous venons d'utiliser notre première bibliothèque : le module `math`. Il existe de nombreuses manières de les rendre accessibles. La plus simple est d'écrire `import` suivi du nom de la bibliothèque. Les fonctions et constantes seront toutes disponibles, préfixées par le nom du module. Si vous souhaitez seulement en utiliser certaines sans avoir à taper à chaque fois le nom du module en entier, vous pouvez les importer pour qu'elles soient directement accessibles.

```
>>> from math import cos, pi
>>> cos(pi / 17)
 0.982973099684
```

Il est possible d'importer tous les composants du module math avec la commande :

```
>>> from math import *
```

C'est cependant une opération déconseillée par les programmeurs Python car on ne sait rapidement plus d'où vient notre fonction. Une bonne solution est de renommer le nom de la bibliothèque en un nom très court. Il y a des conventions unanimement acceptées en Python. Par exemple, la bibliothèque `numpy` qui comporte l'essentiel de la bibliothèque `math` et bien plus est souvent abbreviée de la manière suivante :

```
>>> import numpy as np
>>> np.cos(np.pi / 17)
>>> np.sqrt(2)
 1.41421356237
>>> np.sqrt(-1.0)
 nan
```

On notera que `sqrt` renvoie un nombre flottant spécial appelé NaN (Not a Number) lorsqu'elle est appelée avec un nombre négatif. On retiendra qu'avec Numpy, les fonctions travaillant avec des nombres flottants renverront NaN dès que l'on sort de leur domaine de définition. C'est la bibliothèque que nous utiliserons en priorité car elle est beaucoup plus respectueuse du standard IEEE-754 sur les nombres flottants.

4 Nombres complexes

Les nombres complexes sont des cousins des nombres flottants. Ils sont formés de deux nombres flottants : un représentant leur partie réelle et un autre représentant leur partie imaginaire. Le nombre imaginaire i est représenté par la lettre j que l'on accole à la partie imaginaire du nombre. Bien entendu, on peut mélanger entiers, flottants et nombres complexes. Tout se passe comme si tous les nombres étaient convertis en nombres complexes avant d'effectuer les opérations.

```
>>> type(1.0 + 2.0j)
 complex
>>> 1 / (1.0 + 2.0j)
 (0.2-0.4j)
```

Le module, la partie réelle et la partie imaginaire sont aussi accessibles avec les fonctions `abs`, `real` et `imag`.

```
>>> import numpy as np
>>> np.abs(1.0 + 1.0j)
 1.41421356237
>>> np.real(2.0 + 3.0j)
 2.0
```

Les types que nous venons de découvrir, `bool`, `int`, `float` et `complex` forment ce que l'on appelle les types numériques. Commençons d'abord par remarquer qu'il peut paraître surprenant que `bool` soit un type numérique. C'est cependant le cas si on identifie `False` à 0 et `True` à 1. C'est ce que fait Python pour des raisons historiques. Il vous est cependant demandé d'oublier au plus vite cette curiosité.

Cette suite de type a la particularité intéressante que l'on peut presque considérer que chaque type peut représenter de manière exacte toutes les valeurs du type qui le précède. En effet, on verra que les nombres entiers de taille raisonnable sont représentables de manière exacte par des nombres flottants et les nombres flottants sont tous représentables par des nombres complexes. Toute expression mélangeant ces types numériques sera donc évaluée en convertissant tous les types vers le type le plus large présent. C'est ce que l'on appelle la promotion automatique.

5 Chaînes de caractères

Python nous permet aussi de travailler avec du texte. Pour cela, on utilise des chaînes de caractères que l'on décrit entre en utilisant soit des « " », soit des « ' ».

```
>>> "Hello world!"
 'Hello world!'
>>> 'Ça dépend, ça dépasse.'
 'Ça dépend, ça dépasse.'
```

Python permet d'utiliser des accents dans les chaînes de caractères. Vous pouvez même utiliser des caractères espagnols, allemands, russes, hébreux, arabes ou chinois. Bref, tous les caractères de l'Unicode sont supportés. Même les smileys en font partie.

