# Expressions, types et variables en Python

2015-08-26

## 1 Expressions

Les *valeurs* désignent les données manipulées par un algorithme ou une fonction. Une valeur peut ainsi être : un nombre, un caractère, une chaîne de caractères, une valeur de vérité (`Vrai` ou `Faux`) et bien d'autres ... En `Python`, comme dans la plupart des langages de programmation, ces valeurs sont *typées* selon l'objet qu'elles représentent : il y a ainsi des valeurs de type entier, flottant, chaîne de caractères, booléens ... Leur représentation en mémoire varie beaucoup d'un langage à l'autre, mais ce sont souvent les mêmes objets que l'on cherche à traduire.

Une *expression* est une suite de caractères faisant intervenir des valeurs et des opérations, afin d'obtenir une nouvelle valeur. Pour calculer cette nouvelle valeur, la machine doit *évaluer* l'expression. Voici des exemples d'expressions : `42` , `1+4` , `1.2 / 3.0` , `x+3` .
En `Python`, pour évaluer une expression, il suffit de la saisir dans un interpréteur, qui calcule et affiche alors la valeur qu'il a calculée :

```
>>> 42
42
>>> 1+4
5
```

## 2 Types simples

En programmation, associer un type à une valeur permet :

1. de classer les valeurs en catégories similaires. Par exemple, tous les entiers (type int), tous les flottants (type float)...

2. de contrôler quelles opérations sont faisables sur ces valeurs : par exemple, tout entier (type int) pourra être utilisé dans une soustraction, ..., alors qu'une chaîne ne le pourra pas. Il sera également impossible d'additioner un entier et un booléen. Dans certains langages, par exemple en Caml, il est même impossible d'additioner unentier et un flottant.

Une expression n'a a priori pas de type, car le type de la valeur qu'elle retourne dépend des types des sous-expressions. Ainsi `a+b` sera un entier (resp. un flottant, une chaîne) si `a` et `b` le sont, mais sera un flottant si `a` est un entier et `b` un flottant.

Pour afficher le type d'une valeur ou d'une expression après l'avoir évaluée, on utilise `type` :

```
>>> type(42)
<class 'int'>
>>> type(1.2/3.0)
<class 'float'>
```

Le mot qui suit `class` indique le type de la valeur, entier (`int` en anglais) pour la première et flottant (`float` en anglais) pour la seconde. Le mot `class` fait référence au fait que `Python`est un langage *orienté objet*, ce que nous ignorerons pour l'instant.

Passons en revue les types et les opérations les plus courants en `Python`.

## 2.1 Entiers

Ce type est noté `int` en `Python`.

• **Constantes :** Ce sont les entiers écrits en base 10, et ils ne sont pas bornés en `Python`.

• **Opérateurs :**

1. `+` : addition usuelle.
2. `-` : soustraction usuelle (`15-9` renvoie `6`), mais aussi opposé (`-4`).
3. `//` : *division entière* : `a//b` correspond au quotient de la division euclidienne de `a` par **b** si `b` est strictement positif (`256 // 3` renvoie `85` car $256 = 85 * 3 + 1$). Mais si `b` est strictement négatif, alors `a//b` renvoie ce quotient moins 1 ($15 = (-4) \times (-4) - 1 = (-4) \times (-3) + 3$, le quotient de la division euclidienne de 15 par $-4$ est $-3$, mais `15//-4` renvoie `-4`). Cette division n'est pas définie si `b` est nul.
4. $\%$ : *modulo* : même remarque que dans le point précédent, relativement au reste de la division euclidienne cette fois (`256 % 3` renvoie `1`, `15%-4` renvoie `-1`).
5. `**` : exponentiation (`2**3` renvoie `8`).

Les *règles de précédence*, autrement dit les règles de priorité entre opérations, sont similaires aux règles mathématiques usuelles.

## 2.2 Flottants

Ce type est noté `float` en `Python`.

• **Constantes :** Ce sont les *nombres à virgule flottante*. Nous en donnerons une définition précise dans un chapitre ultérieur : pour simplifier, disons pour l'instant que ce sont des nombres à virgule, avec un nombre borné de chiffres dans leur écriture.

• **Opérateurs :**

1. `+` : addition usuelle.
2. `-` : soustraction usuelle, et aussi opposé.
3. `/` : division usuelle.
4. `**` : exponentiation. Remarquer la différence entre `2**100` et `2.**100` ou `2**100.`.

