Chapitre 2

Premiers pas avec Python

2.1 Liens de téléchargement

Tout commence par une visite du site officiel du langage `Python` : http://www.python.org

On suit le lien DOWNLOAD puis on installe l'une de deux versions les plus récentes :

- d'une part `Python2` : http://www.python.org/download/releases/2.7.4/ (datée du 15 mai 2013)
- d'autre part `Python3` : http://www.python.org/download/releases/3.3.2/ (datée du 15 mai 2013)

Chaque installation produit un dossier dans lequel on trouvera une application nommée `Idle`.

Dans toute la suite de ce document on utilisera `Python3`.

2.2 L'application `Idle`

L'application `Idle` (*Integrated DeveLopment Environment*) permet à la fois :

- d'utiliser `Python` en mode interactif (entrer des commandes, recevoir une réponse, recommencer...)
- d'écrire et sauvegarder des programmes (on dit aussi des *scripts* ou mieux des *modules*) puis de les exécuter.

L'éditeur de l'application `Idle` propose en outre les fonctionnalités suivantes :

- coloration syntaxique (certains éléments du langage reçoivent une couleur spécifique)
- autocomplétion (avec la touche Tab), rappels syntaxiques (à la parenthèse ouvrante d'une fonction)
- indentation automatique après le caractère « : »
  les blocs de langage sont reconnus par `Python` en fonction de leur niveau d'indentation (c'est-à-dire de leur décalage par rapport à la marge gauche). Le passage à la ligne après le caractère « : » signifie l'ouverture d'un nouveau bloc (qui sera automatiquement indenté). La fin d'un bloc de niveau $n+1$ (et donc le retour au bloc de niveau n qui le contient) est obtenue par un "effacement arrière" après retour à la ligne.
- débogueur intégré : possibilité de placer des points d'arrêt, de poursuivre l'exécution du script en mode "pas à pas" et de faire le point à chaque étape (ça n'est quand même pas le point fort de `Idle`).

Il existe de nombreux environnements de développement pour `Python`, plus ou moins sophistiqués, mais on se contentera ici d'utiliser l'application `Idle`, parfaitement adaptée à l'apprentissage du langage `Python`.

On lance donc (d'une façon qui dépend du système d'exploitation utilisé) l'application `Idle`.

Un message d'information apparaît, puis le curseur se positionne juste après le « prompt » représenté ici par `>>>`

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 01:25:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> _
```

On trouvera une aide complète sur `Idle` à l'adresse suivante : http://docs.python.org/3.3/library/idle.html

2.3 Premiers essais avec Idle en mode « calculatrice »

À l'invite du « prompt » de l'application Idle, on peut commencer par entrer et évaluer des expressions très simples (en général sur une seule ligne) avec retour au prompt après évaluation.

Tout ce qui suit le caractère # est considéré comme un commentaire.

```
>>> 2**10     # ici on calcule 2 élevé à la puissance 10
------------------------
>>>
```

NB : '**' désigne ----------.

Astuces :

- si on place le curseur sur une ligne déjà ancienne (donc a priori inaccessible sauf par copier-coller), un simple appui sur « Entrée » renvoie le contenu de cette ligne (ou celui de la zone sélectionnée) au niveau du prompt.

- les combinaisons de touches Ctrl+P et Ctrl+N (ou Alt+P et Alt+N) permettent de circuler dans l'historique des lignes de commande déjà entrées (P pour *Previous*, N pour *Next*).

Le mode interactif permet d'utiliser Python comme une calculatrice.

Les parenthèses permettent de contrôler l'ordre des opérations arithmétiques qui, sinon, sont soumises aux règles de priorité habituelles.

Astuce ter : il est possible de se référer au résultat du calcul précédent avec le caractère de soulignement _ (mais ça n'est valable qu'en mode interactif) :

```
>>> 111**2                  # élève 111 au carré
------------------------
>>> _**2                    # élève le résultat précédent au carré
151807041
>>> _**2                    # élève le résultat précédent au carré
------------------------
```

La présence d'un point décimal force le passage en mode « virgule flottante » :

```
>>> 2**10                   # 2 à la puissance 10, calcul exact
------------------------
>>> _*2.                    # le double du résultat précédent, en mode flottant
------------------------
```

Il est possible d'entrer une expression longue de plus d'une ligne (ou de forcer des passages à la ligne ne devant pas être interprétés comme des demandes d'évaluation) avec le caractère \ (mais cela devrait rester exceptionnel).

```
>>> 'Ceci est une chaîne de caractères,\
 entrée sur plusieurs lignes,\
 mais affichée sur une seule!'
'Ceci est une chaîne de caractères, entrée sur plusieurs lignes, mais affichée sur une seule!'
```

Bien sûr, on peut toujours faire des erreurs (ici une division par 0, c'est ballot) :

```
>>> 1/(1+2-3)
Traceback (most recent call last):
  File "<pyshell#135>", line 1, in <module>
    1/(1+2-3)
ZeroDivisionError: division by zero
```

NB : le numéro #135 n'est pas significatif ici, il est en fait incrémenté à toute nouvelle entrée interactive.

2.4 Variables : initialisation avant utilisation

Pour mémoriser des *valeurs*, on les *associe* (on les *lie*, on les *affecte*, on les *assigne*) à des *identificateurs*.
Le couple identificateur/valeur est appelé une *variable*.

