Une petite référence Python

(mise à jour la plus récente: 3 juin 2013)

Les nouveaux programmes des CPGE scientifiques (rentrée 2013) comportent un enseignement d'informatique *pour tous*, et prévoient notamment l'utilisation du langage `Python`.

Cette introduction se concentre sur les questions prioritaires qui se posent aux débutants en `Python` : quelle est l'idée générale, quels sont les types de données, les structures de contrôle, etc. Elle n'aborde pas des points importants : programmation objet, construction d'interfaces graphiques, utilisation du module `numpy`, etc.

Pour toute suggestion, on peut me contacter à mon adresse électronique.

Jean-Michel Ferrard
Mathématiques, lycée Saint-Louis
44 Boulevard Saint-Michel,
75006, Paris

jean-miche.ferrard@ac-paris.fr

Table des matières

jean-miche.ferrard@ac-paris.fr

Chapitre 1

Premiers pas avec Python

1.1 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 les versions les plus récentes :

- d'une part `Python2` : http://www.python.org/download/releases/2.7.4/ (version au 3 juin 2013)
- d'autre part `Python3` : http://www.python.org/download/releases/3.3.1/ (version au 3 juin 2013)

Il est préférable d'installer à la fois `Python2` et `Python3` (il y a quelques incompatibilités, les plus notables concernant la fonction `print` et la division en nombres entiers).

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

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

1.2 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

1.3 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**100     # ici on calcule 2 élevé à la puissance 100
1267650600228229401496703205376
>>>
```

NB : pour l'exponentiation, on utilisera ** et non ^ (qui désigne en fait le « ou exclusif »).

Dans toute la suite, on omettra le caractère >>> final.

Les capacités d'édition dans la boucle interactive sont limitées. On ne peut pas, par exemple, placer le curseur sur une ligne déjà évaluée pour la modifier ou l'évaluer à nouveau. En revanche, on peut copier le contenu d'une ligne déjà évaluée et « coller » la copie de cette ligne au niveau du prompt.

Astuce : 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.

Astuce bis : 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
12321
>>> _**2                    # élève le résultat précédent au carré
151807041
>>> _**2                    # élève le résultat précédent au carré
23045377697175681
```

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

```
>>> 2**100                  # 2 à la puissance 100, calcul exact
1267650600228229401496703205376
>>> _*2.                    # le carré du résultat précédent, en mode flottant
2.535301200456459e+30
```

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.

1.4 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 = 2013              # on initialise la variable (nommée a) avec la valeur 2013
>>> a*(a+1)*(a+2)         # on calcule l'expression a(a+1)(a+2)
8169176730
```

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).

jean-miche.ferrard@ac-paris.fr

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
False
```

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
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
4444
```

`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
2468
>>> n = "1234"            # l'identificateur n est maintenant lié à une chaîne de caractères
>>> n + n                 # l'opérateur + est ici synonyme de concaténation
'12341234'
>>> 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)
[1,2,3,4,1,2,3,4]
```

1.5 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! '
'bonjour! bonjour! bonjour! bonjour! bonjour! '
>>> a + c                             # la somme de l'entier a et du flottant c est un flottant
8.14
```

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)
(1, 1, 1)
>>> 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)
(3, 8, 3, 8)
```


jean-miche.ferrard@ac-paris.fr

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
[7, 3]
```

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
(2, 3, 4, 5, 1)
```

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
(9, 4)
```

1.6 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
(9, 81)
```

1.7 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
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
5
```

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

jean-miche.ferrard@ac-paris.fr

1.8 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)` ⇒ `5` ; `abs(3+4j)` ⇒ `5.0`
`bin`	chaîne binaire	`bin(1096)` ⇒ `'0b10001001000'` ; `bin(2**10-1)` ⇒ `'0b1111111111'`
`bool`	valeur Vrai/Faux	`bool(1)` ⇒ `True` ; `bool(-3)` ⇒ `True` ; `bool(0)` ⇒ `False`
`chr`	caractère de code donné	`chr(35)` ⇒ `'#'` ; `chr(65)` ⇒ `'A'` ; `chr(97)` ⇒ `'a'`
`complex`	forme un complexe	`complex(1)` ⇒ `(1+0j)` ; `complex(3,4)` ⇒ `(3+4j)`
`divmod`	quotient/reste entiers	`divmod(42,5)` ⇒ `(8,2)` ; `divmod(42.,5)` ⇒ `(8.0, 2.0)`
`eval`	évalue une chaîne	`eval('2*5')` ⇒ `10` ; `eval('2'*5)` ⇒ `22222` ; `eval(2*'5')` ⇒ `55`
`float`	convertit en « flottant »	`float(123)` ⇒ `123.0` ; `float(2**100)` ⇒ `1.2676506002282294e+30`
`help`	aide sur un nom	`help(divmod)` ⇒ *`... divmod(x, y) -> (div, mod) ...`*
`hex`	chaîne héxadécimale	`hex(123456789)` ⇒ `'0x75bcd15'` ; `hex(2**15-1)` ⇒ `'0x7fff'`
`input`	entrée au clavier	`n = int(input('Choisissez un nombre entier: '))`
`int`	convertit en entier	`int(3.7)` ⇒ `3` ; `int(-3.7)` ⇒ `-3` ; `int("110",2)` ⇒ `6`
`len`	longueur d'un objet	`len('abcd')` ⇒ `4` ; `len([2,4,6])` ⇒ `3` ; `len(range(3,8))` ⇒ `5`
`max`	calcul de maximum	`max(3,9,2)` ⇒ `9` ; `max([3,9,2])` ⇒ `9` ; `max('a','B','Z')` ⇒ `'a'`
`min`	calcul de minimum	`min(3,9,2)` ⇒ `2` ; `min([3,9,2])` ⇒ `2` ; `min('a','B','Z')` ⇒ `'B'`
`ord`	code d'un caractère	`ord('#')` ⇒ `35` ; `ord('A')` ⇒ `65` ; `ord('a')` ⇒ `97`
`pow`	calcul de puissance	`pow(2,5)` ⇒ `32` ; `pow(0,0)` ⇒ `1` ; `pow(16,1/2)` ⇒ `4.0`
`print`	affiche à l'écran	`print(2+3)` ⇒ (*`affiche`* `5`) ; `print('2+3')` ⇒ (*`affiche`* `'2+3'`)
`range`	intervalle de valeurs	`range(3,11)` ⇒ intervalle de valeurs 3,4,5,6,7,8,9,10 `range(3,11,2)` ⇒ intervalle de valeurs 3,5,7,9
`repr`	convertit en chaîne	Le résultat est destiné à être réutilisable par l'interpréteur `Python`
`round`	arrondi	`round(2.5)` ⇒ `2` ; `round(2.6)` ⇒ `3` ; `round(1.456789,2)` ⇒ `1.46`
`sorted`	copie triée d'un objet	`sorted([1,5,2,8,3])` ⇒ `[1,2,3,5,8]` `sorted("aebfc")` ⇒ `['a','b','c','e','f']`
`str`	convertit en chaîne	Le résultat est destiné à être affiché de façon naturelle
`sum`	calcul d'une somme	`sum([1,2,3])` ⇒ `6` ; `sum([1,2,3],1000)` ⇒ `1006`
`type`	type (classe) d'un objet	`type(5)` ⇒ `<class 'int'>` ; `type(5.)` ⇒ `<class 'float'>` `type('5')` ⇒ `<class 'str'>` ; `type([1,2])` ⇒ `<class 'list'>`

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.

jean-miche.ferrard@ac-paris.fr

1.9 La fenêtre d'édition dans l'application Idle

Dès qu'on commence à écrire des programmes de quelques lignes, on ne peut se contenter du mode interactif de l'application Idle (le "Python Shell"). Pour sauvegarder son travail, le modifier, l'utiliser (le "lancer"), on utilise la fenêtre d'édition de l'application Idle. Voici comment tout cela peut se passer :

- on ouvre une nouvelle fenêtre d'édition (File/New Window)
- on y écrit une succession de définitions et d'instructions, dans l'ordre où on pourrait les entrer en mode interactif
- on sauvegarde le document avec l'extension .py (sans être obligatoire, cette extension est recommandée)
- on peut tester la syntaxe du module (Run/Check Module)
- on lance l'exécution du module (Run/Run Module), ce qui ramène dans la fenêtre du mode interactif (Python Shell)
- à tout moment, on peut revenir dans la fenêtre d'édition, modifier le module, le lancer à nouveau (l'application Idle nous suggérant de sauvegarder les modifications)

Quand on lance l'exécution du script (du module) :

- s'il ne l'était pas déjà, le dossier où se trouve le module devient le dossier courant de Python
- les instructions qui composent le module sont évaluées dans l'ordre, comme si elles étaient entrées en mode interactif
- les variables qui sont créées au "top-level" du module sont maintenant considérées comme des variables globales (leur contenu reste donc accessible après l'évaluation du module)
- l'évaluation du module (avec Run/Run Module) relance l'interpréteur Python, effaçant donc les définitions qui préexistaient à cette évaluation
- en particulier, on peut ouvrir plusieurs modules sauvegardés préalablement, mais avec cette méthode on ne pourra exécuter qu'un seul d'entre eux (si on veut utiliser conjointement plusieurs modules sauvegardés dans des fichiers différents, on utilisera l'instruction import)

Voici par exemple à quoi pourrait ressembler un module contenant la définition d'une fonction de tri d'une liste (par la méthode d'insertion), et permettant de tester cette fonction sur un exemple simple (les explications concernant tel ou tel aspect de la programmation seront abordées plus tard) :

```
def tri_insertion(L, steps=True):                # tri par insertion d'une liste L
    '''trie L par insertions successives         # docstring de la fonction
       steps=False n'affiche pas les étapes'''
    for i in range(1,len(L)):                    # on va insérer l'élément n°i
        j = i; v = L[i]                          #   v = valeur à insérer, j position d'insertion
        while j > 0 and L[j-1] > v:              #   tant que l'élt à gauche de L[j] est > v
            L[j] = L[j-1]; j -= 1                #   décale cet élt vers la droite et actualise j
        L[j] = v                                 # insère la valeur v en position j
        if steps:                                # si steps=True, affiche étape
            print("Après étape n°{}".format(i),L)

from random import sample                        # importe la fonction sample du module sample

L = sample(range(10,100),10)                     # 10 nombres différents de deux chiffres

print("Liste à trier:",L,"\n")                   # affiche la liste à trier

input("(Appuyer sur Entrée pour trier)\n")

tri_insertion(L)                                 # trie la liste L (sur place)

print("Liste après le tri:",L)                   # affiche la liste triée
```

Quelques remarques sur le module précédent :

- les lignes vides ne sont pas significatives.
- les commentaires, facultatifs, ne sont là que pour favoriser la relecture. Ils sont en revanche utiles dans la phase de mise au point pour masquer/démasquer certaines lignes de codes.
- on remarquera les caractères \n, destinés à ajouter des sauts de ligne dans les instructions print.
- la fonction tri_insertion est munie d'un argument facultatif steps, avec la valeur par défaut True, permettant de spécifier si on souhaite afficher les étapes de la procédure de tri.

jean-miche.ferrard@ac-paris.fr

Après avoir sauvegardé ce module dans le fichier nommé `triparinsertion.py`, on lance son exécution depuis la fenêtre de l'éditeur (choix `Run/Run Module`). Voici ce qu'on peut voir alors dans la fenêtre `Python Shell` (on est donc revenu en mode interactif). La ligne RESTART au début indique bien que l'interpréteur `Python` a été réinitialisé.

```
>> ================================ RESTART ================================
>>>
Liste à trier: [85, 65, 26, 42, 32, 50, 40, 83, 19, 60]

(Appuyer sur Entrée pour trier)
après étape n°1 [65, 85, 26, 42, 32, 50, 40, 83, 19, 60]
après étape n°2 [26, 65, 85, 42, 32, 50, 40, 83, 19, 60]
après étape n°3 [26, 42, 65, 85, 32, 50, 40, 83, 19, 60]
après étape n°4 [26, 32, 42, 65, 85, 50, 40, 83, 19, 60]
après étape n°5 [26, 32, 42, 50, 65, 85, 40, 83, 19, 60]
après étape n°6 [26, 32, 40, 42, 50, 65, 85, 83, 19, 60]
après étape n°7 [26, 32, 40, 42, 50, 65, 83, 85, 19, 60]
après étape n°8 [19, 26, 32, 40, 42, 50, 65, 83, 85, 60]
après étape n°9 [19, 26, 32, 40, 42, 50, 60, 65, 83, 85]

Liste après le tri: [19, 26, 32, 40, 42, 50, 60, 65, 83, 85]
>>>
```

Après l'exécution de ce module, toutes les définitions qu'il contient restent disponibles, notamment la fonction `tri_insertion`, et la fonction `sample` importée depuis le module `random` (qui fait partie de la libraire standard).

Nous avons muni la fonction `tri_insertion` d'une *docstring*, que nous pouvons afficher avec `help` (nous voyons à cette occasion que, dans le mode interactif, nous sommes dans un module prédéfini dont le nom est `__main__`) :

```
Help on function tri_insertion in module __main__:
tri_insertion(L, steps=True)
    trie L par insertions successives
    steps=False n'affiche pas les étapes
```

On peut alors utiliser la fonction `sample` pour créer une nouvelle liste, que nous trions avec la fonction `tri_insert` (mais cette fois-ci en spécifiant l'argument `steps=False` pour ne pas voir s'afficher les étapes du tri).

```
>>> L = sample(range(100,1000),15); L          # 15 entiers différents à trois chiffres
[620, 230, 185, 842, 268, 574, 110, 515, 233, 331, 760, 231, 707, 663, 181]
>>> tri_insertion(L,steps=False)               # trie L par insertion, sans afficher les étapes
>>> L                                          # la liste L a été triée sur place
[110, 181, 185, 230, 231, 233, 268, 331, 515, 574, 620, 663, 707, 760, 842]
```

1.10 Importer un module personnel en mode interactif

Nous avons besoin ici de quelques explications techniques, mais nous nous limiterons à l'essentiel.

On a vu qu'il est possible, dans l'application `Idle`, d'importer des modules intégrés (par exemple le module `math`), ou seulement certaines fonctions d'un module (par exemple la fonction `sample` du module `random`), soit en mode interactif, soit dans un module personnel (comme le module `triparinsertion.py` de la section précédente).

Quand on importe un module, `Python` doit pouvoir le trouver. Pour celà, il cherche dans le "répertoire courant", et sinon il utilise une liste (appelée `Search Path`) indiquant les dossiers dans lesquels il doit effectuer cette recherche : `Python` parcourt ces dossiers dans l'ordre où ils figurent dans cette liste, et il charge le module spécifié dès qu'il le trouve.

Bien sûr, c'est sans problème pour les modules intégrés à `Python`, car leurs dossiers sont automatiquement ajoutés au `Search Path`. Pour connaître le contenu du `Search Path`, on utilise l'attribut `path` du module `sys` (nous avons ici considérablement raccourci le résultat). Par ailleurs la fonction `getcwd` du module `os` donne le répertoire "courant".

Voila ce que nous obtenons après avoir redémarré l'interpréteur `Python` par Ctrl-F6 dans l'application `Idle` (le système utilisé est ici un Mac tournant sous *Mountain Lion* : les utilisateurs d'autres systèmes d'exploitation traduiront) :

```
>>> import sys, os                          # importe les modules sys et os
>>> sys.path                                # demande le Search Path
['', '/Users/jmf/Documents', '/Library/Frameworks/Python.framework/Versions/<...>]
>>> os.getcwd()                             # demande le répertoire courant
'/Users/jmf/Documents'
```


jean-miche.ferrard@ac-paris.fr

On remarque que la première chaîne de cette liste est vide, ce qui est caractéristique du fait que l'interpréteur a été invoqué en mode interactif (et dans ce cas, cette chaîne vide indique simplement le "répertoire courant").

Nous allons reprendre l'exemple du module écrit dans la section précédente, que nous supposons sauvegardé sous le nom `triparinsertion.py` et placé dans le sous-dossier `python-modulesperso` du dossier `Documents`.

Quand on lance un module depuis le menu `Run/Run Module` de l'application `Idle`, le dossier où se trouve ce module est ajouté en tête du `Search Path`, et il devient le nouveau "répertoire courant". Il est donc le premier dossier a être exploré par `Python` quand il est à la recherche du code source d'un module.

Après avoir ouvert `triparinsertion.py` dans l'éditeur de `Idle` et l'avoir lancé par `Run/Run Module`, on voit effectivement que le dossier de ce module apparaît en tête du `Search Path`, et qu'il est maintenant le "répertoire courant" :

```
>>> import sys, os                          # importe les modules sys et os
>>> sys.path
['/Users/jmf/Documents/python-modulesperso', '/Users/jmf/Documents', '/Library/Frameworks/<...>']
>>> os.getcwd()
'/Users/jmf/Documents/python-modulesperso'
```

Si on peut importer les modules de la librairie standard, il doit être possible d'importer ceux que nous avons écrits. Mais pour cela, ces modules doivent se trouver dans le "dossier courant", ou dans un de ceux listés dans le `Search Path`.

Essayons par exemple d'importer `triparinsertion.py`, en mode interactif et après avoir redémarré l'interpréteur `Python`.

```
>>> ================================ RESTART ================================
>>> import triparinsertion                          # pas d'extension py dans l'import de modules!
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
    import triparinsertion
ImportError: No module named 'triparinsertion'
```

Le message d'erreur est dû au fait que nous venons de redémarrer `Python` en mode interactif, donc que le `Search Path` commence à nouveau par une chaîne vide (en tout cas, il ne contient plus le dossier exact où se trouve notre module).

```
>>> import sys, os                          # importe les modules sys et os
>>> sys.path                                # demande le Search Path
['', '/Users/jmf/Documents', '/Library/Frameworks/Python.framework/Versions/<...>]
>>> os.getcwd()                             # demande le répertoire courant
'/Users/jmf/Documents'
```

Il existe des solutions pour modifier le `Search Path` de `Python`, d'une façon temporaire (jusqu'au prochain démarrage de l'interpréteur) ou permanente (notamment en modifiant la variable d'environnement `PYTHONPATH` ou en utilisant des fichiers avec extension `pth`, mais ça dépasse le cadre de cette introduction à `Python`.

Un solution économique et radicale est de placer tous ses modules personnels dans un dossier qui figure par défaut dans le `Search Path`, par exemple ici dans `'/Users/jmf/Documents'`.

Une autre solution consiste à changer simplement le dossier courant, avec la fonction `chdir` du module `os` (le changement de dossier s'effectue de façon absolue ou relative par rapport au dossier courant) :

```
>>> os.chdir('python-modulesperso')       # passe dans le sous-dossier python-modulesperso
```

Le nouveau dossier de travail contient le module `triparinsertion.py`, que nous pouvons maintenant importer :

```
>>> import triparinsertion                 # rappel: ne pas mettre l'extension py ici
Liste à trier: [20, 30, 32, 53, 95, 36, 11, 44, 65, 92]

(Appuyer sur Entrée pour trier)
après étape n°1 [20, 30, 32, 53, 95, 36, 11, 44, 65, 92]
après étape n°2 [20, 30, 32, 53, 95, 36, 11, 44, 65, 92]
après étape n°3 [20, 30, 32, 53, 95, 36, 11, 44, 65, 92]
après étape n°4 [20, 30, 32, 53, 95, 36, 11, 44, 65, 92]
après étape n°5 [20, 30, 32, 36, 53, 95, 11, 44, 65, 92]
après étape n°6 [11, 20, 30, 32, 36, 53, 95, 44, 65, 92]
après étape n°7 [11, 20, 30, 32, 36, 44, 53, 95, 65, 92]
après étape n°8 [11, 20, 30, 32, 36, 44, 53, 65, 95, 92]
après étape n°9 [11, 20, 30, 32, 36, 44, 53, 65, 92, 95]

Liste après le tri: [11, 20, 30, 32, 36, 44, 53, 65, 92, 95]
```


jean-miche.ferrard@ac-paris.fr

L'exemple précédent appelle plusieurs remarques :

– Le contenu du module `triparinsertion.py` a été exécuté intégralement, comme s'il avait été ouvert puis lancé par `Run/Run Module` depuis la fenêtre d'édition de `Idle` : ce n'est pas forcément le but recherché, notamment si on souhaite seulement importer la définition de la fonction `tri_insertion` (nous allons revenir sur ce point).

– une deuxième tentative d'importation ne donne cette fois-ci aucun écho à l'écran :

```
>>> import triparinsertion
>>>
```

c'est normal dans la mesure où `Python` sait déjà que le module a été importé, et qu'il n'y a pas lieu de recommencer.

– si nous demandons le contenu de l'espace de nom global, nous avons la confirmation que le module `triparinsertion.py` a bien été chargé en mémoire (on y trouve aussi la trace des modules importés `os` et `sys`) ; on ne voit pas la fonction `tri_insertion` ici, car elle est en quelque sorte enfermée dans le module `triparinsertion.py` (à suivre) :

```
>>> globals()
{'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 'triparinsertion': <module 'triparinsertion' from './triparinsertion.py'>,
 '__package__': None, '__builtins__': <module 'builtins'>, '__name__': '__main__',
 '__doc__': None, 'sys': <module 'sys' (built-in)>,
 'os': <module 'os' from '/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/os.py'>}
```

– une autre façon de confirmer la présence du module `triparinsertion` en mémoire est d'utiliser la fonction `dir` ; à part quelques noms réservés (encadrés par des `__`) on y retrouve la liste L utilisée pour tester le module, la fonction `sample` utilisée pour construire L (et importée du package `random`), et notre fonction `tri_insertion`.

```
>>> dir(triparinsertion)
['L', '__builtins__', '__cached__', '__doc__', '__file__', '__initializing__', '__loader__',
 '__name__', '__package__', 'sample', 'tri_insertion']
```

– si on essaie d'utiliser la fonction `tri_insertion` directement, on est déçu !

```
>>> tri_insertion([8,2,7,6,3,5,1,4])
<...>
NameError: name 'tri_insertion' is not defined
```

l'explication est la suivante : l'importation du module `triparinsertion` s'accompagne de la création d'un "espace de noms", dans lequel on trouve les noms des objets définis dans le module ; ces noms sont alors vus commes des variables locales au module et sont ainsi protégés contre tout risque d'homonymie.

– pour utiliser la fonction `tri_insertion`, il faut donc "qualifier" son nom par celui du module dont elle est issue :

```
>>> triparinsertion.tri_insertion([8,2,7,6,3,5,1,4])
après étape n°1 [2, 8, 7, 6, 3, 5, 1, 4]
après étape n°2 [2, 7, 8, 6, 3, 5, 1, 4]
après étape n°3 [2, 6, 7, 8, 3, 5, 1, 4]
après étape n°4 [2, 3, 6, 7, 8, 5, 1, 4]
après étape n°5 [2, 3, 5, 6, 7, 8, 1, 4]
après étape n°6 [1, 2, 3, 5, 6, 7, 8, 4]
après étape n°7 [1, 2, 3, 4, 5, 6, 7, 8]
```

– évidemment, c'est un peu contraignant... on peut raccourcir tout ça en utilisant un *alias* lors de l'importation :

```
>>> import triparinsertion as tpi       # importe triparinsertion.py en lui donnant l'alias tpi
>>> L = [8,2,7,6,3,5,1,4]               # crée une liste de test
>>> tpi.tri_insertion(L,steps=False)    # appelle tri_insertion avec son nom qualifié
>>> L                                   # la liste a été triée sur place
[1, 2, 3, 4, 5, 6, 7, 8]
```

– ce qui est plus étonnant est que la fonction `sample`, qui fait partie du module intégré `random`, et qui a été importée par notre module `triparinsertion` (alias `tpi`) appartient à celui-ci

```
>>> sample(range(10,100),10)            # la fonction sample est inconnue dans l'espace global
<...>
NameError: name 'sample' is not defined
>>> tpi.sample(range(10,100),10)        # en revanche, elle est dans notre module
[46, 51, 20, 65, 80, 95, 87, 34, 96, 21]
```


jean-miche.ferrard@ac-paris.fr

– il reste une dernière possibilité, qui est d'utiliser la syntaxe d'importation `from triparinsertion import *` ; les définitions du module entrent alors dans l'espace de nom global, et peuvent être évoquées par leur nom court.

```
>>> from triparinsertion import *              # rend globales toutes les définitions du module
>>> L = sample(range(100,1000),15); L          # sample pour une liste de 15 entiers à 3 chiffres
[952, 900, 713, 356, 583, 895, 748, 760, 262, 182, 143, 482, 382, 644, 428]
>>> tri_insertion(L,steps=False)               # appelle tri_insertion (nom court)
>>> L                                          # renvoie la liste L triée sur place (pas d'étapes)
[143, 182, 262, 356, 382, 428, 482, 583, 644, 713, 748, 760, 895, 900, 952]
>>>
```

1.11 Importation simultanée de plusieurs modules personnels

Nous allons reprendre l'exemple du module `triparinsertion`, et nous allons ajouter un module `triparselection`.

Avant cela, il faut savoir que `Python` utilise en permanence un certain nombre d'identificateurs à son usage personnel (mais qu'il est tout à fait possible de consulter et d'utiliser).

Il en est ainsi du nom `__name__`, qui désigne le nom du module en cours d'exécution (au "top-level").

En mode interactif, le nom de ce module est `__main__` :

```
>>> __name__                    # demande le nom du module en cours d'exécution
'__main__'                      # c'est '__main__' en mode interactif
```

Nous avons vu que tout le code incorporé à un module est exécuté lors de la première importation, exactement comme si ce module était lancé depuis la fenêtre d'édition (menu `Run/Run Module`). Il y a cependant une grosse différence :

– dans le premier cas (avec `import`), le processus maître (désigné par `__name__`) est le nom du module lui-même.

– dans le second cas (exécution du module par `Run/Run Module`) le processus maître est `'__main__'`.

– il suffit donc d'utiliser un test `if __name__ == "__main__"` dans le corps du module lui-même pour isoler une portion du code qui ne doit être exécutée que si le module est lancé dans `Idle` par `Run/Run Module` (et pas si le module est simplement importé).

Pour illustrer ces notions, on a modifié le module `triparinsertion.py`, et on a ajouté un module `triparselection.py`

On suppose que ces modules sont sauvegardés dans un même dossier (peu importe lequel).

On a modifié le fonctionnement par défaut des deux procédures de tri (pas d'affichage des étapes du calcul). En revanche, dans le bloc qui suit le test `if __name__ == "__main__"`, c'est-à-dire dans le bloc qui n'est exécuté que si le module est lancé par `Run/Run Module` dans l'application `Idle`, on a écrit quelques lignes de test de la procédure de tri, avec cette fois une indication `steps=True` pour que les étapes soient affichées.

```
#-----------------------------------#
#---- module triparinsertion.py ----#
#-----------------------------------#

def tri_insertion(L,steps=False):
    '''trie L par insertions
       steps=True affiche les étapes'''
    for i in range(1,len(L)):
        j = i; v = L[i]
        while j > 0 and L[j-1] > v:
            L[j] = L[j-1]; j -= 1
        L[j] = v
        if steps:
            print("étape n°{}".format(i),L)


if __name__ == "__main__":
    from random import sample
    L = sample(range(10,100),10)
    print("Liste à trier:",L,)
    input("\n(Appuyer sur Entrée)")
    tri_insertion(L,steps=True)
    print("\nListe après tri_insertion:",L)
```

```
#-----------------------------------#
#---- module triparselection.py ----#
#-----------------------------------#

def tri_selection(L,steps=False):
    '''trie L par sélections
       steps=True affiche les étapes'''
    n = len(L)
    for i in range(n-1):
        p = i
        for j in range(i+1,n):
            if L[j] < L[p]: p = j
        L[i], L[p] = L[p], L[i]
        if steps:
            print("étape n°{}".format(i),L)

if __name__ == "__main__":
    from random import sample
    L = sample(range(10,100),10)
    print("Liste à trier:",L,)
    input("\n(Appuyer sur Entrée)")
    tri_selection(L,steps=True)
    print("\nListe après tri_selection:",L)
```


jean-miche.ferrard@ac-paris.fr

On ouvre `triparselection.py` dans l'éditeur de `Idle`, et on le lance par `Run/Run Module`.

Comme on le voit ci-dessous, la partie du code située après le test `if __name__ == "__main__"` est exécutée (on y voit une liste de 10 entiers à deux chiffres, triée par sélections successives de l'élément minimum).

