Département Informatique & Mathématiques Appliquées

Prise en main de `Python`

J. Gergaud[1], E. Navarro[2] et D. Rupprecht[3]

22 avril 2013

1. INP-ENSEEIHT, `joseph.gergaud@enseeiht.fr`
2. INP-ENSEEIHT, `emmanuel.navarro@enseeiht.fr`
3. Lycée Pierre de Fermat, `david@rupprecht.fr`

Table des matières

Chapitre 1

Introduction

On trouvera à la fin de ce document quelques références, dont le polycopié du Professeur D. Rupprecht dont on s'est fortement inspiré.

I Quelques dates

- une première version 0.9.0 publié par Guido Van Rossum en février 1991 (les débuts remontent à fin 1989)
- janvier 1994 : version 1.0
- octobre 2000 : version 2.0
- décembre 2008 : version 3.0 - changement majeur car pas de compatibilité avec la version 2
- juillet 2010 : version 2.7 - dernière version majeure pour la branche 2, pour prendre en compte certaines améliorations de Python 3.1
- septembre 2012 : version 3.3

II Qu'est-ce que l'informatique ?

- Leçon inaugurale de Gérard Berry au collège de France
  `http://www.college-de-france.fr/default/EN/all/inn_tec2007/lecon_inaugurale_.htm`
- Slides
  Slides
- Interview sur France culture
  `http://www.franceculture.fr/emission-l-eloge-du-savoir-algorithmes-machines-et-langages-2013-04-02`

III Les avantages et inconvénients de `Python`

III.1 `Python` un langage utilisé dans l'industrie

- You tube est codé en `Python` `http://python.about.com/b/2006/12/13/holy-web-apps-youtube-is-written-in-py`
  `htm`
- Python est utilisé à EDF pour leur code de mécanique de structures Code_Aster :
  `http://innovation.edf.com/recherche-et-communaute-scientifique/logiciels/code-aster-41195.html`
  - Logiciel de simulation numérique en calcul des structures développé depuis 20 ans par EDF R&D pour les besoins de recherche et les études d'expertise des installations de production et de transport d'électricité
  - Constamment développé, maintenu et enrichi de nouveaux modèles, Code_Aster comporte désormais près de 1.200.000 lignes de code source, pour la plupart en langages Fortran et Python
  - Pour satisfaire les exigences de qualité requises par l'industrie nucléaire, les fonctionnalités du code sont qualifiées par des campagnes de validations indépendantes, par comparaisons avec des solution analytiques ou expérimentales, des benchmarks avec d'autres codes. De plus, 2.000 tests sont gérés en configuration : exécutés chaque semaine, ils sont dévolus à la validation élémentaire et sont utiles comme base pédagogique
  - La documentation de Code_Aster, entièrement accessible sur un site internet, comporte plus de 12.000 pages

III.2 Les avantages de `Python`

- c'est un langage interprété

- c'est un langage typé
- c'est un langage orienté objet
- c'est un langage polivalent :
  - piloter un robot ;
  - créer une interface graphique ;
  - créer un site web très complet.
- il est multi-plateforme et gratuit
- Il existe plusieurs environnements (IDLE, IEP, Spyder, IPython, ...) et beaucoup de documentation ...

III.3 Et les inconvénients !

- 2 versions majeures non compatibles : `Python` 2 et `Python` 3. `Python` 3 est mieux, mais tous les modules ne sont pas compatibles
- c'est un langage interprété :
  - il est lent (mais les calculs sont optimisés avec les bibliothèques `NumPy` et `SciPy`) ;
  - beaucoup d'erreurs ne peuvent être détectées qu'à l'exécution
- le typage est dynamique
- langage très riche avec de nombreuses façons de faire
- langage peu contraignant : c'est au programmeur de savoir ce qu'il fait
- syntaxe simple mais beaucoup d'implicite (convention de nommage...)
- certains choix sont critiquables pour une bonne pratique de la programmation :
  - variable globale ;
  - traitement du type boolean ;
  - ...
- ⇒ **Il est très important d'apprendre à bien programmer**

IV Lancer python en mode interactif

- Vous connecter : login et mot de passe dans vos valisettes,
- Pour vous déconnecter, il suffit de cliquer en haut à droite¬ `Se déconnecter ...` puis de valider.
- ouvrir un terminal : `Application¬Accessoires¬terminal`
- lancer `Python` 3. Pour cela taper dans la fenêtre terminal : `python`

```
(python_stage_3.3)python36@kenobi :~$ python
Python 3.3.1 (default, Apr  8 2013, 16 :15 :57)
[GCC 4.4.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()                # ou quit()
(python_stage_3.3)python36@kenobi :~$
```

V Quelques généralités

- Tout ce qui est après le caractère #est un commentaire.
- Une instruction par ligne.
- Il est conseillé de ne pas avoir plus de 79 caractères par ligne.
- Importance de l'indentation dans `Python`.
- Un identificateur (nom de variable , d'une fonction, ...) commence par une lettre ou le caractère _ et est suivi par une lettre ou le caractère _ ou un chiffre.
- `Python` fait la distinction entre les minuscules et les majuscules.
- Certains noms sont interdits car il s'agit de mots clés (*cf.*. ci-après).

```
>>> a = 1        # affectation de la variable a
>>> A = 2
>>> print("A = ", A, " a = ", a)
A =  2  a =  1
>>> b = 2.0
>>> type(a)      # a est de type entier
```

```
<class 'int'>
>>> type(b)     # b est de type float
<class 'float'>
>>> b = "hello"
>>> print(b)
hello
>>> type(b)     # b est maintenant de type str (string, chaîne de caractère)
<class 'str'>
>>> A + a
3
>>> b
'hello'
>>> b+b
'hellohello'
>>> b + b
'hellohello'
>>> a, b = 3, "coucou"              # affectation multiple
>>> a
3
>>> b
'coucou'
>>> for = 1                         # for est un mot clé
  File "<stdin>", line 1
    for = 1
        ^
SyntaxError : invalid syntax
>>> 3/2                             # sous Python 2 cela donne l'entier 1
1.5
>>> 3.0/2                           # sous Python 2 cela donne aussi 1.5
1.5
>>> 2**3
8
```

Et la commande `help`

```
>>> help()

Welcome to Python 3.2!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http ://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

and                 elif                import              raise
as                  else                in                  return
assert              except              is                  try
break               finally             lambda              while
class               for                 nonlocal            with
continue            from                not                 yield
```

```
def                     global                  or
del                     if                      pass

help> modules

Please wait a moment while I gather a list of all available modules...

IN                      binhex                  itertools               shelve
IPython                 bisect                  json                    shlex
__future__              builtins                keyword                 shutil
_abcoll                 bz2                     lib2to3                 signal
_ast                    cProfile                linecache               site
_bisect                 calendar                locale                  smtpd
_codecs                 cgi                     logging                 smtplib
_codecs_cn              cgitb                   macpath                 sndhdr
_codecs_hk              cherrypy                macurl2path             socket
_codecs_iso2022         chunk                   mailbox                 socketserver
_codecs_jp              cmath                   mailcap                 sqlite3
_codecs_kr              cmd                     marshal                 sre_compile
_codecs_tw              code                    math                    sre_constants
_collections            codecs                  mimetypes               sre_parse
_compat_pickle          codeop                  mmap                    ssl
...

Enter any module name to get more help.  Or, type "modules spam" to search
for modules whose descriptions contain the word "spam".



help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>
```

On voit qu'il y a un module `math`, on peut essayer de l'utiliser :

```
>>> math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(2 * math.pi/3)
0.8660254037844388
>>> help('math.sin')
```

Dans la fenêtre apparaît alors l'aide suivant sur la fonction sin. Pour quitter cet aide et revenir à `Python` 3 il faut taper sur le caractère `q`.

```
Help on built-in function sin in math:

math.sin = sin(...)
    sin(x)

    Return the sine of x (measured in radians).
(END)
```

On affiche des résultats avec `print`, en séparant les arguments par des virgules :

```
>>> a = 3
>>> print(2*a, a*a, a**10)      # la syntaxe Python 2 est différente
6 9 59049
```

Les arguments peuvent être à peu près de tout type (en fait, il suffit que dans la classe, il existe une méthode particulière `__str__`qui renvoie une chaîne de caractère qui va représenter l'objet - le résultat est alors utilisé par `print`).

On peut préciser deux arguments particuliers `end=`et `sep=`pour donner ce qui va être la fin de la chaîne ou la partie qui sépare chaque argument à l'affichage :

```
>>> a = 3
>>> print(2*a, a*a, a**10, sep='---')
6---9---59049
>>> print(a); print(a)
3
3
>>> print(a, end=' :'); print(a)
3 :3
```

Remarques : le caractère de retour à la ligne est `n`. Le `;`permet de donner plusieurs instructions sur une même ligne, non recommandé.

Cette fois on utilise `input`

```
>>> x = input("donner un entier  : ")
donner un entier  : 43
>>> x
'43'
>>> type(x)
<class 'str'>
```

bon il va falloir apprendre à convertir...

Chapitre 2

Les types de référence

On peut consulter `http://fr.wikibooks.org/wiki/Programmation_Python/Type` pour une liste des types, ainsi que les différences entre Python2 et Python3 à ce sujet. Dans cette partie on ne rentrera pas en détails sur tous les types. Certains auront un chapitre dédié. On peut aussi consulter le site de référence Python pour tous les types : `http://docs.python.org/3.3/library/stdtypes.html`

I Type, id et références

Comme déjà dit, la fonction `type` permet d'obtenir le type d'un objet ou d'une variable. Un seconde fonction importante pour comprendre comment sont stockés les objets est `id`. Cette fonction donne l'identifiant de l'objet (en gros sa place en mémoire).

```
>>> a = 3
>>> id(a)
9157056
```

II Flottants, booléens et complexes

II.1 Float

On a déjà vu le type `int` [1]. Il existe un type flottant (`float`) standard (environ 16 chiffres significatifs, codés sur 64 bits). Un nombre est détecté comme `float`dès qu'il contient un point décimal ou un exposant de 10 défini avec `e`.

```
>>> 234.5
234.5
>>> type(10.)
<class 'float'>
>>> 3e10
30000000000.0
```

Il existe des librairies pour le calcul multi-précision, comme mpmath [2] qui est entre autre utilisé par Sage, ou encore gmpy [3].

II.2 Complex

Il existe un type `complex`qui s'obtient en collant j à la partie imaginaire :

```
>>> 3+2j
(3+2j)
>>> 3+2*j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
```

1. pour Python2 il existe deux types distincts `int`et `long`(avec conversion automatique lorsque l'entier était trop grand)
2. `http://code.google.com/p/mpmath/`
3. `http://code.google.com/p/gmpy/`

```
>>> 3+2*4.1j
(3+8.2j)
```

II.3 Opérations

Pas grand chose de spécial, on a les opérations classiques.

`+,-,*`	opérations classiques	
`//`	division entière	
`/`	division flottante (resp. entière) sous `Python 3` (resp. `Python 2`)	convertit si besoin
`%`	modulo	
`divmod(x,y)`	quotient et reste de la division	
`abs(x)`	valeur absolue	int, float, complex
`int(x)`	convertit en entier en retirant la partie décimale	différent de partie entière
`float(x)`	convertit en float	
`pow(x,y)` ou `x**y`	puissance	avec int, float et complex

II.4 Booléens

Enfin on dispose d'un type booléens (`bool`) avec deux valeurs True et False (attention aux majuscules !). Les opérations sont `and`, `or`, `not`.

```
>>> a = True
>>> type(a)
<class 'bool'>
>>> b = 5.4 < 2.2
>>> type(b)
<class 'bool'>
>>> a or b
True
```

III Les séquences : tuple, listes et chaînes

Ces trois types d'objets à priori différents font partie d'une même catégorie appelée « séquence ». Ils ont de nombreux points communs notamment lorsqu'on cherche à en extraire une partie. Ils ont également un comportement semblable en tant qu'`itérateurs` dans les boucles for. Commençons par un rapide tour de ces objets.

III.1 Tuple

Un tuple est une suite (ordonnée) d'éléments (pas forcément de même type), séparée par une virgule (entourés par des parenthèses, même si ce n'est pas obligatoire dans la définition).

```
>>> l = (2, 3, 6, "a", 1.3)
>>> type(l)
<class 'tuple'>
```

Comme les autres séquences, on peut accéder à un élément quelconque par sa position : attention le premier élément est en position 0 !

```
>>> l[0]
2
>>> l[3]
'a'
>> len(l)
```

On verra dans une prochaine section, toutes les manières d'extraire une sous-séquence.