```
>>> a = 2017                # on initialise la variable (nommée a) avec la valeur 2017
>>> a*(a+1)*(a+2)           # on calcule l'expression a(a+1)(a+2)
------------------------
```

Quand une variable a été créée, et si on évalue une expression contenant l'identificateur correspondant, celui-ci est remplacé par la valeur de la variable *à ce moment précis* (cette valeur peut bien sûr changer au cours du temps).
On note que l'opérateur d'affectation (d'une valeur à un identificateur) est le signe `=`
Pour tester l'égalité de deux valeurs (résultat `True` si oui, `False` si non), on utilisera l'opérateur `==`

```
>>> a = 6           # ici on donne la valeur 6 à la variable a
>>> a == 7          # puis on teste si le contenu de a est égal à 7
------------------------
```

On gardera toujours à l'esprit la chronologie des affectations :

```
>>> x = 1           # d'abord on donne à x la valeur 1
>>> y = x           # maintenant y et x ont la même valeur 1
>>> x = 0           # finalement on donne à x la valeur 0
>>> y               # mais y vaut toujours 1
------------------------
```

Le nom d'une variable peut être arbitrairement long, et `Python` différencie les majuscules des minuscules.

```
>>> nom_un_peu_long = 1234
>>> Nom_Un_Peu_Long = 5678
>>> Nom_Un_Peu_Long - nom_un_peu_long
------------------------
```

`Python` n'est pas un outil de calcul formel. Il faut donc toujours *initialiser* une variable avant de l'utiliser :

```
>>> a = 10           # on initialise la variable a avec la valeur 10
>>> a + x            # on demande la valeur a+x mais on n'a jamais initialisé la variable x
Traceback (most recent call last):
  File "<pyshell#134>", line 1, in <module>
    a+x
NameError: name 'x' is not defined
```

Une même variable (en fait un même identificateur) peut être successivement lié à des valeurs de *types* différents (entiers, chaînes de caractère, etc.). Il n'y a donc pas lieu de préciser au préalable le *type* de valeur que l'on désire placer dans telle ou telle variable (tout cela est réalisé au moment de l'évaluation : c'est ce qu'on appelle le typage dynamique).

```
>>> n = 1234            # l'identificateur n est d'abord lié à une valeur entière
>>> n + n               # il s'agit ici bien sûr de l'addition des entiers
------------------------
>>> n = "1234"          # l'identificateur n est maintenant lié à une chaîne de caractères
>>> n + n               # l'opérateur + est ici synonyme de concaténation
------------------------
>>> n = [1,2,3,4]       # l'identificateur n est maintenant lié à une valeur de type liste
>>> n + n               # là encore, l'opérateur + est synonyme de concaténation (des listes)
------------------------
```

2.5 Variables : affectations simultanées

Il est possible d'effectuer simultanément plusieurs affectations de variables (de même type ou non) :

```
>>> a, b, c = 5, 'bonjour! ', 3.14    # ici a reçoit 5, b reçoit 'bonjour! ', c reçoit 3.14
>>> a * b                             # le résultat est a (donc 5) fois la chaîne 'bonjour! '
------------------------------------------------------------------------
>>> a + c                             # la somme de l'entier a et du flottant c est un flottant
------------------------
```

On peut initialiser plusieurs variables avec une même valeur en utilisant des = successifs.

```
>>> x = y = z = 1                     # initialise les trois variables x,y,z à la valeur 1
>>> x, y, z                           # forme le triplet (x, y, z)
--------------------
>>> a, b = c, d = 3, 8                # pose a = c = 3, et b = d = 8
>>> (a, b) = (c, d) = (3, 8)          # idem, mais c'est plus lisible comme ça
>>> a, b, c, d                        # forme le quadruplet (a, b, c, d)
--------------------
```

L'affectation simultanée est un bon moyen d'échanger le contenu de deux variables :

```
>>> x, y = 3, 7                        # on donne à x la valeur 3, à y la valeur 7 >>> x, y = y, x
# on échange les valeurs de x et y
>>> [x, y]                             # on forme ici la liste des valeurs x puis y
--------------------
```

On peut bien sûr effectuer toute permutation sur un nombre quelconque de variables.

```
>>> a, b, c, d, e = 1, 2, 3, 4, 5
>>> a, b, c, d, e = b, c, d, e, a      # permutation circulaire sur a, b, c, d, e
>>> a, b, c, d, e                      # forme le tuple (a, b, c, d, e) des nouvelles valeurs
--------------------
```

Comme dans toute évaluation, l'expression qui *suit* le signe = est évaluée en premier (donc avant que la première des affectations ne soit réalisée).

```
>>> u, v = 2, 3                  # ici u reçoit 2, et v reçoit 3
>>> u, v = v*v, u*u              # à gauche de =, lire la séquence 9, 4 (avant toute affectation)
>>> u, v                         # donc ne pas croire qu'on a effectué u = v² = 9 puis v = u² = 81
--------------------
```

2.6 Le séparateur d'instructions « ; »

Il est toujours possible d'évaluer plusieurs instructions consécutivement sur une même ligne (et par exemple plusieurs affectations de variables). Il suffit pour cela de les séparer par le caractère « ; ».

