# 2 Variables

## 2.1 Définition

Une **variable** est une zone de la mémoire de l'ordinateur dans laquelle une **valeur** est stockée. Aux yeux du programmeur, cette variable est définie par un **nom**, alors que pour l'ordinateur, il s'agit en fait d'une adresse, c'est-à-dire d'une zone particulière de la mémoire.

En Python, la **déclaration** d'une variable et son **initialisation** (c'est-à-dire la première valeur que l'on va stocker dedans) se font en même temps. Pour vous en convaincre, testez les instructions suivantes après avoir lancé l'interpréteur :

```
1  >>> x = 2
2  >>> x
3  2
```

Ligne 1. Dans cet exemple, nous avons déclaré, puis initialisé la variable `x` avec la valeur 2. Notez bien qu'en réalité, il s'est passé plusieurs choses :

- Python a « deviné » que la variable était un entier. On dit que Python est un langage au **typage dynamique**.
- Python a alloué (réservé) l'espace en mémoire pour y accueillir un entier. Chaque type de variable prend plus ou moins d'espace en mémoire. Python a aussi fait en sorte qu'on puisse retrouver la variable sous le nom `x`.
- Enfin, Python a assigné la valeur 2 à la variable `x`.

Dans d'autres langages (en C par exemple), il faut coder ces différentes étapes une par une. Python étant un langage dit de *haut niveau*, la simple instruction `x = 2` a suffi à réaliser les 3 étapes en une fois !

Lignes 2 et 3. L'interpréteur nous a permis de connaître le contenu de la variable juste en tapant son nom. Retenez ceci car c'est une **spécificité de l'interpréteur Python**, très pratique pour chasser (*debugger*) les erreurs dans un programme. Par contre, la ligne d'un script Python qui contient seulement le nom d'une variable (sans aucune autre indication) n'affichera pas la valeur de la variable à l'écran lors de l'exécution (pour autant, cette instruction reste valide et ne générera pas d'erreur).

Sachez par ailleurs que l'opérateur d'affectation `=` s'utilise dans un certain sens. Par exemple, l'instruction `x = 2` signifie qu'on attribue la valeur située à droite de l'opérateur `=` (ici, `2`) à la variable située à gauche (ici, `x`). D'autres langages de programmation comme *R* utilisent les symboles `<-` pour rendre l'affectation d'une variable plus explicite, par exemple `x <- 2`.

Enfin, dans l'instruction `x = y - 3`, l'opération `y - 3` est d'abord évaluée et ensuite le résultat de cette opération est affecté à la variable `x`.

## 2.2 Les types de variables

Le **type** d'une variable correspond à la nature de celle-ci. Les trois principaux types dont nous aurons besoin dans un premier temps sont les entiers (*integer* ou *int*), les nombres décimaux que nous appellerons *floats* et les chaînes de caractères (*string* ou *str*). Bien sûr, il existe de nombreux autres types (par exemple, les booléens, les nombres complexes, etc.). Si vous n'êtes pas effrayés, vous pouvez vous en rendre compte ici.

Dans l'exemple précédent, nous avons stocké un nombre entier (*int*) dans la variable `x`, mais il est tout à fait possible de stocker des *floats*, des chaînes de caractères (*string* ou *str*) ou de nombreux autres types de variable que nous verrons par la suite :

```
>>> y = 3.14
>>> y
3.14
>>> a = "bonjour"
>>> a
'bonjour'
>>> b = 'salut'
>>> b
'salut'
>>> c = """girafe"""
>>> c
```

```
'girafe'
>>> d = '''lion'''
>>> d
'lion'
```

**Remarque**

Python reconnaît certains types de variable automatiquement (entier, *float*). Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets (doubles, simples, voire trois guillemets successifs doubles ou simples) afin d'indiquer à Python le début et la fin de la chaîne de caractères.