**Avertissement :** ce qui précéde est valable en `Python`3. Attention à l'Opérateur `/` en `Python`2 ! ! ! !

## 2.3 Booléens

En logique et mathématiques, le mot *booléen* a été donné en hommage au mathématicien britannique George Boole. Ce type est noté `bool` en `Python`.

• **Constantes :** Il y en a deux : `True` et `False`.

• **Opérateurs :**

1. `not` : négation usuelle.
2. `and` : conjonction usuelle.
3. `or` : disjonction usuelle.

**Exemple 2.3.1.** `True or False` renvoie `True`, `not ( False or True) and True` renvoie `False`.

• **Opérateurs de comparaison :**

1. `==` : test d'égalité : `2==3` renvoie `False`, `4==4` renvoie `True`. Problème : `0.1+0.2==0.3` renvoie `False` ! Nous y reviendrons plus tard. Il ne faut pas confondre `==` avec `=`, symbole de l'*affectation*.
2. `!=` : `a != b` est une écriture équivalente à `not (a == b)`.
3. `<,>,<=,>=` : ce à quoi on s'attend.

**Remarque 2.3.2.** `Python`permet les chaînes de comparaisons : `1<2<3` renvoie `True`, et `(1<2<-2) and (-5<2<6)` renvoie `False`.

## 2.4 Conversions

Il est possible de convertir en entier en flottant. La réciproque est possible dans une certaine mesure : `float(2)` renvoie `2.0`, `int(2.5)` renvoie `2`, et `int(-3.5)` renvoie `-3`.

# 3 Types composés

On dit qu'une valeur est de *type composé* si elles formées à partir de plusieurs autres valeurs de types plus simples. De nombreuses constructions sont définies sur tous les types composés.

## 3.1 $n$-uplets ou tuples

Ce type est noté `tuple` en `Python`.

**• Construction et composantes :**

Les $n$-uplets généralisent les couples. Un couple s'écrira `(a,b)`, un triplet `(a,b,c)` ... etc. Il existe même des 1-uplets : il faut les écrire `(a,)`, et non `(a)` : cette dernière écriture est équivalente à `a`, tout simplement.
Il n'est pas obligatoire que les éléments d'un tuple soient de même type. Par exemple, `(1,1.2,True)` est un triplet tout à fait valable.
On accède à la $k$-ème coordonnée d'un tuple `t` par la commande `t[k]`. Attention : **en `Python`, les coordonnées sont numérotées à partir de 0** ! On dit que $k$ est l'*indice* de `t[k]`. Ainsi, si `t = (1,2,3)`, `t[0]` vaut 1, `t[2]` vaut 3, et `t[3]` n'existe pas.

On dit que les tuples sont *immuables* : il n'est pas possible de changer un élément d'un tuple.

**• Opérateurs :**

1. `+` : concaténation. `(1,2,3)+(1,4,5,6)` renvoie `1,2,3,1,4,5,6`.
2. `len` : *longueur* du tuple. `len(1,2,3)` renvoie `3`.
3. `in` : test d'appartenance. `3 in (1,2,3)` renvoie `True`, alors que `3 in (1,2,4,5)` renvoie `False`.
4. le *slicing*, qui permet de construire des sous-tuples : `(5,2,6,7,8,9)[1:5]` renvoie `(2,6,7,8)` (attention à la borne de droite !).

## 3.2 Chaînes de caractères

Ce type est noté `str` en `Python`, pour *string* en anglais.

**• Construction et composantes :**

Ce sont des suites finies de caractères, un caractère étant en python un caractère du clavier. On les note entre guillemets ou apostrophes : `"Ceci est un chaîne de caractères"` ou `'Ceci est une chaîne de caractères'`. La chaîne vide se note `""` ou `''`. Un caractère est une chaîne de longueur 1.
On accède aux composantes d'une chaîne de la même manière que pour un tuple.

**• Opérateurs :**
Ce sont les mêmes que pour les tuples.

## 3.3 Listes

Ce type est noté `list` en `Python`.

Comme les tuples, les *listes* sont des suites finies de valeurs arbitraires, mais cette fois ils sont *mutables* : on peut changer la valeur d'une composante d'une liste.
Les listes s'écrivent entre crochets. On change la valeur de la $k$-ème composante d'une liste `L` grâce à la commande `L[k] = n`, où `n` est la nouvelle valeur. Ainsi :

```
>>> L = [1,2,3]
>>> L[0] = 4
>>> L
[4,2,3]
```

Les autres opérateurs sont les mêmes que pour les tuples et les chaînes.