```
>>> ============================== RESTART ==============================
>>>
Liste à trier: [40, 68, 69, 37, 54, 16, 70, 98, 52, 83]

(Appuyer sur Entrée)
étape n°0 [16, 68, 69, 37, 54, 40, 70, 98, 52, 83]
étape n°1 [16, 37, 69, 68, 54, 40, 70, 98, 52, 83]
étape n°2 [16, 37, 40, 68, 54, 69, 70, 98, 52, 83]
étape n°3 [16, 37, 40, 52, 54, 69, 70, 98, 68, 83]
étape n°4 [16, 37, 40, 52, 54, 69, 70, 98, 68, 83]
étape n°5 [16, 37, 40, 52, 54, 68, 70, 98, 69, 83]
étape n°6 [16, 37, 40, 52, 54, 68, 69, 98, 70, 83]
étape n°7 [16, 37, 40, 52, 54, 68, 69, 70, 98, 83]
étape n°8 [16, 37, 40, 52, 54, 68, 69, 70, 83, 98]

Liste après tri_selection: [16, 37, 40, 52, 54, 68, 69, 70, 83, 98]
>>>
```

Puisque nous avons lancé le module `triparselection.py` par `Run/Run Module` (donc comme si son contenu avait été exécuté, ligne par ligne, en mode interactif), nous avons accès à la fonction `sample` du module `random` (mais il s'agit plutôt ici d'un "effet de bord") et bien sûr de la fonction `tri_selection`.

Nous utilisons ici cette fonction sur une liste de 15 nombres à trois chiffres (et par défaut, l'argument `steps` vaut `False` : il n'y a donc pas affichage des résultats intermédiaires) :

```
>>> L = sample(range(100,1000),15); L
[236, 739, 876, 121, 225, 813, 604, 411, 732, 870, 877, 379, 213, 188, 364]
>>> tri_selection(L); L
[121, 188, 213, 225, 236, 364, 379, 411, 604, 732, 739, 813, 870, 876, 877]
```

Nous avons dit que les modules `triparselection.py` et `triparinsertion.py` se trouvaient dans le même dossier, qui est maintenant (du fait de ce qui précède) le dossier en cours.

Nous pouvons donc *importer* le module `triparinsertion.py` (attention à ne pas utiliser l'extension `.py` avec `import` !!).

```
>>> import triparinsertion              # importation avec les noms qualifiés
>>>
```

La première remarque qui vient à l'esprit est que "rien ne s'est passé" !

En fait, le module a bien été chargé, mais le code situé après le test `if __name__ == "__main__"` n'a pas été exécuté. En effet, durant l'importation, c'est le module importé qui "prend la main" (sans mauvais jeu de mots, car pendant ce temps là `__name__` ce n'est plus `"__main__"` mais `"triparinsertion"`).

La meilleure preuve est que nous pouvons maintenant appeler l'aide sur l'objet `triparinsertion` :

```
>>> help(triparinsertion)

Help on module triparinsertion:

NAME
    triparinsertion

FUNCTIONS
    tri_insertion(L, steps=False)
        trie L par insertions
        steps=True affiche les étapes

FILE
    /Users/jmf/Documents/python-modulesperso/triparinsertion.py
```

L'autre preuve, bien sûr, est que nous pouvons utiliser la fonction `tri_insertion` :

```
>>> L = sample(range(100,1000),15); L        # sample vient du ''Run Module'' sur triparselection
[111, 478, 257, 418, 416, 824, 262, 288, 571, 469, 947, 283, 396, 275, 835]
>>> triparinsertion.tri_insertion(L); L      # attention, utilisation d'un nom qualifié !
[111, 257, 262, 275, 283, 288, 396, 416, 418, 469, 478, 571, 824, 835, 947]
```


jean-miche.ferrard@ac-paris.fr

On va maintenant faire un petit bilan.

Si on veut importer conjointement `triparinsertion.py` et `triparselection.py`, le mieux est d'écrire un module (disons `test_tris.py`, placé dans le même répertoire qu'eux) qui importera nos deux modules.

Voici en quoi pourrait consister notre module `test_tris.py` :

```
#-----------------------------#
#---- module test_tris.py ----#
#-----------------------------#

from triparinsertion import *

from triparselection import *

if __name__ == "__main__":
    print("Le module test_tris.py est chargé")
    print("Il donne accès aux définitions suivantes")
    print("1: tri_insertion(list,steps=False)")
    print("2: tri_selection(list,steps=False)")
```

On ouvre `test_tris.py` dans l'éditeur de `Idle`, on fait `Run/Run Module`, et on obtient :

```
>>> ================================ RESTART ================================
>>>
Le module test_tris.py est chargé
Il donne accès aux définitions suivantes
1: tri_insertion(list,steps=False)
2: tri_selection(list,steps=False)
>>>
```

Nous avons ici utilisé des importations avec noms courts (`from ... import *`, s'il n'y a pas de risque d'homonymie c'est le plus simple), et nous pouvons utiliser directement les identificateurs *non qualifiés* `tri_insertion` et `tri_selection`.

Bien entendu, au risque de nous répéter, nous avons accès aux *docstrings* de ces fonctions :

```
>>> help(tri_selection)
Help on function tri_selection in module triparselection

tri_selection(L, steps=False)
    trie L par sélections
    steps=True affiche les étapes
```

Pour tester tout ça, on importe `sample` du module `random` (car elle n'a pas été importée par la méthode précédente !)

```
>>> from random import sample
>>> L = sample(range(10,100),10); L2 = L[:]; L;      # L2 = copie de sauvegarde de L
[84, 68, 44, 79, 77, 67, 63, 22, 61, 21]
>>> tri_selection(L,steps=True)                      # on va trier L sur place, avec les étapes
étape n°0 [21, 68, 44, 79, 77, 67, 63, 22, 61, 84]
étape n°1 [21, 22, 44, 79, 77, 67, 63, 68, 61, 84]
étape n°2 [21, 22, 44, 79, 77, 67, 63, 68, 61, 84]
étape n°3 [21, 22, 44, 61, 77, 67, 63, 68, 79, 84]
étape n°4 [21, 22, 44, 61, 63, 67, 77, 68, 79, 84]
étape n°5 [21, 22, 44, 61, 63, 67, 77, 68, 79, 84]
étape n°6 [21, 22, 44, 61, 63, 67, 68, 77, 79, 84]
étape n°7 [21, 22, 44, 61, 63, 67, 68, 77, 79, 84]
étape n°8 [21, 22, 44, 61, 63, 67, 68, 77, 79, 84]
>>> tri_insertion(L2,steps=True)                    # on va trier L2 sur place, avec les étapes
étape n°1 [68, 84, 44, 79, 77, 67, 63, 22, 61, 21]
étape n°2 [44, 68, 84, 79, 77, 67, 63, 22, 61, 21]
étape n°3 [44, 68, 79, 84, 77, 67, 63, 22, 61, 21]
étape n°4 [44, 68, 77, 79, 84, 67, 63, 22, 61, 21]
étape n°5 [44, 67, 68, 77, 79, 84, 63, 22, 61, 21]
étape n°6 [44, 63, 67, 68, 77, 79, 84, 22, 61, 21]
étape n°7 [22, 44, 63, 67, 68, 77, 79, 84, 61, 21]
étape n°8 [22, 44, 61, 63, 67, 68, 77, 79, 84, 21]
étape n°9 [21, 22, 44, 61, 63, 67, 68, 77, 79, 84]
```


jean-miche.ferrard@ac-paris.fr

Chapitre 2

Types numériques, comparaisons, intervalles

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.

2.1 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
<class 'bool'>
>>> type(1 + 2 + 3)       # l'expression 1+2+3 appartient à la classe 'int' des entiers
<class 'int'>
>>> type(1 + 2 + 3.)      # à cause du point décimal, l'expression 1+2+3. est un 'float'
<class 'float'>
>>> type(2 + 3j)          # un nombre complexe. Attention, noter j ou J et pas i ou I
<class 'complex'>         # par ailleurs on notera 1j, ou 1.j, plutôt que j seul.
>>> type(2 ** 100)        # la valeur 2^100 est de type 'int'
<class 'int'>
```

Le dernier exemple est un peu particulier. Jusqu'à la version 2 de `Python`, on disposait de deux classes d'entiers : la classe 'int' (entiers compris dans l'intervalle $J = [-2^{n-1}, 2^{n-1}[$, avec $n = 32$ ou $n = 64$ suivant les systèmes) et la classe 'long' (entiers longs) pour les résultats entiers qui sortent de J (le passage de la classe 'int' à la classe 'long' s'effectuant automatiquement).

À partir de `Python3`, les classes 'int' et 'long' ont été fusionnées en une seule classe '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
7
>>> 2 + 1j                # mais ici c'est le nombre complexe 2 + i
(2+1j)
```

2.2 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' (il serait plus exact de parler d'injections canoniques de 'int' vers 'float', et de 'float' vers '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'
118
>>> 2013 / 17         # le résultat dans la division dans ℝ: le résultat est un 'float'
118.41176470588235
>>> 2013 // 17.       # division entière aussi, mais 17. est un 'float', donc résultat 'float'
118.0
>>> (1+2j)/(3-5j)     # division de deux complexes (ici // donnerait une erreur de type)
(-0.20588235294117646+0.3235294117647059j)
```

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)
(2, 6)
>>> divmod(-20,7)     # −20 = 7q + r, avec q = −3 et r = 1      (et on a bien 0 ≤  r < 7)
(-3, 1)
>>> divmod(20,-7)     # 20 = −7q + r, avec q = −3 et r = −1     (et on a bien −7 <  r ≤  0)
(-3, -1)
>>> divmod(-20,-7)    # −20 = −7q + r, avec q = 2 et r = −6     (et on a bien −7 <  r ≤  0)
(2, -6)
>>> divmod(20.,7)     # ici les résultats sont des floats car le dividende est un float
(2.0, 6.0)
```

2.3 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`	

Voici trois exemples d'utilisation de l'opérateur avec assignation += sur des valeurs de différents types :

```
>>> x = 9; x += 1; x                # x contient l'entier 9, et lui ajoute 1
10
>>> x = "abc"; x += "de"; x         # x contient la chaîne 'abc', et lui ajoute la chaîne 'de'
'abcde'
>>> x = [1,2,3]; x += [4,5]; x      # x contient la liste [1,2,3], et lui ajoute la liste [4,5]
[1, 2, 3, 4, 5]
```

2.4 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):
  File "<pyshell#0>", line 1, in <module>
    sqrt(5)
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):
  File "<pyshell#2>", line 1, in <module>
    sqrt(5)
NameError: name 'sqrt' is not defined
>>> math.sqrt(5)                         # comme ça c'est mieux
2.23606797749979
```

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

```
>>> import math
>>> math.pi * math.log(1 + math.sqrt(2) + math.sqrt(3))
4.467997504231714
```


jean-miche.ferrard@ac-paris.fr

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))
4.467997504231714
```

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)

2.5 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

jean-miche.ferrard@ac-paris.fr

2.6 Arithmétique des entiers

`Python` permet de manipuler des entiers en décimal (base 10), binaire (base 2), octal (base 8) ou hexadécimal (base 16).

Pour exprimer un entier en base 2, (resp. 8, resp. 16), on le fait précéder de `0b` (resp. `0o`, resp. `0x`).

Par exemple : `0xD5B`, `0o6533`, et `0b110101011011` représentent tous les trois la valeur 3419.

Voici les trois opérations binaires ("ou" inclusif, "ou" exclusif, "et") spécifiques au type `int`.

Les résultats sont renvoyés au format décimal, et on peut les convertir au format binaire, octal ou hexadécimal (mais sous forme de chaîne de caractères cette fois) avec les fonctions `bin`, `oct` et `hex`. On réalise pourquoi l'opérateur `^` ne peut pas être utilisé pour désigner l'exponentiation ! Si veut calculer a^n, on écrira donc `a**n` ou `pow(a,n)`.

Fonction	Signification	Exemples

`x | y`	ou logique (or)	`bin(0b11001 | 0b01101)` renvoie `'0b11101'`
		`0b11001 | 0b01101` renvoie `29`
		`hex(0x19 | 0xd)` renvoie `'0x1d'`
		`0x19 | 0xd` renvoie `29` ; `25 | 13` renvoie `29`

`x ^ y`	ou exclusif (xor)	`bin(0b11001 ^ 0b01101)` renvoie `'0b10100'`
		`0b11001 ^ 0b01101` renvoie `20`
		`hex(0x19 ^ 0xd)` renvoie `'0x14'`
		`0x19 ^ 0xd` renvoie `20` ; `25 ^ 13` renvoie `20`

`x & y`	et logique (and)	`bin(0b11001 & 0b01101)` renvoie `'0b1001'`
		`0b11001 & 0b01101` renvoie `9`
		`hex(0x19 & 0xd)` renvoie `'0x9'`
		`0x19 & 0xd` renvoie `9` ; `25 & 13` renvoie `9`

Voici les deux opérations de décalage de bits (multiplication ou division par des puissances de 2).

Fonction	Signification	Exemples
`x << n`	décalage à gauche de n bits multiplication de x par 2^n	`13 << 5` renvoie `416` ; `bin(13 << 5)` renvoie `'0b110100000'` `0b1101 << 5` renvoie `416` ; `bin(0b1101 << 5)` renvoie `'0b110100000'`
`x >> n`	décalage à droite de n bits division entière de x par 2^n	`417 >> 5` renvoie `13` ; `bin(417 >> 5)` renvoie `'0b1101'` `0b110100001 >> 5` renvoie `13` ; `bin(0b110100001 >> 5)` renvoie `'0b1101'`
`~ x`	complément à 2 $x \to -x-1 \to x$	`bin(~0b111000)` renvoie `'-0b111001'` `bin(~0b100000)` renvoie `'-0b100001'`

2.7 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)
(<class 'bool'>, <class 'bool'>)
```

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)

jean-miche.ferrard@ac-paris.fr

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
(True, False, True, True, True)
```

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
True
```

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'
True
>>> 'ABC' < 'ABc' < 'AbC' < 'Abc' < 'aBC' < 'aBc' < 'abC' < 'abc'      # huit chaînes différentes
True
```

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
False
>>> 1/2 < 1/3 or 1/4 < 1/3 or 1/2 < 1/0
True
```

Si x et y sont des valeurs de types différents (l'une d'elle au moins étant non numérique) les comparaisons $x == y$ et $x \,!= y$ sont possibles (et renvoient bien sûr respectivement `False` et `True`). En revanche 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
False
>>> 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
False
>>> x + (y < z)        # ici, y < z vaut True, c'est-à-dire 1, et donc on calcule x + 1
2
```


jean-miche.ferrard@ac-paris.fr

2.8 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).

Le mieux est de prendre des exemples, en commençant par du très simple :

```
>>> 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 2013
4343755504
>>> 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
4343774064
>>> 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
False
```

Continuons sur la lancée de 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
(4343755504, 4343755504)
>>> (x is y, x == y)      # il y a égalité physique, donc égalité structurelle
(True, True)
```

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
(2013, 2014)
>>> (id(x), id(y))       # x pointe vers la même adresse, mais pas y
(4343755504, 4343773200)
>>> (x == y, x is y)     # il n'y a plus égalité structurelle, et encore moins égalité physique
(False, False)
```

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
(False, True)
```

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
([1, 2, 3], [1, 2, 3, 1, 2, 3])
```


jean-miche.ferrard@ac-paris.fr

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
(4361208752, 4361208752)
>>> 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 !!
([999, 2, 3], [999, 2, 3])
>>> (id(x), id(y))  # en fait les variables x et y pointent toujours sur la même adresse
(4361208752, 4361208752)
>>> x is y          # il s'agit donc d'égalité physique
True
```

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
([1, 2, 3], [1, 2, 3])
>>> (id(x),id(y))           # mais les listes ne sont pas à la même adresse
(4298413336, 4361210768)
>>> (x is y, x == y)        # il y a égalité structurelle, mais pas égalité physique
(False, True)
>>> y[0] = 999              # on modifie le premier élément de la liste y
>>> (x, y)                  # mais ça ne se répercute plus sur la liste x
([1, 2, 3], [999, 2, 3])
```

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]]
```


jean-miche.ferrard@ac-paris.fr

Chapitre 3

Initiation à la programmation Python

On a jusqu'ici utilisé Python en mode interactif, à la manière d'une calculatrice. Les instructions (ou *commandes*) sont ainsi, l'une après l'autre, entrées au clavier, interprétées et suivies d'un résultat souvent réutilisé par la suite.

Si elle en vaut la peine, une séquence d'instructions peut être sauvegardée dans un fichier texte (on parle de *script* Python).

On peut dès lors ouvrir ce script et l'exécuter, de façon automatisée, comme si les instructions qu'il contient étaient à nouveau entrées au clavier (chronologiquement de la première à la dernière ligne).

Dérouler les mêmes instructions dans le même ordre doit bien sûr posséder un minimum d'intérêt. Pour apporter un peu de profondeur et/ou de fantaisie à tout ça, on peut, à l'intérieur du script lui-même :

- répéter un certain nombre de fois un bloc d'instructions
- n'exécuter certaines instructions que si (ou que tant qu') une condition est vraie
- appeler un autre script (et pourquoi pas le même ? à suivre...)
- orienter le déroulement du script suivant certaines informations fournies par l'utilisateur (informations qui pourraient par exemple être passées au démarrage du script : on parle alors des paramètres d'appel).

Évidemment, tout cela est possible en Python. On se contentera ici d'une première approche modeste (d'autant que de nombreuses caractéristiques du langage n'ont pas encore été abordées).

3.1 Entrée au clavier (input) et affichage à l'écran (print)

Les interactions d'un script avec l'utilisateur se feront essentiellement par l'attente de données entrées au clavier, et par l'affichage de résultats ou de messages à l'écran.

L'expression `input(message)` affiche *message* à l'écran (sans passer à la ligne), attend que l'utilisateur valide par "Entrée" une réponse au clavier, et renvoie cette réponse sous forme de chaîne de caractères.

```
>>> n = input('entrez un entier: ')    # ici on va placer la réponse dans la variable n
entrez un entier: 421
>>> n                                  # le contenu de la variable n est une chaîne de caractères
'421'
```

On peut bien sûr directement convertir la réponse reçue par input en un type particulier (int et float notamment) :

```
>>> n = int(input('entrez votre réponse: '))
entrez votre réponse: 1234
>>> n                                  # ici la variable n contient bien un entier
1234
```

La fonction print permet d'afficher des informations à l'écran. Il s'agit le plus souvent de messages informatifs qui peuvent intervenir à tout moment de l'exécution du script. On ne confondra pas avec l'instruction return dont le rôle est de *renvoyer* le résultat d'une fonction (et qui met fin à l'exécution de celle-ci).

Les exemples ci-après illustrent quelques possibilités de la fonction print.

On voit que les chaînes de caractères sont affichées sans leurs délimiteurs extérieurs (guillemets ou apostrophes).

On voit également qu'il est possible d'afficher successivement plusieurs objets sur une même ligne (ils sont alors séparés par un espace, mais on peut préciser un autre séparateur avec l'option sep=). Un affichage par print se termine par un passage à la ligne (mais on peut préciser un autre mode de terminaison avec l'option end=)

```
>>> print('a', 11, 'b', 2222, 'c', 333); print('d', 1/3, 'e', 1/7);
a 11 b 2222 c 333
d 0.3333333333333333 e 0.14285714285714285
>>> print('a', 1, 'b', 2, 'c', 3, sep='**')
a**1**b**2**c**3
>>> print('a', 1, 'b', 2, 'c', 3, end=' --- '); print('d', 5, 'e', 7);
a 1 b 2 c 3 --- d 5 e 7
```

3.2 Nécessité de délimiter des blocs d'instructions

Dans de nombreux langages de programmation (dans tous ?), on est amené à grouper des instructions successives, et à considérer ce groupe comme une seule "méta"-instruction. Mais encore faut-il trouver un moyen de marquer (par des éléments du langage et/ou par des dispositions visuelles) les limites d'un tel groupe d'instructions.

Notons par exemple `groupe` une suite d'instructions `instruction_1` ; `instruction_2` ; ... ; `instruction_p`

Imaginons également une condition `test`, vraie ou fausse à un moment donné du déroulement du script.

On cherche à écrire une séquence d'instructions du genre :

Commencer ici...
Si la condition `test` *est vraie, alors évaluer les instructions de* `groupe`
De toutes façons, continuer là...

Voici deux formulations possibles. La solution de gauche est pour le moins ambiguë (est-ce qu'on évalue uniquement `instruction_1` si la condition `test` est vraie ?). Seule la solution de droite est correcte car elle indique clairement les limites du groupe d'instructions à évaluer si la condition `test` est vraie.

Formulation incorrecte	Formulation correcte

Commencer ici...
Si la condition `test` *est vraie, alors*
`instruction_1`
`instruction_2`
...
`instruction_p`
De toutes façons, continuer là...
	Commencer ici...
	Si la condition `test` *est vraie, alors*
	Début de groupe
	`instruction_1`
	`instruction_2`
	...
	`instruction_p`
	Fin de groupe
	De toutes façons, continuer là...

On peut aussi imaginer une séquence d'instructions du genre :

Commencer ici...
Tant que la condition `test` *est vraie, évaluer les instructions de* `groupe`
De toutes façons, continuer là...

On encore, si `var` est une variable parcourant un ensemble `ens` de valeurs, on peut chercher à écrire :

Commencer ici...
Pour chacune des valeurs possibles de `var` *dans* `ens`*, évaluer les instructions de* `groupe`
De toutes façons, continuer là...

Voici quelles seraient les formulations correctes pour traduire ces intentions :

Commencer ici...
Tant que la condition `test` *est vraie, alors*
Début de groupe
`instruction_1`
`instruction_2`
...
`instruction_p`
Fin de groupe
De toutes façons, continuer là...
	Commencer ici...
	Pour chacune des valeurs de `var` *dans* `ens`
	Début de groupe
	`instruction_1`
	`instruction_2`
	...
	`instruction_p`
	Fin de groupe
	De toutes façons, continuer là...

jean-miche.ferrard@ac-paris.fr

Il faut donc un moyen non ambigu de poser les limites d'un groupe d'instructions. Dans le même temps, il est d'usage d'accentuer visuellement le groupe par une légère *indentation* (décalage vers la droite).

Dans les exemples précédents, `instruction_1` ; ... ; `instruction_p` peuvent également être des instructions composées, et donc contenir des groupes (des *blocs*) d'instructions.

Chaque instruction du programme figure donc à un certain niveau de *profondeur* (si on convient que le niveau initial est le niveau 0, on trouvera des instructions de niveau 1, de niveau 2, etc.). Ces niveaux sont d'autant plus faciles à identifier pour le lecteur que le programmeur aura pris soin de les marquer visuellement par des indentations progressives.

Dans la plupart des langages, ces indentations n'ont rien d'obligatoire : elles constituent un simple confort visuel. Elles peuvent varier légèrement, à l'intérieur d'un même bloc, sans conséquence pour la logique du programme.

3.3 L'importance fondamentale de l'indentation en Python

La solution apportée par Python au problème précédent est radicale :

Dans un même bloc, deux instructions de même profondeur logique doivent avoir strictement la même indentation

Avec une telle convention, il est inutile de marquer le début et la fin d'un bloc par des éléments du langage (comme des accolades { et }, ou les mots réservés `begin` et `end`).

La contre-partie est un respect scrupuleux des indentations, mais on est aidé en cela par l'éditeur de `Idle`, qui augmente automatiquement l'indentation après chaque instruction d'en tête, et qui conserve cette indentation à l'intérieur du bloc courant.

Pour sortir d'un bloc (donc diminuer l'indentation), un simple appui sur la touche d'effacement arrière suffit.

Venons-en à quelques définitions :
(voir http://docs.python.org/3.3/reference/compound_stmts.html)

- une *instruction composée* est formée d'une ou de plusieurs *clauses*.
- une clause est formée :
  a) d'un *en-tête* se terminant par le caractère : (deux-points),
  b) d'une *suite*, contrôlée par l'instruction d'en-tête.
- cette *suite* peut se réduire à une seule instruction simple, ou consister en une instruction composée.
- une suite *simple* peut être placée sur la même ligne que l'en-tête, après le caractère " : "
- une suite *composée*, appelée ici un *bloc*, doit débuter à la ligne qui suit l'en-tête et, par rapport à celui-ci elle doit être indentée de façon **uniforme**.

Si la ligne d'en-tête d'une clause est au niveau d'indentation n, le bloc qui la suit est donc au niveau d'indentation $n+1$ (la convention est d'utiliser quatre caractères pour séparer les deux niveaux).

On termine le bloc par un retour à l'indentation du niveau n (c'est-à-dire celle de la ligne d'en-tête).

Un *bloc* (au niveau $n+1$ si sa ligne d'en-tête est au niveau n) peut lui-même inclure des *instructions composées* donc des successions de clauses (chacune d'elles pouvant donner lieu à la création de blocs qui sont alors au niveau $n+2$).

On a représenté ci-contre une situation imaginaire (mais typique) d'une portion de script comportant une "clause 1" (instruction "en-tête1" suivie du "bloc1") et une "clause 2" (instruction "en-tête2", suivie du "bloc2").

Le "bloc1" contient lui-même une "clause3", et le "bloc2" contient une "clause4" qui contient elle-même une "clause5".

```
instruction
...
instruction
en-tête1 : #(suivi du bloc 1)
    instruction
    en-tête3 : #(suivi du bloc 3)
        instruction
        ...
        instruction
    instruction
instruction
...
instruction
en-tête2 : #(suivi du bloc 2)
    instruction
    ...
    instruction
    en-tête4 : #(suivi du bloc 4)
        instruction
        ...
        instruction
        en-tête5 : #(suivi du bloc 5)
            instruction
            ...
            instruction
        instruction
    instruction
    ...
    instruction
instruction
```

Rappel important : le bloc qui suit une ligne d'en-tête peut se réduire à une seule instruction. Dans ce cas, il est possible de le placer à la suite du caractère "deux-points" qui termine la ligne d'en-tête.

Remarque : éviter d'utiliser des tabulations (plutôt que des espaces) pour l'indentation des blocs !

3.4 Branchements conditionnels `if...elif...else...`)

Pour dévier le flot des instructions en fonction de la valeur (vraie/fausse, non_nulle/nulle, non_vide/vide) d'une expression (appelée ici *condition*), `Python` met à notre disposition les clauses `if`, `elif` et `else`.

La clause `if` simple permet d'exécuter un bloc d'instructions si la condition est vraie :

```
if condition :                  # si condition est vraie
    bloc_si_condition_vraie     #    alors on parcourt ce bloc
```

On peut rajouter une clause `else` (facultative, donc) pour exécuter un autre bloc si la condition est fausse :

```
if condition :                  # si condition est vraie
    bloc_si_condition_vraie     #    alors on parcourt ce bloc
else :                          # sinon
    bloc_si_condition_fausse    #    alors on parcourt ce bloc
```

Plutôt que d'emboîter des clauses `if`, on peut utiliser une clause `if` suivie par une (des) clause(s) `elif` :

```
if condition1 :                         # si condition1 est vraie
    bloc1_si_C1_vraie                   #    alors on exécute ce bloc1
elif condition2 :                       # sinon si condition2 est vraie
    bloc2_si_C1_fausse_mais_C2_vraie    #    alors on exécute ce bloc2
elif condition3 :                       # sinon si condition3 est vraie
    bloc3_si_C1C2_fausses_mais_C3_vraie #    alors on exécute ce bloc3
...
elif conditionN :                       # sinon si conditionN est vraie
    blocN_si_CN_première_à_être_vraie   #    alors on exécute ce blocN
else :                                  # (facultatif) si toutes les conditions sont fausses
    bloc_else                           #    alors on exécute ce bloc
```

Remarques :

- dans toutes les constructions ci-dessus, au plus un bloc est parcouru
- ne pas oublier le caractère "deux points" qui termine chacune des lignes d'en-tête!!!
- se souvenir que si un bloc se réduit à une instruction, on peut le placer directement après le "deux points"

```
if x % 2:                           # comprendre ici "si x est non nul modulo 2"
    print("x est impair")
elif x % 4:                         # ici x est pair, et on teste son reste modulo 4
    print("x est pair, mais pas multiple de 4")
elif x % 8:                         # ici x est multiple de 4, et on teste son reste modulo 8
    print("x multiple de 4, mais pas de 8")
else:                               # si on en est là, c'est que x est multiple de 8
    print("x est multiple de 8")
```

3.5 Expressions conditionnelles

`Python` offre la possibilité de former des *expressions* dont l'évaluation est soumise à une condition.