Dans l'interpréteur, l'affichage direct du contenu d'une chaîne de caractères se fait avec des guillemets simples, quel que soit le type de guillemets utilisé pour définir la chaîne de caractères.

En Python, comme dans la plupart des langages de programmation, c'est le point qui est utilisé comme séparateur décimal. Ainsi, `3.14` est un nombre reconnu comme un *float* en Python alors que ce n'est pas le cas de `3,14`.

## 2.3 Nommage

Le nom des variables en Python peut être constitué de lettres minuscules ( `a` à `z` ), de lettres majuscules ( `A` à `Z` ), de nombres ( `0` à `9` ) ou du caractère souligné ( `_` ). Vous ne pouvez pas utiliser d'espace dans un nom de variable.

Par ailleurs, un nom de variable ne doit pas débuter par un chiffre et il n'est pas recommandé de le faire débuter par le caractère `_` (sauf cas très particuliers).

De plus, il faut absolument éviter d'utiliser un mot « réservé » par Python comme nom de variable (par exemple : `print` , `range` , `for` , `from` , etc.).

Enfin, Python est sensible à la casse, ce qui signifie que les variables `TesT` , `test` ou `TEST` sont différentes.

## 2.4 Écriture scientifique

On peut écrire des nombres très grands ou très petits avec des puissances de 10 en utilisant le symbole `e` :

```
>>> 1e6
1000000.0
>>> 3.12e-3
0.00312
```

On appelle cela écriture ou notation scientifique. On pourra noter deux choses importantes :

- `1e6` ou `3.12e-3` n'implique pas l'utilisation du nombre exponentiel *e* mais signifie $1 \times 10^6$ ou $3.12 \times 10^{-3}$ respectivement ;
- Même si on ne met que des entiers à gauche et à droite du symbole `e` (comme dans `1e6`), Python génère systématiquement un *float*.

Enfin, vous avez sans doute constaté qu'il est parfois pénible d'écrire des nombres composés de beaucoup de chiffres, par exemple le nombre d'Avogradro $6.02214076 \times 10^{23}$ ou le nombre d'humains sur Terre (au 26 août 2020) 7807568245. Pour s'y retrouver, Python autorise l'utilisation du caractère « souligné » (ou *underscore*) `_` pour séparer des groupes de chiffres. Par exemple :

```
>>> avogadro_number = 6.022_140_76e23
>>> print(avogadro_number)
6.02214076e+23
>>> humans_on_earth = 7_807_568_245
>>> print(humans_on_earth)
7807568245
```

Dans ces exemples, le caractère `_` est utilisé pour séparer des groupes de 3 chiffres mais on peut faire ce qu'on veut :

```
>>> print(7_80_7568_24_5)
7807568245
```

## 2.5 Opérations

### 2.5.1 Opérations sur les types numériques

Les quatre opérations arithmétiques de base se font de manière simple sur les types numériques (nombres entiers et *floats*) :

```
>>> x = 45
>>> x + 2
47
>>> x - 2
43
>>> x * 3
135
>>> y = 2.5
>>> x - y
42.5
>>> (x * 10) + y
452.5
```

Remarquez toutefois que si vous mélangez les types entiers et *floats*, le résultat est renvoyé comme un *float* (car ce type est plus général). Par ailleurs, l'utilisation de parenthèses permet de gérer les priorités.

L'opérateur `/` effectue une division. Contrairement aux opérateurs `+`, `-` et `*`, celui-ci renvoie systématiquement un *float* :

```
>>> 3 / 4
0.75
>>> 2.5 / 2
1.25
```

L'opérateur puissance utilise les symboles `**` :

```
>>> 2**3
8
>>> 2**4
16
```

Pour obtenir le quotient et le reste d'une division entière (voir ici pour un petit rappel sur la division entière), on utilise respectivement les symboles `//` et modulo `%` :

```
>>> 5 // 4
1
>>> 5 % 4
1
>>> 8 // 4
2
>>> 8 % 4
0
```

Les symboles `+`, `-`, `*`, `/`, `**`, `//` et `%` sont appelés **opérateurs**, car ils réalisent des opérations sur les variables.