## 3.4 Conversions

On peut convertir des types simples en chaînes avec la fonction `str` : `str(2.3)` renvoie `'2.3'`.
À l'inverse, `int`, `float` et `bool` permettent de faire l'inverse quand cela est cohérent : **bool('True')** renvoie `True`, mais `int('2.3')` renvoie une erreur.

Il est également possible de passer d'un type composé à un autre grâce aux fonctions `str`, `tuple` et `list` : `str( (1,'a',[1,2]) )` renvoie `"(1,'a',[1,2])"`, `tuple( [1,2,3] )` renvoie `(1,2,3)` et `list('Coucou?')` renvoie `['C', 'o', 'u', 'c', 'o', 'u', ' ', '?']`.

# 4 Variables

## 4.1 Qu'est-ce qu'une variable ?

Une **variable** désigne une zone mémoire de l'ordinateur dans la RAM.
Il s'agit d'un endroit où l'on peut **stocker** une valeur, y **accéder** et **changer** cette valeur.

Pour faire référence à une variable, on utilise un nom de variable, en général composé d'une ou plusieurs lettres. Dans la mesure du possible, on choisit un nom explicite, ce qui permet une meilleure lecture du programme.

## 4.2 Affectation

Quand on stocke une valeur `d` dans une variable `var`, on dit qu'on **affecte** `d` à la variable `var`. `d` est encore appelée **la donnée**.

En `Python`, cette affectation est faite avec la commande `=`.

```
>>> var  = d
```

Les variables (et donc l'affectation) sont incontournables : si vous ne stockez pas une donnée (résultat par exemple d'un calcul), vous ne pourrez pas la réutiliser, ni la modifier.

**Exemple 4.2.1.**
```
>>> x = 42       # x prend la valeur 42.
>>> x = x + 2    # on additionne à la donnée stockée dans x
                 # le nombre 2 et on stocke le résultat de
                 # nouveau dans x
>>> x            # on accède à la valeur qui est mémorisée
                 # dans x.
 44
```

Dans la variable affectée, la nouvelle donnée "écrase" la donnée précédente : on perd son ancienne valeur.
Dans un programme `Python`, on utilisera l'instruction `print(x)` pour afficher dans la console la valeur de la variable `x`.

Il est également possible d'affecter des valeurs à plusieurs variables en une fois, en utilisant des tuples. Ainsi : `a,b = (1,2)` affecte la valeur 1 à `a` et la valeur 2 à `b`.

# 5 Fonctions

## 5.1 Objectif : modularité des programmes

Pour répondre à un problème donné, il est souvent nécessaire d'enchaîner plusieurs voire un nombre important d'instructions.

Pour améliorer la lisibilité d'un programme et aussi pour pouvoir réutiliser ces instructions classiques (dans le même programme ou dans un autre), on privilégie les programmes simples (ou sous-programmes) appelés **fonctions**.

Dans chaque langage, il y a déjà un nombre incalculable de fonctions déjà construites (en `Python`, par exemple, `print`) mais on s'aperçoit très vite que l'on a besoin de créer ses propres fonctions.

## 5.2 Écriture en langage naturel

On appelle `DefFonction` l'opération qui consiste à définir une nouvelle fonction nommée **nom-de-la-fonction** présentée comme suit :

`DefFonction`

**nom_ de_la_fonction**(*paramètres*)
« commentaire expliquant la fonction »
bloc d'instructions
sortie du résultat
**fin**

**Exemple 5.2.1.** On veut convertir en secondes une durée donnée en heures, minutes et secondes :

`DefFonction`

**conversion**(h,m,s)
« convertit en secondes une durée donnée en heures,
minutes et secondes »
$t \leftarrow 3600 * h + 60 * m + s$
Retourner t
**fin**

**Une fois définie, pour l'utiliser, on écrit son nom suivi, entre parenthèses, des paramètres.**

Pour convertir 2h35mn45s, on écrit **conversion**(2,35,45) (réponse : 9345).

**Remarques 5.2.2.**

— Pour bien se faire comprendre, il est important de choisir un nom de fonction explicite. Il faut aussi mettre des commentaires pour expliquer ce que fait la fonction.

— Il est nécessaire de bien cibler les paramètres dont on a besoin c'est-à-dire les données nécessaires à l'exécution de la fonction.

— Il faut repérer ce que retourne la fonction : rien (exemple : un simple affichage ou la modification d'un fichier...), un nombre entier, un flottant, une liste...
Reprenons l'exemple de la conversion

**1er cas.**

| | |
|---|---|
| `DefFonction` | **conversion**(h,m,s)<br>« convertit en secondes une durée donnée en heures, minutes et secondes »<br>$t \leftarrow 3600 * h + 60 * m + s$<br>Retourner t<br>**fin** |

Si on écrit `rep`=**conversion**(h,m,s), on trouvera dans `rep` un nombre entier.

