

Introduction à la programmation Python

par Jean-Daniel Bonjour, puis Samuel Bancal, EPFL-ENAC-IT, © Creative Commons BY-SA
(version préliminaire, rev. 2020-08-28)

- 1 Introduction
 - 1.1 Avant-propos
 - 1.2 Historique et caractéristiques du langage Python
 - 1.3 Quelques références
- 2 Bases de la programmation en Python
 - 2.1 Généralités
 - 2.1.1 En-têtes d'un script
 - 2.1.2 Instructions sur plusieurs lignes, commentaires
 - 2.1.3 Variables, mots-clés réservés
 - 2.1.4 Typage
 - 2.1.5 Types simples
 - 2.1.6 Assignment simple et multiple
 - 2.1.7 Interaction avec l'utilisateur (écran et clavier)
 - 2.1.8 Opérateurs de base
 - 2.1.9 Opérateurs de comparaison et logiques
 - 2.2 Types containers
 - 2.2.1 Chaînes de caractères
 - 2.2.2 Listes
 - 2.2.3 Tuples
 - 2.2.4 Dictionnaires
 - 2.2.5 Sets
 - 2.2.6 Frozensets
 - 2.2.7 Récapitulation des différents types de base
 - 2.2.8 Le slicing
 - 2.2.9 Les fonctions range et enumerate
 - 2.3 Structures de contrôle
 - 2.3.1 Indentation des blocs de code
 - 2.3.2 Exécution conditionnelle if - elif - else
 - 2.3.3 Boucle for
 - 2.3.4 Boucle while
 - 2.3.5 Instructions continue et break
 - 2.4 Fonctions, modules, packages, scripts
 - 2.4.1 Fonctions
 - 2.4.2 Modules
 - 2.4.3 Packages
 - 2.4.4 Scripts

- 2.5 Opérations sur les chaînes
 - 2.5.1 Formatage
 - 2.5.2 Fonctions et méthodes de chaînes
 - 2.5.3 Expressions régulières
- 2.6 Manipulation de fichiers
 - 2.6.1 Ouverture et fermeture
 - 2.6.2 Lecture
 - 2.6.3 Écriture ou ajout
 - 2.6.4 Autres méthodes sur les fichiers
 - 2.6.5 Sérialisation et stockage d'objets avec le module pickle
- 2.7 Quelques fonctions built-in
- 2.8 Quelques modules de la librairie standard
 - 2.8.1 os
 - 2.8.2 glob
 - 2.8.3 shutil
 - 2.8.4 filecmp
 - 2.8.5 sys, platform
 - 2.8.6 math, random
 - 2.8.7 time
 - 2.8.8 datetime
 - 2.8.9 calendar
 - 2.8.10 this
- Programmation objet avec Python
- Scientific Python
- Installation et utilisation de Python et outils associés

1. Introduction

1.1 Avant-propos

Ce support de cours a pour objectif de vous **introduire à la programmation Python** en se basant essentiellement sur des **exemples** et partant du principe que vous connaissez déjà d'autre(s) langage(s) de programmation. Nous ferons parfois quelques parallèles avec [MATLAB](#) / [GNU Octave](#), langages généralement bien connus des ingénieurs.



⚠ Nous avons résolument opté, dans ce support de cours, pour la **version 3** de Python. Lorsque des différences importantes existent entre Python v2 et v3, nous les signalons

avec ce symbole **P2**. Avec la version 3 de Python, apparue en 2008, la fondation Python a en effet décidé de gommer certaines imperfections de jeunesse du langage. La compatibilité arrière avec les versions ≤ 2 n'est donc pas garantie, un programme écrit en Python v2 ne tournant généralement pas sans adaptations sous un interpréteur Python v3. Cette version 3 étant cependant disponible depuis suffisamment longtemps, la plupart des modules, packages, frameworks... ont été adaptés, et nous estimons il est temps de passer définitivement à Python v3.