```
>>> u = 2; v = 3                 # ici u reçoit 2, puis v reçoit 3
>>> u = v*v; v = u*u             # ensuite u reçoit v² = 9,  puis v reçoit u² = 81
>>> u, v
--------------------
```

2.7 Noms de variables et mots réservés

Comme nom de variable, on peut utiliser tous les identificateurs de son choix, à l'exception de quelques mots qui sont strictement réservés par le langage, et donc voici la liste :

and	assert	break	class	continue	def	del	elif	else	except
exec	finally	for	from	global	if	import	in	is	lambda
not	or	pass	print	raise	return	try	while	yield	

Toute tentative d'utiliser l'un de ces mots réservés comme identificateur se traduit par une erreur de syntaxe (il est à noter que l'application `Idle` affiche les mots réservés avec une couleur spéciale).

```
>>> lambda = 12345                  # attention lambda est un mot réservé du langage
SyntaxError: invalid syntax
>>>
```

En revanche, les autres mots du langage (et principalement les fonctions chargées en mémoire lors du lancement de l'interpréteur `Python`) peuvent être "surchargées" (en général involontairement) sans provoquer d'erreur (du moins au début !). C'est le cas, pour prendre un exemple, de la fonction `len` qui renvoie la longueur d'une chaîne de caractères.

```
>>> len('abracadabra')              # la chaîne de caractères est de longueur 11
--------------------
>>> len = 5                         # volontairement ou non, on pose len = 5
>>> len('abracadabra')              # si on demande à nouveau la longueur d'une chaîne, erreur!
<...>
TypeError: 'int' object is not callable
```

Dans l'exemple précédent, Python nous dit que `len` est maintenant un objet de type `int` et qu'on ne peut pas l'*appeler* c'est-à-dire l'utiliser comme nom d'une fonction : il n'est plus *callable*. Voici maintenant une situation où il ne s'agit plus d'une erreur à l'exécution, mais d'une erreur logique (à moins qu'on ne sache précisément ce qu'on veut faire). On définit en effet une fonction `len` qui remplace la fonction initiale (rendant donc l'original inaccessible) :

```
>>> def len(x): return(5)  # on définit une fonction len, renvoyant constamment la valeur 5
                           # la ligne vide termine la définition de la fonction
>>> len('abracadabra')     # la fonction initiale de longueur de chaîne est maintenant inaccessible
--------------------
```

NB : on verra plus loin comment définir des fonctions beaucoup plus élaborées (et intelligentes) avec `Python` !

2.8 Quelques fonctions intégrées

Au lancement de l'application `Idle` (donc de l'interpréteur `Python`), un certain nombre de fonctions sont automatiquement chargées en mémoire (finalement pas si nombreuses : les fonctions plus spécialisées sont accessibles en *important* explicitement des *modules*, comme on le verra plus tard). La liste des fonctions intégrées (*built-in functions*) à Python est consultable à l'adresse suivante :

http://docs.python.org/3.3/library/library/functions.html

Voici une liste très partielle de ces fonctions (uniquement celles qui ont un sens dans la perspective d'une première introduction au langage `Python`) :

Function	**Signification**	**Exemples**
`abs`	valeur absolue (module)	`abs(-5)` ⇒ `----` ; `abs(3+4j)` ⇒ `----`
`bin`	chaîne binaire	`bin(1096)` ⇒ `----------` ; `bin(2**10-1)` ⇒ `----------`
`bool`	valeur Vrai/Faux	`bool(1)` ⇒ `----` ; `bool(-3)` ⇒ `----` ; `bool(0)` ⇒ `-----`
`chr`	caractère de code donné	`chr(35)` ⇒ `---` ; `chr(65)` ⇒ `---` ; `chr(97)` ⇒ `---`
`complex`	forme un complexe	`complex(1)` ⇒ `-----` ; `complex(3,4)` ⇒ `-----`
`divmod`	quotient/reste entiers	`divmod(42,5)` ⇒ `----` ; `divmod(42.,5)` ⇒ `-------`
`eval`	évalue une chaîne	`eval('2*5')` ⇒ `---` ; `eval('2'*5)` ⇒ `-----` ; `eval(2*'5')` ⇒ `--`
`float`	convertit en « flottant »	`float(123)` ⇒ `-----` ; `float(2**100)` ⇒ `--------------------`
`help`	aide sur un nom	`help(divmod)` ⇒ *`... divmod(x, y) -> (div, mod) ...`*
`hex`	chaîne héxadécimale	`hex(123456789)` ⇒ `-------` ; `hex(2**15-1)` ⇒ `--------`
`input`	entrée au clavier	`n = int(input('Choisissez un nombre entier: '))`
`int`	convertit en entier	`int(3.7)` ⇒ `---` ; `int(-3.7)` ⇒ `---` ; `int("110",2)` ⇒ `---`
`len`	longueur d'un objet	`len('abcd')` ⇒ `--` ; `len([2,4,6])` ⇒ `--` ;`len(range(3,8))` ⇒ `---`
`max`	calcul de maximum	`max(3,9,2)` ⇒ `---` ;`max([3,9,2])` ⇒ `---` ;`max('a','B','Z')` ⇒ `---`
`min`	calcul de minimum	`min(3,9,2)` ⇒ `---` ;`min([3,9,2])` ⇒ `---` ;`min('a','B','Z')` ⇒ `---`
`ord`	code d'un caractère	`ord('#')` ⇒ `----` ; `ord('A')` ⇒ `----` ; `ord('a')` ⇒ `----`
`pow`	calcul de puissance	`pow(2,5)` ⇒ `----` ; `pow(0,0)` ⇒ `----` ; `pow(16,1/2)` ⇒ `----`
`print`	affiche à l'écran	`print(2+3)` ⇒ (*`affiche ----`*) ; `print('2+3')` ⇒ (*`affiche ----`*)
`range`	intervalle de valeurs	`range(3,11)` ⇒ `----------` `range(3,11,2)` ⇒ `----------`
`repr`	convertit en chaîne	Le résultat est destiné à être réutilisable par l'interpréteur `Python`
`round`	arrondi	`round(2.5)` ⇒ `---` ;`round(2.6)` ⇒ `----` ;`round(1.456789,2)` ⇒ `----`
`sorted`	copie triée d'un objet	`sorted([1,5,2,8,3])` ⇒ `----------` `sorted("aebfc")` ⇒ `----------`
`str`	convertit en chaîne	Le résultat est destiné à être affiché de façon naturelle
`sum`	calcul d'une somme	`sum([1,2,3])` ⇒ `----` ; `sum([1,2,3],1000)` ⇒ `-----`
`type`	type (classe) d'un objet	`type(5)` ⇒ `----------` ; `type(5.)` ⇒ `----------` `type('5')` ⇒ `----------` ; `type([1,2])` ⇒ `----------`

Remarque : les chaînes de caractères sont indifféremment encadrées par des guillemets simples ' ou doubles ". L'utilisation de guillemets simples (resp. doubles) permet d'incorporer des guillemets doubles (resp. simples) comme un caractère normal de la chaîne.

Chapitre 3

Types numériques, comparaisons

Toutes les valeurs utilisées en `Python` possèdent un *type* (on devrait plutôt dire une *classe*).

On va se limiter ici aux types numériques simples (booléens, entiers, flottants, nombres complexes), et reporter à plus tard l'étude du type "chaîne de caractères" et des types composés (listes, tuples, intervalles, dictionnaires, etc.) dont les valeurs sont obtenues par regroupements de valeurs de type simple.

3.1 Quelques types (classes) de base

Il suffit d'interroger la fonction `type` pour confirmer la classe de quelques valeurs autorisées :