Attention de ne pas confondre un objet et un tuple à un seul élément :

```
>>> a = (2)
>>> type(a)
<class 'int'>
>>> b = (2, )
>>> type(b)
<class 'tuple'>
>>> c = 2,      # les parenthèses ne sont pas obligatoires
>>> type(c)
<class 'tuple'>
```

III.2 Listes

Il y a beaucoup de façons différentes de définir une liste (il y aura tout un chapitre sur les listes). La plus simple ressemble à la définition d'un tuple, mais encadré par des crochets

```
>>> une_liste = [2, 4.3, "bonjour", 3]
>>> une_liste[2]
'bonjour'
>>> print(une_liste)
[2, 4.3, 'bonjour', 3]
```

Attention : comme d'habitude, une variable est un alias sur un objet en mémoire. Cela peut sembler étrange car le comportement suivant n'est pas forcément celui auquel on s'attend :

```
>>> l = [2,3,4]
>>> l2 = l
>>> l2[1] = 0
>>> l2
[2, 0, 4]
>>> l
[2, 0, 4]
```

ainsi `l2` n'est pas une copie de la liste `l` mais un nouvel alias vers le même objet. Une modification de l'une modifie l'autre. Pour comprendre regardons l'`id` :

```
>>> id(l)
4300853472
>>> id(l2)
4300853472
```

Si on veut une vraie copie, il faut utiliser le module `copy` (*cf.* le chapitre sur les listes).

III.3 Chaînes de caractères

Plusieurs façons de définir des chaînes de caractères (string - classe `str`) : avec des guillemets ou des apostrophes. Suivant ce qu'on utilise pour délimiter, l'autre caractère est accessible dans la chaîne :

```
>>> ex1 = "Une premiere chaine"
>>> ex2 = "avec des '', ca marche !"
>>> print(ex2)
avec des '', ca marche !
>>> ex2 = "mais pas avec " puisque la chaine est terminee"
  File "<stdin>", line 1
    ex2 = "mais pas avec " puisque la chaine est terminee"
                         ^
SyntaxError : invalid syntax
>>> "ou alors comme ca \", ca marche"
'ou alors comme ca ", ca marche'
>>> 'ou encore \' si on prefere'
"ou encore ' si on prefere"
```

Le caractère d'échappement \ a plusieurs utilisations. Il permet décrire sur plusieurs lignes une ligne Python trop grande. Il permet aussi d'accéder à certains caractères dans les chaînes : pour utiliser les " ou ', pour certains caractères spéciaux (les plus courants sont \n pour le retour à la ligne, \t pour une tabulation) :

```
>>> ex3 = "Bonjour !\nLa chaine est sur plusieurs lignes\nenfin presque"
>>> ex3
'Bonjour !\nLa chaine est sur plusieurs lignes\nenfin presque'
>>> print(ex3)
Bonjour !
La chaine est sur plusieurs lignes
enfin presque
```

Une dernière méthode pour saisir des chaînes avec des caractères spéciaux : les triples guillemets ou triples apostrophes.

```
>>> ex4 = """C'est promis
... c'est la derniere chaine pour cette fois
...     On termine comme on a commence !"""
>>> ex4
"C'est promis\nc'est la derniere chaine pour cette fois\n\tOn termine comme on
   a commence !"
>>> print(ex4)
C'est promis
c'est la derniere chaine pour cette fois
        On termine comme on a commence !
```

Comme avant on a une distinction entre l'objet chaîne et ce qui est affiché avec print.

IV Objets modifiables ou non

Bien entendu, tout le monde s'est demandé pourquoi on a deux types qui ont l'air les mêmes : les tuples et les listes. Dans Python, il y a deux types d'objets : les modifiables (*mutable*) et les non-modifiables (*immutable*). On verra notamment l'intérêt dans le passage des arguments d'une fonction.

```
>>> ex_tuple = (2,3,4) ; ex_liste = [2,3,4]
>>> ex_tuple[1]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> ex_liste[1]=1
>>> ex_liste
[2, 1, 4]
```

Les objets non-modifiables sont principalement les types numériques, les tuples, les strings.

Nous l'avons vu, il n'est pas possible de modifier « en place » un objet non-modifiable. Par contre certaines opérations de modification « en place », deviennent des modifications par copie sur des objets non-modifiables :

```
>>> a = [1, 2]
>>> id(a)            # rappel : id(obj) donne l'adresse mémoire de l'objet
18155136
>>> a += [42, 43]
>>> a
[1, 2, 42, 43]
>>> id(a)
18155136

>>> b = (1, 2)
>>> id(b)
18147880
>>> b += (42, 43)
>>> b
(1, 2, 42, 43)
```

```
>>> id(b)
139718140847832
```

V Opérations communes

Pour toutes les séquences, on dispose de certaines opérations communes :

`s + t`	concaténation des deux séquences
`s * n`	création d'une séquence ou s est copiée n fois
`s[i]`	éléments en position i
`s[i:j]`	tranche entre l'élément i (inclus) et j (exclu)
`s[i:j:k]`	tranche entre l'élément i (inclus) et j (exclu) avec un pas de k
`len(s)`	nombre d'éléments de s
`max(s), min(s)`	maximum et minimum de s

Dans cette partie on ne parlera que de la lecture des éléments (par *slicing*). On verra comment cela s'utilise (pour les listes puisque chaîne et tuple ne sont pas modifiables) en affectation dans le chapitre sur les listes.

```
>>> l = [0, 1, 2, 3, 4, 5, 6, 7, 8]   # ou en plus court : l = range(9)
>>> l[3]  # le 3eme element de l
3
>>> l = [3, 2, 4, 8, 9, 1, 8, 0]
>>> l[3]  # le 3eme element de l
8
>>> l[2:4] # du 2eme au 3eme
[4, 8]
>>> len(l)
8
>>> l[1:7:2] # du 1 au 7 (exclu) de 2 en 2
[2, 8, 1]
```

Pour la concaténation :

```
>>> l = [1, 2, 3] + [4, 5, 6]
>>> l
[1, 2, 3, 4, 5, 6]
>>> l2 = l*2
>>> l2
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

Cela fonctionne sur les chaînes de caractères :

```
>>> chaine = "Une chaine de caracteres"
>>> chaine[5]
'h'
>>> chaine[5:15:3]
'hndc'
```

On peut compter les éléments à l'envers avec des nombres négatifs, ou laisser certains indices vides pour aller au début ou à la fin : à vous de comprendre comment sont indexés les termes

```
>>> chaine
'Une chaine de caracteres'
>>> chaine[10:]
' de caracteres'
>>> chaine[10:-3]
' de caracte'
>>> chaine[2::4]
'ea cce'
```

Il reste quelques fonctions supplémentaires :

`s in t`	test d'appartenance
`s not in t`	le contraire
`s.index(x, i, j)`	position de `x` dans `s` entre `i` et `j` (`i` et/ou `j` ne sont pas obligatoires)
`s.count(x)`	nombre d'occurences de `x` dans `s`

```
>>> chaine = "c'est pas toujours facile de trouver de bons exemples"
>>> 'il' in chaine
True
>>> 'bac' not in chaine
True
>>> chaine.index('a', 9)
20
>>> chaine.count('ou')
3
```

Bon d'accord, ce n'est peut-être pas ce qu'il faut montrer aux étudiants en priorité si on veut faire faire un peu d'algorithmique et de parcours de chaînes...

VI Conversions

C'est assez simple, il suffit en général d'une syntaxe sous la forme `type(donnees)` :

```
>>> chaine = '123'
>>> int(chaine) + 3
126
>>> float(chaine) + 3
126.0
>>> str(23 + 12)
'35'
>>> tuple = (2, 4, 5, 3)
>>> list(tuple)
[2, 4, 5, 3]
>>> # la plupart des objets ont une méthode '__str__' pour se transformer en
    chaîne de caractères
>>> str(tuple)
'(2, 4, 5, 3)'
```

On peut aller plus loin et évaluer une chaîne :

```
>>> chaine = "12*45"
>>> int(chaine)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12*45'
>>> eval(chaine)
540
>>> from math import sin, pi
>>> eval("sin(2*pi) + 3")
2.9999999999999996      # et oui, c'est du calcul numérique
```

Chapitre 3

Contrôle du flux

I Introduction

Une des particularité de `Python` est que la structuration du code est lié à l'indentation des instructions. Pour délimiter une série d'instructions dans un test, une boucle, il n'y a pas de délimiteurs particuliers (de type begin/end ou { }), l'indentation suit les blocs d'instructions (ou le contraire, à vous de voir).

Qu'est ce que l'indentation : au moins un espace ou une tabulation. Il est très très fortement recommandé d'utiliser systématiquement une suite de 4 espaces (ou moins, mais dans ce cas, il faut rester cohérent et garder toujours le même nombre). La plupart des éditeurs dédié à `Python` convertissent automatiquement les tabulations en une suite de 4 espaces.

La première ligne de ces instructions, comme la définition d'une fonction, se termine toujours par deux points (le caractère :).

II Instructions `if then else`

```
x=2
if x>0:
   print("le nombre est positif")
   x+=1  # on incremente - aucun interet, c'est pour montrer
else:
   print("x est negatif")
   x=3
print(x)
```

On peut compliquer avec des `elif`(else if) :

```
if test1:
    bloc1
elif test2:
    bloc2
elif test3:
    bloc3
else:
    bloc4
```

et bien entendu, on peut imbriquer plusieurs blocs de cette façon :

```
if test1:
    debut_bloc_1
    if sous_test1:
        sous_bloc_interne
    else:
        sous_bloc_sinon
    fin_bloc_1
```

On peut également condenser lorsque les blocs sont très simples (une instruction) :

```
if x>=0: y = x
else: y = -x    # au passage, il est encore plus simple de faire y = abs(x)
```

le code doit rester lisible, et c'est déjà assez limite...

Quelques remarques :

- le test d'égalité se fait avec `==` (ne pas mélanger avec l'affectation)
- On a bien entendu les règles de comparaison standard : `<`, `>`, `<=` et `>=`, ainsi que `!=` (pour différent), avec tous les objets qui ont les fonctions de tests correspondantes (c'est fait pour les entiers, flottant, les chaînes, les listes... à vous de tester pour comprendre quel critère est utilisé).
- il y a le test d'appartenance `in`
- on dispose des opérateurs `and` et `or` pour combiner plusieurs tests, ainsi que des parenthèses pour regrouper convenablement les tests. Il existe également la négation `not`.
- évaluation paresseuses : les expressions booléennes ne sont pas forcément évaluées complétement : si `test1` est `False`, alors `test1 and test2` renverra `False` sans même évaluer le second test (et c'est similaire avec `or`).

```
>>> (2>3) or (3<4)
True
>>> 2>3 and 4<3    # les parentheses ne sont pas obligatoires
False
>>> 2<3<4          # on peut enchainer
True
>>> 2<3>=1         # a ne pas faire !
True
>>> True or fonction_de_test()  # la fonction_de_test ne serra pas évaluée
True
```

En fait les tests utilisés dans `if` ou `while` ne demande pas forcément un vrai test (au sens où on l'entend en général). Les valeurs considérées comme `False` sont :

- évidemment la valeur `False`,
- la constante `None` (constante du langage),
- tout valeur numérique égale à 0 (0, 0.0, $0.0j$),
- les chaînes, listes, tuples, dictionnaires vides.