La syntaxe est la suivante : *expression1* `if` *condition* `else` *expression2*

L'opérateur `if else` est ici un opérateur *ternaire* (il a trois arguments).

Le résultat est bien sûr l'évaluation de *expression1* si la *condition* est vraie, et sinon c'est celle de *expression2*.

Remarque : cette construction ne permet pas l'utilisation du mot réservé `elif` (mais on voit sur le deuxième exemple ci-dessous comment emboîter deux expressions conditionnelles).

```
>>> x = -1; print("x positif" if x>0 else "x négatif ou nul")
x négatif ou nul
>>> x = 0; print("x positif" if x>0 else "x négatif" if x<0 else "x nul")
x nul
```

3.6 Répétitions conditionnelles (`while`)

Pour répéter un bloc d'instructions *tant qu'une condition* est réalisée, `Python` nous propose la clause `while` :

```
while condition :                # tant que la condition est vraie
    bloc_si_condition_vraie      #    alors on parcourt ce bloc
```

Quelques remarques classiques sur ce genre de construction :

- si *condition* est fausse dès le départ, le bloc qui suit n'est jamais parcouru.
- dans la plupart des cas, le bloc qui suit l'instruction d'en-tête `while` agit sur la *condition*, de sorte que celle-ci, vraie au départ, devient fausse et provoque la sortie de la clause.
- on peut écrire une clause `while` avec une condition toujours vraie (par exemple `while 1:` ou `while True:`) à condition (pour éviter une boucle infinie) de sortir par un autre moyen (notamment par `break` ou `return`).

Dans l'exemple ci-dessous, on illustre la clause `while` avec l'exemple classique de la suite dite *de Syracuse*.
Celle-ci est définie par une valeur initiale x_0 et la règle suivante : si x_n est pair alors $x_{n+1} = x_n/2$ sinon $x_{n+1} = 3x_n + 1$.
Une conjecture célèbre dit que l'un des x_n vaut 1 (et la suite boucle alors sur les valeurs $1 \to 4 \to 2 \to 1$).
On étudie ici le comportement très intéressant obtenu pour la valeur initiale $x_0 = 27$:

```
>>> x = 27                       # on part de la valeur 27.
>>> while x != 1:                # tant que x est différent de 1
        if x % 2: x = 3*x+1      #    si x est impair, on le remplace par 3x+1
        else: x = x // 2         #    sinon, on le divise par 2
        print(x, end=' ')        #    on affiche la valeur de x (sans passer à la ligne)

82 41 124 62 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137 412 206 103 310 155
466 233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336 668 334 167 502 251 754 377 1132
566 283 850 425 1276 638 319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288 3644 1822 911
2734 1367 4102 2051 6154 3077 9232 4616 2308 1154 577 1732 866 433 1300 650 325 976 488 244 122
61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

Le bloc qui fait suite à l'instruction `while` peut contenir deux instructions particulières, souvent attachées à un test `if` :

- `break` provoque la sortie immédiate de la clause `while`.
- `continue` ramène à l'évaluation de la condition (ce qui restait du bloc après `continue` est donc ignoré).

La construction `while` peut être complétée par une clause `else`, exécutée si la condition qui suit le `while` est fausse :

```
while condition :                # tant que la condition est vraie
    bloc1_si_condition_vraie     #    alors on exécute ce bloc
else :                           # mais quand la condition devient fausse
    bloc2_si_condition_fausse    #    alors on exécute ce bloc
                                 #    sauf si on est sorti de bloc1 par l'instruction break
```

L'utilisation de `else` attaché à un `while` est rare. Le *bloc2* ci-dessus est en effet parcouru une seule fois, quand la *condition* devient fausse, sauf si on sort du bloc n ° 1 par une instruction `break` (auquel cas *bloc2* est ignoré).

3.7 Notion d'intervalle

On est souvent amené à répéter plusieurs fois un bloc d'instructions en fonction des valeurs successives d'un compteur. Une façon simple (mais c'est loin d'être la seule) d'indiquer la plage des valeurs possibles de ce compteur est d'utiliser un *intervalle* (ou encore *range* dans le langage `Python`) :

- la syntaxe est `range(a,b,h)` où a, b, h sont des valeurs entières.
- la valeur a (le *début* de l'intervalle) est facultative, et par défaut elle vaut 0.
  si a est présent, la valeur h (le *pas* de l'intervalle) est facultative, et vaut 1 par défaut.
- l'intervalle `range(a,b,h)` est formé des valeurs $x = a + kh$, avec k entier tel que $\begin{cases} a \leq x < b & \text{si } a < b \\ b < x \leq a & \text{si } b < a \end{cases}$
- retenons que la valeur b est toujours **exclue** de l'intervalle `range(a,b,h)`

L'intervalle `range(`a`,`b`,`h`)` doit être vu comme une succession de valeurs, en partant de a, et en progressant vers b (sans jamais l'atteindre !), dans le sens croissant ou décroissant selon que le *pas* est positif ou négatif. Ainsi :

- l'intervalle `range(7)` représente la succession des valeurs $0, 1, 2, 3, 4, 5, 6$
- l'intervalle `range(1,7)` représente la succession des valeurs $1, 2, 3, 4, 5, 6$
- l'intervalle `range(1,7,2)` représente la succession des valeurs $1, 3, 5$
- l'intervalle `range(7,2)` est vide (ici le pas a sa valeur par défaut, c'est-à-dire 1)
- l'intervalle `range(7,2,-1)` représente la succession des valeurs $7, 6, 5, 4, 3$
- si $a \leq b$, l'intervalle `range(a,b)` est formé de $b - a$ valeurs (il est notamment vide si $a = b$)

Pour tester l'appartenance d'une valeur à un intervalle, on utilise le mot réservé `in` (résultat `True` ou `False`).

Attention : les intervalles dont il est question ici sont des échantillons de valeurs entières, ce ne sont donc pas des intervalles au sens mathématique du terme. Voici quelques exemples :

```
>>> r = range(100,1000,2)   # intervalle des entiers pairs, de 100 inclus à 1000 exclu.
>>> 100 in r                # 100 fait-il partie de l'intervalle, réponse oui.
True
>>> 1000 in r               # 1000 fait-il partie de l'intervalle, réponse non.
False
>>> 513 in r                # 513 fait-il partie de l'intervalle, réponse non.
False
```

3.8 Répétitions inconditionnelles (boucles `for`)

Pour répéter un certain nombre de fois un bloc d'instructions, on utilisera la construction suivante :

```
for variable in objet :     # pour chaque élément (nommé pour l'occasion variable) de objet
    bloc_d'instructions     #    on parcourt ce bloc
```

En fait, *objet* est ici est toute construction susceptible d'être *parcourue* : on pense bien sûr aux intervalles (`range`), mais aussi aux chaînes (parcourues caractère par caractère), aux listes, aux tuples, aux dictionnaires (on en parlera plus tard).

```
for k in range(0,10):       # pour k = 0, puis k = 1, etc. jusqu'à k = 9
    bloc_d'instructions     #    on parcourt ce bloc
```

```
for x in 'abcdef':          # pour x = 'a', puis x = 'b', etc. jusqu'à x = 'f'
    bloc_d'instructions     #    on parcourt ce bloc
```

Le bloc qui fait suite à l'instruction `for` peut contenir deux instructions particulières, souvent attachées à un test `if` :

- `break` provoque la sortie immédiate de la clause `for`.
- `continue` passe directement à l'étape suivante de la boucle (ce qui reste du bloc après `continue` est donc ignoré).

La construction `for` peut être complétée par une clause `else`, parcourue quand la boucle est terminée :

```
for variable in objet :     # pour chaque élément (nommé pour l'occasion variable) de objet
    bloc1_d'instructions    #    on parcourt ce bloc
else:                       # quand la boucle est terminée
    bloc2_d'instructions    #    on parcourt ce bloc
```

L'utilisation de `else` attaché à une boucle `for` est assez rare. Le *bloc2* ci-dessus est en effet parcouru une seule fois, quand la boucle est terminée, sauf si on sort du bloc n°1 par une instruction `break` (auquel cas *bloc2* est ignoré).

3.9 L'instruction `pass`

L'instruction `pass` ne fait... rien. Mais elle a son utilité dans la phase de mise au point d'un programme.

```
if n == 1:
    <... boc de traitement du cas n=1...>
elif n == 2: pass                          # là on ne sait pas encore
else:
    <... boc de traitement si n ∉ {1,2}...>
```


jean-miche.ferrard@ac-paris.fr

Chapitre 4

Écrire des fonctions Python

Une fonction est un bloc d'instructions qui a reçu un nom, dont le fonctionnement dépend d'un certain nombre de paramètres (les *arguments* de la fonction) et qui renvoie un résultat (au moyen de l'instruction `return`).

```
def nom_de_la_fonction(arguments):  # le nom de la fonction, et les paramètres d'appel
    bloc_d'instructions             #    on parcourt ce bloc
```

4.1 La valeur None, et l'instruction return

`Python` propose le type `NoneType`, dont la seule valeur est `None` : c'est une représentation du "rien", de l'absence.

La séquence des arguments d'une fonction peut très bien être vide. Si f est une telle fonction, l'instruction $y = f()$ *appelle* la fonction (provoque son exécution) et place le résultat dans la variable y. On notera que les parenthèses sont indispensables, pour distinguer l'objet f qui est une fonction de l'objet $f()$ qui est le résultat d'un appel à cette fonction.

Dans un tel cas, il est préférable de penser que f prend tout de même un argument, à savoir la valeur `None`.

La valeur de retour d'une fonction est obtenue par une instruction `return` *valeur* dans le corps de la fonction.

Il peut y avoir plusieurs instructions `return` dans une fonction (évidemment on ne sortira réellement de la fonction qu'en passant par l'une et l'une seulement de ces instructions).

L'oubli de `return` est une erreur classique. Dans ce cas, la fonction renvoie la valeur `None` (c'est peut-être le but).

Un affichage obtenu par `print` dans le corps d'une fonction ne doit pas être considéré comme le résultat de la fonction (c'est un simple *effet de bord*). D'ailleurs la fonction `print` renvoie la valeur `None`.

L'exemple suivant, extrêmement simple, illustre la nécessité de l'instruction `return` :

```
>>> def f(x): print(x*x)     # la fonction f, d'argument x, affiche le carré de x
>>> y = f(5)                 # on appelle f avec l'argument 5 et on met le 'résultat' dans y
25                           # bien sûr, cet appel provoque l'affichage de la valeur 25
>>> y                        # mais ce n'était qu'un 'effet de bord'
                             # en fait y ne contient rien (ou plutôt si, la valeur None)
>>> type(y)                  # on le voit bien en demandant le type de y
<class 'NoneType'>
>>> print(y)                 # ou plus clairement en affichant le 'contenu' de y par print
None
```

On va maintenant modifier la fonction f :

```
>>> def f(x): x*x            # on calcule toujours x², mais il n'y a pas de print ni de return
>>> y = f(5)                 # on appelle f avec l'argument 5 et on met le 'résultat' dans y
>>> type(y)                  # il ne s'est rien passé à l'écran (pas de print)
<class 'NoneType'>           #    et on voit que la valeur 5² est perdue, car f(5) a renvoyé None
```

On opère une nouvelle modification de la fonction f :

```
>>> def f(x): return x*x  # on calcule toujours x², on ne l'affiche pas, mais on le renvoie
>>> y = f(5)              # on appelle f avec l'argument 5 et on met le résultat dans y
>>> y                     # effectivement, y contient maintenant la valeur 25
25
```

Remarque : un *return* vide (c'est-à-dire suivi d'aucune expression) renvoie la valeur `None`.

4.2 l'espace de noms global

Pendant toute la durée de l'évaluation d'un script, `Python` peut ouvrir, utiliser et refermer des *espaces de nom*. Il s'agit d'éviter des conflits d'homonymie (deux identificateurs de même nom, mais de contenus différents, et auxquels on veut continuer à accéder sans que l'un "masque" l'autre).

Au niveau interactif, on est dans l'espace "global". Son contenu peut d'ailleurs être visualisé avec la fonction `globals` (qui ne prend pas d'arguments). Voici par exemple le contenu de l'espace de noms *global* quand on vient juste de lancer l'interpréteur (l'objet affiché est de type *dictionnaire*) :

```
>>> ================================ RESTART ================================
>>> globals()       # on vient de relancer l'interpréteur Python, et on demande les noms globaux
{'__package__': None, '__name__': '__main__',
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__builtins__': <module 'builtins'>, '__doc__': None}
```

Juste après, on crée trois variables globales x, y, z, et on appelle à nouveau la fonction `globals`. On voit que les noms x, y, z et les contenus correspondants ont été ajoutés au dictionnaire :

```
>>> x, y, z = 1, 2, 3          # on crée les variables globales x, y, z
>>> globals()                  # elles apparaissent maintenant dans l'espace de nom global
{'__package__': None, 'y': 2, 'x': 1, '__name__': '__main__',
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__builtins__': <module 'builtins'>, 'z': 3, '__doc__': None}
```

Continuons, en important cette fois le module `math` (avec l'instruction `import math`). En appelant à nouveau la fonction `globals`, on voit que le module `math` a été ajouté, mais qu'il ne représente qu'une seule entrée du dictionnaire (pour des raisons de présentation, on a un peu raccourci le chemin d'accès au module) :

```
>>> import math
>>> globals()
{'__package__': None, 'y': 2, 'x': 1, '__name__': '__main__',
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__builtins__': <module 'builtins'>, 'z': 3,
 'math': <module 'math' from '/Library/.../python3.3/lib-dynload/math.so'>, '__doc__': None}
```

Dans la méthode précédente de chargement du module `math`, les noms des fonctions qui composent ce module ne sont pas directement présents dans l'espace global. Ces noms sont en effet présents dans l'*espace de nom du module* ***`math`***, et on peut y accéder à l'aide du préfixe "math." (par exemple, on écrira `math.sqrt(5)`).

En revanche, on peut décider de charger le module `math` avec l'instruction `from math import *`.

Un nouvel appel à la fonction `globals` montre qu'alors tous les noms de ce module appartiennent à l'espace global (et le dictionnaire est beaucoup plus volumineux : on a ici considérablement raccourci l'affichage).

```
>>> from math import *
>>> globals()
{'isinf': <built-in function isinf>, 'tanh': <built-in function tanh>,
 'expm1': <built-in function expm1>, 'isnan': <built-in function isnan>,
 'log1p': <built-in function log1p>, 'copysign': <built-in function copysign>,
 <...>
 'e': 2.718281828459045, 'log2': <built-in function log2>,
 'hypot': <built-in function hypot>, 'asin': <built-in function asin>}
```

Avec cette nouvelle méthode, les fonctions du module `math` sont donc directement accessibles par leur nom court (par exemple `sqrt` plutôt que `math.sqrt`). C'est peut-être ce qu'on recherche, mais on s'expose à des problèmes d'homonymie : si deux fonctions portent le même nom, la dernière à avoir été chargée en mémoire occulte la précédente.

jean-miche.ferrard@ac-paris.fr

4.3 L'espace de noms local d'une fonction

Quand une fonction f est appelée (et que débute son évaluation), elle crée un nouvel *espace de noms* : cela signifie que les variables qui sont créées dans cette fonction ne sont visibles qu'à l'intérieur de celle-ci (elles y sont *locales*). Notre fonction f peut cependant continuer à accéder aux variables de l'espace global. De même, si notre fonction f contient la définition d'une fonction g, celle-ci crée un espace de noms inclus dans celui de f.

On peut lire les définitions locales d'une fonction en évaluant l'expression `locals()`.

Le mieux est de prendre quelques exemples, après avoir redémarré l'interpréteur `Python`.

```
>>> ================================ RESTART ================================
>>> a, b = 1, 2      # au niveau interactif (au niveau 'global'), on pose a = 1 et b = 2
>>> globals()        # voici le dictionnaire de l'espace de nom global
{'b': 2, 'a': 1, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__doc__': None, '__builtins__': <module 'builtins'>, '__name__': '__main__'}
>>> locals()         # au niveau interactif, 'local' et 'global' c'est pareil
{'b': 2, 'a': 1, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__doc__': None, '__builtins__': <module 'builtins'>, '__name__': '__main__'}
```

On définit ensuite une fonction f (prenant deux arguments x et y) à l'intérieur de laquelle on définit (puis on appelle) une fonction g (prenant un argument z).

Les deux fonctions f et g créent des variables et affichent, à plusieurs reprises, le contenu de l'expression `locals()` (pour qu'on puisse retracer l'évolution de l'espace de nom local en cours).

On accompagne cet affichage d'une lettre majuscule (de 'A' à 'G') pour mieux suivre la chronologie.

À un moment donné, à l'intérieur de la fonction g, on demande également l'affichage du dictionnaire de l'espace de noms global.

```
def f(x,y):
    print('A:',locals())
    a = 3
    print('B:',locals())
    def g(z):
        print('C:',locals())
        a = 4; b = 5
        print('D:',locals())
        print('E:',globals())
    print('F:',locals())
    g(6)
    print('G:',locals())
```

On commence par rappeler le contenu du dictionnaire de l'espace de noms global (on y trouve bien sûr un enregistrement relatif à la fonction f). On voit à cette occasion que dans ce dictionnaire le contenu de la variable f est représenté par "`function f at...`", avec indication de l'adresse du code de la fonction en mémoire.

```
>>> globals()
{'f': <function f at 0x1040f0710>, 'b': 2, 'a': 1, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__doc__': None,
 '__builtins__': <module 'builtins'>, '__name__': '__main__'}
```

On appelle maintenant f avec les argments 10 et 20. On observe (et on commente après) les affichages obtenus :

```
>>> f(10,20)
A: {'y': 20, 'x': 10}
B: {'a': 3, 'y': 20, 'x': 10}
F: {'g': <function f.<locals>.g at 0x1040b8680>, 'a': 3, 'y': 20, 'x': 10}
C: {'z': 6}
D: {'z': 6, 'a': 4, 'b': 5}
E: {'f': <function f at 0x1040f0710>, 'b': 2, 'a': 1, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__doc__': None,
 '__builtins__': <module 'builtins'>, '__name__': '__main__'}
G: {'g': <function f.<locals>.g at 0x1040b8680>, 'a': 3, 'y': 20, 'x': 10}
```

Voici comment on peut interpréter les résultats précédents :

A : on vient d'*entrer* dans la fonction f, et celle-ci a créé deux variables locales x et y, qui ont reçu respectivement les valeurs 10 et 20 passées lors de l'appel à f.

B : dans la fonction f, on a créé une variable locale a (et on lui a donné la valeur 3).

jean-miche.ferrard@ac-paris.fr

F : on est ici juste après la définition de g à l'intérieur de f. On voit que g a été ajoutée au dictionnaire de l'espace de noms local à f, et la description du contenu de g (à savoir `<function f.<locals>.g at 0x1040b8680>`) indique bien qu'il s'agit ici d'une fonction n'existant que dans l'espace de noms local à f.

C : on vient d'entrer dans la fonction g (appelée avec l'argument 6). La fonction g a donc créé une variable locale z avec la valeur 6. On voit que l'espace de noms local est ici celui de g, et qu'il se réduit pour l'instant à la variable z.

D : La fonction g vient de créer deux variables locales a et b, avec les valeurs 4 et 5.

E : Depuis la fonction g on a demandé l'affichage du dictionnaire de l'espace de noms global. On y retrouve les enregistrements relatifs à f et aux variables globales a et b (définies avant l'appel de f) : on voit qu'elles ont conservé les valeurs 1 et 2, et qu'elles ne sont donc pas affectées par les définitions locales de a et de b dans l'espace de g.

G : On est maintenant "sorti" de la fonction g, et on retrouve le dictionnaire de nom local à la fonction f. On voit notamment que les variables locales z, a, b définies dans g n'existent plus : l'enregistrement relatif à la variable a évoque en fait la variable locale définie dans f par l'instruction $a = 3$.

Quand cet appel à la fonction f est terminé, on revient au niveau interactif (global) et on retrouve le dictionnaire de l'espace de nom global dans l'état où il était avant l'évaluation de l'expression $f(10, 20)$. On y retrouve notamment les deux variables globales $a = 1$ et $b = 2$:

```
>>> globals()        # re-voici le dictionnaire de l'espace de nom global
{'b': 2, 'a': 1, '__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__doc__': None, '__builtins__': <module 'builtins'>, '__name__': '__main__'}
```

On voit maintenant une situation différente, dans laquelle les variables concernées ont un contenu de type liste (donc un contenu composite et *mutable*).

On redémarre `Python`, et on définit les fonctions f, g ci-contre. On place la liste $[1, 2, 3]$ dans la variable globale a.

```
a = [1, 2, 3]
def f(x):
    print('A: a=',a,'x=',x,'id(a)=',id(a),'id(x)=',id(x))
    x = [4,5,6]
    print('B: a=',a,'x=',x,'id(a)=',id(a),'id(x)=',id(x))
def g(x):
    print('C: a=',a,'x=',x,'id(a)=',id(a),'id(x)=',id(x))
    x[0] = 9
    print('D: a=',a,'x=',x,'id(a)=',id(a),'id(x)=',id(x))
```

Avec les définitions précédentes, on évalue l'expression $f(a)$. On voit (affichage 'A') que la fonction f crée la variable locale x et qu'elle y place le contenu de a. En fait, à cet instant, les deux variables a (globale) et x (locale) pointent sur une liste unique à une adresse bien précise en mémoire (celle donnée par la fonction `id`).

Ensuite, on redéfinit la variable locale x, en y plaçant $[4, 5, 6]$. On voit (affichage 'B') que les deux variables a et x pointent sur deux listes différentes en mémoire. C'est normal, car les valeurs sont différentes, mais le résultat aurait été le même si on avait écrit $x = [1, 2, 3]$: on aurait défini une nouvelle liste en mémoire (égalité structurelle mais pas égalité physique).

```
>>> f(a)
A: a= [1, 2, 3] x= [1, 2, 3] id(a)= 4316768304 id(x)= 4316768304
B: a= [1, 2, 3] x= [4, 5, 6] id(a)= 4316768304 id(x)= 4316769960
```

On évalue maintenant l'expression $g(a)$. On voit (affichage 'C') que g crée la variable locale x et qu'elle y place le contenu de a. À ce moment il y a égalité physique entre les deux. Ensuite l'instruction $x[0] = 9$ ne redéfinit par la liste x, mais seulement l'un de ses éléments, et cela ne modifie pas l'adresse de la liste x en mémoire (ça modifie seulement l'adresse de la valeur correspondant à la première position de cette liste).

On a la confirmation de ce comportement avec l'affichage 'D', où on constate que la modification de la liste placée dans la variable locale x s'est répercutée sur la liste placée dans la variable globale a (c'est normal, les deux variables pointent sur la même adresse, il y a encore égalité physique).

```
>>> g(a)
C: a= [1, 2, 3] x= [1, 2, 3] id(a)= 4352360600 id(x)= 4352360600
D: a= [9, 2, 3] x= [9, 2, 3] id(a)= 4352360600 id(x)= 4352360600
```

On retiendra donc que quand ils sont transmis en tant qu'argument d'une fonction, les objets composites *mutables* (et en particulier les listes, mais c'est valable aussi pour les dictionnaires, qui seront abordés plus loin) sont transmis par "adresse" plutôt que par "valeur".

Rappelons qu'on peut toujours faire une copie *indépendante* d'une liste a en écrivant l'instruction $x = a[:]$

jean-miche.ferrard@ac-paris.fr

4.4 Remarques sur les espaces de noms emboîtés

Chaque fonction crée son propre espace de noms. Il est possible (et courant) d'emboîter des définitions de fonctions, donc des espaces de noms. Depuis le plus récent d'entre eux, on peut lire les variables définies dans les espaces de noms qui le contiennent. À l'intérieur d'un même espace de noms, les variables deviennent visibles dans l'ordre chronologique de leurs définitions.

Par exemple, on définit ici la variable globale $a = 1515$.

Ensuite on définit une fonction g à l'intérieur d'une fonction f.

La variable locale a est définie dans f. Sa définition suit celle de g mais (et c'est le plus important) elle précède l'appel à g.

Dans ces conditions (et on le constate lors d'un appel à f), la fonction g "voit" la variable locale a (qui lui masque d'ailleurs la variable globale $a = 1515$).

```
>>> a = 1515
>>> def f():
        def g(): print(a)
        a = 2013
        g()
>>> f()
2013
```

Autre exemple : on définit la variable globale $a = 1515$.

On définit ensuite la fonction g à l'intérieur de la fonction f.

La fonction g ne redéfinit plus ici de variable locale a.

Quand on appelle f (qui appelle elle-même g), on constate que g a accès en lecture à la variable globale $a = 1515$.

```
>>> a = 1515
>>> def f():
        def g(): print(a)
        g()
>>> f()
1515
```

Dans l'exemple suivant, on tente de créer la variable locale $a = 2013$ dans la fonction f, mais *après* l'appel à g.

Or la fonction g fait référence à la variable a. Il y a dans ce cas une ambiguïté entre les deux définitions de a.

On voit que `Python` réagit par un message d'erreur approprié (qu'on a ici abrégé).

```
>>> a = 1515
>>> def f():
        def g(): print(a)
        g()
        a = 2013
>>> f()
Traceback (most recent call last): <...>
NameError: free variable 'a' referenced before assignment in enclosing scope
```

Voici une autre situation qui conduit à un message d'erreur (différent du précédent). La fonction f tente d'ajouter 1 au contenu de la variable a. Mais l'écriture `a = a+1` provoque l'évaluation dans f et tente de créer une variable *locale* a.

Le problème est que le second membre $a + 1$ est évalué d'abord, ce que `Python` rejette car l'ambiguité est trop grande entre les deux significations de la variable a.

```
>>> a = 1515
>>> def f():
        a = a + 1; print(a)
>>> f()
Traceback (most recent call last): <...>
UnboundLocalError: local variable 'a' referenced before assignment
```

Si on souhaite utiliser (non seulement pour la lire mais pour la modifier), depuis l'intérieur d'une fonction, une variable globale, il faut le préciser (au début de la fonction) par la directive `global` :

```
>>> a = 1515
>>> def f():
        global a
        a = a + 1; print(a)
>>> f()
1516
>>> a
1516
```


jean-miche.ferrard@ac-paris.fr

4.5 Paramètres positionnels ou nommés, valeurs par défaut

Une fonction peut accepter un nombre quelconque de paramètres : leurs noms n'ont pas beaucoup d'importance (ils servent à nommer localement les valeurs transmises lors de l'appel à la fonction). En revanche l'ordre dans lequel apparaissent ces paramètres dans la définition de la fonction est important.

On peut donner une valeur *par défaut* à un ou plusieurs arguments d'une fonction. Pour cela, dans la définition de cette fonction, et si on note arg le nom de l'argument et val sa valeur par défaut, on remplacera arg par $arg = val$.

Si un argument reçoit une valeur par défaut, il peut donc être omis lors de l'appel de la fonction.

Mais pour qu'il n'y ait aucune ambiguïté, les arguments avec valeurs par défaut (dans la définition de la fonction) doivent *suivre* ceux qui n'en ont pas.