Nous faisons usage, dans ce support de cours, des **conventions de notation** suivantes :

- le code Python est en police de caractère à **chasse fixe** (en utilisant la coloration syntaxique offerte par Pandoc), de même que les touches de clavier ou combinaison de touches entre caractères `< >` (exemple: `<ctrl-D>`)
- l' *italique* indique que vous devez substituer vous-même l'information décrite et non pas saisir littéralement le texte indiqué (exemple: `input(prompt)`)
- les hyper-liens sont colorés (par exemple la table des matières ci-dessus)
- la “syntaxe à point” (mots séparés par un point) fait référence au modèle “orienté objet” de Python (*objet.méthode*, *module.fonction*, etc...) et implique généralement de charger préalablement un *module* (exemple: `sys.exit()` invoque la fonction `exit()` du module `sys` , module qu'il faut donc préalablement charger avec `import sys`)

Sous licence [Creative Commons BY-SA](#), ce support de cours est **accessible en ligne** à l'adresse <https://enacit.epfl.ch/cours/python/introduction/>. Une version PDF formatée pour une belle impression A4 est également disponible sous la forme des deux fichiers téléchargeables suivants : [introduction-python.pdf](#) et [outils-python.pdf](#). L'auteur reçoit volontiers toutes vos remarques (corrections, suggestions de compléments...).

Pour information, ce support de cours a été édité dans le langage de balisage léger *Markdown*, puis traduit en HTML avec le convertisseur *Pandoc*. Si ces techniques vous intéressent, voyez le [manuel Markdown/Pandoc](#) que nous avons réalisé.

1.2 Historique et caractéristiques du langage Python

Quelques dates dans l'histoire de Python (pour davantage de détails, frappez `license()`) :

- 1991 : release la première version publique de Python par [Guido van Rossum](#)
- 1996 : sortie de la librairie NumPy
- 2001 : création de la Python Software Foundation (PSF) qui prend en charge le développement du langage
- 2008 : sortie de Python 2.6 et 3.0
- 2020 : on en est à la version 3.8

Principales caractéristiques du langage Python :

- langage de programmation à usage général de haut niveau : orienté objet, à typage dynamique fort avec types de données évolués, multi-threading, gestion de la mémoire (*garbage collector*), interfaçable avec d'autres langages
- riche et polyvalent : librairie standard Python (*standard library*, i.e. modules intégrés à la distribution de base) couvrant la plupart des domaines (philosophie “*batteries included*”), très nombreux *packages* et *modules* d'extension (calcul scientifiques, visualisation, GUI, SGBD, réseau...) ainsi que *frameworks* (p.ex. Django pour le web, PyGame pour les jeux...)
- interprété, mais néanmoins rapide à l'exécution : utilisable en mode interactif (avec fonctionnalités d'introspection et debugging), permettant non seulement l'élaboration de scripts mais aussi de programmes complexes
- ouvert et multiplateforme/portable : libre et open source (licence PSF *GPL-compatible* sans restriction *copy-left* : voir `license()`) donc librement utilisable et distribuable, disponible sur tous les OS (intégré d'office sous certains, comme Linux et macOS), applications tournant sur toutes les plateformes (le langage étant interprété)

- facilité d'apprentissage et de mise en oeuvre : syntaxe simple/légère, consistance, grande lisibilité, flexibilité et haute efficacité/productivité dans tout le cycle de développement: analyse/design, prototypage, codage, test/debugging, déploiement, maintenance
- diversité des outils de développement : interpréteurs interactifs avec introspection (IPython...), IDEs (Eclipse, Spyder...)
- stable/mature, très répandu et évolutif : très vaste communauté de développeurs et utilisateurs, langage utilisé comme *moteur de script* dans de nombreuses applications métiers (ex: GIMP, Blender, QGIS...).

Ces différentes qualités expliquent pourquoi de plus en plus d'écoles et universités optent, depuis quelques années, en faveur de Python comme premier langage dans l'apprentissage de la programmation. En outre, avec Jupyter Notebook et les bibliothèques NumPy et SciPy notamment, nombre de chercheurs ont aujourd'hui trouvé en Python une véritable alternative à MATLAB, en particulier dans les domaines du calcul numérique, de l'analyse et de la visualisation de données.