```
>>> a = ""
>>> if a:
...     print("quelque chose")
... else:
...     print("rien")
....
rien
```

On peut tester le fait que la chaîne `a`est non vide avec `if a!= "":`, mais il est plus efficace d'utiliser `if a:`.

Il existe beaucoup d'autres méthodes et fonctions qui permettent de tester différents critères :

```
>>> a = 3
>>> isinstance(a, float)  # on regarde si a est une instance de la classe float
False
>>> isinstance(a, int)    # classe int
True
```

et énormément de critères sur les chaînes (voir chapitre correspondant)

```
>>> c = "coucou"
>>> c.islower()
True
>>> c.isdigit()
False
```

III Instruction `for`

Commençons par le plus simple (le principe est le même pour l'indentation) :

```
for i in range(10):
    print(i)
```

On peut manipuler `range` sous les formes `range(n)` pour les entiers de 0 à $n-1$, `range(a,b)` pour ceux entre a et b (strictement) et `range(a,b,p)` lorsqu'on veut ajouter un pas de p.

La boucle `for` ne s'utilise pas forcément avec la fonction `range`. Il est en fait possible d'utiliser des boucles `for` sur tout objet « itérable » (c'est un type particulier d'objets, dont la classe possède des instructions qui permettent de parcourir les données de l'objet).

Les objets itérables que l'on a déjà vu sont :
- `range` : évidemment
- les listes et les tuples
- les chaînes de caractères (on parcourt l'ensemble des caractères de la chaîne)

Quelques exemples simples :

```
>>> l = ['a', 3, 4, 'plus grand']
>>> for e in l:
...     print(e)
...
a
3
4
plus grand
```

ou

```
>>> for e in "les caractères":
...     print(e, end="-")
...
l-e-s- -c-a-r-a-c-t-è-r-e-s-
```

III.1 Modification de la variable

La variable utilisée dans une boucle `for` va décrire un à un les éléments de l'itérateur donné, même si elle est modifiée dans la boucle :

```
>>> for i in range(8) :   # equivalent à "for i in [0, 1, 2, 3, 4, 5, 6, 7] :"
...     i+=2
...     print(i,end=" ")
...
2 3 4 5 6 7 8 9
```

Dans cet exemple, i prend les différentes valeurs imposées par l'itérateur `range`, même si elle est modifiée dans le corps de la boucle, elle prend la valeur suivante de l'itérateur au passage suivant.

Le comportement de `for i in range(10):` est donc différent des boucles de type `for(i = 0; i<10; i++)` en C par exemple.

IV Instruction `while`

Il existe également une instruction `while` :

```
while test :
    bloc1
```

V Quelques compléments

V.1 Introduction

les instructions suivantes ne sont à utiliser que dans des cas très particulier et sont en général à proscrire dans le cadre d'une bonne pratique de la programmation.

V.2 break et continue

Aussi bien dans les boucles `for` que les boucles `while`, on peut utiliser ces deux instructions :
- `break` : sort (violemment) de la boucle et passe à la suite
- `continue` : termine l'itération en cours et passe à la suivante (retour en début de boucle avec l'itération suivante dans une boucle `for` ou retour au test dans `while`).

Quelques exemples sans intérêt :

```
a=0
while (a>=0):
    a+=1
    if (a==5): continue
    elif (a==10): break
    else: print(a)
```

devinez ce qu'il va se passer... et essayez !

Dans le même genre d'idée, qu'affiche ces instructions ?

```
>>> for i in range(11):
        if i%2: continue
        else: print(i)
```

De façon général, il est peu recommandé d'utiliser ces instructions. Vous le voyez cela rend, en général, la compréhension de la boucle difficile !

Mini-exercice : rendre ces deux boucles plus lisibles.

Corrigé :

```
for a in range(1,10):
    if a != 5:
        print(a)
# et
for i in range(0, 11, 2):
    print(i)
```

V.3 What else ?

Pour terminer, on peut ajouter un bloc d'instructions en fin de boucle, qui sera réalisé si la sortie de la boucle s'est faite proprement (c'est-à-dire sans `break`) avec l'instruction `else`

```
while test:
    bloc
else:
    si sortie propre
```

Un cas d'utilisation typique (et souhaitable) de `break` et `else` dans uns boucle de recherche :

```
phrase = ["une", "liste", "de", "chaines"]
for mot in phrase:
    if mot.startswith("li"):
        print("mot trouvé !")
        break
else:
    print("mot non présent...")
```

Chapitre 4

Fonctions

I Introduction

Avant de voir les fonctions, nous allons parler de script et voir comment exécuter un programme écrit en `Python`. Pour cela on va écrire dans un fichier texte un petit programme `Python` que nous exécuterons ensuite. l'éditeur de texte que l'on va utiliser ici s'appelle `geany`. pour cela :

- ouvrir un nouveau terminal (`Application¬Accessoires¬terminal`) ;
- taper `geany &`
  Cela vous ouvre l'éditeur de texte tout en gardant la fenêtre du terminal active.
- Ensuite taper (on reviendra sur la première ligne plus tard)

```
# -*- coding: utf-8 -*-
""" Premier essai de script Python
Petit programme simple affichant une suite de Fibonacci, c.à.d. une suite
de nombre dont chaque terme est égale à la somme des deux précédents

Créé le 20 avril 08:03:50 2013
auteur: gergaud
"""

print("Suite de Fibonacci : ")

a, b = 1, 1               # a & b servent au calcul des termes successifs
print(b)                  # affichage du premier compteur
for i in range(14):       # on affichera 15 termes en tout:
    a, b = b, a + b
    print(b)
```

- enregistrer ce fichier sous le nom `fibo.py` dans le répertoire d'où vous avez lancé `geany`
- Aller dans la fenêtre du terminal et taper `python fibo.py`. Vous obtenez alors l'exécution du programme `Python`.

```
(python_stage_3.3)python36@kenobi :~$ python fibo.py
Suite de Fibonacci :
1
2
3
5
8
13
21
34
55
89
144
233
377
```

```
610
987
```

Tout ceci est un peu lourd, c'est pourquoi, il a été créé des environnements. Nous allons ici utiliser l'environnement `IEP`. Pour le lancer, il suffit de taper dans la fenêtre du terminal `iep` ou de cliquer sur le lien présent sur le bureau.
Pour retrouver et exécuter votre fichier il suffit alors :

- de cliquer sur `File¬Open` et de sélectionner le fichier `fibo.py`
- d'exécuter le fichier en cliquant ensuite sur `Run¬Run file as script` (ou le raccourci `CTRL+SHIFT+E`).

II Définition d'une fonction

- Fichier `premiere_fonction.py`

```
# -*- coding: utf-8 -*-
"""
Créé le 20 avril 08:03:50 2013

auteur: gergaud

Fichier qui ne comprend que la définition d'une fonction très simple
"""

def f(x):
    """cette fonction calcule quelque chose d'assez complique...

       parameters (input)
       ------------------
       x : float

       return
       ------
       y : float
           = 'x'+ 1
       """
    y = x + 1
    return y
```

- Fichier `script2.py`

```
# -*- coding: utf-8 -*-
"""
Créé le 20 avril 08:03:50 2013

auteur: gergaud

Deuxième script : exemple d'appel d'une fonction d'un autre module

"""
import premiere_fonction

a = premiere_fonction.f(3)
print('a = ', a)
```

- Exécuter le programme principal `script2.py`
- Taper `help('premiere_fonction.f')` dans la fenêtre de commande `Python`
- On obtient alors

```
>>> help('premiere_fonction.f')
Help on function f in premiere_fonction:

premiere_fonction.f = f(x)
```

```
    cette fonction calcule quelque chose d'assez complique...

    parameters (input)
    ------------------
    x : float

    return
    ------
    y : float
        = `x`+ 1
```

III Les arguments

Bien entendu, on ne va pas se limiter à des fonctions avec un seul argument... enfin !!!

III.1 Plusieurs arguments

rien de surprenant :

```
>>> def f(x, y):
        return x*y
>>> f(3, 4)      # en plus cela donne le bon résultat
12
```

III.2 Valeurs par défaut

On peut, à la fin de la suite des arguments préciser que certains arguments ont une valeur par défaut lorsque cette valeur n'est pas transmise. Bien entendu, on ne peut pas mélanger les arguments puisque l'interpréteur ne peut pas savoir dans quel ordre les prendre :

```
>>> def f(x, y=1):
        return x*y
>>> f(2, 4)
8
>>> f(6)
6
```

Exercice III.1
Jusqu'où peut-on aller dans les arguments par défaut ? doit-on tous les mettre ou aucun ? à vous de tester.

III.3 Arguments nommés

Il est parfois commode de donner un nom aux différents arguments, cela permet d'appeler la fonction de façon assez naturelle et en transmettant les arguments dans un ordre quelconque.

```
>>> def divise(dividende, diviseur=1):
        return dividende/diviseur
>>> divise(18,4)
4.5
>>> divise(dividende=56, diviseur=3)
18.666666666666668
>>> divise(diviseur=7, dividende=87)
12.428571428571429
```

On n'est alors pas obliger de respecter l'ordre lorsqu'on appelle la fonction du moment que tous les arguments sans valeurs par défaut sont nommés dans l'appel.

III.4 Liste quelconque d'arguments

Il est possible de récupérer dans un tuple un nombre quelconque d'arguments. La syntaxe est sous la forme `def f(*args)`. On peut également combiner tous ce qui est dit avant avec l'ordre arguments, arguments avec valeurs par défaut, et tuple d'arguments.

```
>>> def f(x, y=0, *liste):
        print('les deux premiers : ', x, y)
        for i in liste: print(i, end=' ')
>>> f(3, 4, 5, 3, 4)
les deux premiers : 3 4
5 3 4
```

Exercice III.2
les incontournables : écrire une fonction qui détermine le maximum d'une suite d'éléments. Même chose avec la moyenne.

III.5 Liste quelconque d'arguments nommés

Pour finir, on peut en plus transmettre un nombre quelconque d'arguments nommés : c'est par exemple utile si une fonction peut avoir plusieurs options et qu'on doit filtrer suivant ces options. Imaginons qu'on ait à tracer un objet, on peut alors spécifier dans la fonction de tracé différentes options (couleur, pointillé...) sans savoir au final combien de telles options il peut y avoir (il n'y a pas besoin de reprendre l'entête de la fonction si on ajoute un type mais seulement modifier le corps en conséquence).

Pour cela, on a besoin de parler de dictionnaire... hélas ce sera fait plus tard.

La syntaxe est sous la forme : `def f(arguments, *args, **kargs)`

Remarque : Il est préférable, en général, d'éviter d'utiliser `*args` et `**kwargs`.

IV Portée des variables et espace de noms

La portée d'une variable provient directement de l'endroit où elle est créée. Une variable créée dans une fonction n'aura plus de visibilité lorsqu'on va quitter cette fonction.

L'ordre standard lorsque l'interpréteur doit évaluer le contenu d'une variable, d'un nom :

- espace local
- espace global : lorsque la variable n'a pas été définie localement,
- espace interne : les variables et fonctions de `Python` (`None`, `len`, ...)

```
>>> def f(x):
...   print(a, x)
>>> f(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
NameError: global name 'a' is not defined
>>> a = 3
>>> f(2)
3 2
>>> x = 6
>>> f(2)
3 2
```

Comment sont modifiées les variables à l'intérieur d'une fonction ?

```
>>> def f():
...     a = 3
...     print(a)
>>> a = 1
>>> f()
3
>>> a
```

```
1
```

Cela semble complètement cohérent. Dans la fonction f, la variable a est créée (dans l'espace de nom de la fonction f) mais ne modifie pas la variable globale.

On peut accéder à une variable globale dans une fonction en le spécifiant par la directive `global`

```
>>> def f():
...     global a
...     a += 1
...     print(a)
>>> a = 2
>>> f()
3
>>> a
3
```

et si on ne précise pas

```
>>> def f():
...     a += 1
...     print(a)
>>> a=2
>>> f()    # alors ???
```

Si l'on ne précise pas `global`, les variables globales ne peuvent être modifiées.

continuons :

```
>>> liste = [2, 3, 4]
>>> def f(l):
...     print(l)
...     l = []
...     print(l)
...
>>> f(liste)
[2, 3, 4]
[]
>>> liste
[2, 3, 4]
```

mais...