```
def f(x, y, z, t):          # f est une fonction avec quatre arguments, sans valeurs par défaut
def g(x, y, z, t=10):       # définition de g où, par défaut, t = 10
def h(x, y, z=5, t=10):     # définition de h où, par défaut, z = 5 et t = 10
def k(x, y=2, z, t=10):     # cette définition serait illégale
```

Dans les exemples précédents :

- pour appeler la fonction f, il faut écrire explicitement $f(x, y, z, t)$ en précisant les valeurs des quatre arguments.
- évaluer $g(x, y, z)$ équivaut à évaluer $g(x, y, z, 10)$;
  on peut bien sûr écrire $g(x, y, z, 1000)$ si on tient à donner la valeur 1000 au quatrième argument t.
- évaluer $h(x, y)$ équivaut à évaluer $h(x, y, 5, 10)$; évaluer $h(x, y, 999)$ équivaut à évaluer $h(x, y, 999, 10)$;
  on peut bien sûr écrire $g(x, y, 9, 6)$ si on tient à donner la valeur 9 à l'argument z et la valeur 6 à l'argument t.
- la définition de la fonction k est illégale ;
  par exemple, dans $k(1, 3, 5)$, comment distinguer $(x = 1, y = 3, z = 5, t = 10)$ de $(x = 1, y = 2, z = 3, t = 5)$?

Les arguments avec valeurs par défaut offrent une possibilité intéressante lors de l'*appel* de la fonction.

On peut en effet remplacer $f(\cdots, valeur, \cdots)$ par $f(\cdots, arg = valeur, \cdots)$. Il faut bien sûr que le(s) nom(s) arg utilisés ici dans l'appel de f correspondent au(x) nom(s) arg utilisés dans la définition de cette fonction.

Il faut également que les arguments ainsi nommés *suivent* les arguments non nommés lors de l'appel (ces derniers sont alors simplement appelés *arguments positionnels*). Enfin, et c'est là le principal intérêt, les *arguments nommés* peuvent être donnés dans n'importe quel ordre.

Pour comprendre comment ça fonctionne, prenons un exemple très simple. La fonction f prend cinq arguments : les deux premiers sont obligatoires et les trois suivants sont optionnels (ils reçoivent une valeur par défaut dans la définition de f).

Notre fonction f se résume à afficher les valeurs qu'elle reçoit pour les cinq variables locales x, y, z, t, u.

Voici la définition de f :

```
>>> def f(x, y, z=666, t=777, u=888):
        print('x=',x,' y=',y,' z=',z,' t=',t,' u=',u,sep='')
```

Ensuite, on effectue plusieurs appels à la fonction f, en omettant tout ou partie des arguments par défaut.

```
>>> f(1, 2)                 # ici z,t,u reçoivent leur valeur par défaut
x=1 y=2 z=666 t=777 u=888
>>> f(1, 2, 3)              # ici t et u reçoivent leur valeur par défaut
x=1 y=2 z=3 t=777 u=888
>>> f(1, 2, 3, 4)           # ici seul u reçoit sa valeur par défaut
x=1 y=2 z=3 t=4 u=888
>>> f(1, 2, 3, 4, 5)        # ici les cinq arguments de f  reçoivent une valeur
x=1 y=2 z=3 t=4 u=5
```

Dans cette deuxième série d'exemples, on voit qu'il est possible de préciser des valeurs pour un ou plusieurs des arguments optionnels, et ce dans un ordre quelconque, à condition de préciser d'abord les valeurs des deux arguments positionnels (c'est-à-dire non nommés) x et y.

```
>>> f(1, 2, u=5)            # ici on précise u = 5, mais z et t gardent leur valeur par défaut
x=1 y=2 z=666 t=777 u=5
>>> f(1, 2, t=4, u=3)       # ici on précise u = 3 et t = 4, mais par défaut z = 666
x=1 y=2 z=666 t=4 u=3
>>> f(1, 2, u=5, t=4, z=3)  # ici on précise z = 3, u = 5 et t = 4, dans un ordre quelconque
x=1 y=2 z=3 t=4 u=5
```


jean-miche.ferrard@ac-paris.fr

4.6 Rattrapage des exceptions

Quand une erreur se produit dans un script, elle provoque l'arrêt du programme et l'affichage d'un message d'erreur.

Pour éviter cette interruption brutale, on peut appliquer un traitement spécifique. Plus généralement, on peut traiter une situation exceptionnelle dont on ne sait pas forcément où et quand elle va survenir à l'intérieur d'un bloc donné. Plutôt que de parler d'erreur, on emploiera donc le terme "exception". Et prévoir une réaction adaptée à une exception, c'est la "rattraper".

Pour le traitement des exceptions, Python offre la clause `try: ... else`.

La forme la plus simple est la suivante, où le bloc2 est parcouru si une exception (quelle qu'elle soit) est rencontrée dans le bloc1 (cette exception provoque l'arrêt du bloc1 et le passage immédiat au bloc2) :

```
try :
    bloc1_dans_lequel_une_exception_peut_survenir
except :
    bloc2_de_rattrapage_de_toutes_les_exceptions
```

On peut également prévoir un traitement particulier pour telle ou telle exception. Dans ce cas, on utilisera une construction un peu plus élaborée, du genre :

```
try :
    bloc1_dans_lequel_une_exception_peut_survenir
except nom_de_l_exception_A :
    blocA_de_rattrapage_de_l_exception_A
except nom_de_l_exception_B :
    blocB_de_rattrapage_de_l_exception_B
...
except nom_de_l_exception_Z :
    blocZ_de_rattrapage_de_l_exception_Z
except :                                                    # (facultatif)
    bloc_de_rattrapage_de_toutes_les_exceptions_non_prévues_à_ce_stade
```

On peut appliquer un même traitement à plusieurs exceptions. Pour celà, on écrira : `except` $(err_1, err_2, \ldots)$

La construction `except` peut être complétée par une clause `else`, exécutée si aucune exception n'a été levée/rattrapée.

Python possède beaucoup d'exceptions prédéfinies (cf http://docs.python.org/3.3/library/exceptions.html)

En voici juste quelques-unes :

IndexError	se produit si on cherche à accéder à un élément d'une liste, en dehors des limites de celle-ci
NameError	se produit quand on évoque une variable (locale ou globale) dont le nom n'existe pas
SyntaxError	se produit quand Python échoue à lire une expression dont la syntaxe est erronée
TypeError	se produit quand une opération est appliquée à un objet qui n'est pas du type adéquat
ZeroDivisionError	se produit quand un calcul arithmétique conduit à une division par 0

Le programme ci-dessous demande une expression numérique au clavier, et il renvoie la valeur de cette expression (et s'il y a une erreur, quelle qu'elle soit, il demande à nouveau d'entrer une expression numérique) :

```
def calcul():
    fini = False
    while not fini:
        try: result = float(eval(input('Entrez une expression numérique: ')))
        except: print('Il y a eu une erreur, recommencez')
        else: fini = True
    print('Le résultat est:',result)
```


jean-miche.ferrard@ac-paris.fr

Voici un exemple d'utilisation. On entre ici plusieurs expressions erronées. L'erreur n'est jamais la même, mais le traitement indifférencié par `try:... except:` ne permet pas de s'en rendre compte.

```
>>> calcul()
Entrez une expression numérique: [1,2,3]
Il y a eu une erreur, recommencez
Entrez une expression numérique: 1**
Il y a eu une erreur, recommencez
Entrez une expression numérique: 1/0
Il y a eu une erreur, recommencez
Entrez une expression numérique: 1/2+1/3
Le résultat est: 0.8333333333333333
```

On modifie maintenant le programme `calcul` en affinant un peu le traitement des erreurs. Il y a un message personnalisé pour les exceptions les plus probables, mais la dernière clause `except:` permet de rattraper toutes les exceptions non traitées à ce stade.

```
def calcul():
    fini = False
    while not fini:
        try: result = float(eval(input('Entrez une expression numérique: ')))
        except (TypeError, ValueError): print("Erreur de Valeur/Type")
        except SyntaxError: print("Il y a une erreur de syntaxe")
        except ZeroDivisionError: print("Il y a une division par zéro")
        except: print("Quelque chose n'a pas fonctionné, mais quoi?")
        else: fini = True
    print('Le résultat est:',result)
```

Et voici un exemple d'utilisation de la procédure `calcul` :

```
>>> calcul()
Entrez une expression numérique: [1,2,3]
Erreur de Valeur/Type
Entrez une expression numérique: 1**
Il y a une erreur de syntaxe
Entrez une expression numérique: 1/0
Il y a une division par zéro
Entrez une expression numérique: sqrt(5)
Quelque chose n'a pas fonctionné, mais quoi?
Entrez une expression numérique: 1/2+2/3
Le résultat est: 1.1666666666666665
```

Les erreurs sont souvent subies (puis traitées), mais elles peuvent aussi être provoquées (on dit alors *levées*).

Cela peut correspondre à la réalisation d'une situation très particulière ("exceptionnelle") et l'utilisation de clauses `try... except...` peut constituer une bonne alternative aux clauses `if... then...`.

Dans le petit programme suivant, on part à la recherche d'un élément *cible* dans une *liste*. Si l'élément *cible* est trouvé, on lève `Exception` (ce qui est la classe d'exception la plus générale) et on répond par le message "Trouvé" (le parcours de la liste est alors interrompu, bien sûr). Sinon (et il faut pour cela aller jusqu'au bout de la liste), et donc s'il n'y a pas d'erreur, la clause `else` affiche le message "Non trouvé".

```
def cherche(cible,liste):
    try:
        for elt in liste:
            if elt == cible: raise Exception
    except: print("Trouvé")
    else: print("Non trouvé")
```

Voici un exemple d'utilisation de la fonction `cherche` :

```
>>> cherche(8,[2,6,1,9,8,3,4,0,7])
Trouvé
>>> cherche(5,[2,6,1,9,8,3,4,0,7])
Non trouvé
```


jean-miche.ferrard@ac-paris.fr

4.7 Fonctions lambda

En Python, une *fonction lambda* est une fonction anonyme (à laquelle on n'a pas donné de nom), et qu'on peut appliquer "à la volée" dans une expression.

La syntaxe est : `lambda` *paramètres* : expression (remarque : `lambda` est un des mots réservés du langage).

Les fonctions lambda sont réservées à des situations relativement simples. Leur définition doit tenir sur une seule ligne, et elles ne peuvent pas contenir d'instructions composées (pas d'affectation, pas de boucle, etc.). Elles consistent donc essentiellement en la définition d'une expression calculée en fonction des paramètres qui lui sont passés.

Pour prendre un exemple simpliste (et pas très utile), les deux définitions suivantes de la fonction f sont équivalentes :

```
>>> def f(x,y,z):
        return 100*x+10*y+z
>>> f(1,2,3)
123
```

```
>>> f = lambda x,y,z: 100*x+10*y+z
>>> f(1,2,3)
123
>>>
```

On peut utiliser des fonctions anonymes dans des constructions avec `map` (où il s'agit d'appliquer une même fonction à tous les éléments d'une liste, par exemple) ou `filter` (où il s'agit cette fois de sélectionner des éléments répondant à un certain critère). Mais il est toujours possible de contourner l'utilisation des fonctions `lambda` en utilisant des "listes par compréhension" (cette notion sera abordée plus loin).

Voici par exemple deux façons de calculer la liste des carrés des dix premiers entiers positifs.

La deuxième méthode est plus élégante (et surtout plus naturelle) :

```
>>> list(map(lambda x: x*x, range(1,11)))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
>>> [x*x for x in range(1,11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Autre exemple, on forme la liste de tous les entiers de $[1, 20[$ qui ne sont pas divisibles par 3.

Là encore, la deuxième méthode est préférable :

```
>>> list(filter(lambda x: x%3,range(20)))
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

```
>>> [x for x in range(20) if x%3]
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

Les fonctions `lambda` conduisent souvent à un style un peu difficile à lire, et on vient de voir qu'on peut s'en passer.

On verra cependant un exemple un peu plus convaincant dans la section suivante.

4.8 Documentation des fonctions

Il est recommandé de placer des commentaires dans les fonctions qu'on écrit, et d'éviter un code trop dense.

Prenons un exemple. La fonction suivante trie une liste par la méthode d'insertion :

```
def tri_insertion(L):
    for i in range(1,len(L)):
        v=L[i]; j=i
        while j>0 and L[j-1]>v: L[j]=L[j-1]; j-=1
        L[j]=v
```

Le code précédent est trop dense, et non commenté, ce qui ne favorise pas sa relecture. Il est recommandé d'utiliser plutôt le style suivant (même si, pour ce qui est des commentaires, on force peut-être ici un peu le trait) :

```
def tri_insertion(L):            # trier une liste par insertion
    for i in range(1,len(L)):
        # pré-condition de ce passage dans la boucle for: L[0],...,L[i-1] sont triés
        v = L[i]                 #   il s'agit d'insérer v = L[i] parmi L[0],...,L[i-1]
        j = i;                   #   soit j la position d'insertion de v (par défaut j = i)
        # tant qu'on n'est pas arrivé au début de la liste
        # et tant que l'élément à gauche de L[j] est encore supérieur à la valeur v
        while j > 0 and L[j-1] > v:
            L[j] = L[j-1]        # alors on le décale à droite pour faire de la place
            j -= 1               # et on décrémente la (future) position d'insertion de v
        L[j] = v                 # on insère la valeur v à la position j
        # post-condition de ce passage dans la boucle for: L[0],...,L[i] sont triés
```


jean-miche.ferrard@ac-paris.fr

Voici un exemple d'utilisation de cette fonction `tri_insertion` (on importe la fonction `sample` du module `random`, pour créer ici une liste d'entiers différents d'un intervalle donné).

```
>>> from random import sample          # importe la fonction sample du module random
>>> seq = sample(range(10,101), 15)    # liste de 15 entiers différents dans [10,100]
>>> seq_backup = seq[:]                # copie de sauvegarde
>>> seq                                # voici la liste qu'on va trier
[82, 93, 28, 36, 22, 34, 24, 46, 73, 10, 35, 95, 83, 59, 30]
>>> tri_insertion(seq)                 # on appelle la fonction tri_insertion sur cette liste
>>> seq                                # la liste a été triée 'sur place'
[10, 22, 24, 28, 30, 34, 35, 36, 46, 59, 73, 82, 83, 93, 95]
>>> seq_backup                         # mais on a conservé l'original
[82, 93, 28, 36, 22, 34, 24, 46, 73, 10, 35, 95, 83, 59, 30]
```

La fonction `help` de `Python` permet d'afficher une aide succinte sur les fonctions intégrées.

Voici par exemple l'aide associée à la fonction `divmod` :

```
>>> help(divmod)
Help on built-in function divmod in module builtins:
divmod(...)
    divmod(x, y) -> (div, mod)
    Return the tuple ((x-x%y)/y, x%y).  Invariant: div*y + mod == x.
```

Incorporer des commentaires à une fonction, c'est très bien si on souhaite relire le code de celle-ci. Mais si on est seulement intéressé par l'utilisation de cette fonction, il est intéressant de pouvoir disposer d'une aide sur les arguments qu'elle attend, et sur la nature du résultat qu'elle renvoie.

La solution est d'utiliser une "chaîne de documentation" (*docstring* en anglais) placée juste après la ligne de définition, et délimitée par une paire de triples guillemets (ou de triples apostrophes). L'utilisation de ces délimiteurs triples permet de faire courir la chaîne de documentation sur plusieurs lignes.

À titre d'exemple, nous allons modifier la fonction `tri_insertion` en lui ajoutant un nouveau paramètre formel sous la forme d'une fonction pour personnaliser la façon dont la liste est triée.

```
def tri_insertion(L, f = lambda x: x):
    '''tri_insertion(L,f) trie la liste L dans l'ordre croissant
       des valeurs de la fonction f (par défaut, f est l'identité).'''
    for i in range(1,len(L)):
        j = i; v = L[i]
        while j > 0 and f(L[j-1]) > f(v):
            L[j] = L[j-1]; j -= 1
        L[j] = v
```

Une fois validée cette nouvelle définition, la *docstring* est affichée en tapant `help(tri_insertion)` :

```
>>> help(tri_insertion)
Help on function tri_insertion in module __main__:
tri_insertion(L, f=<function <lambda>>)
    tri_insertion(L,f) trie la liste L dans l'ordre croissant
    des valeurs de la fonction f (par défaut, f est l'identité).
```

Voici comment trier une liste d'entiers relatifs selon leurs valeurs absolues, ou par ordre décroissant. Cette nouvelle définition de `tri_insertion` et l'exemple du tri décroissant illustrent une utilisation commode des fonctions lambda :

```
>>> from random import sample               # importe la fonction sample du module random
>>> seq = sample(range(-99,100),10)         # liste de 10 entiers différents dans [-99,99]
>>> seq_backup = seq[:]; seq                # copie de sauvegarde, contenu initial de seq
[13, -5, -79, 68, -95, -59, 36, -14, -91, 30]
>>> tri_insertion(seq,abs); seq             # tri par valeurs absolues, et nouveau contenu de seq
[-5, 13, -14, 30, 36, -59, 68, -79, -91, -95]
>>> seq = seq_backup[:]                     # récupération de la sauvegarde
>>> tri_insertion(seq,lambda x: -x); seq    # tri décroissant, et nouveau contenu
[68, 36, 30, 13, -5, -14, -59, -79, -91, -95]
```


jean-miche.ferrard@ac-paris.fr

Chapitre 5

Les séquences (chaînes, tuples, listes)

5.1 Propriétés communes aux séquences (hors "mutations")

Il n'y a pas à proprement parler de type *séquence* en `Python`, mais on désigne comme tels les objets qui appartiennent à l'un des trois types suivants : les *chaînes de caractères*, les *listes*, et les *tuples* (il faudrait y ajouter les objets de type *bytes* ou *array bytes*, qui seront évoqués plus tard). Les *intervalles* (c'est-à-dire les objets créés par l'instruction `range`) peuvent aussi être considérés comme des séquences. Au-delà des différences entre ces types de données, les séquences ont un certain nombre de propriétés importantes en commun :

- elles sont composées d'un nombre fini d'éléments auxquels on peut accéder par un indice. Ainsi `seq[k]` désigne l'élément situé en position k dans la séquence *seq* (la numérotation commence à 0).
- un indice négatif signifie qu'on compte à partir de la fin. Ainsi `seq[-1]` désigne le dernier élément d'une séquence.
- on peut effectuer des *coupes* (le terme anglais est *slice*). Ainsi `seq[i:j]` désigne la séquence (chaîne, liste ou tuple) formée des éléments qui sont en position i (inclus) à j (exclu) dans la séquence *seq*
- on peut tester l'appartenance d'un élément à une séquence. La syntaxe est *elt* `in` *seq* (résultat booléen)
- on peut parcourir une séquence au sein d'une boucle `for`. La syntaxe est `for` *elt* `in` *seq*
- la longueur d'une séquence `seq` (le nombre d'éléments dont elle est constituée) est donnée par `len(seq)`

Dans le tableau ci-dessous, s et t désignent des séquences (chaînes, listes ou tuples), x désigne un objet pouvant appartenir à s, et on note i, j, k, n des entiers :

Opérations communes à tous les types de séquences			
`x in s`	True si x est dans s, False sinon	`x not in s`	False si x est dans s, True sinon
`s + t`	Concaténation de s et t	`s*n, n*s`	concatène n copies de s
`len(s)`	longueur de la séquence s	`s[i]`	le i-ème élément de s
`s[i:j]`	coupe de s pour indices $[i, j[$	`s[i:j:k]`	idem, mais avec un "pas" k
`min(s)`	le plus petit élément de s	`max(s)`	le plus grand élément de s
`s.index(x)`	le 1er indice où x est dans s	`s.count(x)`	le nombre de x dans s

Dans les pages qui suivent, on reviendra en détail sur l'utilisation spécifique des listes, des tuples et des chaînes de caractères, mais voici un bref aperçu :

- Les **chaînes de caractères** sont délimitées par des guillemets doubles ("abc") ou simples ('abc'). Elles sont constituées de caractères au format unicode (qui permet de représenter les lettres de tous les alphabets).
- Les **listes** sont des successions d'objets (eux-mêmes séparés par des virgules), délimitées par des crochets `[` et `]`. Elles peuvent contenir des objets de type quelconque. Un exemple de liste est `[1,'a',[3.14,'xyz']]`.
- Les **tuples** sont des successions d'objets (séparés par des virgules), délimitées par des parenthèses `(` et `)`.
  Les tuples peuvent contenir des objets de type quelconque. Un exemple de tuple est : `(1,'a',[3.14,'xyz'])`.

  Contrairement aux listes, les tuples ne sont pas *mutables* (voir plus loin). On ne peut donc pas modifier ou supprimer leurs éléments individuellement (on peut juste redéfinir un tuple dans sa totalité).

Remarquons que les fonctions intégrées `list`, `tuple` et `str` permettent de passer d'un type de séquence à l'autre :

```
>>> s = [1,[2,3],"abc"]           # une liste
>>> str(s)                        # on la transforme en chaîne
"[1, [2, 3], 'abc']"
>>> tuple(s)                      # on la transforme en tuple
(1, [2, 3], 'abc')
>>> t = ("xyz",(1,3.14),[0,2])    # un tuple
>>> str(t)                        # on le transforme en chaîne
"('xyz', (1, 3.14), [0, 2])"
>>> list(t)                       # on le transforme en liste
['xyz', (1, 3.14), [0, 2]]
>>> c = "abcdef"                  # une chaîne de caractères
>>> list(c)                       # on la transforme en la liste de ses caractères
['a', 'b', 'c', 'd', 'e', 'f']
>>> tuple(c)                      # on la transforme en le tuple de ses caractères
('a', 'b', 'c', 'd', 'e', 'f')
```

5.2 Séquences mutables ou non

Rappelons que quand on crée une variable, donc lorsqu'on lie un identificateur à un objet par l'instruction *nom=obj*, on associe en fait à cet identificateur l'adresse en mémoire où se trouve physiquement l'objet. On exprime cette situation en disant qu'on crée une *référence*, ou encore un *pointeur*, vers l'objet.

Il y a une différence importante entre les listes d'une part, et les chaînes et tuples de l'autre : les listes sont dites *mutables*, mais les chaînes et les tuples ne le sont pas.

Dire qu'un objet est *mutable*, c'est dire qu'on peut en modifier (voire en supprimer) un (ou plusieurs) élément(s), sans pour autant créer une nouvelle référence vers l'objet ainsi modifié (l'adresse du début de l'objet reste inchangée).

Par exemple, si on pose $x = y = [5, 2, 9]$, on définit deux variables x et y contenant en fait l'adresse d'une unique liste. On peut poser $x[2] = 10$: on *mute* donc la liste dont le contenu est maintenant $[5, 10, 9]$, mais dont l'adresse ne change pas en mémoire. Les variables x et y pointent donc toujours vers la même adresse, à laquelle on trouve la liste $[5, 10, 9]$ (pour l'utilisateur, tout se passe donc comme si la mutation de la liste x s'était répercutée sur la liste y).

Le caractère mutable des listes peut réserver quelques surprises. Considérons l'exemple suivant :

```
>>> x = [1,2,3]; id(x)      # la variable x contient en fait l'adresse de la liste [1,2,3]
4301154856
>>> y = [x,x,x,x]; id(y)    # la variable y contient l'adresse de la liste [x,x,x,x]
4359646312
>>> [id(e) for e in y]      # les éléments de la liste y pointent l'adresse de la liste [1,2,3]
[4301154856, 4301154856, 4301154856, 4301154856]
>>> x = [4,5,6]; id(x)      # on redéfinit le contenu de x. L'adresse change!
4359645952
>>> y                       # mais le contenu de la liste y n'a pas changé
[[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]
>>> [id(e) for e in y]      # normal car les adresses pointent toujours sur la liste [1,2,3]
[4301154856, 4301154856, 4301154856, 4301154856]
```

Reprenons l'exemple précédent, en le modifiant légèrement :

```
>>> x = [1,2,3]; id(x)      # la variable x contient l'adresse de la liste [1,2,3]
4359646384
>>> y = [x,x,x,x]; id(y)    # la variable y contient l'adresse de la liste [x,x,x,x]
4359646240
>>> [id(e) for e in y]      # les éléments de la liste y pointent l'adresse de la liste [1,2,3]
[4359646384, 4359646384, 4359646384, 4359646384]
>>> x[0] = 9; id(x)         # on mute la liste x, on ne la redéfinit pas: adresse inchangée!
4359646384
>>> y                       # toutes les composantes de y ont changé en même temps
[[9, 2, 3], [9, 2, 3], [9, 2, 3], [9, 2, 3]]
>>> [id(e) for e in y]      # normal car elles pointent la même adresse, où on a maintenant [9,2,3]
[4359646384, 4359646384, 4359646384, 4359646384]
```


jean-miche.ferrard@ac-paris.fr

Dans l'exemple suivant, on place dans la variable z une liste contenant quatre exemplaires de la liste $[1, 2, 3]$.

Dans un premier temps, on *redéfinit* l'élément $z[0]$. Cela n'a pas d'impact sur $z[1]$, $z[2]$ et $z[3]$.

Mais si on *mute* l'élément $z[1]$, par exemple, cette mutation se répercute sur les éléments $z[2]$ et $z[3]$ (car $z[1]$, $z[2]$ et $z[3]$ continuent à pointer sur la même adresse).

```
>>> z = [[1,2,3]] * 4            # on définit z par concaténation de quatre listes identiques
>>> z
[[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]
>>> z[0] = [5,6]                 # ici on redéfinit l'élément z[0]
>>> z                            # ça n'a pas d'influence sur z[1], z[2] et z[3]
[[5, 6], [1, 2, 3], [1, 2, 3], [1, 2, 3]]
>>> z[1][2] = 2013               # ici on mute l'élément z[1]
>>> z                            # ça se répercute sur z[2] et z[3]
[[5, 6], [1, 2, 2013], [1, 2, 2013], [1, 2, 2013]]
```

5.3 Listes définies "en compréhension"

Les listes peuvent être formées :

- en évaluant l'expression `list()` ou `[]`. On obtient la liste vide.
- en combinant des éléments `[elt`$_0$`, elt`$_1$`,..., elt`$_n$`]`, ou en convertissant une séquence par `list(seq)`.
- en "compréhension", par `[`*expr* `for` *indice* `in` *iterable*`]`, ou `[`*expr* `for` *indice* `in` *iterable* `if` *condition*`]`.
  La liste est ici formée des valeurs de *expr* quand *indice* parcourt *iterable* (et où *condition* est faculative). Un "itérable" est tout objet qui peut être traversé, parcouru : (les séquences, les ensembles, les dictionnaires...)
  L'expression suivante forme par exemple la liste des x^2 où $1 \leq x < 100$, en se limitant à $x \equiv 3$ modulo 10.

```
>>> [x*x for x in range(1,100) if x%10 == 3]
[9, 169, 529, 1089, 1849, 2809, 3969, 5329, 6889, 8649]
```

En fait, les listes peuvent être construites "en compréhension" d'une façon plus générale encore.

La syntaxe est alors

```
[ expression for indice_1 in iterable_1 [if condition_1]
              for indice_2 in iterable_2 [if condition_2]
              ...
              for indice_n in iterable_n [if condition_n] ]
```

Chacune des conditions est facultative (et la condition n° i s'applique à l'indice n° i).

Dans cette syntaxe, *expression* est évaluée en fonction des valeurs des n-uplets $(indice_1, indice_2, \ldots, indice_n)$ successifs (dans l'ordre lexicographique) et le résultat est la liste de ces évaluations de *expression*.

Il faut bien comprendre la chronologie : pour chacune des valeurs possibles de $indice_1$ (définies par $iterable_1$ et le test éventuel $condition_1$), on fait varier $indice_2$ (attention : $iterable_2$ et $condition_2$ peuvent dépendre de $indice_1$), puis (à $indice_1$ et $indice_2$ fixés) on fait varier $indice_3$, etc.

Ainsi la boucle sur $indice_1$ "contient" la boucle sur $indice_2$, qui contient elle-même, etc., jusqu'à la boucle sur $indice_n$.