1.3 Quelques références

La littérature sur Python est extrêmement abondante. Nous ne retenons ici que quelques références significatives :

- [P3DOC] : [Python v3 Documentation](#) : documentation officielle Python v3
- [P3TUT] : [The Python Tutorial](#) : tutoriel officiel Python v3
- [SWIM] : [Apprendre à programmer avec Python 3](#) : ouvrages de Gérard Swinnen, disponibles en ligne et sous forme papier (éditions Eyrolles)
- [PSLN] : [Python Scientific Lecture Notes](#) : introduction à l'écosystème Scientific Python (Scipy)
- [POINT] : [Python 3 Cheat Sheet](#) | [Python 3.2 Reference Card](#) | [Python 2.4 Quick Reference Card](#) : cartes de références Python, par Laurent Pointal

2. Bases de la programmation en Python

Pour des conseils concernant l'**installation de Python** sur votre machine, voyez notre chapitre "[Installation d'un environnement Python v3 complet](#)".

2.1 Généralités

2.1.1 En-têtes d'un script

Nous verrons plus loin ce que sont précisément les *scripts* Python. Pour simplifier et comprendre les exemples qui suivent, disons à ce stade qu'il s'agit de fichiers au format texte dont le nom se termine par l'extension `.py`, contenant des instructions Python et débutant en général par 2 lignes d'en-têtes.

Interpréteur du script

La première ligne d'en-tête du script (appelée *she-bang* en raison des 2 premiers caractères `#` et `!`) spécifie l'*interpréteur* qui doit être utilisé pour l'exécuter. S'agissant de Python, vous verrez parfois : `#!/usr/bin/python`, mais cette forme est moins portable que : `#!/usr/bin/env python` (exemple ci-contre) qui fonctionne indépendamment de l'emplacement (*path*) d'installation de l'interpréteur Python.

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 print("Hééé, bien le bonjour !")
```

Sous GNU/Linux Ubuntu, pour explicitement utiliser l'interpréteur Python v3 provenant des dépôts officiels Ubuntu, il faut utiliser le *she-bang* `#!/usr/bin/env python3`.

Cette première ligne est nécessaire si l'on veut pouvoir lancer le script depuis un shell en frappant : `./nom_du_script.py`. Elle n'est cependant pas nécessaire (mais tout de même recommandée) si on le lance avec `python nom_du_script.py` ou `python3 nom_du_script.py`.

Encodage des caractères

Pour faire usage de caractères accentués dans un script Python (même si l'on n'utilise ceux-ci que dans des commentaires !), il peut être nécessaire de définir, mais pas plus loin que la seconde ligne du fichier, un pseudo-commentaire `coding` (voir l'exemple ci-dessus).

Depuis Python v3 (qui a systématisé Unicode pour la gestion des chaînes de caractères) :

- si le fichier du script est encodé UTF-8 (ce qui est recommandé), la directive `# -*- coding: utf-8 -*-` n'est pas nécessaire, mais c'est une bonne pratique de la mettre
- s'il est encodé ISO-latin/ISO-8859 (déconseillé), il est dans ce cas nécessaire de spécifier `# -*- coding: iso-8859-1 -*-` (ou `# -*- coding: latin-1 -*-`)

P2 Sous Python v2 (où l'encodage par défaut est ASCII) :

- si le fichier du script est encodé ISO-latin/ISO-8859, il est nécessaire de spécifier `# -*- coding: iso-8859-1 -*-` (ou `# -*- coding: latin-1 -*-`)
- s'il est encodé UTF-8, il est nécessaire de spécifier `# -*- coding: utf-8 -*-`

Explications techniques : [Unicode](#) est le jeu de caractères universel prenant en compte les écritures de tous les pays du monde. [UTF-8](#) est un encodage Unicode dans lequel un caractère est codé respectivement sur 1, 2 ou 4 octets selon qu'il fait partie de la plage ASCII sans accents ($\text{code} \leq 127$), de la plage d'extension ($128 \leq \text{code} \leq 255$) ou des plages universelles ($256 \leq \text{code}$).

2.1.2 Instructions sur plusieurs lignes, commentaires

Pour des raisons de lisibilité, une instruction Python ne devrait pas dépasser environ 80 caractères. On peut si nécessaire la répartir sur plusieurs lignes en utilisant, en fin de lignes, le caractère de continuation de ligne `\` (back-slash).

Cependant les parties de code délimitées par `()`, `[]`, `{ }` (*tuples*, *listes*, *dictionnaires*...), triple apostrophe `'''` ou triple guillemets `"""` (*chaînes* multi-lignes) peuvent s'étendre sur plusieurs lignes sans faire usage du caractère `\`.