```
>>> def g(l):
...     print(l)
...     l[0] = 'modifie'
...     print(l)
...
>>> liste
[2, 3, 4]
>>> g(liste)
[2, 3, 4]
['modifie', 3, 4]
>>> liste
['modifie', 3, 4]
```

Quelle différence entre ces deux fonctions ? le mieux est de regarder les identifiants

```
def f(l):
    print("liste dans f : ", id(l))
    l=[]
    print("nouvelle liste : ", id(l))

def g(l):
    print("liste dans g :", id(l))
```

```
    print("élément 0 dans g : ", id(l[0]))
    l[0] = 'modifie'
    print("élément 0 modifié dans g : ", id(l[0]))
    print("liste après modif dans g : ", id(l))

liste = [2, 3, 4]
f(liste)
print("id global : ", id(liste))
print()
g(liste)
print("id global final : ", id(liste))
print("element 0 global : ", id(liste[0]))
```

avec, lorsqu'on exécute :

```
liste dans f :  139889367296264
nouvelle liste :  38856320
id global :  139889367296264

liste dans g : 139889367296264
element 0 dans g :  9157024
element 0 modifie dans g :  139889367713584
liste apres modif dans g :  139889367296264
id global final :  139889367296264
element 0 global :  139889367713584
```

L'identifiant `l` garde bien une portée globale dans la seconde fonction (c'est un alias vers la liste identifiée également par `liste`) puisqu'il n'est pas défini localement. Le comportement est donc totalement logique.

Pour terminer :

- `globals()` renvoie un dictionnaire avec les éléments globaux sous la forme nom :valeur
- `locals()` renvoie le dictionnaire des éléments locaux.

Exercice IV.1

Écrire une fonction remplace qui prend en argument

- *une chaîne, un premier caractère c_1, une second caractère c_2*
- *deux arguments optionnels debut et fin*

et qui renvoie une chaîne dans laquelle chaque caractère c_1 entre les positions début et fin est remplacé par c_2.

IV.1 Construction d'un argument par défaut

Lorsqu'une fonction est définie avec certains arguments par défaut, cette valeur par défaut est construite à la création de la fonction. Cela n'a aucune conséquence sur un objet non mutable (il ne bougera jamais), mais beaucoup plus sur un mutable : si on évalue le code suivant

```
>>> def f(l=[]):
...     l.append(1)
...     print(l)

>>> liste = [2, 2, 2]
>>> print("avant :", liste)
avant : [2, 2, 2]
>>> f(liste)
[2, 2, 2, 1]
>>> print("apres :", liste)
apres : [2, 2, 2, 1]   # tout est standard
>>> f()
[1]
>>> f()
[1, 1]
>>> f(liste)
[2, 2, 2, 1, 1]
>>> f()
```

```
[1, 1, 1]
```

Lors de la création de la fonction f, une liste vide est construite et est ensuite utilisée lors d'un appel où l'objet l n'est pas spécifié. Dit autrement la valeur de l'argument par défaut est construit une fois pour toute

Si vous avez bien compris que fait cette suite d'instructions ?

```
>>> def f(l=[]):
...     l.append(1)
...     print(l)
...     l=[1,2,3]
...     print(l)
>>> f()   # premier appel
[1]
[1, 2, 3] # pas de surprise
>>> f()   # seconde appel...
```

Ce comportement étrange est une source d'erreur classique en python. Pour avoir des arguments par défaut mutables, mais bien initialisés à chaque appel, on est obligé de faire :

```
def f(l=None):
    """ Fonction avec argument ayant une valeur par défaut mutable

    parameters (input)
    ------------------
    l : list
        si None alors `l` prend la valeur par défaut `[]`
    """
    if l is None:
        l = []
    l.append(1)
    print(l)
```

On peut tester :

```
>>> f()   # premier appel
[1]
>>> f()   # second appel...
[1]
```

V Synthèse

(i) Portée des variables :
- chaque appel à une fonction définit un nouvel espace de noms,
- toutes les variables affectées dans une fonction sont automatiquement locales,
- l'ordre de résolution est : local > global > interne

(ii) Passage de paramètres :
- tous les passages se font par paramètre,
- tous les paramètres passés sont locaux (modifier le contenu d'un mutable ne modifie pas l'identifiant du mutable).

(iii) Les valeurs par défauts sont construites une seule fois, lors de la création de la fonction.

VI Récursivité

Pas de soucis, on peut faire de la programmation récursive, avec ses avantages et ses inconvénients si on s'y prend mal...

```
def fact(n):                    # il n'y a pas plus original
    if n>1:
        return n * fact(n-1)
    else:
```

```
        return 1

print("6! = ", fact(6))
```

VI.1 La tortue

Une bonne méthode pour expérimenter la récursivité est d'utiliser cette bonne vieille tortue tout droit héritée du logo. Pour cela, on importera le module correspondant (dans le shell ou dans le module) avec

```
from turtle import *
```

On dispose alors de quelques fonctions intéressantes :

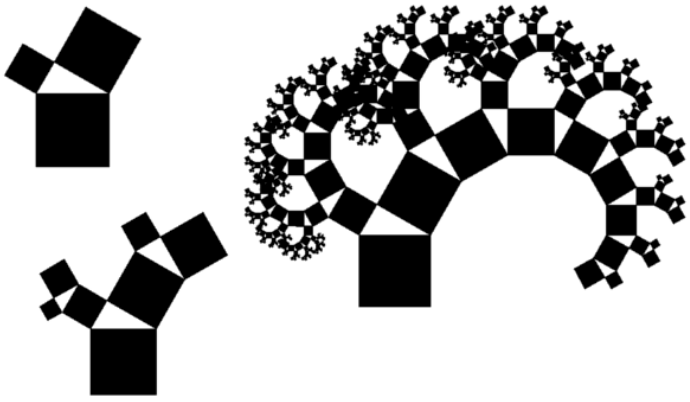
`reset()`	efface la fenêtre
`goto(x,y)`	se déplace sans écrire
`forward(d)` et `backward`	avance ou recule de d
`up()` et `down()`	monte et descend le crayon
`left(a)` et `right(a)`	tourne de a (degrés)
`penup()` et `pendown()`	monte ou baisse le crayon
`begin_fill()` et `end_fill()`	début/fin d'une zone à remplir

```
from turtle import *

reset()
for i in range(20):
    forward(100)
    left(150)
```

Exercice VI.1
utiliser le module turtle pour écrire une procédure qui trace un triangle (ou un flocon) de Von Koch avec une profondeur donnée

Exercice VI.2
un petit arbre pythagoricien :



On pourra charger le module math avec `from math import *` *afin d'ajouter quelques fonctions intéressantes comme le sinus (ou le nombre π).*

Chapitre 5

Modules

I Chargement d'un module externe

Afin d'avoir un interpréteur très léger, très peu de fonctions sont intégrées. Il faut donc charger ces fonctions qui se trouvent dans des modules. Python dispose d'un nombre impressionnant de modules. Certains font partie de la distribution standard (module math, random, copy par exemple), d'autres sont à télécharger et installer indépendamment (NumPy, SciPy, Matplotlib par exemple - même si certaines distributions non officielles de `Python` les intègrent). Il y a différentes possibilités pour charger un module - cela se place en début du module que l'on programme.

– la totale :

```
from module import *
```

cela charge le module, l'éxécute et place toutes les fonctions du module (et variables) dans l'espace de noms courant - cela peut être dangereux car on peut alors écraser une version précédente de la fonction. Les fonctions et variables s'appellent alors directemet par leur nom :

```
>>> from module import *
>>> sin(pi+e)
-0.41078129050290935
```

On peut ne charger que certaines fonctions du module avec la syntaxe

```
from module import fonctions_a_importer
```

Par exemple

```
>>> from math import sin, cos
>>> sin(1)
0.8414709848078965
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
```

– on ne mélange pas :

```
import module
```

Comme précédemment on importe le module. En revanche les fonctions ne sont plus placées dans l'espace de noms courants mais dans un espace propre. Pour y accéder, il faut préciser qu'on utilise la fonction dans cet espace de noms :

```
>>> import math
>>> math.sin(math.pi+2)
-0.9092974268256817
```

Évidemment c'est plus lourd, surtout si le nom du module est long... On peut simplifier en précisant un nom pour cet espace de noms :

```
>>> import math as m
>>> m.sin(m.pi+1)
-0.8414709848078964
```

Les deux utilisations sont possibles suivant les situations. Pour les petits programmes, on peut directement importer dans l'espace de noms. Pour les programmes plus conséquents, il vaut mieux importer le module avec `import module` afin d'éviter les conflits (sauf lorsqu'on utilise très intensivement les fonctions du module, par exemple tkinter qui permet de faire des interfaces graphiques).

Un module est encore un objet comme un autre :

```
>>> import math
>>> type(math)
<class 'module'>   # c'est un objet de type module
>>> del math       # on efface l'objet de la memoire
>>> type(math)     # il n'existe plus...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

II Quelques modules importants

II.1 Au sujet de la complétion

Petite remarque générale sur la complétion automatique sous IEP (ou sur un autre éditeur mais avec éventuellement un autre raccourci). L'éditeur propose en effet les commandes qui commencent par ce qu'on a tapé, ou ce qui s'en rapproche le plus. Par exemple si on a importer le module `math` et que l'on tape
`math.s` dans la fenêtre "traitement de texte" alors IEP propose `sin, sinh` et `sqrt`. Il faut appuyer sur `TAB` pour valider un choix de complétion.

II.2 math

Pour la documentation complète : `http://docs.python.org/3.3/library/math.html`. On y retrouve la plupart des fonctions et constantes usuelles, par exemple :

- les constantes : `pi, e`
- quelques fonctions classiques : `sqrt, exp, log, log10`, ainsi que `ceil, floor, fabs, factorial` (pour un entier)
- les fonctions trigonométriques diverses : `sin, cos, tan, asin, acos, atan`
- les fonctions hyperboliques : `sinh, cosh, tanh, asinh, acosh, atanh`
- quelques fonctions spéciales : `gamma, lgamma` ($\ln \circ \Gamma$), `erf` (avec $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$) ainsi que sa fonction complémentaire `erfc`
- quelques autres plus ou moins utilisées...

Il existe un module assez proche pour traiter le cas des fonctions d'une variable complexe (module `cmath`).

II.3 random

Pour toutes les fonctions : `http://docs.python.org/3.3/library/random.html`. On n'en présente que quelques-unes ici, les plus habituelles

`random`	réel dans $[0, 1[$ (loi uniforme)
`randrange([debut, ]fin)`	entier entre debut et fin (non compris)
`randrange(debut, fin, pas)`	entier dans range(debut,fin,pas)
`randint(a, b)`	entier entre a et b compris (=`randrange(a,b+1)`)
`uniform(a, b)`	réel dans $[a, b[$ - loi uniforme
`gauss(mu,sigma)`	loi gaussienne de paramètre *mu* et *sigma*
`seed(a=None, version=2)`	initilialisation du générateur : l'élément a est un entier/chaîne ou rien - dans ce cas l'heure actuelle est utilisée
`choice(seq)`	retourne un élément de la séquence
`shuffle(liste)`	mélange la liste l (modifie la liste, ne renvoie pas une nouvelle liste)

II.4 copy

Un petit module qui permet de copier des listes (essentiellement - en fait tout type d'objet qui contient des objets mutables).

```
>>> import copy
>>> l = list(range(6))
>>> g = copy.copy(l)
>>> g
[0, 1, 2, 3, 4, 5]
>>> id(l); id(g)
140027462515024   # on obtient bien une copie de la liste (identifiant diffé
   rents)
140027462515096
```

La fonction `copy` ne fait une copie qu'au premier niveau

```
>>> l = [[0], [1]]
>>> m = copy.copy(l)
>>> id(l); id(m)
46948152
47001896
>>> id(l[1]); id(m[1])
46947720
46947720
>>> l[1][0] = 2
>>> m
[[0], [2]]
```

Pour créer complètement un nouvel élément en descendant dans les listes :

```
>>> l = [[0],[1]]
>>> m = copy.deepcopy(l)
>>> id(l); id(m)
46948296
46948152
>>> id(l[1]); id(m[1])
46948224
47001824
>>> l[1][0] = 2
>>> m
[[0], [1]]
```

II.5 time

Le module `time` (`http://docs.python.org/3.3/library/time.html`) permet de faire des manipulations sur le temps

```
>>> import time
>>> time.gmtime()
time.struct_time(tm_year=2013, tm_mon=3, tm_mday=24, tm_hour=17, tm_min=24,
    tm_sec=1, tm_wday=6, tm_yday=83, tm_isdst=0)
# OK ce n'est pas tres explicite, mais on peut formater cela
>>> time.strftime("%A %d %B %Y %H :%M :%S", time.gmtime())
'Sunday 24 March 2013 17 :26 :14'
# et si on essaie en francais
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'fr_FR.UTF-8')
'fr_FR.UTF-8'
>>> time.strftime("%A %d %B %Y %H :%M :%S", time.gmtime())
'dimanche 24 mars 2013 17 :28 :55'
# et oui faut travailler aussi le dimanche
```

On peut notamment utiliser la fonction `clock()` (ou, à partir de Python3.3, `process_time()`) afin de déterminer le temps passé dans une fonction

```
>>> t1=time.clock()
>>> max([x*y for x in range(10000) for y in range(10000)])
99980001
>>> t2=time.clock()
>>> t2-t1
7.079999999999999
```

La fonction ne mesure pas un vraiment temps écoulé, mais un temps « processeur ».