```
>>> type(1 == 2)           # l'égalité (fausse) 1==2 appartient à la classe 'bool' des booléens
-----------------
>>> type(1 + 2 + 3)        # l'expression 1+2+3 appartient à la classe 'int' des entiers
-----------------
>>> type(1 + 2 + 3.)       # à cause du point décimal, l'expression 1+2+3. est un 'float'
-----------------
>>> type(2 + 3j)           # un nombre complexe. Attention, noter j ou J et pas i ou I
-----------------          # N.B. on notera 1j, ou 1.j, plutôt que j seul.
>>> type(2 ** 100)         # la valeur 2^100 est de type 'int'
-----------------
```

On notera que le nombre complexe i est noté `j` ou `J` en Python, et que cette lettre `j` (ou `J`) doit être obligatoirement être utilisée comme suffixe d'une valeur (de type 'int' ou 'float') afin d'être reconnue sans ambiguïté :

```
>>> j = 5                  # on donne la valeur 5 à la variable j
>>> 2 + j                  # ici c'est la somme de 2 et du contenu de j
-----
>>> 2 + 1j                 # mais ici c'est le nombre complexe 2 + i
-----
```

3.2 Opérations entre types numériques

On considère ici les types (les *classes*) 'int', 'float' et 'complex'.

Les valeurs de ces types (on dit aussi les *instances* de ces classes) partagent un certain nombre d'opérations arithmétiques communes (addition, produit, etc.).

Quand on évalue une expression arithmétique contenant des valeurs numériques, on peut considérer la classe 'int' comme incluse dans la classe 'float', elle-même incluse dans la classe 'complex.

Lors de l'évaluation d'une telle expression, le calcul s'effectue dans la classe la plus étroite qui soit compatible avec tous les arguments, et c'est dans cette classe qu'est renvoyé le résultat (c'est pour cette raison, par exemple, que la somme de l'entier 2 et du flottant 3.0 est le flottant 5.0).

Voici les principales opérations sur les types numériques :

`x+y  x-y  x*y  x/y`	somme, différence, produit, quotient
`pow(x,y)  x**y`	calcule x à la puissance y. Par défaut $0^0 = 1$
`x//y  x%y`	quotient et reste dans une division euclidienne entre entiers
`divmod(x,y)`	renvoie la paire `(x//y, x%y)`
`abs(x)`	valeur absolue, module
`int(x)  float(x)`	conversion vers le type 'int', ou vers le type 'float'
`round(x)  round(x,n)`	arrondit à l'entier le plus proche, ou valeur approchée à 10^{-n} près
`z.real  z.imag  z.conjugate()`	partie réelle, partie imaginaire, conjugué
`complex(x,y)  x+y*1j`	renvoie le nombre complexe $x+iy$

NB : une source d'erreur classique concerne les divisions par `/` ou `//` (c'est une différence entre `Python2` et `Python3`). Avec `Python3`, l'opérateur `//` fournit le quotient entier dans la division euclidienne.

```
>>> 2013 // 17        # le quotient dans la division euclidienne: le résultat est un 'int'
-----
>>> 2013 / 17         # le résultat dans la division dans ℝ: le résultat est un 'float'
--------------
>>> 2013 // 17.       # division entière aussi, mais 17. est un 'float', donc résultat 'float'
--------------
>>> (1+2j)/(3-5j)     # division de deux complexes (ici // donnerait une erreur de type)
------------------------------
```

NB : dans la division (dite entière) par `//` et `%`, le quotient et le reste dans la division de x par y sont respectivement l'entier q et le réel r tels que : $x = qy + r$, avec $0 \leq r < y$ si $y > 0$, et $y < r \leq 0$ si $y < 0$. Si l'un au moins de x ou y est un flottant, alors q et r sont des flottants (bien que le flottant q ait une valeur entière...)