```
1 print("Affichage d'une longue ligne \
2 de texte. Blabla blabla blabla \
3 blabla blabla blabla blabla blabla.")
4
5 liste = ['un', 'deux', 'trois',
6         'quatre', 'cinq', 'six']
7
8 chaine = """Une chaîne de caractères
9 multi-lignes""" # intégrant le caractère de
10                # retour à la ligne \n
```

Pour insérer des commentaires dans du code Python :

- ce qui débute par le caractère `#` (pour autant qu'il ne fasse pas partie d'une chaîne de caractères) jusqu'à la fin de la ligne est considéré comme un commentaire
- les commentaires multi-lignes sont définis à la façon des chaînes de caractères multi-lignes, c-à-d. précédé et suivi de 3 apostrophes `'''` ou 3 guillemets

```
"""
```

```
21 # Commentaire d'une ligne
22
23 print('x =', x) # Commentaire après instruction
24
25 '''Un commentaire chevauchant
26 plusieurs lignes...'''
27
28 """et un autre commentaire
29 multi-ligne"""
```

2.1.3 Variables, mots-clés réservés

⚠ Les noms de variables (ou de tout autre objet Python tel que *fonction*, *classe*, *module*...) débutent par une lettre majuscule ou minuscule (**A-Z** ou **a-z**) ou `_` (souligné) suivie de 0, 1 ou plusieurs lettres, chiffres ou soulignés. Ils sont sensibles aux majuscules/minuscules (**VAR1** et **var1** désignant ainsi deux variables distinctes). Depuis Python v3, on pourrait utiliser tout caractère Unicode, mais il est préférable de s'en tenir aux caractères ASCII (donc sans caractères accentués).

Les mots-clés ci-contre (que vous pouvez afficher dans un interpréteur Python avec la commande `help("keywords")`) sont réservés et ne peuvent donc pas être utilisés pour définir vos propres identifiants (variables, noms de fonction, classes...).

False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

Pour le reste, Python ne protège pas l'utilisateur contre l'écrasement de noms. Par exemple le fait de définir un objet `abs=12` masque la fonction *built-in* `abs()` (valeur absolue d'un nombre), fonction que l'on retrouvera en déréférençant l'objet avec `del abs`.

2.1.4 Typage

Python est un langage à *typage dynamique fort* :

- *dynamique* : car on n'a pas à déclarer explicitement les variables utilisées, et le type d'une variable peut changer au cours de son utilisation (selon le type de l'objet référencé par la variable)
- *fort* : car l'interpréteur peut signaler des erreurs de type (par exemple si l'on définit : `2 + "3 francs"`) par opposition à un langage *faiblement typé* comme PHP (qui retournerait `5` dans ce cas)

Pour vérifier le type d'une variable (ou plutôt le type de la donnée/objet référencé par la variable), on utilisera la fonction `type(objet)`. Sous IPython, on peut aussi utiliser la commande `%whos`.

2.1.5 Types simples

Python propose **4 types simples** de base (*built-in*). Ils sont dits *simples* (ou *sclaires*, *atomiques*) car ils permettent de stocker une seule donnée par variable, contrairement aux *types containers*. Il s'agit de :

- *bool* : booléen (état vrai ou faux)
- *int* : entier (en précision illimitée depuis Python v3 !)
- *float* : réel flottant (double précision 64 bits)
- *complex* : complexe flottant

On verra plus loin que ces 4 types sont *immuables* (contrairement par exemple aux *listes* et *dictionnaires*).

On dispose de fonctions d'arrondi et de conversion, notamment :

- troncature et conversion *float*→*int* : `int(flottant)`
- arrondi, et le cas échéant conversion *float*→*int* : `round(flottant, ndigits)` : arrondi à *ndigits* chiffres après la virgule (si *ndigits* positif), ou avant la virgule (si négatif) ; par défaut *ndigits*=0, c'est-à-dire conversion à l'entier supérieur (si *flottant* est positif) ou inférieur (si négatif)
- conversion *int*→*float* : `float(entier)`
- pour faire une conversion en entier base 10 d'un *nombre* exprimé sous forme de chaîne en n'importe quelle *base* (2= binaire, 8= octale, 16= hexadécimale, etc...), on peut utiliser `int('nombre', base)`

Pour récupérer la valeur absolue d'un *entier* ou *flottant*, on dispose de la fonction habituelle `abs(entier_ou_flottant)`.