II.6 this

```
>>> import this
```

III Écrire son module

Lorsqu'on développe un projet conséquent, on a fortement envie de scinder son code en module indépendant : des modules qui contiennent un ensemble de fonctions qui peuvent être réutilisées dans différentes situations. On incorpore alors son module de la même façon qu'un module fourni (il faut simplement qu'il soit accessible - en général en le mettant dans le dossier du module principal, ça tourne).

Quelques commentaires sur `__name__` :

```
>>> __name__
'__main__'
```

on dispose donc d'une variable donnant le nom de l'espace de nom actuel (cette variable `__name__`). Par exemple, on dispose d'un module enregistré sous le nom `mod.py` qui contient cela :

```
def f(x):
    print(__name__)
    return(x+2)
```

on l'importe dans le shell :

```
>>> from mod import *
>>> __name__
'__main__'
>>> f(2)
mod
4
```

On sait donc à un moment dans quel espace de nom on se situe. Quel est l'intérêt simple de la chose... écrire dans un module, une portion de code qui ne sera exécutée que lorsque ce sera le module principal : on fait régulièrement cela pour inclure plusieurs éléments de tests du module, ces tests n'étant réalisés que lorsqu'on exécute directement le module. Un exemple : on crée un module calcul.py qui contient cela :

```
print("exemple de print à ne pas faire !")

def f(x,y):
   return(x*y)

if __name__=='__main__':
    print(f(3, 4))
    print(f("abc", 2))
    print(f([1,2,3], 3))
```

Lorsqu'on exécute ce module, on obtient :

```
(python_stage_3.3)python36@kenobi :~$ python calcul.py
Exemple de print à ne pas faire !
12
abcabc
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

En revanche, si on importe ce module, la partie de tests n'est pas évaluée.

```
>>> from calcul import *
Exemple de print à ne pas faire !
>>> f(3, 4)
12
```

IV Quelques exercices

Exercice IV.1
Utiliser Python et le module `random` *pour obtenir une réponse approchée aux questions suivantes :*
Une urne contient n boules blanches et 2 boules noires.

- *On tire successivement les boules de l'urne. Quelle est la position moyenne de la première boule noire tirée ? de la seconde ?*
- *On tire avec remise une boule de l'urne. On s'arrête lorsqu'on a tiré p fois une boule noire (p étant fixé). Quel est le nombre moyen de tirages ?*

Chapitre 6

Quelques remarques

I Sur le codage des caractères

À l'origine de l'informatique, les seuls caractères utilisés étaient les caractères latins sans accents. On pouvait donc coder ces caractères sur 7 bits (128 combinaisons possibles). Le code utilisé est le code ASCII [1]. Ensuite, il y a eu différents codes pour les autres caractères que ceux de la langue anglaise : les caractères accentués pour le français, les caractères grecques, ... Ainsi, pour le français, il y a eu la norme `latin-1`, mais aussi par exemple la norme windows ! Aujourd'hui, c'est la norme `Unicode` qui est la référence.

C'est cette norme qu'utilise `Python` 3 par défaut, mais pas `Python` 2. C'est pourquoi, il est préférable de dire à `Python` que l'on travaille avec un encodage `utf8` (encodage standard de la norme `Unicode`). Ceci se fait simplement en ajoutant, à tous vos fichiers, la première ligne :

```
# -*- coding : utf-8 -*-
```

Pour illustrer ceci, considérons les fichiers `car.py` et `car_utf8.py` suivant

```
a = "caractères accentués"
print(a)
```

```
# -*- coding : utf-8 -*-
a = "caractères accentués"
print(a)
```

Alors si on exécute ces modules sont `Python` 3 (commande `python3` ou commande `python`) et `Python` 2(commande `python2`) on obtient

```
(python_stage_3.3)python36@kenobi :~$ python3 car.py
caractères accentués
(python_stage_3.3)python36@kenobi :~$ python3 car_utf8.py
caractères accentués
(python_stage_3.3)python36@kenobi :~$ python2 car.py
  File "car.py", line 1
SyntaxError : Non-ASCII character '\xc3' in file car.py on line 1, but no
   encoding declared ; see http ://www.python.org/peps/pep-0263.html for details
(python_stage_3.3)python36@kenobi :~$ python2 car_utf8.py
caractères accentués
```

Remarque I.1
Bien sûr si vous le souhaitez vous pouvez travailler avec les codes `latin1` *ou* `windows`*. Pour cela il faut respectivement mettre à la première ligne* `# -*- coding: latin-1 -*-` *ou* `# -*- coding: CP1252 -*-`

II Pour une bonne pratique de la programmation

Il est préférable quand on programme en `Python` de respecter quelques usages :

1. American Standard Code for Information Interchange

- 4 espaces par niveau d'indentation.
- Pas de tabulations.
- Ne jamais mixer des tabulations et des espaces.
- Un saut de ligne entre les fonctions.
- Deux sauts de ligne entre les classes.
- Ne pas mettre trop de blancs (ne pas en mettre après ou avant des parenthèses, crochets ou accolades) :
  - Yes : `spam(ham[1], eggs : 2)`
  - No : `spam( ham[ 1 ], eggs : 2 )`

  Mais en mettre où il faut
  - Yes : `if x == 4 : print x, y ; x, y = y, x`
  - No : `if x == 4 : print x , y ; x , y = y , x`
  - Yes : `x = 2*x*y - 10`
  - No : `x=2*x*y-10`
- Mettre les noms des variables et fonctions en minuscules
- Utiliser des majuscules pour les noms des classes : StudlyCaps pour les classes.
- Garder une taille de ligne inférieure à 80 caractères.
- Utilisez la continuité implicite des lignes au sein des parenthèses/crochets/accolades :

```
def __init__(self, first, second, third,
             fourth, fifth, sixth):
    output = (first + second + third
              + fourth + fifth + sixth)
```

- ...

Pour ces conventions voir :
- http://www.python.org/dev/peps/pep-0008/
- https://larlet.fr/david/biologeek/archives/20080511-bonnes-pratiques-et-astuces-python/

Enfin pour faire une bonne documentation aller voir
- Un bon tutoriel : https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt
- Avec un exemple : https://github.com/numpy/numpy/blob/master/doc/example.py

Chapitre 7

Listes

I Construction

On a déjà vu comment construire des listes simples :

```
>>> liste1 = [1, 2, 3]
>>> liste2 = ['abc', 5]
>>> liste3 = list(range(10))  # Pour transformer un type 'range' en une liste
>>> mois = ['janvier', 'fevrier', 'mars',
...     'avril', 'mai', 'juin',                    # on peut ecrire une liste
...     'juillet' 'aout', 'septembre',             # sur plusieurs lignes
...     'octobre', 'novembre', 'decembre']
```

`Python` offre de plus un moyen élégant et très efficace de générer des listes : les définitions par *compréhension*. L'idée est de parcourir une (ou plusieurs) liste(s) (ou tout objet itérable), filtrer, modifier leurs éléments pour constuire une nouvelle liste. Quelques exemples pour comprendre le principe :

```
>>> l1 = [i**2 for i in range(11)]
>>> print(l1)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> l2 = [n**p for n in [2,3,5] for p in range(5)]
>>> print(l2)    # avec deux 'iterables'
[1, 2, 4, 8, 16, 1, 3, 9, 27, 81, 1, 5, 25, 125, 625]

>>> l3 = [i**2 for i in range(50) if i%2 == 0 and i%3 == 1]
>>> print(l3)    # avec un filtre
[16, 100, 256, 484, 784, 1156, 1600, 2116]
```

Exercice I.1
Essayez de prévoir le résultat et vérifiez...

```
>>> l = [[2, 3], [4, 5]]
>>> [x for b in l for x in b]
```

II Modification

On a déjà vu comment modifier un élément en utilisant son indice :

```
>>> liste = [1, 2, 3, 4, 5]
>>> liste[2] = 'modif'
>>> print(liste)
[1, 2, 'modif', 4, 5]
```

On peut appliquer le même principe pour modifier plusieurs éléments **successifs** d'une liste. Il faudra alors donner une liste (ou tout objet itérable) en remplacement :

```
>>> liste = list(range(11))
>>> print(liste)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> liste[6:9] = [0,0,0]
>>> print(liste)
[0, 1, 2, 3, 4, 5, 0, 0, 0, 9, 10]

>>> liste[2:4] = ['a','b','c','d','e']
>>> print(liste)    # on peut remplacer par une liste plus longue
[0, 1, 'a', 'b', 'c', 'd', 'e', 4, 5, 0, 0, 0, 9, 10]

# Note: on pourrait faire exactement la meme chose avec
liste[2:4] = 'abcde'
# car, quand on modifie une tranche, on la remplace par un objet iterable

>>> liste[3:8] = [3.14]
>>> print(liste)                    # ... ou plus courte
[0, 1, 'a', 3.14, 5, 0, 0, 0, 9, 10]

>>> liste[5:] = []
>>> print(liste)      # la liste vide permet d'effacer un bout de la liste
[0, 1, 'a', 3.14, 5]
```

Attention, les deux commandes suivantes donnent des résultats différents : `liste[3] = []` et `liste[3 :4] = []`. Pour effacer un ou plusieurs éléments d'une liste, on peut aussi utiliser la commande `del` :

```
>>> liste = list(range(11))
>>> del(liste[2])
>>> print(liste)
[0, 1, 3, 4, 5, 6, 7, 8, 9, 10]
>>> del(liste[7:])
>>> print(liste)
[0, 1, 3, 4, 5, 6, 7]
```

On peut aussi modifier en une seule commande des éléments **non successifs** d'une liste. Mais cette fois-ci, pour d'évidentes raisons de cohérence, la liste de remplacement doit posséder exactement autant d'éléments que ceux que l'on souhaite changer.

```
>>> liste = list(range(11))
>>> liste[3:8:2] = [0, 0, 0]
>>> print(liste)
[0, 1, 2, 0, 4, 0, 6, 0, 8, 9, 10]
>>> liste[3:8:2] = [1, 2, 3, 4]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: attempt to assign sequence of size 4 to extended slice of size 3
```

Exercice II.1
le crible d'Erathostène - écrire une fonction qui prend en paramètre un entier n et détermine la liste des entiers premiers inférieurs à n. Le principe est de partir d'une liste de 2 à n, de chercher le premier élément non nul (ici 2), de rayer ses multiples, puis de recommencer jusqu'à arriver au bout.

III Méthodes associées aux listes

Les listes étant des objets, elles ont plusieurs méthodes associées pour effectuer les tâches les plus courantes.

`l.append(a)`	ajoute l'élément `a` à la fin de la liste `l`
`l.insert(indice, a)`	insert l'élément `a` à l'indice `indice`. Si cet indice dépasse la taille de la liste, l'élément est ajouté à la fin
`l.extend(iter)`	ajoute à la fin de `l` la liste créée à partir de l'objet itérable `iter`
`l.pop(indice)`	retourne l'élément d'indice `indice` et le retire de la liste `l`. Par défaut, c'est le dernier élément qui est retiré. Si la liste est vide, la méthode renvoie une erreur
`l.reverse()`	inverse l'ordre des éléments de la liste `l`
`l.sort()`	tri la liste `l`
`l.index(a)`	retourne l'indice de l'élément `a` dans la liste `l` s'il est présent dans cette liste et une erreur sinon
`l.count(a)`	compte le nombre d'occurence de l'élément `a` dans la liste `l`

Il faut bien comprendre que ces commandes modifient la liste. Attention donc si on les utilise dans une fonction car il ne faut pas oublier que les paramètres sont passés par référence.

IV Fonctions appliquées à une liste

IV.1 Un peu de lambda-fonction

On n'a parfois pas besoin/envie de définir une fonction pour faire certains types de calculs (on va par exemple appliquer une fonction aux éléments d'une liste). On peut pour cela itiliser la directive `lambda` :

```
>>> f = lambda x: x*x
>>> f(3)
9
```

ou peut également préciser plusieurs arguments, voire même des valeurs par défaut :

```
>>> g = lambda x, y=2: x*y
>>> g(5, 7)
35
>>> g(6)
12
```

Cela permet de définir des fonctions courtes simplement.

IV.2 map, filter et zip

On peut se servir de `map` pour appliquer une fonction aux éléments d'une liste :

```
>>> l = list(range(6))
>>> l
[0, 1, 2, 3, 4, 5]
>>> result = map(lambda x: (x+1)**2, l)
>>> result
<map object at 0x28e67d0>
# 'result' n'est plus une liste, mais un itérateur.
# le résultat du map n'est construit que lorsque l'on itère dessus.
# Attention: comportement différent en python2
>>> list(result)
[1, 4, 9, 16, 25, 36]
```

Cela dit il est plus clair d'utiliser une construction par compréhension :

```
>>> [(x+1)**2 for x in l]
[1, 4, 9, 16, 25, 36]
```

Le filtrage d'une liste se fait de la même manière avec la fonction `filter` : cela renvoie de nouveau un itérateur.