De nombreux caractères « spéciaux » existent. Par exemple, le retour à la ligne est un « caractère » que l'on obtient en écrivant « `\n` ». De même, la tabulation est un caractère que l'on obtient en écrivant « `\t` ». La plupart des éditeurs de texte affichent la tabulation en la remplaçant par 2 ou 4 espaces, mais il faut être conscient que c'est un caractère à part entière. Il est cependant très découragé de l'utiliser et un éditeur de texte bien configuré insère des espaces plutôt que le caractère de tabulation lorsque l'on utilise la touche « tab ». Enfin, si vous souhaitez utiliser des guillemets dans une chaîne de caractère et que votre choix du délimiteur vous en empêche, vous pouvez l'insérer avec « `\"` » ou « `\'` ». Par exemple :

```
>>> "What do you mean \"ew\"? I don't like Spam!"
 'What do you mean "ew"? I don't like Spam!'
```

Les chaînes de caractère ont leur propre type : le type `str`.

```
>>> type("Il suffit pas d'y dire, y faut aussi y faire.")
 str
```

On ne confondra pas les chaînes de caractères et les entiers. En particulier, si on essaie d'ajouter un entier à une chaîne de caractères en écrivant « `2 + "2"` », on obtient une erreur de type. Cependant, on peut utiliser le symbole `+` entre deux chaînes de caractères, ce qui a pour effet de les concaténer.

```
>>> "Tic" + "Tac"
 'TicTac'
```

De la même manière, il est possible de multiplier une chaîne de caractères par un entier.

```
>>> "G" + 10 * "o" + "al"
 'Gooooooooooal'
```

On peut convertir une chaîne de caractères en un entier, un nombre flottant, ou même un nombre complexe. Il suffit pour cela d'appliquer la fonction portant le nom du type désiré à la chaîne de caractères.

```
>>> n = int("2")
>>> x = float("13.1")
>>> n * x
 26.2
```

Si la chaîne de caractères ne peut être interprétée comme une valeur du type demandée, une exception sera renvoyée. Enfin, l'opération inverse est possible avec la fonction `str`.

```
>>> "Fahrenheit " + str(45) + str(1)
 'Fahrenheit 451'
```

Le standard ASCII a associé un entier entre 0 et 127 à chaque caractère de l'alphabet anglais. Plus tard, le standard Unicode a étendu cette table et a associé un entier à chaque caractère présent dans les langues internationales. Pour obtenir cet entier, on utilise la fonction `ord`.

```
>>> ord("A")
 65
>>> ord("a")
 97
>>> ord("é")
 233
```

L'opération inverse est évidemment possible. C'est d'ailleurs le moyen le plus simple d'écrire des caractères non disponibles sur votre clavier.

```
>>> chr(233)
 'é'
>>> "I " + chr(9786) + " Lazaristes."
 'I :-) Lazaristes'
```

6 Comparaisons

Pour savoir si deux valeurs sont égales, on utilise le symbole « `==` ».

```
>>> 1 + 1 == 2
 True
```

La valeur renvoyée par un test d'égalité est un *booléen*. En général, deux valeurs de types différents ne sont pas égales.

```
>>> 2 == "2"
 False
```

Cependant, cette règle connait une exception pour les types numériques où les valeurs sont promues au type le plus large avant d'effectuer le test. On a donc :

```
>>> 1.0 == 1
 True
```

Excepté pour les types numériques, on ne comparera que des valeurs de même type. La règle générale est que Python renverra `True` si les valeurs sont les mêmes et `False` si elles sont distinctes. Si vous tenez vraiment à comparer des valeurs non numériques de types différents, Python vous renverra `False`.

Pour les types numériques autres que les nombres complexes, il est possible d'utiliser les opérateurs de comparaison `<=`, `>=`, `<` et `>`. Ainsi :

```
>>> x = 3.14
>>> (3 <= x) and (x <= 4)
 True
```

Python permet de simplifier cette expression en « `3 <= x <= 4` », mais nous éviterons ce type d'expression dans nos programmes.

Les chaînes de caractères sont aussi comparables. L'ordre utilisé est l'ordre lexicographique.

```
>>> "OL" > "OM"
 False
```

Faites cependant très attention à l'ordre des caractères. Les minuscules sont bien évidemment dans l'ordre alphabétique, tout comme les majuscules. Mais la lettre « Z » est avant la lettre « a ». Les caractères sont en effet ordonnés par l'entier qui leur est associé dans la table Unicode.