**2ème cas.**

| | |
|---|---|
| `DefFonction` | **conversion**(h,m,s)<br>« convertit en secondes une durée donnée en heures, minutes et secondes »<br>$t \leftarrow 3600 * h + 60 * m + s$<br>Afficher t<br>**fin** |

Si on écrit `rep`=**conversion**(h,m,s), on ne trouvera rien dans `rep` mais la valeur s'affichera à l'écran.

— Enfin, il faut toujours tester sa fonction sur un ensemble significatif de valeurs pour repérer d'éventuelles erreurs ou manquements.

## 5.3 Écriture en `Python`

```
def nom_de_la_fonction(parametres):
    """ commentaire expliquant la fonction"""
    bloc d'instructions
    return(resultat)
```

**Remarque 5.3.1.** L'indentation des blocs n'est pas là que pour décorer. Dans la plupart des langages de programmation on *conseille* d'indenter les différents blocs afin de faciliter la lecture du code. En `Python`, l'indentation est *signifiante* : après le mot-clé `def`, chaque ligne indentée fait partie de la fonction. La première ligne non indentée rencontrée marque la fin de la fonction : cette ligne ne fait plus partie de la fonction, mais celles qui précéde en font partie. Il est donc *impératif* d'indenter quand il le faut, et seulement quand il le faut. On rencontrera ce phénomène constamment en `Python`.

# 6 Exercices

**Exercice 6.0.2.** Voici des affectations successives des variables $a$ et $b$. Dresser un tableau donnant les valeurs de $a$ et $b$ à chaque étape.

```
>>>  a = 1
>>>  b = 5
>>>  a = b-3
>>>  b = 2*a
>>>  a = a
>>>  a = b
```

**Exercice 6.0.3.** Écrire une séquence d'instructions qui échange les valeurs de deux variables.

**Exercice 6.0.4.** Écrire, sans variable supplémentaire, une suite d'affectation qui permute circulairement vers la gauche les valeurs des variables $x$, $y$, $z$ : $x$ prend la valeur de $y$ qui prend celle de $z$ qui prend celle de $x$.

**Exercice 6.0.5.** Combien d'affectations sont suffisantes pour permuter circulairement les valeurs des variables $x_1, \cdots, x_n$ sans utiliser de variable supplémentaire ? Et en utilisant autant de variables supplémentaires que l'on veut ?

**Exercice 6.0.6.** Mêmes questions que l'exercice précédent en remplaçant suffisantes par nécessaires.

**Exercice 6.0.7.** Supposons que la variable $x$ est déja affectée, et soit $n \in \mathbb{N}$. On veut calculer $x^n$ sans utiliser la puissance, avec uniquement des affectations, autant de variables que l'on veut, mais avec le moins de multiplications possible. Par exemple, avec les 4 instructions :

```
>>> y1 = x * x
>>> y2 = y1 * x
>>> y3 = y2 * x
>>> y4 = y3 * x
```

on calcule $x^5$, qui est la valeur de `y4`.
Mais 3 instructions suffisent :

```
>>> y1 = x * x
>>> y2 = y1 * y1
>>> y3 = y2 * x
```

En fonction de $n$, et avec les contraintes précédentes, quel est le nombre minimum d'instructions pour calculer $x^n$ ?

**Exercice 6.0.8.** Définir la fonction $f$ qui à $x$ associe $\begin{cases} 2 & \text{si} \quad x \in [-4,-2] \\ -x & \text{si} \quad x \in [-2,0] \\ 0 & \text{si} \quad x \in [0,4] \end{cases}$

**Exercice 6.0.9.** Ecrire une fonction calculant le produit des entiers impairs de 1 à $2n+1$.

**Exercice 6.0.10.** Soit $(a, b, c) \in \mathbb{R}^2$, $a \neq 0$.

Ecrire une fonction qui renvoie les solutions $ax^2 + bx + c = 0$ si celles-ci sont réelles, une phrase disant qu'il n'y a pas de solutions réelles sinon.

Modifier pour introduire le cas de la racine double.

**Exercice 6.0.11.**

1. Ecrire une fonction qui à un nombre entier associe le chiffre des unités.
2. Ecrire une fonction qui à un nombre entier associe le chiffre des dizaines.
3. Ecrire une fonction qui à un nombre entier associe le chiffre des unités en base 8.