Enfin, il existe des opérateurs « combinés » qui effectue une opération et une affectation en une seule étape :

```
>>> i = 0
>>> i = i + 1
>>> i
1
>>> i += 1
>>> i
2
>>> i += 2
>>> i
4
```

L'opérateur `+=` effectue une addition puis affecte le résultat à la même variable. Cette opération s'appelle une « incrémentation ».

Les opérateurs `-=`, `*=` et `/=` se comportent de manière similaire pour la soustraction, la multiplication et la division.

### 2.5.2 Opérations sur les chaînes de caractères

Pour les chaînes de caractères, deux opérations sont possibles, l'addition et la multiplication :

```
>>> chaine = "Salut"
>>> chaine
'Salut'
>>> chaine + " Python"
'Salut Python'
>>> chaine * 3
'SalutSalutSalut'
```

L'opérateur d'addition `+` concatène (assemble) deux chaînes de caractères.

L'opérateur de multiplication `*` entre un nombre entier et une chaîne de caractères duplique (répète) plusieurs fois une chaîne de caractères.

**⚠ Attention**

Vous observez que les opérateurs `+` et `*` se comportent différemment s'il s'agit d'entiers ou de chaînes de caractères : `2 + 2` est une addition alors que `"2" + "2"` est une concaténation. On appelle ce comportement **redéfinition des opérateurs**. Nous serons amenés à revoir cette notion dans le chapitre 19 *Avoir la classe avec les objets*.

### 2.5.3 Opérations illicites

Attention à ne pas faire d'opération illicite car vous obtiendriez un message d'erreur :

```
>>> "toto" * 1.3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'
>>> "toto" + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Notez que Python vous donne des informations dans son message d'erreur. Dans le second exemple, il indique que vous devez utiliser une variable de type *str* c'est-à-dire une chaîne de caractères et pas un *int*, c'est-à-dire un entier.

## 2.6 La fonction `type()`

Si vous ne vous souvenez plus du type d'une variable, utilisez la fonction `type()` qui vous le rappellera.

```
>>> x = 2
>>> type(x)
<class 'int'>
>>> y = 2.0
>>> type(y)
<class 'float'>
>>> z = '2'
>>> type(z)
<class 'str'>
```

Nous verrons plus tard ce que signifie le mot *class*.

**⚠ Attention**

Pour Python, la valeur `2` (nombre entier) est différente de `2.0` (*float*) et est aussi différente de `'2'` (chaîne de caractères).

## 2.7 Conversion de types

En programmation, on est souvent amené à convertir les types, c'est-à-dire passer d'un type numérique à une chaîne de caractères ou vice-versa. En Python, rien de plus simple avec les fonctions `int()`, `float()` et `str()`. Pour vous en convaincre, regardez ces exemples :

```
>>> i = 3
>>> str(i)
'3'
>>> i = '456'
>>> int(i)
456
>>> float(i)
456.0
>>> i = '3.1416'
>>> float(i)
3.1416
```

On verra au chapitre 7 *Fichiers* que ces conversions sont essentielles. En effet, lorsqu'on lit ou écrit des nombres dans un fichier, ils sont considérés comme du texte, donc des chaînes de caractères.

Toute conversion d'une variable d'un type en un autre est appelé *casting* en anglais, il se peut que vous croisiez ce terme si vous consultez d'autres ressources.

## 2.8 Note sur la division de deux nombres entiers

Notez bien qu'en Python 3, la division de deux nombres entiers renvoie par défaut un *float* :