Voici deux premiers exemples où une liste est formée par compréhension avec deux `for` imbriqués. On voit que cela produit deux variables locales i et j, et que la première boucle "contient" la deuxième. Cette chronologie est importante, comme on le voit sur le troisième exemple, qui conduit à une erreur.

```
>>> [100*i+j for i in range(1,5) for j in range(1,4)]
[101, 102, 103, 201, 202, 203, 301, 302, 303, 401, 402, 403]
>>> [100*i+j for i in range(1,5) for j in range(1,i+1)]
[101, 201, 202, 301, 302, 303, 401, 402, 403, 404]
>>> [100*i+j for j in range(1,i+1) for i in range(1,5)]
<...>
NameError: name 'i' is not defined
```

Voici quatre autres exemples, qui doivent être soigneusement comparés avec les précédents. Ici la liste calculée s'écrit `[`*expression* `for` *indice* `in` *intervalle*`]`, où *expression* est elle-même définie comme une liste en compréhension. La boucle permettant de calculer *expression* est parcourue "sous le contrôle" de la boucle principale.

jean-miche.ferrard@ac-paris.fr

```
>>> [[100*i+j for i in range(1,5)] for j in range(1,4)]
[[101, 201, 301, 401], [102, 202, 302, 402], [103, 203, 303, 403]]
>>> [[100*i+j for j in range(1,4)] for i in range(1,5)]
[[101, 102, 103], [201, 202, 203], [301, 302, 303], [401, 402, 403]]
>>> [[100*i+j for j in range(1,i+1)] for i in range(1,5)]
[[101], [201, 202], [301, 302, 303], [401, 402, 403, 404]]
>>> [[100*i+j for i in range(1,5)] for j in range(1,i+1)]
<...>
NameError: name 'i' is not defined
```

L'utilisation de listes en compréhension permet des constructions élégantes. L'imbrication des définitions en compréhension peut cependant conduire à des formulations assez délicates à relire, comme l'atteste cette fonction qui renvoie la liste des facteurs premiers strictement inférieurs à n^2, où n est l'entier passé en argument :

```
def cribleobscur(n):
    return [p for p in range(2,n*n) if p not in [j for i in range(2,n) for j in range(2*i,n*n,i)]]
```

```
>>> cribleobscur(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

5.4 Opérations de mutation de listes

Dans le tableau suivant, s est une séquence *mutable* (une liste, a priori), et x est un objet qu'on écrit, ajoute, supprime ou recherche dans s. On note t un objet "itérable" dont les éléments sont écrits dans s ou ajoutés à la fin de s.

Opérations communes aux séquences mutables, donc applicables aux listes			
`s[i] = x`	remplace $s[i]$ par x	`del s[i]`	supprime l'élément $s[i]$
`s[i:j] = t`	remplace $s[i], \ldots, s[j-1]$ par les élts de t	`s[i:j:k] = t`	idem mais avec le pas k
`del s[i:j]`	supprime $s[i], s[i+1], ..., s[j-1]$	`del s[i:j:k]`	idem mais avec le pas k
`s.append(x)`	ajoute un élément x à s	`s.clear()`	efface tous les éléments de s
`s.copy()`	copie indépendante de s (idem `s[:]`)	`s.extend(t)`	ajoute les élts de l'itérable t à s
`s.insert(i,x)`	insère x en position i dans s	`s.pop(i)`	supprime et renvoie le $i^{\text{ème}}$ élt de s
`s.remove(x)`	supprime la $1^{\text{ère}}$ occurence de x dans s	`s.reverse()`	inverse l'ordre des éléments de s

Attention : la plupart des instructions précédentes renvoient la valeur `None` mais *mutent* la liste (sans changer son adresse). Prenons quelques exemples, en plaçant une même liste dans s et t, et en faisant une copie "fraîche" de cette liste dans u. Les modifications sur s se répercutent dans t mais pas dans u :

```
>>> s = t = list(range(10,20)); t            # place dans s et t la même liste (même adresse!)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> u = s.copy(); u                          # met dans u une copie indépendante de l'original
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> s[3] = 'coucou'; t                       # remplace s[3], modification répercutée sur t
[10, 11, 12, 'coucou', 14, 15, 16, 17, 18, 19]
>>> del s[3]; t                              # supprime s[3], modification répercutée sur t
[10, 11, 12, 14, 15, 16, 17, 18, 19]
>>> s.insert(3,0); t                         # insère 0 en position 3 dans s (idem dans t)
[10, 11, 12, 0, 14, 15, 16, 17, 18, 19]
```

À la suite des instructions précédentes, on voit comment utiliser des coupes (*slices*) de la liste s :

```
>>> s[1:9:2]                                 # lit une coupe de longueur 4 dans s
[11, 0, 15, 17]
>>> s[1:9:2] = 'abcd'; t                     # remplace ces 4 élts par ceux de 'abcd'
[10, 'a', 12, 'b', 14, 'c', 16, 'd', 18, 19]
>>> del s[1:5]; t                            # efface les éléments de positions 1 à 4
[10, 'c', 16, 'd', 18, 19]
```


jean-miche.ferrard@ac-paris.fr

Terminons cette série d'exemples par quelques manipulations supplémentaires sur la liste s :

```
>>> s.append('xyz'); t                          # ajoute l'objet 'xyz' à la fin de la liste
[10, 'c', 16, 'd', 18, 19, 'xyz']
>>> s.extend('uvw'); t                          # complète la liste par les éléments de 'uvw'
[10, 'c', 16, 'd', 18, 19, 'xyz', 'u', 'v', 'w']
>>> s.pop(5)                                    # renvoie s[5], et le supprime de la liste
19
>>> s.remove(16); t                             # supprime la première occurence de la valeur 16
[10, 'c', 'd', 18, 'xyz', 'u', 'v', 'w']
>>> s.reverse(); t                              # inverse la liste s (opération 'sur place')
['w', 'v', 'u', 'xyz', 18, 'd', 'c', 10]
>>> u                                           # pendant ce temps-là, l'original n'a pas changé
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Pour trier "sur place" la liste contenue dans une variable s, on écrit `s.sort()`, et pour obtenir une copie triée de cette liste, mais sans affecter l'original, on évalue l'expression `sorted(s)`. Dans tous les cas, il y a deux arguments facultatifs : `key=`*fonction* (défaut `key=None`) pour préciser une clef de tri, et `reverse=True/False` (défaut `reverse=False`) pour inverser l'ordre du tri.

```
>>> from random import sample                  # importe la fonction sample du module random
>>> s = sample(range(100,1000),10); s          # liste de 10 entiers différents à trois chiffres
[831, 348, 562, 879, 130, 864, 758, 886, 756, 355]
>>> sorted(s)                                  # renvoie la liste triée
[130, 348, 355, 562, 756, 758, 831, 864, 879, 886]
>>> s                                          # mais l'original n'a pas été modifié
[831, 348, 562, 879, 130, 864, 758, 886, 756, 355]
>>> s.sort(); s                                # ici le tri s'effectue sur place
[130, 348, 355, 562, 756, 758, 831, 864, 879, 886]
```

5.5 Les tuples

Les tuples sont des séquences non mutables d'objets séparés par une virgule, et délimitées par des parenthèses `(` et `)`.

Les tuples peuvent être formés :

- en évaluant l'expression `tuple()` ou `()`. On obtient le tuple vide.
- en combinant des éléments `(`$\texttt{elt}_0$`,` $\texttt{elt}_1$`,...,` $\texttt{elt}_n$`)`, ou en convertissant une séquence par `tuple(seq)`.
- en "compréhension", par `tuple(`*expr* `for` *indice* `in` *iterable*`)`, ou `(`*expr* `for` *indice* `in` *iterable* `if` *condition*`)` ;

  on peut également utiliser la syntaxe plus générale suivante, (cf "listes en compréhension").

```
tuple( expression for indice_1 in iterable_1 [if condition_1]
                  for indice_2 in iterable_2 [if condition_2]
                  ...
                  for indice_n in iterable_n [if condition_n] )
```

Quelques remarques :

- Les parenthèses aux extrémités sont facultatives (l'important, ce sont les virgules) mais recommandées pour la lisibilité.

  Pour former un tuple à un seul élément, il faut faire suivre cet élément d'une virgule.
- Les opérations sur les séquences non mutables s'appliquent aux tuples : appartenance (avec `in` t et `not in` t), concaténation (avec $t+t'$) et répétition (avec t `*` n), longueur avec `len(`t`)`, accès indexés et coupes (avec $t[i]$, $t[i:j]$, $t[i:j:k]$), minimum/maximum (`max` et `min`), recherche de valeur x par `index.`t`(`x`)` et occurences par t`.count(`x`)`.

  Les tuples sont indiqués pour "packer" ensemble des données, si on ne souhaite pas les modifier individuellement (ce qui serait de toutes façons impossible ici pour cause de "non mutabilité") ni augmenter ou diminuer leur nombre.

  Tout comme les listes et les chaînes, les tuples sont *itérables*, et peuvent donc être parcourus par une boucle `for`.

jean-miche.ferrard@ac-paris.fr

Commençons par un premier exemple de tuple défini "en extension" :

```
>>> t = (123, 'abc', 3.14); type(t)          # on définit un tuple de longueur 3
<class 'tuple'>
>>> t[1]                                     # on accède en lecture à l'élément en position 1
'abc'
>>> t[1] = 'uvw'                             # impossible de modifier un élément !!!
<...> TypeError: 'tuple' object does not support item assignment
```

Voici quelques exemples de manipulations d'un tuple t défini en compréhension :

```
>>> t = tuple(x*x for x in range(1,9)); t    # un tuple défini en compréhension
(1, 4, 9, 16, 25, 36, 49, 64)
>>> t[::-1]                                  # le même tuple, mais à l'envers
(64, 49, 36, 25, 16, 9, 4, 1)
>>> id(t)                                    # voici l'adresse de t en mémoire
4361652568
>>> t = t + (2013,); t                       # on ajoute l'élément 2013 (NB la virgule!)
(1, 4, 9, 16, 25, 36, 49, 64, 2013)
>>> id(t)                                    # mais attention, c'est un tout nouvel objet!!!
4361652432
>>> t[1:6:2]                                 # le tuple (t[1],t[3],t[5])
(4, 16, 36)
>>> t[-1]                                    # le dernier élément du tuple t
2013
```

Terminons par quelques opérations possibles sur les tuples :

```
>>> (1,2,3)+(4,5)                            # concaténation de deux tuples
(1, 2, 3, 4, 5)
>>> (1,2,3),(4,5)                            # on forme un '2uple' (un '3uple' puis un '2uple')
((1, 2, 3), (4, 5))
>>> (0,)*5                                   # le '1uple' (0) répété 5 fois
(0, 0, 0, 0, 0)
>>> x,y,z = (5,6,7)                          # on 'dépacke' un tuple
>>> z, y, x                                  # on repacke, mais en changeant l'ordre
(7, 6, 5)
>>> tuple('pqrs')                            # transforme 'pqrs' (chaîne donc itérable) en tuple
('p', 'q', 'r', 's')
>>> tuple([1,2,3,4])                         # transforme une liste (donc itérable) en tuple
(1, 2, 3, 4)
```

5.6 Les chaînes de caractères

Les chaînes sont des séquences de caractères non mutables.

On ne peut donc pas modifier un ou plusieurs caractères. Si on ajoute un caractère (ou une autre chaîne) par concaténation à une chaîne existante, le résultat est la création d'une nouvelle chaîne à une nouvelle adresse en mémoire.

Les chaînes sont délimitées par des guillemets simples `'bonjour'`) doubles `"bonjour"`) ou triples (`'''bonjour'''`).

L'utilisation de guillemets triples permet de faire courir une chaîne de caractères sur plusieurs lignes, comme on l'a vu dans la section consacrée aux "chaînes de documentation" (*docstrings*).

La possibilité de choisir entre les délimiteurs simples ou doubles permet d'insérer des guillemets dans une chaîne (on pourra par exemple former la chaîne "c'est l'automne").

Les opérations sur les séquences non mutables s'appliquent aux chaînes :

- appartenance (avec `in` ch et `not in` ch),
- concaténation (avec $ch + ch'$) et répétition (avec $ch * n$),
- longueur avec `len(`ch`)`, accès indexé et coupes (avec $ch[i]$, $ch[i:j]$, $ch[i:j:k]$)
- minimum/maximum (`max` et `min`)
- recherche de x par ch`.index(`x`)` et occurences par ch`.count(`x`)`.

Tout comme les listes et les tuples, les chaînes sont *itérables*, et peuvent donc être parcourues dans une boucle `for`.
Comme on le voit ici, on concatène des chaînes avec l'opérateur `+` (ou un simple espace), et on les répète par `*` :

```
>>> 'abc' + 'uvwxyz'                          # concaténation de deux chaînes
'abcuvwxyz'
>>> 'abc' * 5                                 # répétition d'une même chaîne
'abcabcabcabcabc'
```

Voici quelques exemples d'opérations possibles sur les chaînes de caractères :

```
>>> st = 'abxyztcabaabaabcabw'                # définition d'une chaîne
>>> for c in st: print(ord(c),end=' ')        # boucle d'affichage des codes des caractères
97 98 120 121 122 116 99 97 98 97 97 98 97 97 98 99 97 98 119
>>> len(st), st[2:5], st[-1]                  # lecture de caractères par leurs indices
(19, 'xyz', 'w')
>>> st[::-1]                                  # inverser l'ordre des caractères
'wbacbaabaabactzyxba'
>>> ('abc' in st, 'abcd' in st, 'ab' not in st)  # appartenance ou non
(True, False, False)
>>> st.count('aba'), st.index('aba')          # nombre d'occurences, première occurence
(2,7)
>>> st.index('aba',8)                         # 1ère occurence à partir de la position 8
10
>>> st[2]='h'                                 # impossible de modifier un caractère
<...> TypeError: 'str' object does not support item assignment
>>> list('abcbde')                            # la liste des caractères d'une chaîne
['a', 'b', 'c', 'b', 'd', 'e']
```

Les chaînes possèdent un très grand nombre de *méthodes* qui leur sont propres ; ces méthodes ont souvent un comportement par défaut qui peut être personnalisé en précisant la valeur de paramètres facultatifs. Comme il est impossible de tout citer, voici une simple sélection :
(pour plus de détails consulter : http://docs.python.org/3.3/library/stdtypes.html#text-sequence-type-str)
Voir également la section suivante pour une analyse détaillée de certaines méthodes importantes.

str`.endswith(`*suffix*`)`	renvoie `True` si la chaîne *str* se termine par la chaîne *suffix*
str`.startswith(`*prefix*`)`	renvoie `True` si la chaîne *str* commence par la chaîne *prefix*
str`.isalnum()`	renvoie `True` si *str* est formée uniquement de caractères alphanumériques
str`.isalpha()`	renvoie `True` si *str* est formée uniquement de caractères alphabétiques
str`.isdigit()`	renvoie `True` si *str* est formée uniquement de chiffres
str`.lower()`	renvoie une chaîne obtenue par passage en minuscules
str`.upper()`	renvoie une chaîne obtenue par passage en majuscules
str`.strip()`	renvoie une chaîne obtenue en supprimant les blancs au début et à la fin
str`.replace(`*old*`,`*new*`[,`*n*`])`	renvoie une chaîne obtenue en remplaçant *old* par *new*, au plus *n* fois
str`.find(`*sub*`)`	indice de la première occurence de *sub* (et -1 si non trouvé)
str`.index(`*sub*`)`	comme `find`, mais erreur `ValueError` si non trouvé
str`.rfind(`*sub*`)`	indice de la dernière occurence de *sub* (et -1 si non trouvé)
str`.center(`*n*`[,`*c*`])`	renvoie *str* centrée dans une chaîne de longueur *n*, bordée par des *c*
str`.ljust(`*n*`[,`*c*`])`	renvoie *str* justifiée à gauche dans une chaîne de longueur *n*, complétée par des *c*
str`.rjust(`*n*`[,`*c*`])`	renvoie *str* justifiée à droite dans une chaîne de longueur *n*, complétée par des *c*

jean-miche.ferrard@ac-paris.fr

5.7 Méthodes importantes sur les chaînes (split, join, format)

– La méthode **split** permet de découper une chaîne en une liste de sous-chaînes, en effectuant les coupures sur des caractères bien précis (par défaut les espaces)

```
>>> t = "être ou ne pas être, là est la question"
>>> t.split()                # on découpe la chaîne sur les espaces (comportement par défaut)
['être', 'ou', 'ne', 'pas', 'être,', 'là', 'est', 'la', 'question']
>>> t.split(sep=',')         # ici on découpe sur les virgules (il n'y en a qu'une)
['être ou ne pas être', ' là est la question']
>>> t.split('être')          # on peut choisir une chaîne comme séparateur
['', ' ou ne pas ', ', là est la question']
>>> t.split(maxsplit=4)      # ici, on autorise au plus quatre coupures
['être', 'ou', 'ne', 'pas', 'être, là est la question']
>>> t="   plein de   vide   dans cette    chaîne   "
>>> t.split()                # dans un split.() tous les espaces sont supprimés.
['plein', 'de', 'vide', 'dans', 'cette', 'chaîne']
>>> t="--plein-de--tirets---dans-cette--chaîne--"
>>> t.split('-')            # ça n'est pas pareil si on splite sur un autre caractère
['', '', 'plein', 'de', '', 'tirets', '', '', 'dans', 'cette', '', 'chaîne', '', '']
```

– La méthode **join** est l'inverse de la méthode **split**. L'expression *sep*.join(*iterable*) regroupe les chaînes éléments de l'objet *iterable* (une liste de chaînes, par exemple), en utilisant la chaîne *sep* comme séparateur.

```
>>> ''.join(['abc','def','ghi'])       # joint les chaînes de la liste, avec un séparateur vide
'abcdefghi'
>>> '*'.join(['abc','def','ghi'])      # idem mais en séparant par *
'abc*def*ghi'
>>> '-?-'.join(['abc','def','ghi'])    # on peut séparer par une chaîne quelconque
'abc-?-def-?-ghi'
```

La méthode **join** doit être considérée comme une propriété de la chaîne qui est spécifiée comme séparateur.

La méthode **join** est économique en mémoire et doit être préférée à une boucle mettant les différentes chaînes bout à bout (complexité linéaire plutôt que quadratique).

– La méthode **format** permet de formater une chaîne (pour l'afficher, le plus souvent), en utisant un canevas contenant des champs (qui servent à spécifier le format) et des arguments pour renseigner ces champs.
La syntaxe est *canevas*.format(*arguments*). Les champs de formatage (dans la chaîne *canevas*) sont délimités par des accolades, et chaque champ est renseigné par l'argument qui lui correspond. Il y a énormément de variantes !

On trouvera une description détaillée ici : http://docs.python.org/3.3/library/string.html#formatstrings

On trouvera des exemples ici : http://docs.python.org/3.3/library/string.html#formatexamples

Voici quelques exemples simples :

```
>>> canevas = 'Nom: {}, prénom: {}, date de naissance: {}'
>>> canevas.format('William','Shakespeare',1613)
'Nom: William, prénom: Shakespeare, date de naissance: 1613'
>>> canevas = 'Nom: {n}, prénom: {p}, date de naissance: {d}'
>>> canevas.format(d=1613,p='William',n='Shakespeare')
'Nom: Shakespeare, prénom: William, date de naissance: 1613'
>>> canevas = 'Le nombre décimal {0:d} s'écrit {0:b} en binaire et {0:x} en hexadécimal'
>>> canevas.format(2013)
"Le nombre décimal 2013 s'écrit 11111011101 en binaire et 7dd en hexadécimal"
>>> canevas = 'On sait bien que {0}+{1}+{2} = {1}+{0}+{2} = {0}+{2}+{1}'
>>> canevas.format(15,87,23)
'On sait bien que 15+87+23 = 87+15+23 = 15+23+87'
>>> canevas = '{0}/{1} vaut {2:.2f} (2 décimales) et {2:.7f} (7 décimales)'
>>> x, y = 20132013, 71; canevas.format(x,y,x/y)
'20132013/71 vaut 283549.48 (2 décimales) et 283549.4788732 (7 décimales)'
```


jean-miche.ferrard@ac-paris.fr

5.8 Objets de type bytes et bytearray

On sait que le code "Ascii" (American Standard Code for Information Interchange) permet de représenter, dans un registre qui va de 0 à 127, les caractères de l'alphabet anglais (ainsi que quelques caractères non imprimables) et qu'il a été complété (pour un registre de codes allant de 128 à 255) dans différents formats pour les caractères accentués (notamment le code "ISO 8859-1", ou encore "Latin-1", utilisé pour les langues d'Europe de l'ouest).

Le code "Ascii étendu" identifie donc caractères et octets (valeurs binaires sur 8 bits, de 0 à $2^8 - 1 = 255$).

Pour coder tous (?) les caractères et symboles de tous (?) les alphabets (par exemple les caractères des langues d'Asie), il a fallu mettre au point une norme mondiale, dite "Unicode".

Les chaînes de caractères Python sont des séquences non mutables de caractères, et elle utilisent le format Unicode.

Cette universalité se paie du prix qui est la taille suivant laquelle chaque caractère est codé. Mais Python est suffisamment optimisé pour qu'on n'ait pas à s'en soucier. On remarque simplement que la taille d'une chaîne de caractères peut dépendre de la nature de ceux-ci, comme le montrent les exemples suivants.

```
>>> from sys import getsizeof    # la fonction getsizeof permet de calculer une 'empreinte mémoire'
>>> getsizeof("abcdef")          # six caractères non accentués
55
>>> getsizeof("àbcdef")          # on a remplacé la lettre 'a' par la lettre accentuée 'à'
79
>>> getsizeof("αβγδϵθ")          # six caractères de l'alphabet grec
86
```

Les bytes ressemblent aux chaînes de caractères : ils sont des séquences non mutables d'entiers compris entre 0 et 255 (c'est-à-dire codés sur un octet, "byte" en anglais). On peut tout à fait les considérer comme des chaînes de caractères, à condition de se limiter aux caractères du code ascii standard (pas de caractère accentué).

On peut les créer avec le préfixe b (suivi d'une chaîne alphabétique) ou par la fonction bytes (appliquée à un "iterate").

```
>>> B = b'xyztu'; C = bytes(range(65,91))  # les deux objets B et C sont de type bytes
>>> (B, C)                                 # voila comment Python les affiche
(b'xyztu', b'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
>>> B[0], list(B), B[::-1], C[0::5]        # on peut y accéder par index ou par coupe
(120, [120, 121, 122, 116, 117], b'utzyx', b'AFKPUZ')
>>> D = bytes(range(0,20)); D              # le bytes des entiers de 0 à 19 (affichés en hexa)
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13'
>>> list(D)                                # on convertit ce bytes en liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

On peut appliquer aux "bytes" les fonctions communes sur les séquences non mutables. Comme ils s'apparentent étroitement aux chaînes de caractères, ils possèdent également la plupart des méthodes de la classe string.

Les objets de type bytearray sont des tableaux mutables d'entiers compris entre 0 et 255.

On peut appliquer aux objets de type bytearray les fonctions communes sur les séquences non mutables, mais également les méthodes spécifiques aux séquences mutables (c'est-à-dire les méthodes applicables aux listes). Comme ils s'apparentent étroitement aux chaînes de caractères, ils possèdent également la plupart des méthodes de la classe string.

```
>>> B = bytearray(10); B              # un "bytearray" de taille 10, dont tous les élts sont nuls
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> for k in range(0,9,2): B[k]=65+k  # dans cette boucle, on modifie certains éléments de B
>>> B                                 # affiche le nouveau contenu de B
bytearray(b'A\x00C\x00E\x00G\x00I\x00')
>>> list(B)                           # convertit ce contenu en une liste
[65, 0, 67, 0, 69, 0, 71, 0, 73, 0]
>>> B.reverse()                       # l'objet B, mutable, possède la méthode reverse
>>> list(B)                           # on le vérifie en convertissant B en liste
[0, 73, 0, 71, 0, 69, 0, 67, 0, 65]
>>> B.append(96); list(B)             # nouvelle preuve de la mutabilité d'un bytearray
[0, 73, 0, 71, 0, 69, 0, 67, 0, 65, 96]
```

On retiendra que les objets de type bytes et bytearray sont adaptés au traitement des informations sur des entiers de l'intervalle [0,255]. Pour plus d'information, on pourra évaluer help(bytes) et help(bytearray).

jean-miche.ferrard@ac-paris.fr

Chapitre 6

Dictionnaires, ensembles, itérateurs, générateurs, fichiers

6.1 Dictionnaires

Les *dictionnaires* sont des structures mutables, non ordonnées, formées d'enregistrements du type `clé:valeur`

Le seul moyen d'accéder à une *valeur* particulière est par l'intermédiaire de sa *clé*.

Les dictionnaires peuvent être formés :

- en évaluant l'expression `dict()` ou `{}`. On obtient le dictionnaire vide
- en délimitant par { et } une séquence de paires `clé:valeur` : `{clé`$_1$`:val`$_1$`, clé`$_2$`:val`$_2$`,..., clé`$_n$`:val`$_n$`}`
- en évaluant par exemple : `dict([[clé`$_1$`,val`$_1$`],[clé`$_2$`,val`$_2$`],...,[clé`$_n$`,val`$_n$`]])`
- en ajoutant des paires `clé:valeur` à un dictionnaire existant. On a ici une différence essentielle avec les listes, pour lesquelles il est seulement possible de modifier un élément existant ou d'ajouter un élément supplémentaire à la *fin* de la liste avec la méthode `append`
- en "compréhension", par exemple `D = {x:x*2 for x in range(10)}`

Voici quelques méthodes applicables à un objet *dic* de type dictionnaire. On note ici *cle* une clé et *val* une valeur :

`len(`*dic*`)`	renvoie la longueur du dictionnaire (nombres de clés)
dic`[`*cle*`]`	renvoie la valeur associée à la clé (lève `KeyError` si la clé est absente)
dic`[`*cle*`] =` *val*	crée (ou modifie) une paire clé/valeur
`del` *dic*`[`*cle*`]`	efface une paire clé/valeur
cle `in` *dic*	renvoie `True` si la clé est présente dans le dictionnaire, `False` sinon
cle `not in` *dic*	renvoie `True` si la clé est absente du dictionnaire, `False` sinon
dic`.clear()`	efface le contenu du dictionnaire
dic`.copy()`	renvoie une copie du dictionnaire, indépendante de l'original
dic`.get(`*cle*`)` *dic*`.get(`*cle*`,`*def*`)`	renvoie la valeur associée à la clé, et renvoie `None` si la clé est absente. renvoie la valeur associée à la clé, et la valeur de *def* si la clé est absente
dic`.items()`	renvoie un itérable pour décrire les couples (clé,valeur) dans une boucle `for`
dic`.keys()`	renvoie un itérable pour décrire les clés du dictionnaire dans une boucle `for`
dic`.pop(`*cle*`)` *dic*`.pop(`*cle*`,`*def*`)`	renvoie la valeur, et supprime la paire clé/valeur (`KeyError` si clé absente) renvoie la valeur, et supprime la paire clé/valeur (évalue *def* si clé absente)
dic`.popitem()`	Renvoie une paire arbitraire (clé,valeur) et la supprime de *dic* (`KeyError` si *dic* vide)
dic`.setdefaut(`*cle*`)` *dic*`.setdefaut(`*cle*`,`*def*`)`	renvoie la valeur si la clé est présente, sinon crée le couple *cle*`:None` renvoie la valeur si la clé est présente, sinon crée le couple *cle*`:`*def*
dic`.update(`*dic*2`)`	fusionne le dictionnaire *dic*2 sur le dictionnaire *dic* (remplace les valeurs si clés homonymes). Ici *dic*2 peut aussi être une liste de listes `[`*cle*`,`*valeur*`]`
dic`.values()`	renvoie un itérable pour décrire les valeurs du dictionnaire dans une boucle `for`

Les dictionnaires sont optimisés pour un accès aux valeurs par un "hachage" rapide sur les clés.

L'utilisateur n'a aucun moyen de savoir dans quel ordre les couples clés/valeurs sont placés dans le dictionnaire (et cette question est en fait sans importance).

À ce titre, les dictionnaires ne sont pas des séquences (ils ne peuvent pas être parcourus par un indice de position). Ils sont cependant considérés comme des objets "itérables" (on peut donc les parcourir).

Si les valeurs placées dans un dictionnaire peuvent être quelconques (et par exemple être elles-mêmes des dictionnaires), les clés utilisées pour accéder à ces valeurs doivent être d'un type non mutable (entiers, chaînes, par exemple).

Dans un dictionnaire donné, les clés doivent être uniques, mais elles ne doivent pas nécessairement être du même type.

Voici un exemple (très simple !) de dictionnaire (les clés sont des noms, les valeurs des âges...) :

```
>>> ages = {'Paul':41, 'Léon':25, 'Jeanne':134}  # crée un dictionnaires de trois entrées
>>> ages                                         # l'ordre affiché est non prévisible
{'Paul': 41, 'Jeanne': 134, 'Léon': 25}
>>> 'Paul' in ages, 'Jean' in ages               # teste la présence de deux clés
(True, False)
>>> ages['Jean']                                 # demander l'âge de Jean conduit à une erreur
<...> KeyError: 'Jean'
>>> ages.get('Jean',-1)                          # ici pas d'erreur, mais une valeur par défaut
-1
>>> ages['Marc'] = 33                            # crée un nouvel enregistrement
>>> del ages['Paul']                             # supprime un enregistrement
```

On continue à la suite de l'exemple précédent...

