

# Chapitre : 3 Bases du langage

Yves Guidet pour Édugroupe

V1.4.8 May 3, 2017

Pour l'instant, on va considérer constantes et variables des types les plus simples.  
Allons-y !

# Les types d'objets Python

- ▶ nombres,
- ▶ chaînes,
- ▶ listes,
- ▶ dictionnaires,
- ▶ tuples,
- ▶ fichiers.

- ▶ commençant par un dièse (#)
- ▶ finissent avec la ligne

- ▶ créées automatiquement
- ▶ nom des variables : `_`, lettres (pas d'accent !), chiffres
  - ▶ les noms commençant par un `_` ne sont pas exportés lorsqu'ils se trouvent dans un module;
  - ▶ ne pas commencer par une majuscule (sauf noms de classe)
  - ▶ les noms commençant par deux `_` et finissant par deux `_` sont réservés par le langage lui même,

- \* simples

`x=11`

- \* multiples

`x=y=29`

- \* parallèles

`x,y=29,11`

- ▶ La fonction `type()` permet de connaître le type d'une variable
- ▶ La fonction `dir()` permet de connaître ses méthodes

# Les types numériques : entiers

- ▶ nombres entiers :
  - ▶ int : taille = 32-bit

```
>>> x=7
```

```
>>> y=033
```

```
>>> z=0xff
```

```
>>> x
```

```
7
```

```
>>> y
```

```
27
```

```
>>> z
```

```
255
```

# Les types numériques : entiers longs

- ▶ type long.

```
x=1L
```

```
x=-451
```

```
x=1212121212121212121212121212121
```

```
x=2147483647+1
```

Conversions :

```
x = int(1L) #x est un int
```

```
x = long(1) #x est un long
```

Notons que ces conversions s'écrivent comme les *transtypes* (« casts ») en C++.

# le type long en Py3k

Comme le dit votre poly, le type *long* n'existe plus en Py3k ; une petite manip ? Voilà :

```
>>> def fact(n):  
...     if n==0:  
...         return 1  
...     else:  
...         return n*fact(n-1)  
...  
>>> fact(5)  
120
```

Une jolie fonction récursive, qui a le tort de faire très vite des dépassements de capacité. Essayons :

```
>>> fact(50)  
30414093201713378043612608166064768844377641568960512000000000000  
>>> type(fact(50))  
<class 'int'>
```

Sûr que cet *int* ne tient pas sur 32 bits. Et qu'il n'y a pas de "L" à la fin. Autrement dit les *int* sont devenus des *bigints*, ce qui était réservé aux *longs* en Python2.

Notons que la fonction *type()* répondait *type* en Python2, maintenant elle parle de *class*.

# Les types numériques : flottants et complexes

## ► float

```
x = 1.234
```

```
x = 1.0
```

```
x = 1.
```

```
x = 1.234e54
```

```
x = 1.234E54
```

```
x = -1.454e-2
```

## ► nombres complexes (on y reviendra)

```
x = 1 + 1j
```

```
x = 1.2e3 + 1.5e7j
```

```
x = 5j + 4
```

```
x = 1 + x*1j
```

## ► booléens (True, False)

# Opérateurs mathématiques

```
+  addition  6+4 -> 10
-  soustraction  6-4 -> 2
*  multiplication  6*4 -> 24
/  division  6/4 -> 1.5  # pas toujours !
# faire 6./4
// division entière  6//4 -> 1  # Nouveau !
%  reste de la division entière  6%4 -> 2
```

# La division entière

```
>>> 5/2
```

```
2
```

```
>>> from __future__ import division
```

```
>>> 5/2
```

```
2.5
```

```
>>> 5//2
```

```
2
```

# La division entière en Py3k

On est dans le futur !

```
yves@bella:Chapitre-3-Bases_du_langage$ python3
```

```
Python 3.2.3 (default, Apr 10 2013, 05:07:54)
```

```
[GCC 4.7.2] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more in
```

```
>>> 5/2
```

```
2.5
```

```
>>> 5//2
```

```
2
```

# On n'est pas en C !

```
>>> x=0
>>> x++
File "<stdin>", line 1
    x++
    ^
SyntaxError: invalid syntax
>>> x += 1
>>> x
1
```

Autrement dit `+=` (et les autres) existe alors que `++` non.

On l'a vu, il existe un type prédéfini *complex* en Python :

```
>>> z = complex(0, 1)
>>> z**2
(-1+0j)
```

Un *dir(complex)* nous apprend l'existence des méthodes :

- ▶ *conjugate*,
- ▶ *imag*,
- ▶ et *real*.

Par exemple, en continuant la manip ci-dessus :

```
>>> (z**2).imag
0.0
>>> (z**2).real
1.0
```

## Exercice << Cercle >>

- Ecrire un script calculant (et imprimant) la circonférence d'un cercle à partir de son rayon ; la valeur du rayon ne sera pas demandée à l'utilisateur, mais << codée en dur >> préalablement dans une variable.

# Quelques instructions/fonctions

- ▶ `print` : pour afficher du texte à l'écran. Exemple : `print`  
`<< texte >>`
  - ▶ (pas une fonction !)
- ▶ `input`
- ▶ `raw_input`
  - ▶ argument possible (= message)

Le mot-clé `<< print >>` est devenu une fonction en Python3k.

# Lecture au clavier

```
>>> x=input()
6
>>> x=input()
tralala
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<string>", line 1, in <module>
NameError: name 'tralala' is not defined
>>> x=raw_input()
tralala
>>> x
'tralala'
>>> x=raw_input()
1953
>>> x
'1953'
```

► type de x ?

## Et si on est en python3 ?

Eh bien plus rien ne marche ! Parce que les Entrées/Sorties ont changé.

Considérons par exemple le script :

```
nom = raw_input("Comment t'appelles-tu ? ")
```

```
print "joli nom, ça", nom
```

```
age = input("Et tu as quel âge, {} ? ".format(nom))
```

```
print "{} ans bel âge, ça {}".format(age, nom)
```

Lancé sous python3 on obtient des erreurs.

- ▶ pour ce qui est des sorties, *print* est devenu une fonction et exige des parenthèses
- ▶ *raw\_input()* en Python 2 équivaut à *input()* en Python 3
- ▶ et *input()* en Python 2 devient *eval(input())* en Python 3.

# le script 2to3

Heureusement Python fournit un script 2to3 qui s'utilise comme suit :

```
2to3 -w scr.py
```

Et cela donne :

```
nom = input("Comment t'appelles-tu ? ")
```

```
print("joli nom, ça", nom)
```

```
age = eval(input("Et tu as quel âge, {} ? ".format(nom)))
```

```
print("{} ans bel âge, ça {}".format(age, nom))
```

Au sujet de l'appel à *eval*, notons :

- ▶ que sans lui *age* serait de type *str* et non *int*,
- ▶ qu'on pourrait lui substituer un appel à *int*.

## Exercice << cercle >> (again)

- ▶ on sait maintenant lire le rayon au clavier
- ▶ on saura bientôt que *int* convertit en entier

A vous de jouer !