```
>>> divmod(20,7)      # 20 = 7q + r, avec q = 2 et r = 6       (et on a bien 0 ≤  r < 7)
--------
>>> divmod(-20,7)     # −20 = 7q + r, avec q = −3 et r = 1     (et on a bien 0 ≤  r < 7)
--------
>>> divmod(20,-7)     # 20 = −7q + r, avec q = −3 et r = −1    (et on a bien −7 <  r ≤  0)
--------
>>> divmod(-20,-7)    # −20 = −7q + r, avec q = 2 et r = −6    (et on a bien −7 <  r ≤  0)
--------
>>> divmod(20.,7)     # ici les résultats sont des floats car le dividende est un float
------------
```

3.3 Les opérateurs avec assignation

Il est courant d'avoir à incrémenter une variable. L'instruction `i = i+1` sera avantageusement remplacée par `i += 1`.

On dit que `+=` est un *opérateur avec assignation*.

Voici la liste des opérateurs avec assignation de `Python`. Leur portée va bien au delà des types numériques : ils s'appliquent à tous les types sur lesquels l'opération (addition, produit, etc.) a un sens.

`x+=y` équivaut à `x=x+y`	`x-=y` équivaut à `x=x-y`	`x*=y` équivaut à `x=x*y`
`x/=y` équivaut à `x=x/y`	`x%=y` équivaut à `x=x%y`	`x**=y` équivaut à `x=x**y`
	`x//=y` équivaut à `x=x//y`	

Exemples:

```
>>> x = 9; x += 1; x              # x contient l'entier 9, et lui ajoute 1
-----
>>> x = "abc"; x += "de"; x       # x contient la chaîne 'abc', et lui ajoute la chaîne 'de'
-----

>>> x = [1,2,3]; x += [4,5]; x    # x contient la liste [1,2,3], et lui ajoute la liste [4,5]
----------
```

3.4 Valeurs booléennes et comparaisons

Les branchements du type "si condition alors... (sinon...)" servent à orienter le flux des instructions en fonction de la réponse (vraie ou fausse) à une condition.

Cette condition résulte le plus souvent de la comparaison de deux valeurs.

Pour `Python`, les valeurs "vrai" et "faux" sont respectivement représentées par les constantes `True` et `False` (qui sont d'ailleurs les deux seules valeurs du type `bool`) :

```
>>> type(True), type(False)
----------------------------
```

Mais toutes les valeurs numériques ont une traduction booléenne : 0 représente le "faux", et toute valeur non nulle représente le "vrai" (inversement `True` et `False` sont synonymes des entiers 1 et 0). Plus généralement : tout objet non vide (chaîne de caractères, liste, etc.) est considéré comme "vrai", les objets vides représentant donc la valeur "faux".

Voici les opérateurs qui permettent de comparer des valeurs en `Python` :

$<$	(strictement inférieur à)	$>$	(strictement supérieur à)
$<=$	(inférieur ou égal à)	$>=$	(supérieur ou égal à)
$==$	(égal à)	$!=$	(différent de)

On notera bien sûr la différence entre le test d'égalité == et l'instruction d'affectation =.

Voici maintenant les opérateurs qui permettent de combiner des valeurs booléennes entre elles :

`or`	(ou logique)	`and`	(et logique)	`not`	(négation logique)

Quand deux valeurs numériques de types différents (entier, flottant, nombre complexe) sont comparées, elles sont converties dans le type qui les contient toutes les deux (cela explique, comme on le voit dans l'exemple ci-après, que `Python` considère comme égaux l'entier 4 et le flottant 4.0, ou encore le flottant 0. et le nombre complexe $0j$).

```
>>> 1 < 2, 3 <= 2, 4 != 5, 4 == 4., 0. == 0j
----------------------------------
```

On peut *enchaîner* les comparaisons. Par exemple l'expression $x < y < z$ signifie ($x < y$ et $y < z$).
De même, l'expression $x < y > z != t$ signifie ($x < y$ et $y > z$ et $z \neq t$).

```
>>> 1 < 4 > 2 != 0 < 3    # ici il faut lire: (1 < 4) et (4 > 2) et (2 ≠ 0) et 0 < 3
-------
```

Les chaînes de caractères sont comparées selon l'ordre lexicographique (mais les caractères eux-mêmes sont évalués en fonction de leur code numérique : les minuscules viennent donc avant les majuscules).