```
>>> def f(x): return x**2>2 and 3*x<10
>>> list(filter(f, range(10)))
[2, 3]
```

plus directement avec une lambda-fonction

```
>>> list(filter(lambda x: x<4, range(10)))
[0, 1, 2, 3]
```

La encore il est plus simple de le faire avec une construction directe :

```
>>> l = range(10)
>>> [x for x in l if x**2>2 and 3*x<10]
[2, 3]
```

ou encore

```
>>> filtre = lambda x: x**2>2 and 3*x<10
>>> [x for x in range(10) if filtre(x)]
[2, 3]
```

Pour finir, la commande `zip` prend deux ou plusieurs listes en arguments en renvoie un itérateur composé des tuples formés par les éléments des listes en même position (s'arrête à la plus courte) :

```
>>> l1 = [1, 2, 3]
>>> l2 = ['a', 'b', 'c', 'd']
>>> list(zip(l1, l2))
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> list(zip(l2, l2, l1))
[('a', 'a', 1), ('b', 'b', 2), ('c', 'c', 3)]
```

Exercice IV.1
On vous donne une liste d'entier. Déterminer le nombre d'éléments non nuls de cette liste.

Exercice IV.2
Écrivez un programme qui analyse un par un tous les éléments d'une liste de nombres (par exemple celle de l'exercice précédent) pour générer deux nouvelles listes. L'une contiendra seulement les nombres pairs de la liste initiale, et l'autre les nombres impairs.

Exercice IV.3
Déterminer le plus grand et le plus petit des produits de deux nombres distincts dans une liste de nombres.

Exercice IV.4
Écrivez une fonction qui prend en paramètre un tableau de nombres entiers, et qui recherche, dans ce tableau, la plus grande différence (en valeur absolue), entre un élément et son successeur. Votre fonction doit retourner cette différence.

Exercice IV.5
Les reines
On se donne sur un échiquier ($8x8$) les positions de reines (liste de couples). Déterminer le nombre de cases qui ne sont pas accessibles à ces reines.

Exercice IV.6
On a fait une jolie décoration sur un gateau avec différents fruits (confits si vous aimez) qui font le tour extérieur du gateau. On aimerait se couper une part qui contient tous les fruits, mais comme on est raisonnable, on veut qu'elle soit la plus petite possible. Quelle est la taille ?
Par exemple on décrira la répartition par une chaine `abaaacaccbbcbbbabbbaabbabbccbcbbccaccb` *(ou une liste) et on cherchera la plus petite sous chaine avec les différentes lettres (n'oubliez pas que les bouts se rejoignent).*

Exercice IV.7

Programmez le tri par insertion pour les listes.

Algorithme du tri par insertion : *on parcourt dans l'ordre toute la liste à trier et on classe l'élément i parmi les éléments qui le précéde.*

Exercice IV.8

Programmez le tri rapide pour les listes.

Algorithme du tri rapide :

- *Si la liste est vide ou ne possède qu'un élément, elle est triée.*
- *Sinon, on extraie le premier élément e et on sépare les éléments restant en deux listes ℓ_1 et ℓ_2 : ceux qui sont inférieurs ou égaux à e et ceux qui sont strictement supérieurs à e. On renvoie alors la concaténation de la liste ℓ_1 triée, de $[e]$ et de la liste ℓ_2 triée.*

Exercice IV.9

Programmez l'algorithme de Luhn qui permet de faire la somme de contrôle des cartes bancaires.

Algorithme de Luhn : *il procède en trois étapes.*

- *L'algorithme multiplie par deux un chiffre sur deux, en commençant par l'avant dernier et en se déplaçant de droite à gauche. Si un chiffre qui est multiplié par deux est plus grand que neuf, on considère alors son reste par la division euclidienne par 9.*
- *La somme de tous les chiffres obtenus est effectuée.*
- *Si le reste de la division par 10 est égal à zéro, alors le nombre original est valide.*

Exercice IV.10

Puissance 4

Programmez un jeu de puissance 4 : vous avez une grille de 6 lignes et 7 colonnes remplie de 0. A tour de rôle les joueurs choisissent une colonne (le programme affiche le tableau avec des 0, 1 ou 2) et le jeu se termine dès qu'il y a un alignement de 4 chiffres identiques.

Exercice IV.11

Stations de métro

On vous donne en paramètre la description d'un plan de métro, sous la forme d'une liste des couples de stations entre lesquelles il existe une ligne de métro sur laquelle ces deux stations sont consécutives. On peut donc se déplacer entre ces deux stations sans passer par une quelconque autre station.

On vous indique une station de départ, et vous devez trouver la station la plus éloignée de cette station, lorsque l'on s'y rend en métro, c'est-à dire celle qui nécessite de passer par le plus grand nombre de stations intermédiaires. On ne tient pas compte du temps passé aux éventuelles correspondances.

Les stations de métro sont identifiées par des numéros. Tout ce qu'on vous donne, ce sont des couples de nombres, identifiant deux stations reliées directement.

Exercice IV.12

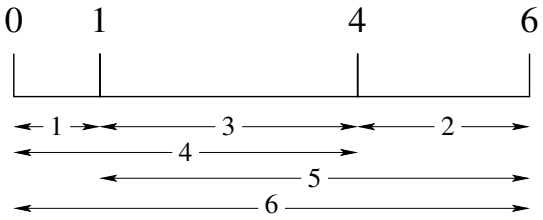
Labyrinthe

Vous disposez d'une grille de taille $N \times N$, remplie au hasard par M fois l'entier 1, les autres cases étant nulles. Les deux coins aux extrémités contiennent la valeur 1. Peut-on joindre ces deux coins, en se déplaçant hirizontalement et verticalement en ne passant que par des 1 ? On créera une fonction qui prend N et M en paramètre et qui créé cette grille avec exactement M fois 1 (dont les extrémités). Puis une autre qui teste si le parcours est possible (et éventuellement une qui crée le chemin le plus court...)

Exercice IV.13

En mathématiques, une règle de Golomb est une règle, munie de marques à des positions entières, telle que deux paires de marques ne soient jamais à la même distance ; en d'autres termes, chaque couple de marques mesure une

longueur différente des autres. Exemple de règle :



Quelques définitions :

- *L'ordre d'une règle de Golomb signifie le nombre de marques présentes. La règle ci-dessus est d'ordre 4.*
- *La longueur d'une règle de Golomb représente l'écart maximal entre 2 de ses marques. (par convention, la règle commence à zéro, donc c'est la valeur de la plus haute marque). La longueur de la règle ci-dessus vaut 6.*
- *Une règle de Golomb est dite optimale si sa longueur est la longueur minimale pour son ordre. La règle ci-dessus est optimale.*
- *Une règle de Golomb est dite parfaite si elle permet de mesurer toutes les longueurs entre 1 et n (n = longueur). La règle ci-dessus est aussi parfaite.*

(i) Coder une fonction qui prends une règle candidate en entrée, et retourne un tuple de booléens (Valide, Parfaite).

(ii) Réaliser une fonction prenant en paramètre un ordre et une longueur, et retournant toutes les règles de Golomb trouvées pour ces paramètres.

Chapitre 8

Les dictionnaires

I Principe

Les dictionnaires sont un autre type de données. Ce sont des tableaux associatifs : plutôt que d'avoir des cases numérotées, on a des cases désignées par un identifiant, appelé **clé** (en général une chaîne de caractères) et de nouveau un contenu quelconque associé à chaque identifiant. Les dictionnaires ne sont pas des séquences (il n'y a pas d'ordre sur les éléments), mais sont des objets modifiables (comme les listes) et sont aussi des itérateurs.

I.1 Définir un dictionnaire

Un dictionnaire est défini entre accolades, sous la forme `cle :valeur`. La clé est souvent une chaîne mais en fait, n'importe quel élément non-modifiable peut-être utilisé. Le dictionnaire vide s'écrit `{}`

```
>>> horaire = {'maths':12, 'physique':7, 'chimie':1, 'option':2, 'lv1':2, '
    francais':2}
>>> horaire['maths']  # on accede a un element
12
>>> horaire['chimie'] = 2  # on modifie la valeur de la cle chimie
>>> horaire['TIPE'] = 2    # on cree une nouvelle etiquette
>>> horaire
{'lv1': 2, 'option': 2, 'TIPE': 2, 'physique': 7, 'francais': 2, 'chimie': 2, '
    maths': 12}
```

On retrouve quelques fonctions classiques :

- nombre d'éléments :

```
>>> len(horaire)
7
```

- effacer un élément :

```
>>> del horaire['maths']
>>> horaire
{'lv1': 2, 'option': 2, 'TIPE': 2, 'physique': 7, 'francais': 2, 'chimie':
    2}
```

- test d'appartenance (sur les clés) :

```
>>> 'chimie' in horaire
True
>>> 'maths' in horaire
False
```

- itérer sur les clés

```
>>> for i in horaire: print(i)
lv1
option
TIPE
```

```
physique
francais
chimie
```

I.2 Les méthodes

`keys()`	les clés du dictionnaire (itérateur)
`values()`	les valeurs (itérateur)
`items()`	les couples (tuple) (itérateur)
`clear()`	efface tout le dictionnaire
`copy()`	crée une vraie copie du dictionnaire
`pop(cle[,d])`	retire la clé et renvoie sa valeur (`d` si la clé n'existe pas)
`popitem()`	retire et renvoie un couple clé-valeur
`get(cle[,default])`	comme `dico[cle]` mais retourne `default` si la clé n'existe pas

I.3 Parcourir le dictionnaire

Si on veut parcourir les clés :

```
>>> for i in horaire: print(i, end=" ")
lv1 option TIPE physique francais chimie
```

ou encore

```
for i in horaire.keys(): print(i ,end=" ")
lv1 option TIPE physique francais chimie
```

On peut alors accéder aux valeurs :

```
>>> for i in horaire.keys(): print(i, ":", horaire[i], end=" ")
lv1 : 2 option : 2 TIPE : 2 physique : 7 francais : 2 chimie : 2
```

Dans ce dernier cas, on peut directement parcourir les couples avec la méthodes `items()`