```
>>> for elt in ages: print(elt)                  # itérer un dictionnaire, c'est itérer les clés
Marc
Jeanne
Léo
>>> for (n,a) in ages.items():                   # voici comment itérer sur les paires (clé,valeur)
        print('Son prénom est {} et son âge est {} ans'.format(n,a))
Son prénom est Marc et son âge est 33 ans
Son prénom est Jeanne et son âge est 134 ans
Son prénom est Léon et son âge est 25 ans
>>> list(ages)                                   # la conversion ne donne ici que les clés
['Marc', 'Jeanne', 'Léon']
>>> list(ages.keys())                            # autre façon d'obtenir la liste des clés
['Marc', 'Jeanne', 'Léon']
>>> list(ages.values())                          # la liste des valeurs
[33, 134, 25]
>>> max(ages.values())                           # la valeur maximum
134
>>> [(n,a) for n,a in ages.items()]              # voici la liste des enregistrements
[('Marc', 33), ('Jeanne', 134), ('Léon', 25)]
```


jean-miche.ferrard@ac-paris.fr

6.2 Ensembles

Les objets de type "ensemble" (*set* dans le language `Python`) sont des structures de données qui modélisent la notion mathématique d'ensemble. Un tel objet peut contenir des valeurs de type quelconque, mais une même valeur ne peut apparaître qu'une seule fois. De plus les éléments d'un objet de type ensemble ne sont pas ordonnés (on ne peut pas y accéder par un indice : on peut juste savoir si une valeur est ou n'est pas élément de l'ensemble).

On forme un ensemble par la séquence de ses éléments (valeurs), encadrée par { et }, ou en utilisant le constructeur `set` (appliqué à un itérable quelconque).

```
>>> e = {1,5,2,3,2,7,5,2,1,3,2}        # forme un ensemble par une séquence de valeurs entre { et }
>>> e = set([1,5,2,3,2,7,5,2,1,3,2])   # même résultat en convertissant une liste
>>> e                                  # dans un ensemble, tous les doublons ont été éliminés
{1, 2, 3, 5, 7}
>>> set('abracadabra')                 # l'ensemble des caractères distincts d'une chaîne
{'d', 'r', 'a', 'c', 'b'}
```

Les objets de type "ensemble" sont mutables. Voici quelques méthodes applicables à un objet *ens* de type ensemble (*set*). On note *elt* une valeur susceptible d'appartenir ou d'être ajoutée à l'ensemble *ens* :

`len(`*ens*`)`	renvoie le cardinal (le nombre d'éléments) de l'ensemble
ens`.add(`*elt*`)`	ajoute un élément à un ensemble
ens`.remove(`*elt*`)` *ens*`.discard(`*elt*`)`	retire un élément à un ensemble (`KeyError` si élément absent) retire un élément à un ensemble s'il y est effectivement (pas d'erreur sinon)
elt `in` *ens*	renvoie `True` si l'élément est présent dans l'ensemble, `False` sinon
elt `not in` *ens*	renvoie `True` si l'élément est absent de l'ensemble, `False` sinon
ens`.clear()`	efface le contenu de l'ensemble
ens`.copy()`	renvoie une copie de l'ensemble, indépendante de l'original
ens`.pop()`	renvoie et supprime un élement arbitraire de l'ensemble (`KeyError` si ensemble vide)

On continue sur la lancée de l'exemple précédent (avec l'ensemble $e = \{1, 2, 3, 5, 7\}$) :

```
>>> e.discard(5)          # on retire l'élément 5 (pas d'erreur s'il avait été absent)
>>> e
{1, 2, 3, 7}
>>> e.add(0)              # on ajoute l'élément 0
>>> e
{0, 1, 2, 3, 7}
>>> 4 in e                # on demande si 4 est dans l'ensemble
False
```

Les objets de type "ensemble" ont des méthodes pour les opérations ensemblistes usuelles :

ens_1`.isdisjoint(`ens_2`)`	renvoie `True` si ens_1 et ens_2 sont disjoints
ens_1 `<=` ens_2	renvoie `True` si ens_1 est inclus dans ens_2
ens_1 `<` ens_2	renvoie `True` si ens_1 est inclus strictement dans ens_2
ens_1 `>=` ens_2	renvoie `True` si ens_1 contient ens_2
ens_1 `>` ens_2	renvoie `True` si ens_1 contient strictement ens_2
ens_1 `|` ens_2 `|` ens_3 `|` ...	renvoie l'union des ensembles
ens_1 `&` ens_2 `&` ens_3 `&` ...	renvoie l'intersection des ensembles
ens_1 `-` ens_2	renvoie la différence ensembliste $ens_1 \setminus ens_2$
ens_1 `^` ens_2	renvoie la différence symétrique $ens_1 \Delta ens_2$

jean-miche.ferrard@ac-paris.fr

Voici quelques exemples (on voit que l'ordre des éléments d'un ensemble est imprévisible) :

```
>>> m3 = set(range(0,50,3)); m3     # les multiples de 3 dans l'intervalle [0,50[
{0, 33, 3, 36, 6, 39, 9, 42, 12, 45, 15, 48, 18, 21, 24, 27, 30}
>>> m5 = set(range(0,50,5)); m5     # les multiples de 5 dans l'intervalle [0,50[
{0, 35, 5, 40, 10, 45, 15, 20, 25, 30}
>>> m7 = set(range(0,50,7)); m7     # les multiples de 7 dans l'intervalle [0,50[
{0, 35, 7, 42, 14, 49, 21, 28}
>>> m5 | m7                         # union des multiples de 5 ou de 7
{0, 35, 5, 7, 40, 10, 45, 14, 15, 49, 20, 21, 25, 28, 42, 30}
>>> sorted(m5 | m7)                 # renvoie une liste (pas un ensemble!) triée
[0, 5, 7, 10, 14, 15, 20, 21, 25, 28, 30, 35, 40, 42, 45, 49]
>>> m3 - m7                         # les multiples de 3 qui ne sont pas multiples de 7
{33, 3, 36, 6, 39, 9, 12, 45, 15, 48, 18, 24, 27, 30}
>>> m7 - m3                         # les multiples de 7 qui ne sont pas multiples de 3
{49, 35, 28, 14, 7}
>>> m3 + m7                         # attention, pas d'opération + sur les ensembles !!!
<...> TypeError: unsupported operand type(s) for +: 'set' and 'set'
>>> m3 ^ m5                         # les multiples de 3 ou 5, mais pas de 15
{3, 5, 6, 9, 10, 12, 18, 20, 21, 24, 25, 27, 33, 35, 36, 39, 40, 42, 48}
>>> m3 & m5                         # les multiples de 3 et de 5 (c'est-à-dire ceux de 15)
{0, 45, 30, 15}
```

On dispose aussi d'*opérations avec assignation* permettant de modifier facilement un ensemble.

ens_1 `|=` ens_2	remplace ens_1 par $ens_1 \cup ens_2$	ens_1 `&=` ens_2	remplace ens_1 par $ens_1 \cap ens_2$
ens_1 `-=` ens_2	remplace ens_1 par $ens_1 \setminus ens_2$	ens_1 `^=` ens_2	remplace ens_1 par $ens_1 \Delta ens_2$

En reprenant les ensembles $m7$ et $m3$ précédents, on illustre le caractère mutable des objets de type ensemble :

```
>>> m7bis = m7                  # m7bis et m7 c'est pareil
>>> m7ter = m7.copy()           # m7ter est une copie indépendante de m7
>>> m7 -= m3                    # retire de m7 les éléments de l'ensemble m3
>>> m7                          # on obtient les multiples de 7 non multiples de 3
{35, 7, 14, 49, 28}
>>> m7bis                       # la modification s'est répercutée sur m7bis
{35, 7, 14, 49, 28}
>>> m7ter                       # mais pas sur m7ter, qui garde son indépendance!
{0, 35, 7, 42, 14, 49, 21, 28}
```

Remarque : `Python` propose également le type `frozenset`, pour modéliser les ensembles non mutables.
Les objets de type `frozenset` sont donc des ensembles "gelés", dont on ne peut pas modifier le contenu ni la longueur (à moins de redéfinir l'objet complètement).
Ils disposent des méthodes des objets de type `set` (sauf ceux qui ont trait à la mutabilité, bien sûr).
On pourra se reporter à http://docs.python.org/3.3/library/stdtypes.html#frozenset

jean-miche.ferrard@ac-paris.fr

6.3 Itérateurs

On connait la boucle `for`, qui permet de parcourir les différents éléments d'une liste, d'un tuple, d'une chaîne, ou même d'un dictionnaire ou d'un ensemble. Voici quelques exemple récapitulatifs, où on itère la fonction `print` sur des objets successifs (intervalle, liste, chaîne, tuple, ensemble, dictionnaire) :

```
>>> obj = range(1,10)
>>> for x in obj: print(x,end=' ')
1 2 3 4 5 6 7 8 9
>>> obj = [x*x for x in range(1,10)]
>>> for x in obj: print(x,end=' ')
1 4 9 16 25 36 49 64 81
>>> obj = 'bien le bonjour'
>>> for x in obj: print(x,end=' ')
b i e n   l e   b o n j o u r
>>> obj = tuple(range(1,10))
>>> for x in obj: print(x,end=' ')
1 2 3 4 5 6 7 8 9
```

```
>>> obj = set(range(10,40,3))
>>> obj
{34, 37, 10, 13, 16, 19, 22, 25, 28, 31}
>>> for x in obj: print(x,end=' ')
34 37 10 13 16 19 22 25 28 31
>>> obj = {x:x*x for x in range(1,10)}
>>> for x in obj.keys(): print(x,end=' ')
1 2 3 4 5 6 7 8 9
>>> for x in obj.values(): print(x,end=' ')
1 4 9 16 25 36 49 64 81
>>> for (x,y) in obj.items(): print(x,y,end=' ')
1 1 2 4 3 9 4 16 5 25 6 36 7 49 8 64 9 81
```

Les itérables sont donc les objets qu'on peut parcourir, soit parce qu'ils sont présents physiquement en mémoire, soit parce qu'ils sont capables de produire toutes leurs valeurs *à la demande*. Pour illustrer cette différence un peu subtile, on va construire un itérateur un peu particulier avec la fonction `map` :

```
>>> from sys import getsizeof      # importe la fonction calculant la taille mémoire d'un objet
>>> m = map(x:x*x, range(1,100))  # pour mapper la fonction x → x² sur l'intervalle [1,99[
>>> getsizeof(m)                  # la taille occupée en mémoire est minuscule
64
>>> m                             # en fait m pointe sur un "map object" quelque part en mémoire
<map object at 0x103d6bc90>
>>> next(m), next(m), next(m)     # la fonction next permet d'avancer dans l'itérateur
(1,4,9)
>>> next(m), next(m), next(m)     # après les trois premières valeurs, les trois suivantes
(16, 25, 36)
```

On voit ici que notre objet `map` contient un mécanisme permettant de délivrer des valeurs successives (et qu'il est capable de mémoriser à quel stade il en est resté). On peut ainsi accéder manuellement aux valeurs successives de l'objet m, jusqu'à au dernier (un appel supplémentaire de `next` provoque alors l'erreur `StopIteration`) :

```
>>> while True: print(next(m),end=' ')
49 64 81 100 121 144 169 196 225 256 289 324 361 400 441 484 529 576 625 676 729 784 841
900 961 1024 1089 1156 1225 1296 1369 1444 1521 1600 1681 1764 1849 1936 2025 2116 2209
2304 2401 2500 2601 2704 2809 2916 3025 3136 3249 3364 3481 3600 3721 3844 3969 4096 4225
4356 4489 4624 4761 4900 5041 5184 5329 5476 5625 5776 5929 6084 6241 6400 6561 6724 6889
7056 7225 7396 7569 7744 7921 8100 8281 8464 8649 8836 9025 9216 9409 9604 9801
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    while True: print(next(m),end=' ')
StopIteration
```

Revenons maintenant à la définition initiale de l'objet m, et créons une liste obtenue en convertissant cet objet (avec le constructeur `list`). De cette manière, nous formons une liste $m2$ contenant simultanément toutes les valeurs qu'était capable de produire l'objet m. Deux remarques s'imposent : d'une part la taille de l'objet $m2$ est beaucoup plus grande que celle de m, et d'autre part la conversion de m en la liste $m2$ a visiblement "épuisé" le mécanisme d'itération de m :

```
>>> m = map(x:x*x, range(1,100))  # pour mapper la fonction x → x² sur l'intervalle [1,99[
>>> m2 = list(m)                  # on convertit m en liste
>>> getsizeof(m2)                 # la taille de m2 est beaucoup plus importante
992
>>> m2                            # on a réduit ici l'affichage, qui tient sur plusieurs lignes
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, <...>, 9025, 9216, 9409, 9604, 9801]
>>> list(m)                       # on voit que la conversion en liste a épuisé l'itérateur m !
[]
```

Prenons maintenant un exemple analogue au précédent, mais obtenu avec la fonction `filter`

```
>>> f = filter(lambda x: x%5 == 3, range(1,1000))  # filtre  dans [1,999] les congrus à 3 modulo 5
>>> f                                              # f est un "filter objet"
<filter object at 0x103d37a50>
>>> getsizeof(f)                                   # il occupe très peu de place en mémoire
64
>>> next(f), next(f), next(f)                      # on itère trois fois sur cet objet
(3, 8, 13)
>>> g = [x for x in range(1,1000) if x % 5 == 3 ]  # liste des congrus à 3 modulo 5 dans [0,999]
>>> getsizeof(g)                                   # la taille est beaucoup plus importante
1680
```

Autre exemple, avec la fonction `zip`, qui forme des tuples d'éléments de même position dans différents itérables :

```
>>> z = zip([1,2,3],[4,5,6],[7,8,9])       # on va synchroniser trois listes de même longueur
>>> z                                       # le résultat est un "zip object"
<zip object at 0x103d806c8>
>>> next(z)                                 # la première itération renvoie le tuple (1,4,7)
(1, 4, 7)
>>> next(z)                                 # 2ème itération: le tuple des seconds
(2, 5, 8)
>>> next(z)                                 # 3ème itération: le tuple des troisièmes
(3, 6, 9)
>>> next(z)                                 # une itération de trop car c'était fini
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    next(z)
StopIteration
```

Nous terminons ce tour d'horizon en évocant les "intervalles" (obtenus par la fonction `range` de `Python`), et qui ont été souvent utilisés jusqu'ici (notamment dans les boucles `for`) :

```
>>> r = range(0,10000,25)              # l'intervalle des valeurs de 0 à 9999, avec un pas de 5
>>> getsizeof(r)                       # ça ne prend pas de place en mémoire
48
>>> getsizeof(list(r))                 # en revanche, si on forme la liste des valeurs....
3720
>>> next(r)                            # tiens, les 'range' ne sont donc pas des itérateurs
<...> TypeError: 'range' object is not an iterator
>>> ir = iter(r)                       # en fait, le voici, l'itérateur associé au range r
>>> ir
<range_iterator object at 0x103d24e10>
>>> next(ir), next(ir), next(ir)       # trois itérations 'à la main'
(0, 25, 50)
```

Quand on évalue une boucle `for` pour parcourir un objet itérable, c'est le mécanisme `next` qui est mis en oeuvre dans les coulisses, l'exception `StopIteration` étant traitée par la boucle `for` elle-même.

Mais il en est ainsi de nombreux mécanismes qui opèrent sur des objets itérables (la fonction `in` pour tester l'appartenance, la fonction `sum` pour calculer une somme, etc).

Il y a une différence subtile entre "itérable" et "itérateur", les itérateurs étant des fonctions permettant de parcourir les itérables (mais ça fait un peu jargon, tout ça) ; si cette distinction est pertinente pour les "range", elle ne l'est pas pour les "zip", "map" et autres "filter", qui sont leur propre itérateur (got it ?)

Tout ça est en fait assez transparent dans la pratique. On retiendra surtout que l'utilisation d'itérateurs est un moyen efficace (l'empreinte mémoire est faible) d'accéder successivement aux différentes composantes d'un objet itérable (chaîne, liste, tuple, dictionnaire, ensemble, etc.) en évitant de créer simultanément toutes ces composantes en mémoire.

Dernière distinction intéressante : on peut créer plusieurs itérateurs sur un intervalle, ce qui permet de gérer simultanément plusieurs parcours de celui-ci.

jean-miche.ferrard@ac-paris.fr

6.4 Fonctions utiles sur les itérateurs

6.4.1 La fonction `enumerate`

La fonction `enumerate` prend en argument un objet itérable *seq* et elle crée un itérateur renvoyant les couples (n, x) formés des objets x successifs de *seq* et de leur numéro d'ordre n.

```
>>> from random import sample          # on importe la fonction sample du module random
>>> L = sample(range(10,100),10); L    # on crée une liste de 10 nombres différents à deux chiffres.
[64, 33, 98, 36, 15, 29, 21, 47, 78, 89]
>>> list(enumerate(L))]                # on forme la liste des couples (n,L[n]), avec n ≥ 0
[(0,64), (1,33), (2,98), (3,36), (4,15), (5,29), (6,21), (7,47), (8,78), (9,89)]
>>> list(enumerate(L,start=1))         # ici, on fait débuter la numérotation à 1
[(1,64), (2,33), (3,98), (4,36), (5,15), (6,29), (7,21), (8,47), (9,78), (10,89)]
```

Avec la liste L ci-dessus. Voici comment calculer la somme des $kL[k]$, si la numérotation démarre à $k = 1$:

```
>>> sum((n+1)*L[n] for n in range(len(L)))         # sans enumerate
2932
>>> sum(n*x for (n,x) in enumerate(L,start=1))     # avec enumerate
2932
```

6.4.2 La fonction `zip`

La fonction `zip` renvoie un itérateur calculant les couples (x, y) d'éléments de même position dans deux itérables (qui peuvent très bien ne pas être de la même longueur, on s'arrête alors à la plus courte séquence) :

```
>>> s1 = range(1,10); s2 = 'abcdef'         # chiffres de 1 à 9, puis lettres de a à f
>>> list(zip(s1,s2))                        # on zippe s1 et s2, et on convertit en liste
[(1,'a'), (2,'b'), (3,'c'), (4,'d'), (5,'e'), (6,'f')]
>>> list(zip(s2,s1))                        # on zippe s2 et s1, et on convertit en liste
[('a',1), ('b',2), ('c',3), ('d',4), ('e',5), ('f',6)]
```

La fonction `zip`, associée à l'opérateur `*`, peut être utilisée pour transposer une matrice de tuples :

```
>>> a = [(1, 2, 3), (4, 5, 6)]              # une liste de deux tuples de longueur 3
>>> b = list(zip(*a)); b                    # forme la liste transposée
[(1, 4), (2, 5), (3, 6)]
```

6.4.3 Les fonction `any` et `all`

La fonction `any` permet de savoir si l'un au moins des éléments d'un itérable est "vrai" (c'est-à-dire non nul, non vide).
La fonction `all` permet de savoir si tous les éléments d'un itérable sont "vrais" (c'est-à-dire non nuls, non vides).

```
>>> L = [5236,8075,9876,9503]               # on forme une liste de quatre entiers
>>> any(x%17 for x in L)                    # l'un d'eux au moins est-il non divisible par 17?
True
>>> all(x%17 for x in L)                    # tous sont-ils non divisibles par 17?
False
>>> [x%17 for x in L]                       # voici en fait la liste des restes modulo 17
[0, 0, 16, 0]
```

6.4.4 La fonction `reversed`

La fonction `reversed` permet d'itérer "à l'envers" sur les valeurs d'un itérable.

```
>>> s = t = 0
>>> for n in [8,5,6,2]: s = 10*s + n            # convertit la liste de chiffres en un entier
>>> for n in reversed([8,5,6,2]): t = 10*t + n  # même chose, mais à l'envers
>>> s, t
(8562, 2658)
```

6.5 Générateurs (instruction `yield`)

Rappelons qu'une fonction est un bloc d'instructions qui a reçu un nom, dont le fonctionnement dépend d'un certain nombre de paramètres (les *arguments* de la fonction) et qui renvoie un résultat (au moyen de l'instruction `return`).

```
def nom_de_la_fonction(arguments):   # le nom de la fonction, et les paramètres d'appels
    bloc_d'instructions              #    on parcourt ce bloc
```

Dans la pratique, on appelle la fonction en lui passant des arguments, et elle nous renvoie l'expression qui suit le premier `return` rencontré. Le parcours de la fonction est alors terminé (avec retour au programme "appelant"), et il faut procéder à un nouvel appel à cette fonction pour que son contenu soit à nouveau évalué depuis le début.

Plusieurs appels à cette fonction peuvent ainsi produire une séquence de résultats.

On peut cependant imaginer un autre mécanisme. Notre fonction renverrait un résultat, puis resterait *en sommeil* (dans l'état où elle se trouve alors). Un nouvel appel de la fonction la "réveillerait" là où on l'avait laissée, lui permettant de nous renvoyer un nouveau résultat (et de se mettre, là encore, en attente). Dans ces situations d'attente, la fonction conserverait son "contexte" (notamment la valeur de ses variables locales), pour permettre un redémarrage propre.

Ce mécanisme est exactement celui des *générateurs*, qui sont des fonctions définies comme on l'a vu jusqu'à présent, à ceci près que l'instruction `return` est remplacée par `yield` (mot anglais qui signifie "donner", "produire").

Un exemple classique permettra de comprendre ce dont il s'agit.

On sait que la suite de Fibonacci $(F_n)_{n\geq 0}$ est définie par $F_0 = 0$, $F_1 = 1$, et, pour tout $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$.

Voici trois fonctions calculant à leur manière les éléments successifs de la suite de Fibonacci (seule la troisième est un générateur, car elle rend un résultat au moyen de `yield`).

```
def fibo1(n):
    l = [x] = [0]; y = 1
    for k in range(n):
        x, y = y, x+y
        l.append(x)
    return l
```

```
def fibo2(n):
    x, y = 0, 1
    for k in range(n):
        x, y = y, x+y
    return x
```

```
def fibo3(n):
    x, y = 0, 1
    for k in range(n+1):
        yield x
        x, y = y, x+y
```

```
>>> fibo1(10)                          # fibo1(n) renvoie une liste de n+1 valeurs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> [fibo2(n) for n in range(0,11)]    # liste des F_n renvoyés par fibo2 (0 ≤ n ≤ 9)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> [u for u in fibo3(10)]             # la liste des F_n obtenus avec fibo3, pour (0 ≤ n ≤ 9)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> list(fibo3(10))                    # idem
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

La fonction `fibo1` est un peu particulière puisqu'un seul appel à `fibo(`n`)` génère la liste $[F_0, F_1, \ldots, F_n]$. En revanche l'expression `fibo2(`n`)` renvoie le seul nombre de Fibonacci F_n. Évaluer `[fibo2(n) for n in range(0,11)]` est donc ici assez maladroit puisque ça provoque des recalculs systématiques.

Les exemples ci-dessus ne permettent pas de distinguer la différence fondamentale de fonctionnement entre `fibo3` et les deux autres fonctions. Essayons d'y voir un peu plus clair !

Tout d'abord, rien ne permet de différencier les objets `fibo1`, `fibo2`, et `fibo3` : ce sont trois objets de type "function", situés quelque part en mémoire :

```
>>> fibo1, fibo2, fibo3
(<function fibo1 at 0x103f707a0>, <function fibo2 at 0x103f70830>, <function fibo3 at 0x103f708c0>)
```

La différence essentielle tient au résultat de l'évaluation de `fibo1(`n`)` (c'est une liste), `fibo2(`n`)` (c'est l'entier F_n) et `fibo3(`n`)` (c'est un "générateur"). La première chose à bien comprendre est qu'évaluer `fibo3(`n`)` **ne provoque pas l'exécution du code de la fonction** `fibo3` : cela met seulement en place un mécanisme, prêt à se déclencher à notre demande, et qui va fournir, au coup par coup, chacun des entiers F_0, F_1, etc. jusqu'à F_n.

```
>>> fibo1(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fibo2(10)
55
>>> fibo3(10)
<generator object fibo3 at 0x103f77910>
```


jean-miche.ferrard@ac-paris.fr

Plaçons par exemple le générateur `fibo3(10)` dans la variable f.

Pour demander à f de générer (de "yielder" diraient certains) une nouvelle valeur, on évalue `next(`f`)`

```
>>> next(f)      # Actionne le générateur f. On obtient ici F_0=0.
0
>>> next(f)      # entre deux évaluations de next(f), l'état actuel du générateur est préservé
1
>>> next(f)      # chaque évaluation de next(f) fournit une nouvelle valeur. ici F(2)=1
1
>>> next(f)      # on obtient maintenant F(3)=2
2
>>> next(f), next(f), next(f), next(f)   # quatre next(f) ⇒ quatre nouvelles valeurs F(k)
(3, 5, 8, 13)
>>> next(f), next(f), next(f)            # encore trois nouvelles valeurs
(21, 34, 55)
>>> next(f)                              # c'est la demande de trop!
Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    next(f)
StopIteration
```

L'exemple précédent montre que le générateur f=`fibo3(10)` constitue un mécanisme qui peut être sollicité à tout moment par `next(`f`)`. Chaque appel renvoie la valeur qui suit l'instruction `yield` dans le corps de la fonction `fibo3`.

On voit clairement que le code de la fonction `fibo3` est "suspendu" entre deux appels successifs (avec préservation du contenu des variables locales). D'autre part, le code de la fonction `fibo3` montre que l'instruction `yield` est sollicitée au sein de la boucle `for k in range(0,n)` (avec $n = 10$ dans notre exemple). Chaque évaluation de `next(`f`)` avance d'une étape dans cette boucle. Quand cette boucle est terminée, l'évaluation de `next(`f`)` se traduit par une erreur de type `StopIretation` (et c'est normal car on sort enfin du code de la fonction `fibo3`, mais pas par un `yield`).

La fonction `next` accepte un argument par défaut, qui rattrape `StopIteration` quand on dépasse le nombre d'itérations :

```
>>> next(f,"c'est fini, là, ok?")
"c'est fini, là, ok?"
```

Ce qui est spécialement intéressant dans le fonctionnement d'un générateur, c'est qu'il permet de former les valeurs successives d'une suite virtuellement infinie. Modifions par exemple la fonction `fibo3` en supprimant l'argument n :

```
def fibo4():
    x, y = 0, 1              # le début de la suite de Fibonacci
    while True:              # ad vitam aeternam
        yield x                 # délivre la valeur courante de la suite
        x, y = y, x+y           # actualise les deux termes consécutifs
```

L'expression `fibo4()` crée alors un générateur qui permet de délivrer les valeurs successives de la suite de Fibonacci, ad libitum, et au rythme qu'on veut...

```
>>> f = fibo4()                          # prêt à engendrer les nombres de Fibonacci
>>> [next(f) for k in range(0,10)]       # les dix premières valeurs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> [next(f) for k in range(0,5)]        # les cinq suivantes
[55, 89, 144, 233, 377]
```

Important : si f est un générateur, on peut en solliciter toutes les valeurs au moyen de boucles "`for` *indice* `in f`" (si l'ensemble des valeurs est fini, c'est préférable). Dans ce cas c'est la boucle qui déclenche elle-même le mécanisme "next" et qui rattrape l'exception de fin d'itération.