```
>>> 'A' < 'Z' < 'a' < 'z'
------
>>> 'ABC' < 'ABc' < 'AbC' < 'Abc' < 'aBC' < 'aBc' < 'abC' < 'abc'      # huit chaînes différentes -
-------
```

Dans une répétition b_1 `or` b_2 `or` $\cdots$ `or` b_n de "ou" logiques, les valeurs booléennes $b_1, \ldots, b_n$ sont évaluées de gauche à droite, et l'évaluation s'arrête si l'une d'elles vaut `true` (car alors toute l'expression vaut `true`). Il en est de même dans une répétition b_1 `and` b_2 `and` $\cdots$ `and` b_n de "et" logiques : l'évaluation s'arrête dès que l'une des valeurs b_k vaut `false` : c'est ce qu'on appelle l'*évaluation paresseuse* des booléens.

Dans l'exemple ci-dessous, les comparaisons $1/3 < 1/2$ et $1/4 < 1/3$ renvoient la valeur "vrai", donc l'évaluation continue jusqu'à la comparaison $1/2 < 1/0$ qui provoque une erreur :

```
>>> 1/3 < 1/2 and 1/4 < 1/3 and 1/2 < 1/0
<...>
ZeroDivisionError: division by zero
```

Au contraire, dans le deuxième exemple, la comparaison $1/3 < 1/4$ renvoie la valeur "faux", ce qui interromp l'enchaînement des "et" successifs (donc évite la dernière évaluation qui aurait conduit à une erreur de division par 0).
Dans le dernier exemple, c'est la comparaison vraie $1/4 < 1/3$ qui arrête l'enchaînement des "ou" successifs :

```
>>> 1/3 < 1/2 and 1/3 < 1/4 and 1/2 < 1/0
------
>>> 1/2 < 1/3 or 1/4 < 1/3 or 1/2 < 1/0
------
```

Si x, y sont de types différents (l'une d'elle au moins étant non numérique) les comparaisons $x == y$ et $x != y$ sont possibles (et renvoient respectivement `False` et `True`). Les autres comparaisons (inégalités) renvoient une erreur de type :

```
>>> 123 == "123"          # L'entier 123 et la chaîne "123", ça n'est pas la même chose
------
>>> 123 <= "123"          # Mais de là à les comparer...
------
TypeError: unorderable types: int() <= str()
```

Remarque : les opérateurs `<, <=, >, >=, !=, ==` ont le même degré de priorité, plus faible que celui des opérateurs arithmétiques. Mais ils sont prioritaires devant `not`, lui-même prioritaire devant `and`, lui-même prioritaire devant `or`.
Par exemple $x + y < z$ se lit $(x + y) < z$, et non pas $x + (y < z)$:

```
>>> x = 1; y = 3; z = 4
>>> x + y < z          # ici, on teste si la somme x + y est strictement inférieure à z
-----
>>> x + (y < z)        # ici, y < z vaut True, c'est-à-dire 1, et donc on calcule x + 1
-----
```

3.5 Égalité structurelle et égalité physique

Quand on affecte une valeur à une variable (par exemple $x = 2013$), on crée en fait un *pointeur* (une adresse) pour désigner cette valeur qu'il faut imaginer quelque part en mémoire.

Les variables `Python` sont donc des références (des adresses, des pointeurs) vers des valeurs. Pour connaître l'adresse où se trouve exactement la valeur qu'on a associée à un identificateur, on utilise la fonction intégrée `id`.

L'*égalité structurelle* de deux objets signifie l'égalité de leurs valeurs, alors que l'*égalité physique* signifie l'égalité des adresses de ces valeurs (bien sûr l'égalité physique implique l'égalité structurelle, mais la réciproque est fausse).

Pour tester l'égalité structurelle, on utilise l'opérateur `==`, et pour l'égalité physique, on utilise l'opérateur `is`.

Ces distinctions ne sont pas très importantes quand on manipule des objets dits "non mutables" (comme les types de base : entiers, flottants, nombres complexes, chaînes de caractères, et également les tuples qui seront étudiés plus tard).

En revanche, la distinction entre égalité physique et égalité structurelle prend tout son sens quand on modifie les éléments d'objets "mutables" (comme les listes ou les dictionnaires).

Exemple 1:

```
>>> x = 2013       # on initialise la variable x avec la valeur 2013.
>>> id(x)          # voici exactement à quelle adresse de la mémoire se trouve la valeur

---------
>>> y = 2013       # on initialise la variable y avec la même valeur 2013.
>>> id(y)          # on voit que les deux valeurs 2013 sont à des adresses différentes en mémoire
---------
>>> x == y         # on demande si les valeurs sont égales, la réponse est oui
True
>>> x is y         # on demande si les valeurs sont à la même adresse, la réponse est non
---------
```

Continuons l'exemple précédent.

```
>>> y = x                # on recopie la variable x dans la variable y
>>> (id(x), id(y))       # x et y se réfèrent à la même valeur, à un endroit précis de la mémoire
---------------------
>>> (x is y, x == y)     # il y a égalité physique, donc égalité structurelle
-------------
```

Continuons, en modifiant maintenant le contenu de la seule variable y :

```
>>> y += 1              # on ajoute 1 au contenu de la variable y
>>> (x,y)               # y contient maintenant 2014, mais x a conservé sa valeur 2013
------------
>>> (id(x), id(y))      # x pointe vers la même adresse, mais pas y
--------------------
>>> (x == y, x is y)    # il n'y a plus égalité structurelle, et encore moins égalité physique
-----------
```

Exemple 2: On va maintenant effectuer des opérations analogues, mais sur des listes (les objets de type liste seront étudiés en détail un peu plus loin) :

```
>>> x = [1,2,3]; y = [1,2,3]   # on met la liste [1,2,3] dans x, puis dans y
>>> (x is y, x == y)           # les listes ne sont pas à la même adresse, mais ont la même valeur
------------
```

On continue sur cet exemple, en redéfinissant la liste y

```
>>> y = y + y          # on concatène la liste y à elle-même
>>> (x, y)             # x et y contiennnent donc deux listes de valeurs différentes
------------------------
```

On va maintenant recopier à nouveau x dans y, mais modifier seulement un élément de y.