```
>>> x = 3 / 4
>>> x
0.75
>>> type(x)
<class 'float'>
```

**Remarque**

Ceci n'était pas le cas en Python 2. Pour en savoir plus sur ce point, vous pouvez consulter le chapitre 21 *Remarques complémentaires*.

## 2.9 Note sur le vocabulaire et la syntaxe

Nous avons vu dans ce chapitre la notion de **variable** qui est commune à tous les langages de programmation. Toutefois, Python est un langage dit « orienté objet », il se peut que dans la suite du cours nous employions le mot **objet** pour désigner une variable. Par exemple, « une variable de type entier » sera pour nous équivalent à « un objet de type entier ». Nous verrons dans le chapitre 19 *Avoir la classe avec les objets* ce que le mot « objet » signifie réellement (tout comme le mot « classe »).

Par ailleurs, nous avons rencontré plusieurs fois des **fonctions** dans ce chapitre, notamment avec `type()`, `int()`, `float()` et `str()`. Dans le chapitre 1 *Introduction*, nous avons également vu la fonction `print()`. On reconnaît qu'il s'agit d'une fonction car son nom est suivi de parenthèses (par exemple, `type()`). En Python, la syntaxe générale est `fonction()`.

Ce qui se trouve entre les parenthèses d'une fonction est appelé **argument** et c'est ce que l'on « passe » à la fonction. Dans l'instruction `type(2)`, c'est l'entier `2` qui est l'argument passé à la fonction `type()`. Pour l'instant, on retiendra qu'une fonction est une sorte de boîte à qui on passe un (ou plusieurs) argument(s), qui effectue une action et qui peut renvoyer un résultat ou plus généralement un objet. Par exemple, la fonction `type()` renvoie le type de la variable qu'on lui a passé en argument.

Si ces notions vous semblent obscures, ne vous inquiétez pas, au fur et à mesure que vous avancerez dans le cours, tout deviendra limpide.

## 2.10 Minimum et maximum

Python propose les fonctions `min()` et `max()` qui renvoient respectivement le minimum et le maximum de plusieurs entiers et / ou *floats* :

```
>>> min(1, -2, 4)
-2
>>> pi = 3.14
>>> e = 2.71
>>> max(e, pi)
3.14
>>> max(1, 2.4, -6)
2.4
```

Par rapport à la discussion de la rubrique précédente, `min()` et `max()` sont des exemples de fonction prenant plusieurs arguments. En Python, quand une fonction prend plusieurs arguments, on doit les séparer par une virgule. `min()` et `max()` prennent en argument autant d'entiers et de *floats* que l'on veut, mais il en faut au moins deux.

## 2.11 Exercices

*Conseil* : utilisez l'interpréteur Python pour les exercices suivants.

### 2.11.1 Nombres de Friedman

Les nombres de Friedman sont des nombres qui peuvent s'exprimer avec tous leurs chiffres dans une expression mathématique.

Par exemple, 347 est un nombre de Friedman car il peut s'écrire sous la forme $4 + 7^3$. De même pour 127 qui peut s'écire sous la forme $2^7 - 1$.

Déterminez si les expressions suivantes correspondent à des nombres de Friedman. Pour cela, vous les écrirez en Python puis exécuterez le code correspondant.

- $7 + 3^6$
- $(3 + 4)^3$
- $3^6 - 5$
- $(1 + 2^8) \times 5$
- $(2 + 1^8)^7$

### 2.11.2 Prédire le résultat : opérations

Essayez de prédire le résultat de chacune des instructions suivantes, puis vérifiez-le dans l'interpréteur Python :

- `(1+2)**3`
- `"Da" * 4`
- `"Da" + 3`
- `("Pa"+"La") * 2`
- `("Da"*4) / 2`
- `5 / 2`
- `5 // 2`
- `5 % 2`

### 2.11.3 Prédire le résultat : opérations et conversions de types

Essayez de prédire le résultat de chacune des instructions suivantes, puis vérifiez-le dans l'interpréteur Python :

- `str(4) * int("3")`

- `int("3") + float("3.2")`
- `str(3) * float("3.2")`
- `str(3/4) * 2`