```
>>> for cle, valeur in horaire.items():
...     print(cle, ":", valeur)
lv1 : 2
option : 2
TIPE : 2
physique : 7
francais : 2
chimie : 2
```

Chapitre 9

Chaînes de caractères

I Méthodes sur les chaînes de caractères

On rappelle quelques éléments :

- une chaine (class `str`) est une séquence, avec toutes les opérations associées : `len` pour la longueur, `in` pour l'appartenance (sous-chaine), les techniques de sélection par tranches,
- une chaine est itérateur (caractère par caractère)
- une chaine n'est pas modifiable, toutes les méthodes renvoient donc une nouvelle chaine (ou autre chose)- elles s'utilisent toutes sous la forme `chaine.methode(...)`.

On peut consulter `http://docs.python.org/3.3/library/stdtypes.html#text-sequence-type-str` et, pour le module `string`, `http://docs.python.org/3.3/library/string.html`

I.1 Transformations

`center(longueur[, car])`	centre la chaine en remplissant avec `car` (espace par défaut) sur la longueur donnée
`ljust(longueur[, car])` et `rjust`	idem avec alignement à gauche/droite
`lstrip(cars)` et `rstrip`	retire en début (fin) de chaine les caractères donnés (espace par défaut)
`split(sep=None, max=-1)`	retourne une liste de chaines coupées par la sous-chaine `sep` (un nombre `max` de fois)
`rsplit`	idem par la droite
`splitlines(fin)`	sépare les lignes (caractères `\n,\r`) dans un tableau. Conserve les fin de ligne si `fin` est `True`
`expandtabs([taille])`	remplace les tabulations par des espaces (8 par défaut)
`join(l)`	concatène les éléments de l'itérable `l` en intercalant la chaine entre deux : `"-".join(["06", "07", "08", "09", "10", "11"])`
`lower(), upper(), capitalize()`	diverses transformations
`title(), swapcase()`	idem
`casefold()`	transforme certains caractères spécifiques à la langue
`encode(encoding, errors)`	encode dans le jeu de caractères donné (utf-8 par défaut)

I.2 Recherche

`count(sous[, debut[, fin]])`	retourne le nombre d'apparition de la sous-chaine entre début et fin (les occurences sont non-recouvrantes)
`index(sous[, debut[, fin]])`	position de la première apparition de `sous` - erreur `ValueError` si absente
`find(sous[, debut[, fin]])`	idem, retourne -1 si la sous-chaine est absente
`partition(sous)`	cherche la première occurence de `sous` et retourne un 3-uple avec les trois morceaux (avant, sous, après)
`replace(ancien, nouveau[, nombre])`	remplace `ancien` par `nouveau` - le fait seulement `nombre` de fois si précisé
`rfind`, `rindex`, `rpartition`	comme au dessus mais par la droite

I.3 Méthodes de tests

`isalnum()`	si alphanumérique (et au moins un caractère)
`isalpha()`	si alphabétique (et au moins un caractère)
`isdecimal()`	si ne contient que des caractères décimaux (et au moins un caractère)
`isdigit()`, `isnumeric`	idem mais avec certains caractères unicode spécifiques
`isidentifier()`	si la chaine peut être utilisé comme identifiant
`islower()` et `isupper()`	si en minuscule ou majuscule
`isspace()`	que des espaces (et au moins un)
`isprintable`	chaine affichable (sans caractère d'échappement spéciaux)
`istitle()`	
`endswith(suffixe[, start[, end]])`	si la chaine se termine par `suffixe`
`startswith(suffixe[, start[, end]]`	si la chaine commence par `suffixe`

II Formatage d'une chaîne

Il est fréquent d'avoir à créer une chaine à partir de certaines données (et de formater cette chaine, notamment pour l'affichage). On utilise pour cela la méthode `format`. La méthode décrite ici est la méthode `format` qui s'utilise sous la forme `chaine.format(arguments)`. Le détail sur `http://docs.python.org/3.3/library/string.html#formatstrings`.

```
>>> a, b, c = 2, 5, "fini"
>>> chaine = "premier : {}, second : {} et {}".format(a,b,c)
>>> print(chaine)
premier : 2, second : 5 et fini
```

les termes en accolades sont remplacés successivement par le contenu des variables. On peut utiliser plusieurs fois la même variable et même péciser l'ordre d'utilisation des arguments :

```
>>> chaine = "second : {1} et une autre fois {1}, le premier : {0} et enfin c'
    est {2}".format(a,b,c)
>>> print(chaine)
second : 5 et une autre fois 5, le premier : 2 et enfin c'est fini
```

On peut formater chaque champ en utilisant `<` (alignement à gauche), `>` (à droite), `^` (centré) en précisant le nombre d'espace à mettre (et même en précisant un autre caractère de remplissage)

```
>>> "test : {:<30}".format("chaine")
'test : chaine                         '
>>> "test : {:^30}".format("chaine")
'test :              chaine            '
>>> "test : {:_^30}".format("chaine")
```

```
'test : ____________chaine____________'
>>> "test : {0:_^30} - avec {1:0>5}".format("chaine", 7)
'test : ____________chaine____________ - avec 00007'
```

On peut également nommer les champs plutôt que les définir par leur position

```
>>> "{prenom}, {prenom} {nom}".format(nom="Bond", prenom="James")
'James, James Bond'
```

On peut également utiliser un dictionnaire pour les arguments, mais avec une option de conversion du dictionnaire en une séquence d'arguments nommés (on met ** devant le dictionnaire) :

```
>>> donnees = {'nom':"Bond", 'prenom':"James"}
>>> "{prenom}, {prenom} {nom}".format(**donnees)
'James, James Bond'
```

On peut également le faire avec une liste ou un tuple, mais avec une seule étoile dans ce cas :

```
>>> liste = ["James", "Bond"]
>>> "{0}, {0} {1}".format(*liste)
'James, James Bond'
```

De façon plus générale, on peut convertir ainsi une liste ou tuple en une séquence d'arguments avec * (idem avec ** pour tranformer un dictionnaire en une séquence d'arguments nommés) :

```
>>> liste = ["James", "Bond"]
>>> print(*liste, sep="---")
James---Bond
```

Pour formatter un nombre flottant la syntaxe est sous la forme `{ :x.yf}` où `x` désigne la taille totale, `y` le nombre de chiffre après le point. On peut forcer l'affichage du signe avec `{ :+x.yf}`

```
>>> "{:12.6f}".format(24672.9823498247)
'24672.982350'
>>> "{:12.4f}".format(24672.9823498247)
'  24672.9823'
>>> "{:+12.4f}".format(24672.9823498247)
' +24672.9823'
>>> "{:+12.4f}".format(-24672.9823498247)
' -24672.9823'
```

III Quelques exercices

Exercice III.1
Ecrire une fonction, qui prend en paramètre une chaine caractères, et qui affiche la première lettre de chacun de ses mots, en majuscule. La chaîne qu'on vous donne est composée uniquement de caractères minuscules non-accentués, et d'espaces. Deux mots sont séparés par un ou plusieurs espaces.

Exercice III.2
Ecrivez une fonction qui prend en paramètre une chaîne de caractères, et qui retourne le nombre de caractères du plus long mot de cette chaîne. La chaîne est constituée uniquement de caractères minuscules non-accentués, et les mots sont séparés par des espaces. (Un espace entre chaque mot).

Exercice III.3
Même chose mais en ajoutant différents signes de ponctuations.

Exercice III.4
Parenthésage On vous donne une chaîne contenant une expression parenthésée. Vérifeir si cette expression est bien parenthésée (autant de parenthèse ouvrante que fermante avec un nombre d'ouverture toujours plus grand que celui de fermetures).

Exercice III.5

La RFC 1855 intitulée "Netiquette Guidelines" définit les conventions de politesse sur les réseaux informatiques. Elle précise parmi beaucoup d'autres choses de poster les messages en moins de 80 colonnes.

Vous devez écrire un programme qui prend une chaîne de caractères en arguments, composée de mots séparés par des espaces (un seul espace entre deux mots), et qui doit remplacer certains de ces espaces par des retours à la ligne, de telle sorte que chaque ligne ne contienne pas plus de 80 caractères (elle peut contenir 80 caractères, sans compter le caractère de retour à la ligne). Il ne faut cependant pas passer à la ligne, s'il est encore possible de placer un mot.

Exercice III.6

Rot13

Le Rot13 est une méthode de chiffrement très simple qui consiste à remplacer un caractère par un autre à 13 caractères de là. « A » devient « N », par exemple. Utilisé pour coder les fichiers donnant la fin d'un film, l'humour noir ou provocant, la partie d'un message sur l'Usenet dans laquelle on insulte copieusement l'interlocuteur, etc.

Tous les caractères qui ne sont pas des lettres non accentuées de l'alphabet sont laissés tels-quels. Le principe de cette méthode est que si on réapplique un rot13 sur une chaîne cryptée par rot13, on retrouve le message d'origine (puisqu'il y a 26 lettres dans l'alphabet).

On pourra utiliser les fonctions `ord` *et* `chr`*.*

Chapitre 10

Fichiers

I Utiliser des fichiers

On utilise un fichier en l'ouvrant à partir de la fonction `open`. Cela crée alors un objet (c'est original) avec différentes méthodes. Pour avoir l'aide complète : `help(file)`. Il y a plusieurs paramètres possibles. Les plus standards sont les deux premiers : le nom du fichier et le type d'ouverture (en lecture, en écriture, en ajout...). Par défaut le type est « lecture » (`'r'`). Les autres types usuels sont `'w'` (write) pour l'ouverture en écriture (en vidant le fichier) et `'a'` (append) pour l'ajout à la fin du fichier (et création si le fichier n'existe pas). Je vous renvoie à la documentation complète pour plus d'options : `http://docs.python.org/3.3/library/functions.html#open` ainsi que celle sur les différents type de flux qui peuvent être créé : `http://docs.python.org/3.3/library/io.html`. Dans cette partie, on va essentiellement s'intéresser aux fichiers texte.

Une fois un objet fichier créé, on peut lire les données ou en écrire. À la fin on ferme le fichier :

```
>>> fichier = open("test.txt", "w")
>>> fichier.write("Une chaine dans le fichier\nUne deuxieme ligne.")
46   # la methode retoure le nombre de caracteres ecrits
>>> fichier.close()
```

Lorsqu'on regarde le contenu du fichier test.txt, on obtient (le caractère `\n` est un retour à la ligne).

```
Une chaine dans le fichier
Une deuxieme ligne.
```

On peut ensuite relire ce fichier :

```
>>> fichier = open("test.txt")
>>> a = fichier.read() # on lit l'integralite du fichier
>>> type(a)
<class 'str'>
>>> a
'Une chaine dans le fichier\nUne deuxieme ligne.'
>>> print(a)
Une chaine dans le fichier
Une deuxieme ligne.
>>> fichier.close()
```

On peut éventuellement lire le fichier par petits morceaux :

```
fichier = open("test.txt", "w")
fichier.write("Une grande chaine de caracteres afin de tester la lecture par
   petits morceaux")
fichier.close()
fichier=open("test.txt")
while True:
    qqcar = fichier.read(8)
    print(qqcar)
    if qqcar == "": break
fichier.close()
```

donne

```
Une gran
de chain
e de car
acteres
afin de
tester l
a lectur
e par pe
tits mor
ceaux
```

On peut remarquer que l'arrivée à la fin du fichier ne provoque pas d'erreurs mais seulement une chaine vide (on peut évidemment remplacer le test `if qqcar == "" :` par `if not qqcar :`).

Les *méthodes* importantes :

`read(nombre)`	lit les caractères du fichier, avec un nombre maximal si l'argument est donné. Chaine vide lorsqu'il n'y a plus rien. Lit l'intégralité si nombre n'est pas précisé
`readline()`	lit une ligne complète du fichier (retourne une chaine). Vide si à la fin
`readlines()`	lit les lignes du fichier dans une liste (de chaînes)
`write(chaine)`	écrit la chaîne dans le fichier
`writelines(lignes)`	écrit les lignes de la liste transmise (pas de retour à la ligne ajouté à la fin de chaque ligne)
`close()`	ferme le flux
`tell()`	indique la position courante dans le fichier

Un fichier texte ouvert en lecture possède également les propriétés d'un itérateur : on parcourt alors les lignes du fichier :

```
fichier = open("test.txt")
for ligne in fichier:   # on parcourt les lignes
    traitement...
```

II Le module os

Si vous êtes courageux (ou pire), vous pouvez jeter un coup d'oeil à `http://docs.python.org/3.3/library/os.html` ainsi que `http://docs.python.org/3.3/library/filesys.html` pour tout ce qui peut concerner les fichiers. Ici je ne décrirai que quelques unes des fonctions du module `os`

`getcwd()`	répertoire actuel (current working directory)
`chdir(rep)`	change le répertoire courant
`mkdir(rep)`	crée un répertoire
`listdir(chemin)`	retourne une liste des fichiers dans le répertoire précisé (répertoire courant par défaut)
`remove(chemin)`	efface le fichier donné
`rename(source, dest)`	renomme un fichier ou dossier

III Fichiers csv

Les fichiers csv (Comma-separated values) sont des fichiers texte que l'on peut obtenir en export par un tableur. Chaque ligne du fichier correspond à une ligne du tableau, les champs sont séparés par un élément fixe (virgule, point-virgule...). Chaque champ est éventuellement protégé par des guillemets (afin par exemple de mettre une virgule ou un point-virgule). Ce type de fichier permet de travailler facilement sur une feuille de calcul d'un tableur. Ils sont également utilisés (à la fois en import et en export) par certaines bases de données (par exemple MySQL

permet de sauver le contenu d'une table en csv ou d'importer un fichier csv dans une table). En Python, le module `csv` permet de lire et écrire de tels fichiers.

III.1 Commandes de base

`reader(csvfile, dialect='excel', **param)`	ouvre un flux (itérateur) vers un fichier csv (ouvert avec `open`) avec les paramètres donnés (description après)
`writer(csvfile, dialect='excel', **param)`	ouvre un flux en écriture

III.2 Paramètres de format

Dans les options d'ouverture ou de fermeture (initialisées suivant *dialect* qui par défaut est *excel*), on peut spécifier différentes options (`http://docs.python.org/3.3/library/csv.html#dialects-and-formatting-parameters`). En voici quelques-unes :

`delimiter`	par défaut `,`
`lineterminator`	par défaut `"\r\n"` (pour writer, le reader d'adapte)
`quotechar`	délimitateur de texte (défaut `'"'`)
`quoting`	prend l'une des valeurs QUOTE_ALL, QUOTE_MINIMAL, QUOTE_NONNUMERIC, QUOTE_NONE (essentiellement pour le writer)

III.3 Reader

Une fois le flux ouvert, on obtient un itérateur qu'on parcourt de façon classique. Admettons que nous ayons un fichier fichier.csv sous cette forme :