```
>>> y = x           # on recopie le contenu de x (la liste [1,2,3]) dans la variable y
>>> (id(x), id(y))  # x et y pointent sur une même adresse en mémoire
---------------------
>>> y[0] = 999      # on place 999 en position 0 (le 1er élément) de la liste contenue dans y
>>> (x, y)          # on voit que la modification a été répercutée sur la liste contenue dans x !!
---------------------
>>> (id(x), id(y))  # en fait les variables x et y pointent toujours sur la même adresse
------------------
>>> x is y          # il s'agit donc d'égalité physique
--------
```

L'exemple précédent est important. Les listes `Python` sont des objets composites (formés d'éléments a priori disparates mais rassemblés dans une même structure). Si on modifie un élément d'une liste, `Python` ne modifie pas l'adresse de celle-ci (en fait, il modifie seulement l'adresse de l'élément concerné dans cette liste).

Cela explique qu'après l'instruction $y = x$ (à l'issue de laquelle x et y pointent sur une même liste en mémoire), la modification $y[0] = 999$ semble répercutée sur la liste contenue dans la variable x (tout simplement car les variables x et y continuent à pointer sur la même adresse).

L'instruction $y = x$ n'a donc pas associé à y une nouvelle copie de la liste associée à x, elle a fait pointer y et x sur une même adresse (toute modification d'un des deux éléments $x[k]$ ou $y[k]$ sera donc "répercutée" sur l'autre).

Si on veut créer une copie d'une liste qui soit indépendante de l'original, on procédera de la façon suivante :

```
>>> x = [1,2,3]             # on place une liste dans la variable x
>>> y = x[:]                # on place dans y une copie indépendante de cette liste
>>> (x, y)                  # les deux valeurs sont identiques
-----------------------
>>> (id(x),id(y))           # mais les listes ne sont pas à la même adresse
-----------------------
>>> (x is y, x == y)        # il y a égalité structurelle, mais pas égalité physique
--------------
>>> y[0] = 999              # on modifie le premier élément de la liste y
>>> (x, y)                  # mais ça ne se répercute plus sur la liste x
---------------------
```

Remarque : l'expression `x[:]` est un cas particulier de la syntaxe `x[a:b]` qui renvoie la liste des éléments de l'objet x situés de la position a (incluse) à la position b (exclue). Par défaut a vaut 0 (c'est le début de l'objet x) et b vaut la longueur de cet objet. Ainsi `x[:]` renvoie donc la totalité de l'objet x (mais cette copie est indépendante de l'original).

Attention : la notion de "copie originale", telle qu'elle a été évoquée ci-dessus (l'instruction `y = x[:]`) n'est tout à fait exacte que si la liste source (ici x) ne contient que des objets "non mutables" (nombres, chaînes, tuples). En effet, après la copie, l'adresse de y est différente de celle de x, mais l'adresse de chaque $y[i]$ reste celle de $x[i]$.

Un exemple permettra d'y voir plus clair :

```
>>> x = [0,[10,20],2,[30,31,32]]          # on fabrique une liste composite
>>> [id(x),[id(t) for t in x]]            # l'adresse de x, et celles des x[i] successifs
[4318592032, [4297261120, 4328417832, 4297261184, 4359646240]]
>>> y = x[:]; y                           # on effectue une copie "originale" de x dans y
[0, [10, 20], 2, [30, 31, 32]]
>>> [id(y),[id(t) for t in y]]            # l'adresse de y est nouvelle, pas celle des y[i]
[4359645448, [4297261120, 4328417832, 4297261184, 4359646240]]
>>> y[0] = 1000; y[1][0] = 2013; y        # modifions y[0] (non mutable) et y[1] (mutable)
[1000, [2013, 20], 2, [30, 31, 32]]
>>> x                                     # pas d'impact sur x[0], mais impact sur x[1]
[0, [2013, 20], 2, [30, 31, 32]]
```

3.6 Les fonctions mathématiques du module `math`

La plupart des fonctions mathématiques usuelles (racine carrée, sinus, logarithme, etc.) ne sont pas chargées en mémoire au démarrage de l'interpréteur `Python`. Elles appartiennent à un module nommé `math` qu'il faut donc *importer*. Il y a plusieurs possibilités (l'instruction choisie doit bien sûr précéder l'utilisation des fonctions intégrées au module) :

`import math` charge la totalité du module `math` les fonctions mathématiques sont alors accessibles sous la forme `math.nom_de_la_fonction`
`from math import *` charge la totalité du module `math` on accède aux fonctions du module sans utiliser le préfixe "`math.`"
`from math import sqrt, log, exp` (par exemple) charge seulement certaines fonctions du module on accède aux fonctions chargées sans le préfixe "`math.`"

Remarque : le mécanisme décrit ci-dessus est commun à tous les modules de `Python` (qu'ils soient intégrés à la distribution elle-même ou créés par des développeurs tiers). La méthode `import nom_du_module` est généralement conseillée, car elle évite des conflits de noms potentiels (on sait toujours de quel module vient telle ou telle fonction).

Autre remarque : quand le nom d'un module est un peu long, on peut utiliser un *alias*. Par exemple, pour charger le module `itertools`, on pourra écrire `import itertools as it`. De cette façon `it` devient un alias de `itertools`, et on pourra par exemple utiliser la fonction `accumulate` en écrivant simplement `it.accumulate`.

Pour éviter la lourdeur due à l'utilisation du préfixe "math.", on pourra utiliser la méthode `from math import ...`

On voit ici la première instruction (et la première erreur) au lancement de l'application `Idle` : tant qu'on ne l'a pas importée, on ne peut pas utiliser la fonction `sqrt` car elle est inconnue de l'interpréteur `Python`.

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 01:25:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> sqrt(5)
Traceback (most recent call last):
[...]
NameError: name 'sqrt' is not defined
>>>
```