En reprenant les notations précédentes, voici comment "yielder" les 10 premières valeurs de la suite de Fibonacci.

```
>>> g = fibo3(10)                        # crée le générateur (mais ne calcule pas les F_n !!)
>>> for k in g: print(k,end=' ')         # demande et imprime les dix premiers F_n
0 1 1 2 3 5 8 13 21 34 55
```

Autre possibilité, déjà évoquée, avec le constructeur `list` (qui lui aussi prend en charge le mécanisme d'itération) :

```
>>> list(fibo3(15))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```


jean-miche.ferrard@ac-paris.fr

6.6 Fichiers

On appelera fichier toute collection de données (textuelles ou binaires) enregistrée sur un support physique (un disque dur, une clé usb, etc). Un fichier peut être accessible en lecture seule, ou en lecture/écriture. Pour pouvoir être consulté et/ou modifié, il doit être ouvert d'abord, puis fermé ensuite (mais `Python` se charge souvent de fermer le fichier lui-même quand tout est terminé).

`Python` propose la classe `file` pour modéliser la notion de fichier, et pour faciliter les interactions. Tout dépend beaucoup du système d'exploitation (Windows, Linux, Mac OSX, etc) mais la classe `file` offre des méthodes qui permettent de masquer les différences.

Dans tous les cas, un fichier possède un nom et réside (ou est créé) dans un dossier. Pour accéder au fichier, on a besoin de son nom et du chemin d'accès à son dossier.

Par défaut, il s'agit du "dossier courant", c'est-à-dire celui où réside le script en cours d'exécution (ou alors le dernier script utilisé, ou celui de l'application IDLE si on travaille en mode interactif).

Dans toute la suite de cette section, on suppose qu'on se place uniquement dans le "dossier courant". Les fichiers seront donc désignés par leur nom, simplement, sans indication d'un chemin d'accès.

Mais si on veut un contrôle précis sur les dossiers, on importera le module `os` (pour "operating system") qui fournit quelques fonctions utiles : `os.chdir("chemin")` désigne un nouveau dossier de travail, `os.getcwd()` renvoie le dossier de travail actuel ("get current directory").

On ne confondra pas les objets de type `file` tels qu'ils sont créés et manipulés par `Python` (voir plus loin) avec la concrétisation "physique" de ces fichiers sur le disque. En fait, les objets `file` sont des abstractions permettant de désigner commodément ces fichiers "physiques".

On se contentera ici d'indiquer les principales fonctions ayant trait aux fichiers :

– **Ouverture et fermeture** :

$myfile$ `=open(`str`,`$mode/type$`,`$encoding$`)` et $myfile$`.close()`

Ici $myfile$ est l'identificateur associé à l'objet abstrait de type `file`, lui-même associé au fichier physique dont le nom est la chaîne str, en spécifiant le mode et le type d'accès.

Par défaut, l'accès est `'rt'`, c'est-à-dire `'r'` (lecture) et `'t'` (fichier texte).

Les modes d'accès sont `'r'` (reading : lecture seule), `'w'` (writing : avec effaçage si le fichier s'il existe), `'x'` (création exclusivement, avec erreur si le fichier existe), `'a'` (append : écriture à la fin du fichier s'il existe).

Il convient d'indiquer l'encodage quant on ouvre le fichier (par exemple en écrivant `encoding = 'utf-8'` si le fichier texte est en Unicode, ou `encoding = 'Latin-1'`, etc).
Les deux principaux types d'accès sont `'b'` (binaire), `'t'` (texte).

– **Lecture d'un fichier texte** :

$myfile$`.read(`n`)` (n caractères, et par défaut la totalité du fichier) : le résultat est une chaîne de caractères.

$myfile$`.readline()` renvoie une ligne de texte, terminée par le caractère `\n` de fin de ligne.

$myfile$`.readlines()` renvoie la liste de toutes les lignes de texte.

– **Écriture dans un fichier texte** :

$myfile$`.write(`$chaine$`)` écrit une chaîne de caractères dans un fichier texte.

– **Lecture/écriture dans un fichier binaire** :

Le module `pickle` permet d'écrire/lire des données de type quelconque (on n'est donc plus obligé de préciser soit un nombre de caractères, soit une chaîne). Le fichier doit être ouvert en binaire, en écriture ('wb') ou en lecture ('rb').

Après `import pickle`, la syntaxe est `pickle.dump(`obj`,`$myfile$`)` (enregistrer obj) et obj=`pickle.load(`$myfile$`)`

– **Lecture séquentielle d'un fichier texte** :

La syntaxe `for` $ligne$ `in` $myfile$ permet de lire séquentiellement les différentes lignes d'un fichier texte, celui-ci étant donc traité comme un itérateur (c'est la boucle `for` qui se charge de tout, et en particulier du rattrapage de l'exception quand la lecture du fichier est terminée). Cette syntaxe est supérieure à $myfile$`.readlines()`, qui charge la totalité du fichier en mémoire sous la forme d'une liste de chaînes.

jean-miche.ferrard@ac-paris.fr

Nous allons traiter un petit exemple, à partir du texte du poème "Nuit Rhénane" de Guillaume Apollinaire.

```
>>> import os                          # importe le module os ('operating system')
>>> os.getcwd()                       # renvoie le dossier courant
'/Users/jmf/Documents'
```

On crée un fichier texte, en écriture, encodé en utf-8, de nom "nuit-rhenane.txt" sur le disque. Dans la suite, on doit utiliser l'identificateur `poeme`, et non pas "nuit-rhenane.txt". C'est en effet l'objet `poeme` qui possède les méthodes qu'on va utiliser. On voit que `poeme` est en fait un objet de type `TextIOWrapper` (mais c'est sans grande importance) :

```
>>> poeme = open("nuit-rhenane.txt",'wt',encoding='utf-8')
>>> poeme
<_io.TextIOWrapper name='nuit-rhenane.txt' mode='wt' encoding='utf-8'>
```

On écrit maintenant deux lignes dans le fichier. On les termine par le caractère de fin de ligne `\n`.

À chaque fois, la fonction `write` de l'objet `poeme` renvoie le nombre de caractères enregistrés (y compris `\n`) :

```
>>> poeme.write("Mon verre est plein d'un vin trembleur comme une flamme\n")
56
>>> poeme.write("Écoutez la chanson lente d'un batelier\n")
39
```

On peut bien sûr écrire plusieurs lignes de textes consécutivement, à condition de penser au séparateur `\n`.
Attention, ici le caractère \ qui termine la première ligne (juste après `\n`) ne fait pas partie de la chaîne écrite : c'est une commodité de l'éditeur `Python` pour indiquer un passage à la ligne non significatif (simplement parce que le texte entré ici est trop long pour tenir sur une seule ligne à l'écran) :

```
>>> poeme.write("Qui raconte avoir vu sous la lune sept femmes\n\
Tordre leurs cheveux verts et longs jusqu'à leurs pieds\n"))
102
>>> poeme.close()          # maintenant, on décide de fermer le fichier.
```

Le poème est incomplet, et voici les vers manquants, sous forme d'une liste (avec la place prévue pour les lignes vides). Remarque : dans la saisie d'une liste aussi longue, l'éditeur de `Python` ne nécessite pas d'utiliser le caractère \, et il attend sagement que la liste soit terminée par le caractère `]`.

```
>>> liste = ["",
"Debout chantez plus haut en dansant une ronde",
"Que je n'entende plus le chant du batelier",
"Et mettez près de moi toutes les filles blondes",
"Au regard immobile aux nattes repliées",
"",
"Le Rhin le Rhin est ivre où les vignes se mirent",
"Tout l'or des nuits tombe en tremblant s'y refléter",
"La voix chante toujours à en râle-mourir",
"Ces fées aux cheveux verts qui incantent l'été",
"",
"Mon verre s'est brisé comme un éclat de rire"]
```

Le fichier ayant été fermé (mais incomplet), on décide de le réouvrir, mais attention, en mode `append` (sinon on effacerait ce qui vient d'être enregistré !). À l'aide d'une boucle `for` (en itérant sur les éléments v de *liste*), on écrit chacun des vers manquants au poème (attention à bien rajouter les caractères `\n` de fin de ligne) :

```
>>> poeme = open("nuit-rhenane.txt",'a',encoding='utf-8')
>>> for v in liste: poeme.write(v+"\n");
>>> poeme.close()
```

Ouvrons à nouveau le fichier `poeme`, mais en mode lecture. On lit les deux premières lignes (avec deux appels à `poem.readline()`). On voit que les caractères de terminaison `\n` figurent à la fin de chaque ligne lue :

```
>>> poeme = open("nuit-rhenane.txt",encoding='utf-8')
>>> poeme.readline()
"Mon verre est plein d'un vin trembleur comme une flamme\n"
>>> poeme.readline()
"Écoutez la chanson lente d'un batelier\n"
```


jean-miche.ferrard@ac-paris.fr

NB : si on est arrivé à la fin du fichier, `readline` renvoie une chaîne vide (donc ne lève pas une exception).

Un appel à `readlines()` (attention au " s") renvoie alors le reste du fichier, sous forme d'une liste de chaînes, avec les `\n` de terminaison (on n'a pas tout reproduit ici, même si c'est beau). Puis on referme le fichier (c'est plus prudent).

```
>>> poeme.readlines()
['Qui raconte avoir vu sous la lune sept femmes\n',
"Tordre leurs cheveux verts et longs jusqu'à leurs pieds\n", '\n',
<...>
"Ces fées aux cheveux verts qui incantent l'été\n", '\n',
"Mon verre s'est brisé comme un éclat de rire\n"]
>>> poeme.close()
```

On l'ouvre ensuite (en mode par défaut : "text/read") puis on lit et affiche le poème par l'intermédiaire d'une boucle `for`.

Bien penser à l'option `end=""` sans laquelle les caractères de fin de ligne (déjà présents dans le fichier) seraient doublés par le passage à la ligne inhérent à la fonction `print` :

```
>>> poeme = open("nuit-rhenane.txt",encoding='utf-8')
>>> for c in poeme: print(c,end="")
Mon verre est plein d'un vin trembleur comme une flamme
Écoutez la chanson lente d'un batelier
Qui raconte avoir vu sous la lune sept femmes
Tordre leurs cheveux verts et longs jusqu'à leurs pieds

Debout chantez plus haut en dansant une ronde
Que je n'entende plus le chant du batelier
Et mettez près de moi toutes les filles blondes
Au regard immobile aux nattes repliées

Le Rhin le Rhin est ivre où les vignes se mirent
Tout l'or des nuits tombe en tremblant s'y refléter
La voix chante toujours à en râle-mourir
Ces fées aux cheveux verts qui incantent l'été

Mon verre s'est brisé comme un éclat de rire
```

Remarque importante :

- quand on ouvre un fichier-texte *file* en lecture, l'expression $ch =$`''.join(`$file$`)` renvoie dans *ch* une chaîne obtenue par concaténation de toutes les chaînes du fichier (impressionnant !).
- L'expression *ch*`.split(sep='\n')` renvoie alors la liste des chaînes obtenues par séparation au niveau des `\n`.

Voici maintenant un petit exemple avec le module `pickle`.

On ouvre un fichier binaire en écriture, et on y inscrit les carrés des entiers de 1 à 9.

On l'ouvre ensuite en lecture, comme s'il s'agissait d'un fichier texte. La fonction `read` conduit bien sûr à une erreur (le premier caractère n'est pas reconnu comme un code ascii, mais ce serait pareil si on avait précisé l'encodage utf-8).

```
>>> from pickle import *; myfile = open("essai",'wb')
>>> for k in range(1,10): dump(k*k,myfile)
>>> myfile = open("essai"); myfile.read()
<...>
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 0: ordinal not in range(128)
```

Tout va mieux en ouvrant le fichier en lecture binaire. On récupère les neuf données binaires écrites précédemment :

```
>>> myfile = open("essai",'rb')
>>> [load(myfile) for k in range(1,10)]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```


jean-miche.ferrard@ac-paris.fr

Chapitre 7

Quelques fonctions de quelques modules...

Chaque distribution `Python` est livrée avec un nombre considérable de modules qui augmentent la portée du langage.
On a déjà parlé des modules `math` et `cmath`. On évoquera dans les pages suivantes une petite partie des fonctions et des modules qu'il nous semble utile de connaître.
La documentation de référence se trouve ici : http://docs.python.org/3.3/library/index.html
Toujours pour obtenir de l'aide, une autre possibilité consiste à taper `help('modules')` dans IDLE (pour avoir la liste des modules disponibles), ou `help('random')` si on veut de l'aide, par exemple, sur le module `random`.

7.1 Le module `random`

Adresse : http://docs.python.org/3.3/library/random
Utilité : génération de valeurs pseudo-aléatoires
Syntaxe : `import random`, ou `from random import *`, ou `from random import sample`, *etc.*
Quelques fonctions utiles :

`seed(`n`)`	réinitialise le générateur de nombres aléatoires en utilisant l'entier n.
`seed()`	idem mais en utilisant l'horloge du système.
`randrange(`a`,`b`,`h`)`	un entier aléatoire de $[a, b[$, avec le pas h (par défaut $h = 1$).
`randrange(`b`)`	un entier aléatoire de $[0, b[$.
`randint(`a`,`b`)`	un entier aléatoire de $[a, b]$, donc synonyme de `randrange(`a`,`b`+1)`
`choice(`*seq*`)`	un élément au hasard dans la séquence (non vide) *seq*
`shuffle(`*seq*`)`	"rebat" aléatoirement la séquence (non vide) *seq* (opère sur place)
`sample(`*pop*`,`k`)`	liste de k éléments distincts de *pop* (ensemble ou séquence)
`random()`	le prochain réel pseudo-aléatoire dans $[0, 1[$
`uniform(`a`,`b`)`	un réel pseudo-aléatoire dans $[a, b]$
`gauss(`m`,`σ`)`	réel pseudo-aléatoire, distribution gaussienne, moyenne m, écart-type σ

Voici quelques exemples :

```
>>> from random import *                  # on importe le module random
>>> seed(1)                               # on initialise le générateur de nombres aléatoires
>>> [randint(1,6) for k in range(10)]     # on simule 10 lancers d'un dé honnête
[2, 5, 1, 3, 1, 4, 4, 4, 6, 4]
>>> [randint(1,6) for k in range(10)]     # on relance le dé encore 10 fois
[2, 1, 4, 1, 4, 4, 5, 1, 6, 4]
>>> seed(1)                               # on recale le générateur
>>> [randint(1,6) for k in range(10)]     # on retrouve les 10 premiers lancers
[2, 5, 1, 3, 1, 4, 4, 4, 6, 4]
>>> sample(range(0,10),10)                # voici une permutation aléatoire de {0,1,...,9}
[4, 3, 1, 2, 0, 5, 9, 6, 8, 7]
>>> t = list(range(0,10))                 # on forme la liste [0,1,...,9]
>>> shuffle(t); t                         # on rebat cette liste aléatoirement (sur place)
[2, 9, 4, 1, 5, 7, 6, 3, 8, 0]
```

7.2 Le module decimal

Adresse : http://docs.python.org/3.3/library/decimal.html

Utilité : calculs sur les nombres décimaux.

Syntaxe : `import decimal`, ou `from decimal import *`

Le module `decimal` permet d'effectuer des calculs exacts sur les nombres décimaux, dans les limites d'une précision fixée par l'utilisateur (mais par défaut égale à 28 chiffres significatifs).

Dans la suite, on suppose que le module `decimal` a été importé par la commande `from decimal import *`

L'instruction `getcontext().prec = n` fixe la précision à n chiffres significatifs.

Les "nombres décimaux" (au sens du module `decimal`) sont obtenus par application du constructeur `Decimal` (noter le D majuscule) appliqué à un entier ou à une chaîne de caractères (elle-même une représentation d'un flottant, ce qui permet à l'utilisateur de former des valeurs décimales avec une précision donnée).

Le module `decimal` "surcharge" les opérations arithmétiques usuelles, ce qui permet d'effectuer des calculs exacts sur les décimaux (tant qu'on ne dépasse pas la limite de précision qu'on s'est fixée).

Commençons par un exemple simple, pour comprendre à quoi le module `decimal` peut être utile, et en quoi il s'attaque à des problèmes posés par la représentation binaire habituelle des flottants.

```
>>> 0.7 * 0.7              # pour le carré de 0.7, on s'attendrait à trouver 0.49
0.48999999999999994        # mais il y a une imprécision due à la représentation binaire
>>> 0.7 * 0.7 == 0.49      # plus problématique, Python ne reconnait pas cette égalité évidente
False
>>> [0.5 ** n for n in range(1,8)]      # on calcule (0.5)^n, avec n ∈ {1,···,7} (pas d'erreurs)
[0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625, 0.0078125]
>>> [0.1 ** n for n in [2,3,4]]         # on calcule (0.1)^n, avec n ∈ {2,3,4}
[0.010000000000000002, 0.0010000000000000002, 0.00010000000000000002]
```

Les "erreurs" ci-dessus ne sont que des conséquences de la représentation en base 2 des flottants : même un nombre aussi inoffensif que $0.1 = 1/(2*5)$ n'est pas représenté de façon exacte en mémoire (contrairement à $0.5 = 2^{-1}$).

Voici ce qu'on obtient, avec les mêmes "calculs", en important le module `decimal` :

```
>>> from decimal import *
>>> [Decimal("0.1") ** n for n in range(2,6)]
[Decimal('0.01'), Decimal('0.001'), Decimal('0.0001'), Decimal('0.00001')]
>>> [Decimal("0.5") ** n for n in range(2,6)]
[Decimal('0.25'), Decimal('0.125'), Decimal('0.0625'), Decimal('0.03125')]
```

Tout cela n'est peut-être pas très impressionnant, alors on va augmenter la précision :

```
>>> getcontext().prec = 50              # augmente la précision à 50 chiffres
>>> Decimal(1)/Decimal(17)              # 1/17 avec 50 chiffres significatifs
Decimal('0.058823529411764705882352941176470588235294117647059')
>>> Decimal(2).sqrt()                   # √2 avec 50 chiffres significatifs
Decimal('1.4142135623730950488016887242096980785696718753769')
>>> Decimal(1).exp()                    # le nombre  e avec 50 chiffres significatifs
Decimal('2.7182818284590452353602874713526624977572470937000')
>>> Decimal(2).ln()                     # ln 2 avec 50 chiffres significatifs
Decimal('0.69314718055994530941723212145817656807550013436026')
>>> Decimal(2).ln().as_tuple()[1]       # on peut même extraire les chiffres!
(6, 9, 3, 1, 4, 7, 1, 8, 0, 5, 5, 9, 9, 4, 5, 3, 0, 9, 4, 1, 7, 2, 3, 2, 1, 2,
 1, 4, 5, 8, 1, 7, 6, 5, 6, 8, 0, 7, 5, 5, 0, 0, 1, 3, 4, 3, 6, 0, 2, 6)
```

Toujours avec une précision de 50 chiffres, voici la valeur décimale de la somme $\sum_{n=1}^{999} \frac{1}{n}$:

```
>>> sum([Decimal(1)/Decimal(n) for n in range(1,1000)])
Decimal('7.4844708605503449126565182043339001765216791697082')
```

Un certain nombre des fonctionnalités du module `decimal` sont des ajouts récents. Le nombre de fonctions mathématiques qui sont compatibles avec ce module est essentiellement limité au logarithme, à l'exponentielle et à la racine carrée.

Pour plus d'informations, on lira les "recettes" (http://docs.python.org/3.3/library/decimal.html#recipes) qui montrent comment programmer le calcul de π, e^x, $\cos(x)$, $\sin(x)$ avec une précision donnée.

jean-miche.ferrard@ac-paris.fr

7.3 Le module fractions

Adresse : http://docs.python.org/3.3/library/fractions.html

Utilité : calculs sur les rationnels.

Syntaxe : `from fractions import Fraction`

Le module `fractions` permet d'effectuer des calculs exacts sur les nombres rationnels.

Un nombre rationnel s'obtient par le constructeur `Fraction` qui prend en argument deux entiers (le numérateur, puis le dénominateur qui par défaut vaut 1) ou une chaîne (par exemple '12/17') ou un flottant (sous forme décimale).

Voici quelques exemples :

```
>>> from fractions import Fraction                # importe la classe Fraction du module fractions
>>> Fraction(1,2) + Fraction(1,3)                 # calcule ici 1/2 + 1/3
Fraction(5, 6)
>>> sum([Fraction(1,n) for n in range(1,20)])     # la somme des 1/n, pour 1 ≤ n ≤ 20
Fraction(275295799, 77597520)
>>> float(_)                                      # convertit ce résultat en flottant
3.547739657143682
>>> from math import pi                           # importe la valeur π depuis le module math
>>> Fraction(pi)                                  # approximation rationnelle de π
Fraction(884279719003555, 281474976710656)
>>> fp = Fraction(pi).limit_denominator(1000); fp # idem mais avec dénominateur ≤ 1000
Fraction(355, 113)
>>> fp.numerator, fp.denominator                  # extrait le numérateur et le dénominateur
(355, 113)
```

7.4 Le module string

Adresse : http://docs.python.org/3.3/library/string.html

Utilité : quelques constantes utiles, et opérations de formatage avancé de chaînes.

Syntaxe : `import string`, ou `from string import *`, ou `from string import printable`, *etc*.

Le module `string` permet des opérations avancées de formatage de chaînes de caractères (bien au-delà de ce que permet la fonction intégrée `format`), mais c'est très technique, et donc on se reportera à l'aide intégrée à `Python`.

Le module `string` contient quelques chaînes de caractères constantes qui peuvent se révéler utiles :

ascii_letters	abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
ascii_lowercase	abcdefghijklmnopqrstuvwxyz
ascii_uppercase	ABCDEFGHIJKLMNOPQRSTUVWXYZ
digits	0123456789
hexdigits	0123456789abcdefABCDEF
octdigits	01234567
punctuation	!"#$%&\'()*+,-./:;<=>?@[\\]^_‘{|}~
printable	renvoie digits + ascii_letters + punctuation + whitespace
whitespace	\t\n\r\x0b\x0c (nb : le premier caractère est l'espace)

```
>>> from string import *; from random import *    # importe string et random
>>> maj = list(ascii_uppercase)                   # liste des majuscules
>>> ''.join(sample(maj,10))                       # un mot aléatoire de 10 majuscules distinctes
'UXMRSKJHCP'
>>> shuffle(maj)                                  # rebat la liste maj sur place
>>> ''.join(maj)                                  # renvoie le résultat sous forme de chaîne
'VMZTYDIJRSHUXLCGWAQBEFPKNO'
```


jean-miche.ferrard@ac-paris.fr

7.5 Le module itertools

Adresse : http://docs.python.org/3.3/library/itertools.html

Utilité : création d'itérateurs pour boucles efficaces

Syntaxe : `import itertools`, ou `from itertools import *`, ou `from itertools import accumulate`, *etc.*

Voici une sélection de fonctions issues du module `itertools` :

`count(`n,h`)`	crée un itérateur démarrant à n, de pas h ($h=1$ par défaut)
`cycle(`s`)`	crée un itérateur produisant en boucle les valeurs d'une séquence finie s
`repeat(`x`,`n`)`	crée un itérateur produisant n fois la valeur x (si n absent, répétition infinie)
`accumulate(`s`)` `accumulate(`s,f`)`	crée un itérateur produisant les sommes cumulées d'une séquence finie s. crée un itérateur produisant les valeurs $v_0=s_0$, $v_1=f(v_0,s_1)$, $v_1=f(v_1,s_2)$, etc.
`chain(`$s,s',s'',\cdots$`)`	crée un itérateur produisant les valeurs de s, puis celles de s', celles de s'', etc.
`product(`$s,s',s'',\cdots$`)` `product(`s`,repeat=`n`)`	crée un itérateur produisant les valeurs du produit cartésien $s\times s'\times s''\times\cdots$ crée un itérateur produisant les valeurs du produit cartésien s^n
`permutations(`s,n`)` `permutations(`s`)`	crée un itérateur produisant les n-uples d'éléments distincts crée un itérateur produisant les permutations d'éléments de la séquences s
`combinations(`s,n`)`	crée un itérateur produisant les n-uples ordonnés d'éléments distincts (utiliser `combinations_with_replacement` pour des combinaisons avec répétitions)

Voici quelques exemples d'utilisation.

On a utilisé le constructeur `list` pour visualiser facilement la liste des valeurs rendues par les différents itérateurs créés ici, mais on répète que l'utilité de ces *itérateurs* (on devrait plus précisément parler de *générateurs*) est de nous permettre d'accéder aux valeurs successives d'une séquence, sans pour autant créer la totalité de cette séquence en mémoire.

Les générateurs tels qu'ils sont créés ici sont en général utilisés dans un "contexte d'itération" (boucle `for`). Ils peuvent également être utilisés "en mode manuel", en leur demandant de délivrer leurs valeurs par des `next` successifs.

On visitera http://docs.python.org/3.3/library/itertools.html#itertools-recipes pour de nombreux exemples d'utilisation des fonctions du module `itertools`

```
>>> from itertools import *                          # importe tout le module (noms courts)
>>> c = cycle('abc'); [next(c) for k in range(14)]   # 14 valeurs du cycle a → b → c → a
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c', 'a', 'b']
>>> list(chain('abcd','efg'))                        # enchaîne deux séquences de caractères
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> list(accumulate(range(1,10)))                    # sommes cumulées de {1,2,...,9}
[1, 3, 6, 10, 15, 21, 28, 36, 45]
>>> list(accumulate(range(1,10),lambda x,y: x*y))    # produits cumulés de {1,2,...,9}
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
>>> list(product(range(2),range(4)))                 # éléments de {0,1}×{0,1,2,3}
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3)]
>>> [(x,y) for x in range(2) for y in range(4)]      # la même chose sans itertools
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3)]
>>> list(product(range(3),repeat=2))                 # les éléments de {0,1,2}×{0,1,2}
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> list(product(range(2),repeat=3))                 # les éléments de {0,1}×{0,1}×{0,1}
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> list(permutations(range(3)))                     # les 6 permutations de {0,1,2}
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
>>> list(permutations(range(4),2))                   # les couples d'éléments de {0,1,2,3}
[(0, 1), (0, 2), (0, 3), (1, 0), (1, 2), (1, 3), (2, 0), (2, 1), (2, 3), (3, 0), (3, 1), (3, 2)]
>>> list(combinations(range(5),2))                   # les paires d'éléments de {0,1,2,3,4}
[(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
>>> list(combinations_with_replacement(range(4),2))  # paires avec répétitions de {0,1,2,3}
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```


jean-miche.ferrard@ac-paris.fr

7.6 Les modules operator et functools

Adresses :
http://docs.python.org/3.3/library/operator.html
http://docs.python.org/3.3/library/functools.html

Le module **operator** offre une traduction fonctionnelle d'un grand nombre d'opérateurs. Par exemple, la fonction **add** est la traduction préfixée de l'opérateur d'addition. La liste des traductions *opérateur* → *fonction* est trop longue pour être reproduite ici. On se contentera donc de quelques exemples (d'autant qu'il est a priori possible de contourner l'utilisation de ce module en utilisant des listes en compréhension).

```
>>> from operator import *
>>> list(map(add,[1,2,3],[40,50,60]))          # on mappe l'addition sur deux listes
[41, 52, 63]
>>> [x+y for x,y in zip([1,2,3],[40,50,60])]   # idem avec zip et liste en compréhension
[41, 52, 63]
>>> from random import sample                  # importe la fonction sample du module random
>>> a = sample(range(100,1000),10); a          # liste a de 10 entiers différents à trois chiffres
[632, 138, 575, 233, 912, 985, 654, 245, 731, 589]
>>> b = sample(range(100,1000),10); b          # liste b de 10 entiers différents à trois chiffres
[988, 770, 254, 505, 838, 809, 110, 999, 684, 431]
>>> list(map(lt,a,b))                          # mappe la fonction de comparaison <
[True, True, False, True, False, False, False, True, False, False]
>>> [x<y for x,y in zip(a,b)]                  # idem avec zip et liste en compréhension
[True, True, False, True, False, False, False, True, False, False]
```

Pour ce qui est du module **functools**, on retiendra essentiellement la fonction **reduce** :

reduce(f, s)	calcule $v_1 = s_1$, $v_2 = f(v_1, s_2)$, etc. puis $v_{k+1} = f(v_k, s_{k+1})$ etc. et renvoie la dernière valeur.
reduce(f, s, d)	idem, mais en initialisant à partir d'une valeur d

```
>>> from operator import add                # importe la fonction add du module operator
>>> from functools import reduce            # importe la fonction reduce du module functools
>>> reduce(lambda x,y: 10*x+y, [7,3,4,8,6]) # conversion chiffres → base 10
73486
>>> reduce(add,[1,2,3,4,5])                 # calcule la somme des éléments de la liste
15
>>> sum(x for x in [1,2,3,4,5])             # mais il y a plus simple (et sans module)
15
```

7.7 Le module time

Adresse : http://docs.python.org/3.3/library/time.html

Utilité : accéder à des mesures de durée, de dates, et effectuer des conversions durées/dates.

Syntaxe : **import time**, ou **from time import ***, ou **from time import sleep**, *etc.*

Le module **time** opère sur des dates exprimées en secondes depuis la date 0 du système d'exploitation (le 1er janvier 1970 sur les systèmes Unix), ou exprimées sous une forme structurée (avec indication de l'année, du mois, etc.).

Une date structurée peut être renvoyée au format GMT (*Greenwich Mean Time*) mais le nouveau nom est plutôt UTC (*Coordinated Universal Time*) ou au format local (pour tenir compte du fuseau horaire tel qu'il est réglé sur la machine).