```
"Carottes";2.35;5
"Poireaux";3.20;3
"Courgettes";1.60;4
"Tomates";2.95;4
```

On peut l'ouvrir et le lire avec le programme suivant

```
import csv

file = open("fichier.csv")
csvfile = csv.reader(file,delimiter=";")
for i in csvfile:
    print(csvfile.line_num, type(i), i, sep=" -- ")
file.close()
```

qui retourne

```
1 -- <class 'list'> -- ['Carottes', '2.35', '5']
2 -- <class 'list'> -- ['Poireaux', '3.20', '3']
3 -- <class 'list'> -- ['Courgettes', '1.60', '4']
4 -- <class 'list'> -- ['Tomates', '2.95', '4']
```

Chaque ligne est donc obtenue sous forme d'une liste. On peut également charger un fichier ligne par ligne, sous forme de dictionnaire, la liste des étiquettes étant la première ligne du fichier ou le contenu de l'option `fieldnames`, à l'aide de l'instruction `DictReader` (`http://docs.python.org/3.3/library/csv.html#csv.DictReader`)

Si cette fois le fichier contient une première ligne avec les noms de colonnes :

```
"Nom";"Prix";"Quantite"
"Carottes";2.35;5
"Poireaux";3.20;3
"Courgettes";1.60;4
"Tomates";2.95;4
```

La lecture se fait comme avant

```
import csv

file = open("fichier2.csv")
csvfile = csv.DictReader(file, delimiter=";", quoting=csv.QUOTE_NONNUMERIC)
# les champs numeriques sont convertis en mumerique et pas en chaine
print(csvfile.fieldnames)
for i in csvfile:
    print(csvfile.line_num, type(i), i, sep=" -- ")
file.close()
```

et cela donne

```
['Nom', 'Prix', 'Quantite'] # fieldnames
2 -- <class 'dict'> -- {'Nom': 'Carottes', 'Quantite': 5.0, 'Prix': 2.35}
3 -- <class 'dict'> -- {'Nom': 'Poireaux', 'Quantite': 3.0, 'Prix': 3.2}
4 -- <class 'dict'> -- {'Nom': 'Courgettes', 'Quantite': 4.0, 'Prix': 1.6}
5 -- <class 'dict'> -- {'Nom': 'Tomates', 'Quantite': 4.0, 'Prix': 2.95}
```

III.4 Writer

Le principe est le même. On dispose notamment des méthodes `writerow(sequence)`, `writerows(sequence_list)`, `writeheader()` et `DictWriter(...)`.

Voir : `http://docs.python.org/3.3/library/csv.html#csv.DictWriter`

IV Exercices

Exercice IV.1

Choisir un fichier texte quelque part, et compter le nombre d'occurence de chaque lettre dans ce fichier. Retournez alors la liste des lettres avec leur nombre d'occurrences, tout cela trié en ordre alphabétique, puis en ordre d'apparition.

Exercice IV.2

Écrire un programme qui lit le fichier Notes.csv et

- *calcul la moyenne pour chaque devoir, chaque élève,*
- *crée un classement,*
- *affiche un histogramme de répartition pous chaque DS et pour les notes finales (on comptera les notes dans chaque intervalle $[n, n+1[$ ou $[2p, 2p+2[$). Puisqu'on n'a toujours pas vu comment faire un graphique, on pourra afficher les résultats en texte avec des colonnes de `*`*

Chapitre 11

Les tris

Nous avons déjà vu la méthode `sort` de la classe `list`. Nous allons voir ici d'autres moyens d'effectuer des tris d'une itérable, en fonctions de divers paramètres.

I Version simple

Pour une liste, on peut utiliser la méthode `sort`. Cela modifie la liste en triant ses éléments (avec l'utilisation de la fonction qui compare les éléments de la liste si ils sont comparables) :

```
>>> liste = [3,8,5,19,45,34,2]
>>> id(liste)
20492944
>>> liste.sort()
>>> liste
[2, 3, 5, 8, 19, 34, 45]
>>> id(liste)
20492944
```

On obtient bien le même objet, avec ses éléments triés.

On peut, à la place créer une nouvelle liste triée avec la fonction `sorted`

```
>>> liste = [3,8,5,19,45,34,2]
>>> id(liste)
20493160
>>> l2 = sorted(liste)
>>> id(l2)
20434168
>>> l2
[2, 3, 5, 8, 19, 34, 45]  # on a une nouvelle liste triee
>>> liste
[3, 8, 5, 19, 45, 34, 2]  # cette fois la liste n'est pas modifiee
```

On peut également trier un dictionnaire. Il n'y a bien entendu pas de méthode de tri pour un dictionnaire (c'est non ordonné), mais la fonction `sorted` peut s'utiliser. Le tri est alors réalisé sur les clés.

```
>>> personnes = {"Pierre":['H', 45], "Paule":['F', 53], "Jacques":['H', 57], "
    Marie":['F', 43]}
>>> sorted(personnes)
['Jacques', 'Marie', 'Paule', 'Pierre']  # on recupere les cles triees
```

II Modification de la fonction de tri

On peut spécifier une fonction qui va servir au tri (le tri est effectué sur les valeurs renvoyées par cette fonction). Par exemple pour trier une liste de tuple suivant l'un des éléments :

```
>>> liste_p=[("Pierre", 'H', 45), ("Paule", 'F', 53), ("Jacques", 'H', 57), ("
    Marie", 'F', 43)]
```

```
>>> def f(x) : return(x[2])
>>> sorted(liste_p,key = f)
[('Marie', 'F', 43), ('Pierre', 'H', 45), ('Paule', 'F', 53), ('Jacques', 'H',
    57)]
```

Dans cet exemple, le tri est effectué suivant la valeur de f pour chaque élément. Puisque cette fonction f n'a aucun intérêt en dehors du tri, il n'est pas nécessaire de la définir. On peut à cet endroit utiliser une lambda-fonction

```
>>> liste_p=[("Pierre", 'H', 45), ("Paule", 'F', 53), ("Jacques", 'H', 57), ("
    Marie", 'F', 43)]
>>> sorted(liste_p, key = lambda k : k[2])
[('Marie', 'F', 43), ('Pierre', 'H', 45), ('Paule', 'F', 53), ('Jacques', 'H',
    57)]
```

On peut également ajouter un troisième paramètre `reverse=True` pour trier dans l'ordre décroissant. Ces paramètres (`key` et `reverse`) s'utilise aussi sur la méthode `sort` pour une liste :

```
>>> liste_p.sort(key = lambda k : k[2], reverse=True)
>>> liste_p
[('Jacques', 'H', 57), ('Paule', 'F', 53), ('Pierre', 'H', 45), ('Marie', 'F',
    43)] # liste modifiee
```

III Avec le module operator

III.1 Sélection d'un élément

Le module operator (`http://docs.python.org/3.3/library/operator.html`) permet d'utiliser certaines fonctions usuelles sous forme fonctionnelle (par exemple une fonction `add` au lieu de l'operateur `+`). Cela permet notamment de récupérer certains éléments d'une liste ou d'un dictionnaire et ainsi de faire le tri avec ces éléments. On utilise notamment la fonction `itemgetter` qui permet de définir des fonctions d'extractions :

```
>>> from operator import itemgetter
>>> f = itemgetter(3,1,7) # definit une fonction qui extrait les positions 3, 1
    et 7
>>> liste = list(range(3,19,2))
>>> liste
[3, 5, 7, 9, 11, 13, 15, 17]
>>> f(liste)
(9, 5, 17)
```

On a la même chose avec un dictionnaire :

```
>>> from operator import itemgetter
>>> dico = {'prenom':'david', 'lycee':'Fermat','ville':'Toulouse'}
>>> g = itemgetter('ville','lycee')
>>> g(dico)
('Toulouse', 'Fermat')
>>> itemgetter('prenom')(dico)
'david'
```

On n'a plus qu'à combiner cela avec la fonction `key` pour trier sur un champ quelconque si on a une liste d'objets de type séquence

```
>>> from operator import itemgetter
>>> liste_p = [("Pierre", 'H', 45), ("Paule", 'F', 53), ("Jacques", 'H', 57), (
    "Marie", 'F', 43)]
>>> f = itemgetter(2)
>>> g = itemgetter(1, 2)
>>> sorted(liste_p, key=f) # tri sur le troisieme element
[('Marie', 'F', 43), ('Pierre', 'H', 45), ('Paule', 'F', 53), ('Jacques', 'H',
    57)]
>>> sorted(liste_p, key=g) # tri sur le second puis troisieme element
```

```
[('Marie', 'F', 43), ('Paule', 'F', 53), ('Pierre', 'H', 45), ('Jacques', 'H',
    57)]
```

III.2 Sélection d'un attribut

On dispose également d'une fonction `attrgetter` qui permet de choisir parmi les attributs d'un objet :

```
>>> from operator import attrgetter
>>> liste = [2+3j, 4-4j, 5+2j]
>>> sorted(liste, key=attrgetter('imag'))
[(4-4j), (5+2j), (2+3j)]
```

La liste est triée en comparant l'attribut `imag` de chaque élément (si `z` est de type complexe, on regarde `z.imag`)

Chapitre 12

Quelques références

- le polycopier du professeur Rupprecht du Lycée Pierre de Fermat à Toulouse : ? ? ?
- le livre de G. Swinnen : Très bon livre pour apprendre (on s'en est fortement inspiré), téléchargeable gratuitement et existe à l'achat en version papier.
  `http://inforef.be/swi/python.htm`
- le site du zéro : un apprentissage pas à pas, assez bien fait.
  `http://www.siteduzero.com/informatique/tutoriels/apprenez-a-programmer-en-python`
- le « Python Tutorial » officiel : `http://docs.python.org/3.3/tutorial/`
- un début de livre sur Wikibooks :
  `http://fr.wikibooks.org/wiki/Programmation_Python`
  plus complet en anglais : `http://en.wikibooks.org/wiki/Python_Programming`
- le livre de Xavier Dupré de l'ENSAE téléchargeable gratuitement et existe à l'achat en version papier. La version téléchargeable est beaucoup plus complète, 900 pages ! et contient beaucoup d'exercices (attention c'est fait pour `Python` 2.6).
  `http://www.xavierdupre.fr/mywiki/InitiationPython`

Et sur le Web :

- Documentation officielle de la bibliothèque standard `Python` :
  `http://docs.python.org/3.3/library/index.html`
- Python Module of the Week, une bonne présentation des principales bibliothèques de `Python` :
  `http://pymotw.com/2/`
- Index des bibliothèques tierces : `Python` : `https://pypi.python.org/pypi`
- Librairies scientifiques :

  NumPy : utilisation et traitement rapide de tableaux à plusieurs dimensions, intégration numérique, optimisation, transformée de Fourier, algèbre linéaire numérique...
  `http://www.numpy.org/`

  SciPy : librairie de calcul scientifique pour Python, dépend de Numpy, syntaxe proche de Matlab ou Scilab.
  `http://www.scipy.org/`

  Matplotlib : pour faire plein de jolis dessins.
  `http://matplotlib.org/`

  SymPy : calcul formel en Python.
  `http://sympy.org/fr/index.html`

Bibliographie

[1] Tentative numpy tutorial, document web, 2013. http://www.scipy.org/Tentative_NumPy_Tutorial.

[2] Eli Bressert. *SciPy and NumPy*. O'Reilly Media, 2e edition, 2012. http://www.it-ebooks.info/book/1280/.

[3] NumPy community. Numpy user guide, release 1.8.0.dev-fd6f038, 2013. http://docs.scipy.org/doc/numpy-dev/numpy-user.pdf.

[4] Valentin Haenel, Emmanuelle Gouillart, and Gaël Varoquaux editors. Python scientific lecture notes, release 2013.1. http://scipy-lectures.github.com, 2013.

[5] Gérard Swinnen. *Apprendre à programmer avec Python 3*. Eyrolles, 2e edition, 2010.