On importe donc la totalité du module `math` par l'instruction `import math`. L'erreur se produit à nouveau si on oublie de préfixer par "`math.`" le nom de la fonction `sqrt` :

```
>>> import math                          # importe la totalité du module math
>>> sqrt(5)                              # il faut écrire math.sqrt(5) et pas seulement sqrt(5)
Traceback (most recent call last):
[...]
NameError: name 'sqrt' is not defined
>>> math.sqrt(5)                         # comme ça c'est mieux
------------------
```

Voici comment on pourrait écrire l'expression $\pi \ln(1+\sqrt{2}+\sqrt{3})$ après avoir chargé le module `math` :

```
>>> import math
>>> math.pi * math.log(1 + math.sqrt(2) + math.sqrt(3))
------------------
```

Voici comment écrire la même expression, mais après avoir importé le module `math` par `from math import *` :

```
>>> from math import *
>>> pi * log(1 + sqrt(2) + sqrt(3))
------------------
```

Quand le module `math` a été chargé, on obtient le détail des fonctions qu'il contient en tapant `help(math)`.

Voici l'essentiel des fonctions du module `math`, évaluées en un argument x quelconque :

`e` `pi`	constantes $e = 2.718281828459045$ et $\pi = 3.141592653589793$
`log(x)` `log2(x)` `log10(x)` `log(b,x)`	logarithme népérien, de base 2, de base 10, de base b
`exp(x)` `expm1(x)` `log1p(x)`	calcule e^x, $e^x - 1$ pour x "petit", $\ln(1+x)$ pour x "petit"
`cos(x)` `sin(x)` `tan(x)`	fonctions trigonométriques directes (x en radians)
`degrees(x)` `radians(x)`	conversion radians → degrés, conversion degrés → radians
`acos(x)` `asin(x)` `atan(x)`	fonctions trigonométriques réciproques de x (résultat en radians)
`cosh(x)` `sinh(x)` `tanh(x)`	fonctions hyperboliques directes de x
`acosh(x)` `asinh(x)` `atanh(x)`	fonctions hyperboliques réciproques de x
`floor(x)` `ceil(x)` `trunc(x)`	partie entière $[x]$, entier plafond, tronque vers l'entier direction 0
`sqrt(x)` `fabs(x)` `fmod(x,y)`	calcule $\sqrt{x}$, $\lvert x\rvert$, $x \bmod y$ (résultats de type float)
`factorial(x)` `gamma(x)`	factorielle $x!$ (x dans $\mathbb{N}$), fonction d'Euler $\Gamma(x)$
`fsum(...)`	somme en mode flottant d'un objet itérable (par ex : une liste)

3.7 Le module `cmath`

`Python` met aussi à notre disposition un module `cmath` pour les calculs sur les nombres complexes. La plupart des noms sont identiques à ceux du module `math` mais les fonctions sont ici considérées comme allant de $\mathbb{C}$ dans $\mathbb{C}$ (cette homonymie plaide en la faveur d'une importation par `import ...` plutôt que par `from ... import ...`)

```
>>> import math                    # on charge le package math
>>> math.exp(2 + 3j)               # et on essaie de calculer exp(2+3i), mais ça ne marche pas
<...>
TypeError: can't convert complex to float
>>> import cmath                   # on charge le package cmath
>>> cmath.exp(2 + 3j)              # et ça marche avec la fonction exp du package cth
(-7.315110094901103+1.0427436562359045j)
```

Pour prolonger l'exemple précédent, et revenir sur les questions d'homonymie, si on charge le module `math` (avec la syntaxe `from math import *`, puis le module `cmath` (avec la syntaxe `from cmath import *`), alors les fonctions de `cmath` "recouvrent" celle de `math` (il est facile de deviner ce qui se passe si on inverse les deux chargements de module).

```
>>> from math import *      # importe tout le package math (en mode 'fonctions non préfixées')
>>> from cmath import *     # importe tout cmath (en mode 'fonctions non préfixées')
>>> exp(2 + 3j)             # ici c'est la fonction exp du package cmath qui va s'appliqer
(-7.315110094901103+1.0427436562359045j)
```

En mémoire, un nombre complexe z est représenté par le couple (x, y) formé de ses parties réelle et imaginaire (qui sont accessibles individuellement par `z.real` et `z.imag`). On sait que les fonctions du package `cmath` reprennent (et étendent à $\mathbb{C}$) celles du module `math`. On notera tout de même les trois fonctions suivantes, spécifiques aux nombres complexes.

`phase(z)`	argument de z, exprimé en radians dans l'intervalle $]-\pi, \pi]$
`polar(z)`	forme polaire de z ; équivalent à `(abs(z), phase(z))`
`rect(r,θ)`	passage de la forme polaire $re^{i\theta}$ à la forme cartésienne $x + iy$

Il existe enfin d'autres modules de nature mathématique et numérique. On trouvera l'aide nécessaire à l'adresse suivante (en changeant éventuellement la partie de l'adresse qui est relative au numéro de version de `Python`) :

http://docs.python.org/3.3/library/numeric.html