```
>>> from time import *                  # importe la totalité du module time
>>> lt = localtime(); lt                # renvoie la date structurée locale actuelle
time.struct_time(tm_year=2013, tm_mon=2, tm_mday=3, tm_hour=14, tm_min=26, tm_sec=31,
                 tm_wday=6, tm_yday=34, tm_isdst=0)
>>> asctime(lt)                         # convertit cette date en qqchose de plus lisible
'Sun Feb  3 14:26:31 2013'
>>> mktime(lt)                          # convertit en secondes depuis le 1er janvier 1970 à 0h00
1359897991.0
```


jean-miche.ferrard@ac-paris.fr

Voici quelques fonctions du module `time` (on note *dst* une date structurée et *sec* une durée en secondes).

`gmtime(`*sec*`)`	convertit un nombre de secondes (depuis la date 0) en la date GMT structurée
`localtime(`*sec*`)`	convertit un nombre de secondes (depuis la date 0) en la date locale structurée
`calendar.timegm(`*dst*`)`	convertit une date structurée GMT en un nombre de secondes depuis la date 0
`mktime(`*dst*`)`	convertit une date structurée locale en un nombre de secondes depuis la date 0
`asctime(`*dst*`)`	convertit une date structurée en une chaîne plus lisible
`ctime(`*sec*`)`	convertit des secondes (depuis la date 0) en chaîne (date locale) `ctime(0)`⇒`'Thu Jan 1 01:00:00 1970'` et `ctime(1e9)`⇒`'Sun Sep 9 03:46:40 2001'`
`strftime(`*ft*`,`*sec*`)`	convertit des secondes (depuis la date 0) en chaîne, avec une chaîne de formatage *ft* `strftime("%a, %d %b %Y %H:%M:%S", localtime())`⇒ `'Sun, 03 Feb 2013 15:06:25'`
`strptime(`*ch*`,`*ft*`)`	convertit en date structurée une chaîne *ch* (date lisible), avec formatage par *ft* `time.strptime("30 Nov 00", "%d %b %y")` ⇒ `time.struct_time(tm_year=2013,...`

Plusieurs fonctions permettent de mesurer des durées (c'est utile pour étudier l'efficacité temporelle d'un algorithme).

Le sujet est rendu assez compliqué par les différences qui existent entre les plateformes (Windows, Mac, etc.).

Pour plus d'informations, consulter : http://www.python.org/dev/peps/pep-0418/

Voici quelques fonctions disponibles (pour mesurer une durée, on utilisera la différence entre deux valeurs renvoyées par la fonction choisie, par exemple `perf_counter`, car les origines de ces "horloges" sont indéfinies).

`sleep(`*sec*`)`	suspend l'exécution pendant le nombre indiqué *sec* de secondes
`time()`	donne l'heure locale actuelle, exprimée en secondes depuis la date 0 (déconseillé pour mesurer des courtes durées, par manque de précision)
`clock()`	valeur de l'horloge-processeur à un instant donné, exprimée en microsecondes (efficace sous Windows, beaucoup moins sous Unix)
`monotonic()`	horloge-processeur "monotone", indépendante des modifications du système
`perf_counter()`	horloge précise sur toutes plateformes (depuis `Python3.3`)
`process_time()`	idem, mais n'inclut pas les pauses générées par la fonction `sleep`

Exemple : on va comparer trois fonctions formant la liste *lc* des k^2, pour $0 \leq k \leq n$ (avec n entier passé en argument).

Ces fonctions ne renvoient pas *lc* (elles la fabriquent, juste), mais le temps passé (sur 6 décimales).

`test1` crée *lc* par ajouts des $[k^2]$ (mauvais !), `test2` par des `append` successifs (bien mieux) et `test3` forme la liste en compréhension (encore mieux)

```
def test1(n):
    from time import perf_counter as pc
    top = pc(); lc = []
    for k in range(n): lc = lc + [k*k]
    return round(pc()-top,6)
```

```
def test2(n):
    from time import perf_counter as pc
    top = pc(); lc = []
    for k in range(n): lc.append(k*k)
    return round(pc()-top,6)
```

```
def test3(n):
    from time import perf_counter as pc
    top = pc();
    lc = [k*k for k in range(n)]
    return round(pc()-top,6)
```

NB : on remarquera comment on a renommé localement la fonction `perf_counter` en `pc`.

Voici combien de temps mettent ces trois fonctions, pour $n = 10^3$, $n = 10^4$ et $n = 10^5$ (no comment pour `test1`) :

```
>>> [test1(10**n) for n in [3,4,5]]          # test1 est une très mauvaise solution
[0.001771, 0.174184, 23.001052]
>>> [test2(10**n) for n in [3,4,5]]          # la fonction test2 est bien meilleure
[0.000133, 0.001309, 0.012158]
>>> [test3(10**n) for n in [3,4,5]]          # test3 est encore meilleure (et plus élégante)
[0.000142, 0.000712, 0.00709]
```

Remarque : à la place de la fonction `perf_counter`, on aurait pu utiliser les fonctions `clock`, `time`, `process_time` ou `monotonic`. Il est difficile sur un seul exemple de se rendre compte de ce qui les différencie réellement. La documentation `Python` recommande en tout cas de ne pas utiliser les fonctions `time` ou `clock` (soit par leur imprécision sur des mesures de durées faibles, soit par dépendance relativement au système d'exploitation utilisé).

NB : On pourra aussi consulter les modules `datetime` (http://docs.python.org/3.3/library/datetime.html) et `calendar` (http://docs.python.org/3.3/library/calendar.html)

jean-miche.ferrard@ac-paris.fr

7.8 La classe Counter du module collections

Adresse : http://docs.python.org/3.3/library/collections.html#counter-objects

Syntaxe : `from collections import Counter`

La classe `Counter` permet de créer des dictionnaires spécialisés dans le comptage des occurences d'un motif particulier dans un objet (par exemple le nombre de fois où apparaissent les différentes lettres d'une chaîne de caractères).

La classe `Counter` hérite des méthodes propres aux dictionnaires.

Le constructeur `Counter` accepte un itérable, et il crée un dictionnaire dont les clefs sont les éléments de l'itérable et dont les valeurs correspondantes sont les nombres de fois où ces éléments apparaissent dans l'itérable.

On crée ici un compteur indiquant le nombre d'occurences de chaque lettre d'une chaîne.

```
>>> from collections import Counter        # importe la classe Counter du module collections
>>> cnt = Counter('abracadabra'); cnt      # combien de fois certaines lettres dans 'abracadabra'
Counter({'a': 5, 'r': 2, 'b': 2, 'd': 1, 'c': 1})
>>> cnt['b'], cnt['z']                     # combien de fois 'b', ou 'z'
(5, 0)
>>> list(cnt.elements())                   # la liste des éléments, avec répétitions
['d', 'r', 'r', 'a', 'a', 'a', 'a', 'a', 'b', 'b', 'c']
>>> cnt.most_common()                      # liste triée suivant les fréquences décroissantes
[('a', 5), ('r', 2), ('b', 2), ('d', 1), ('c', 1)]
>>> cnt.most_common(3)                     # les trois éléments les plus fréquents
[('a', 5), ('r', 2), ('b', 2)]
>>> cnt.most_common()[-1]                  # l'élément le plus rare
('c', 1)
```

On peut créer des objets de type `Counter` à partir d'une séquence d'objets du type $valeur = frequence$.

La classe `Counter` autorise des opérations ensemblistes (on agit sur les fréquences).

```
>>> c1 = Counter(a=5, b=3, c=2, d= 6, e=1); c1    # on crée un premier Counter
Counter({'d': 6, 'a': 5, 'b': 3, 'c': 2, 'e': 1})
>>> c2 = Counter(a=1, b=4, d= 6, e=1); c2         # puis un deuxième
Counter({'d': 6, 'b': 4, 'e': 1, 'a': 1})
>>> c1 + c2                                       # ajoute les fréquences
Counter({'d': 12, 'b': 7, 'a': 6, 'e': 2, 'c': 2})
>>> c1 - c2                                       # on les soustrait (ne garde que les positives)
Counter({'a': 4, 'c': 2})
>>> c1 & c2                                       # intersection (minimum des fréquences)
Counter({'d': 6, 'b': 3, 'e': 1, 'a': 1})
>>> c1 | c2                                       # union (maximum des fréquences)
Counter({'d': 6, 'a': 5, 'b': 4, 'c': 2, 'e': 1})
```

Les exemples suivants montrent comment utiliser la classe `Counter` pour mesurer des fréquences dans des simulations d'expériences aléatoires.

```
>>> from random import randint
>>> g = (randint(1,6) for k in range(1000))   # un générateur de 1000 lancers d'un dé
>>> rf = Counter(g); rf                       # dictionnaire des résultats/fréquences
Counter({3: 175, 2: 172, 4: 171, 5: 167, 1: 159, 6: 156})
>>> [rf[r] for r in sorted(rf)]               # extrait fréquences, dans l'ordre des résultats r
[159, 172, 175, 171, 167, 156]
>>> g2 = (randint(1,6)+randint(1,6) for k in range(1000))  # génère 1000 sommes de deux dés
>>> rf2 = Counter(g2); rf2                    # dictionnaire des sommes/fréquences
Counter({7: 190, 6: 137, 8: 126, 9: 116, 5: 102, 4: 87, 10: 73, 3: 58, 11: 51, 2: 32, 12: 28})
>>> [rf2[k] for k in sorted(rf2)]             # les fréquences selon les sommes croissantes
[32, 58, 87, 102, 137, 190, 126, 116, 73, 51, 28]
```


jean-miche.ferrard@ac-paris.fr

7.9 La classe deque du module collections

Adresse : http://docs.python.org/3.3/library/collections.html#collections.deque

Syntaxe : `from collections import deque`

La classe `deque` permet de créer des listes à double entrée (à gauche et à droite). Les objets de la liste peuvent ainsi être ajoutés ou retirés, soit à gauche soit à droite, et en temps constant (on rappelle que les objets habituels de type `list` ne sont accessibles en temps constant qu'en une seule extrémité).

Le constructeur `deque` accepte un itérable (et en argument optionnel une longueur maximum n, par défaut n est "infini"). Si n est précisé, et quand la liste atteint sa longueur maximum, l'ajout d'éléments à gauche (resp. à droite) provoque la "sortie" d'autant d'éléments à droite (resp. à gauche).

Voici un échantillon des fonctions disponibles, à partir d'un exemple simple :

```
>>> from collections import deque           # import la classe deque du module collections
>>> dq = deque([3,7,2],10); dq              # liste à double entrée, longueur maximum 10
deque([3, 7, 2], maxlen=10)
```

On peut ajouter un ou plusieurs éléments, soit à droite, soit à gauche (attention à l'ordre à gauche) :

```
>>> dq.append(8); dq                        # on ajoute 8 à droite
deque([3, 7, 2, 8], maxlen=10)
>>> dq.appendleft(1); dq                    # on ajoute 1 à gauche
deque([1, 3, 7, 2, 8], maxlen=10)
>>> dq.extend(['a','b','c','d']); dq        # on étend par a,b,c,d à droite
deque([1, 3, 7, 2, 8, 'a', 'b', 'c','d'], maxlen=10)
>>> len(dq), maxlen(dq)                     # longueur actuelle, longueur maximum
(9, 10)
>>> dq.extendleft(['x','y','z']); dq        # complète à gauche par x,y,z: c,d sortent à droite
deque(['z', 'y', 'x', 1, 3, 7, 2, 8, 'a', 'b'], maxlen=10)
```

On peut "popper" un élément à gauche, ou à droite :

```
>>> dq.pop(); dq                            # renvoie et efface l'élément le plus à droite
'b'
deque(['z', 'y', 'x', 1, 3, 7, 2, 8, 'a'], maxlen=10)
>>> dq.popleft(); dq                        # renvoie et efface l'élément le plus à gauche
'z'
deque(['y', 'x', 1, 3, 7, 2, 8, 'a'], maxlen=10)
```

On peut supprimer n'importe quel élément, ou tester l'appartenance :

```
>>> dq.remove(7); dq                        # retire la première occurrence d'un élément
deque(['y', 'x', 1, 3, 2, 8, 'a'], maxlen=10)
>>> 7 in dq                                 # on voit que 7 n'est plus dans la liste
False
>>> del(dq[4]); dq                          # supprime le quatrième élément
deque(['y', 'x', 1, 3, 8, 'a'], maxlen=10)
>>> dq[1], dq[-2]                           # le deuxième élément, l'avant-dernier
('x', 8)
```

On peut renverser l'ordre, faire tourner, convertir au format liste "habituel" :

```
>>> dq.reverse(); dq                        # inverse l'ordre
deque(['a', 8, 3, 1, 'x', 'y'], maxlen=10)
>>> dq.rotate(2); dq                        # rotation de deux positions vers la droite
deque(['x', 'y', 'a', 8, 3, 1], maxlen=10)
>>> dq.rotate(-3); dq                       # rotation de trois positions vers la gauche
deque([8, 3, 1, 'x', 'y', 'a'], maxlen=10)
>>> list(dq)                                # convertit en liste au sens usuel
[8, 3, 1, 'x', 'y', 'a']
>>> list(reversed(dq))                      # liste usuelle, à partir d'une copie inversée
['a', 'y', 'x', 1, 3, 8]
```

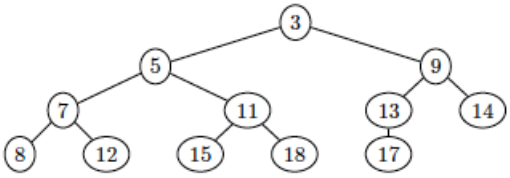

jean-miche.ferrard@ac-paris.fr

7.10 Le module heapq

Adresse : http://docs.python.org/3.3/library/heapq.html

Le module **heapq** implémente un modèle de liste triée "par tas".
On dit qu'une liste ℓ est un un *tas* si pour tout k on a les inégalités : $\ell[k] \leq \ell[2k+1]$ et $\ell[k] \leq \ell[2k+2]$.

Par exemple, `[3,5,9,7,11,13,14,8,12,15,18,17]` est un tas, qu'il est commode de représenter sous forme arborescente.



Les opérations de maintien de la structure de tas (ajouts ou suppressions) y sont en temps logarithmique par rapport à la longueur de la liste. La valeur minimum est située au début du tas.

Remarque : les tas implémentés dans le module **heapq** sont des "tas-min" (chaque élément parent est inférieur ou égal à ses deux enfants), alors que l'habitude est plus souvent de considérer des "tax-max". Pour la théorie, on se reportera à son livre favori sur l'algorithmique et les structures de données.

```
>>> from heapq import *                          # importe le module heapq (avec les noms courts)
>>> t1 = [3,17,11,14,9,12,8,18,5,15,13,7]        # une liste
>>> heapify(t1); t1                              # on transforme t1 en tas, sur place
[3, 5, 7, 14, 9, 11, 8, 18, 17, 15, 13, 12]
>>> t2 = [3,5,9,7,11,13,14,8,12,15,18,17]        # mêmes éléments, mais c'est déjà un tas
>>> heapify(t2); t2                              # heapify ne modifie donc pas t2
[3, 5, 9, 7, 11, 13, 14, 8, 12, 15, 18, 17]
>>> heappush(t2,10); t2                          # ajoute 10 au tas t2, et maintient le tas
[3, 5, 9, 7, 11, 10, 14, 8, 12, 15, 18, 17, 13]
>>> heappop(t2); t2                              # "poppe" le minimum, et actualise le tas t2
3
[5, 7, 9, 8, 11, 10, 14, 13, 12, 15, 18, 17]
>>> heappushpop(t2,16); t2                       # ajoute 16, renvoie le minimum, maintient le tas
5
[7, 8, 9, 12, 11, 10, 14, 13, 16, 15, 18, 17]    # voir aussi la fonction heapreplace
```

Le module **heapq** contient deux fonctions renvoyant la liste (ordonnée) des n plus grands (ou des n plus petits) éléments d'un itérable. C'est efficace si n n'est pas trop grand (sinon utiliser `sorted`) et si $n \neq 1$ (sinon utiliser `min` ou `max`).

```
>>> from random import sample
>>> a = sample(range(100,1000),10); a            # liste de 10 entiers différents de trois chiffres
[961, 196, 764, 487, 759, 608, 426, 697, 553, 420]
>>> nlargest(3,a)                                # la liste des 3 plus grands
[961, 764, 759]
>>> nsmallest(4,a)                               # la liste des 4 plus petits
[196, 420, 426, 487]
>>> nsmallest(3,a,lambda x: x % 10)              # les trois avec le plus petit reste modulo 10
[420, 961, 553]
>>> nsmallest(3,a,lambda x: abs(x-500))          # les trois plus proches de 500
[487, 553, 426]
```

Le module **heapq** contient également une fonction `merge` permettant de fusionner deux ou plusieurs itérables (supposés déjà triés dans l'ordre croissant) en un seul générateur (comme toujours avec les générateurs, l'ensemble des valeurs n'est pas chargé en mémoire en même temps, mais il est disponible à la demande dans un contexte d'itération) :

```
>>> x = [1, 5, 8, 15, 19, 23, 25, 31]            # première liste croissante
>>> y = [2, 3, 6, 12, 14, 19, 26, 28, 30]        # deuxième liste croissante
>>> z = [2, 10, 20, 40]                          # troisième liste croissante
>>> merge(x,y,z)                                 # on les fusionne en un seul générateur
<generator object merge at 0x101603c30>
>>> list(merge(x,y,z))                           # si on veut voir toutes les valeurs d'un coup
[1, 2, 2, 3, 5, 6, 8, 10, 12, 14, 15, 19, 19, 20, 23, 25, 26, 28, 30, 31, 40]
```


jean-miche.ferrard@ac-paris.fr

7.11 Le module bisect

Adresse : http://docs.python.org/3.3/library/bisect.html

Le module bisect permet de rechercher des positions d'insertion (et d'insérer de nouvelles valeurs) dans une liste triée.

Les positions d'insertion peuvent être calculées soit "par la gauche", soit (par défaut) "par la droite" (c'est important si on veut insérer des valeurs qui figurent déjà dans la liste).

```
>>> from bisect import *                    # importe la totalité du module bisect (noms courts)
>>> tab = list(range(0,100,10)); tab        # une liste triée dans l'ordre croissant
>>> bisect_left(tab,25)                     # la valeur 25 viendrait s'insérer en position 3
3
>>> bisect_left(tab,60)                     # 60 s'insèrerait à gauche en position 7
7
>>> bisect(tab,60)                          # 60 s'insèrerait à droite en position 8
8
>>> insort(tab,28); tab                     # on insère 28 dans la liste
[0, 10, 20, 28, 30, 40, 50, 60, 70, 80, 90]
```

On peut également effectuer des insertions (ou chercher des positions) en spécifiant un intervalle de positions (mais attention, l'insertion effective risque alors de briser la nature globalement triée de la liste).

```
>>> tab                                     # rappelons le contenu de la liste tab
[0, 10, 20, 28, 30, 40, 50, 60, 70, 80, 90]
>>> bisect(tab,75)                          # 75 devrait normalement s'insérer en position 9
9
>>> bisect(tab,75,lo=2,hi=5)                # mais si on impose [2,5], il viendrait en position 5
5
>>> insort(tab,75,hi=7); tab                # on décide d'insérer 75 en position maximum 7
[0, 10, 20, 28, 30, 40, 50, 75, 60, 70, 80, 90]
>>> insort(tab,15,lo=6); tab                # on décide d'insérer 15 en position minimum 6
[0, 10, 20, 28, 30, 40, 15, 50, 75, 60, 70, 80, 90]
>>> insort(tab,5,lo=3,hi=7); tab            # insère 5 en imposant l'intervalle de positions [3,7]
[0, 10, 20, 5, 28, 30, 40, 15, 50, 75, 60, 70, 80, 90]
```

La documentation Python contient un certain nombre d'exemples montrant comment "humaniser" un peu les fonctions précédentes, notamment dans les recherches d'occurrences dans une liste triée.

On trouvera ces fonctions ici : http://docs.python.org/3.3/library/bisect.html#searching-sorted-lists

Voici un exemple d'utilisation du module bisect (créer une liste triée par insertions successives d'entiers aléatoires) :

```
>>> from bisect import *                    # importe le package bisect (noms courts)
>>> from random import *                    # importe le package random (noms courts)
>>> seed(4); tab = []                       # initialise graîne aléatoire, tableau vide
>>> for k in range(1,15):                   # à quatorze reprises
       insort(tab,randint(10,100))          #    on insère un entier aléatoire à deux chiffres
       print(tab)                           #    on affiche le tableau après cette insertion
[40]
[40, 48]
[23, 40, 48]
[23, 40, 48, 60]
[23, 40, 48, 60, 71]
[23, 29, 40, 48, 60, 71]
[21, 23, 29, 40, 48, 60, 71]
[18, 21, 23, 29, 40, 48, 60, 71]
[12, 18, 21, 23, 29, 40, 48, 60, 71]
[12, 18, 21, 23, 29, 40, 48, 60, 61, 71]
[12, 18, 21, 23, 29, 40, 48, 60, 61, 71, 80]
[12, 18, 21, 23, 29, 40, 47, 48, 60, 61, 71, 80]
[12, 17, 18, 21, 23, 29, 40, 47, 48, 60, 61, 71, 80]
[12, 17, 18, 21, 23, 29, 38, 40, 47, 48, 60, 61, 71, 80]
```


jean-miche.ferrard@ac-paris.fr

7.12 Le module copy

Adresse : http://docs.python.org/3.3/library/copy.html

Le module copy permet d'effectuer des copies "superficielles" ou "en profondeur" d'objets mutables, notamment de listes. Pour cela, il met à notre disposition les fonctions copy et deepcopy.

Les "copies en profondeur" sont utiles pour les *listes de listes* (les matrices, par exemple), pour créer une copie y totalement indépendante d'un objet original x (non seulement différence des adresses de x et y, mais différence des adresses des éléments qui se correspondent dans x et y). Les exemples ci-dessous illustrent ces subtilités :

```
>>> from copy import *              # importe les fonctions copy et deepcopy
>>> a = list(range(0,5)); a         # place dans a la liste [0,1,2,3,4]
[0, 1, 2, 3, 4]
>>> id(a)                           # voici où est située cette liste en mémoire
4318102808
>>> x = [2013, a, a, a]; x          # dans x, une liste avec 2013 puis trois exemplaires de a
[2013, [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
>>> y = x; y                        # poser y = x, c'est faire pointer y sur la même liste que x
[2013, [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
>>> z = copy(x); z                  # effectue une copie ''superficielle'' (shallow copy)
[2013, [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
>>> id(x), id(y), id(z)             # x et y pointent sur la même adresse, mais pas z
(4318102448, 4318102448, 4318102592)
>>> [id(e) for e in x]              # les adresses des éléments de x (et aussi de y, forcément)
[4322826768, 4318102808, 4318102808, 4318102808]
>>> [id(e) for e in z]              # les adresses internes à z sont les mêmes que pour x !!!
[4322826768, 4318102808, 4318102808, 4318102808]
>>> x[0] = 9999                     # modifier x[0] crée un nouvel objet donc une nouvelle adresse
>>> y                               # voici le nouveau contenu de x et de y
[9999, [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
>>> (id(x),id(y))                   # c'est normal car les adresses de x,y sont restées les mêmes
(4318102448, 4318102448)
>>> z                               # en revanche, la liste pointée par z n'a pas changé
[2013, [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
>>> id(z)                           # normal car l'adresse de z, inchangée, diffère de celle de x
4318102592
>>> a[4] = -1000                    # modifions maintenant seulement a[4]
>>> id(a)                           # l'adresse de la liste a n'a pas changé
4318102808
>>> y                               # donc la modification de a se voit dans x, et dans y
[9999, [0, 1, 2, 3, -1000], [0, 1, 2, 3, -1000], [0, 1, 2, 3, -1000]]
>>> z                               # mais aussi dans z !!!
[2013, [0, 1, 2, 3, -1000], [0, 1, 2, 3, -1000], [0, 1, 2, 3, -1000]]
>>> zzz = deepcopy(x)               # on fait maintenant une copie ''profonde'' de x
>>> [id(e) for e in x]              # rappelons les adresses des composantes de x
[4322826800, 4318102808, 4318102808, 4318102808]
>>> [id(e) for e in zzz]            # les composantes mutables de zzz ont changé d'adresse
[4322826800, 4318102880, 4318102880, 4318102880]
>>> a[4]=777                        # on modifie à nouveau un élément de a
>>> x                               # les composantes de x, pointant sur a, sont modifiées, normal
[9999, [0, 1, 2, 3, 777], [0, 1, 2, 3, 777], [0, 1, 2, 3, 777]]
>>> zzz                             # mais celles de zzz n'ont pas changé (normal aussi)
[9999, [0, 1, 2, 3, -1000], [0, 1, 2, 3, -1000], [0, 1, 2, 3, -1000]]
```

Important : l'instruction `y = x[:]` équivaut à une copie superficielle (c'est-à-dire à `y = copy.copy(x)`), ce qui permet de se passer du module copy (à condition qu'on veuille copier des listes dont les composantes sont non mutables!!!)

7.13 Autres modules et adresses utiles

Voici quelques modules qui pourraient s'avérer utiles... ou même indispensables.

Tout dépend de ce qu'on veut faire avec Python.

Le module tkinter, par exemple, permet d'écrire des interfaces graphiques pilotées par événements (boîtes de dialogues, tracés, gestion de la souris, etc.) et il ne saurait être décrit en quelques pages (il y a des livres pour ça).

- le module os.path (opérations sur les chemins d'accès)
  adresse : http://docs.python.org/3.3/library/os.path.html
- le module pickle (sauvegardes d'objets Python dans des fichiers)
  adresse : http://docs.python.org/3.3/library/pickle.html#module-pickle
- le module os (opérations liées au système d'exploitation)
  adresse : http://docs.python.org/3.3/library/os.html
- le module io (opérations d'entrées-sorties, en mode texte ou binaire)
  adresse : http://docs.python.org/3.3/library/io.html
- le module turtle (programmer les déplacements d'un crayon – d'une "tortue Logo" – à l'écran)
  adresse : http://docs.python.org/3.3/library/turtle.html
- le module tkinter (conception d'interfaces graphiques, programmation par événements)
  adresse : http://docs.python.org/3.3/library/tkinter.html
- le module tkinter.ttk (bibliothèque de widgets pour le module tkinter)
  adresse : http://docs.python.org/3.3/library/tkinter.ttk.html
- le module tkinter.tix (widgets supplémentaires pour le module tkinter)
  adresse : http://docs.python.org/3.3/library/tkinter.tix.html
- le module 2to3 (portage de scripts Python2 vers Python3)
  adresse : http://docs.python.org/3.3/library/2to3.html
- le module timeit (chronométrer précisément des petits bouts de code Python)
  adresse : http://docs.python.org/3.3/library/timeit.html
- le module sys (paramètres et fonctions spécifiques au système d'exploitation)
  adresse : http://docs.python.org/3.3/library/sys.html

La liste précédente est très loin d'être exhaustive !

On pourra en particulier consulter les adresses suivantes :

- la libraire standard de Python : http://docs.python.org/3.3/library/index.html
- L'environnement Canopy de la société ENTHOUGHT.
  Le « must » pour le calcul scientifique avec Python : https://www.enthought.com/products/canopy/
- la page d'accueil de NumPy : http://www.numpy.org
- la page d'accueil de SciPy : http://www.scipy.org
- la page d'accueil de matplotlib : http://www.matplotlib.org/
- la page d'accueil de IPython (« Interactive Python ») : http://www.ipython.org/
- une liste d'éditeurs Python, si Idle ne suffit pas : http://wiki.python.org/moin/PythonEditors
- les milliers de modules disponibles (écrits par des développeurs tiers) : http://pypi.python.org/pypi
- aide à l'installation de modules tiers : http://docs.python.org/3.3/install/index.html

jean-miche.ferrard@ac-paris.fr

Mises à jour

La version la plus récente de ce document est disponible à l'adresse :
http://www.mathprepa.fr/une-petite-reference-python.pdf

Auteur

Jean-Michel Ferrard, jean-miche.ferrard@ac-paris.fr

Licence d'utilisation de ce document