```
1 # bool : booléen (logique)
2 vlog = True
3 type(vlog)          # => builtins.bool
4 not vlog             # => False
5 # notez le 1er car. majuscule de True et False !
6
7 # int : entier de précision illimitée !
8 a = 2 ; i = -12
9 v = 2**80            # => 1208925819614629174706176
10 # définition d'entiers en
11 # binaire, octal ou hexadéc.:
12 k = 0b111           # => 7
13 m = 0o77            # => 63
14 n = 0xff            # => 255
15 # conv. chaînes bin/octal/hexa
16 # en entier & vice-versa:
17 int('111',2)        # => 7
18 # et inverse: bin(7)  => '0b111'
19 int('77',8)         # => 63
20 # et inverse: oct(63)  => '0o77'
21 int('ff',16)        # => 255
22 # et inverse: hex(255) => '0xff'
23
24 # float : flottant 64 bits
25 b = -3.3 ; r = 3e10
26 abs(b)              # => 3.3
27 # arrondi et conversion
28 int(3.67)           # => 3 (int)
29 int(-3.67)          # => -4 (int)
30 round(3.67)         # => 4 (int), comme round(3.67, 0)
31 round(3.67,1)       # => 3.7 (float)
32 round(133,-1)        # => 130 (int)
33 round(133,-2)        # => 100 (int)
34
35 # complex : complexe flottant
36 cplx = 3.4 + 2.1j # ou: cplx = complex(3.4, 2.1)
37 cplx.real           # => 3.4
38 cplx.imag           # => 2.1
```

Vous aurez remarqué dans ces exemples que, contrairement à d'autres langages, il n'y a pas de `;` (point-virgule) à la fin des instructions. Cela ne serait nécessaire que si l'on souhaite définir plusieurs instructions sur la même ligne, ce qui est rarement pratiqué en Python et déconseillé pour des raisons de lisibilité.

Table 4-2. NumPy data types

Type	Type code	Description
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point; compatible with C float
float64	f8 or d	Standard double-precision floating point; compatible with C double and Python float object
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type; a value can be any Python object
string_	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'
unicode_	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')

EXERCICE 2

2.1.6 Assignment simple et multiple

On vient de voir plus haut que l'assignation de variables s'effectue classiquement avec `=`

Python permet de faire aussi de l'assignation multiple, c'est-à-dire :

- assigner la même valeur à plusieurs variables
- assigner un *tuple* à un autre *tuple* (il faut alors autant de variables à gauche qu'à droite)

```
1 x1 = x2 = x3 = 3 # => x1, x2, x3 prennent la val. 3
2
3 y1,y2,y3 = 4,5,6 # => y1=4, y2=5, y3=6
4 y1, y2 = y2, y1 # permutation => y1=5, y2=4
5
6 tpl = (10,'abc') # ici un tuple...
7 v1, v2 = tpl     # => v1=10, v2='abc'
```

On peut déréférencer une donnée (ou tout type d'objet) avec `del objet` ou `del (objet)`. Pour déréférencer un attribut, on utilise `del objet.attr` ou `delattr(objet,'attr')`.

Explications techniques : Python accède aux données en mode “référence”. Soit une variable à laquelle on a assigné une valeur. Si on lui assigne ensuite une autre valeur, la variable se “réfère” dès lors à cette nouvelle donnée. Qu'advient-il alors de l'ancienne donnée ? Pour autant que plus aucune variable ne s'y réfère, le “garbage collector” la supprimera alors automatiquement, libérant ainsi de la mémoire. Sachant cela, on comprend mieux le mécanisme de “typage dynamique” de Python : une variable n'a en elle-même pas de type, elle a celui de la donnée à laquelle elle se réfère couramment !

```
21 v1 = 123 # => création donnée 123 référencée par v1
22 type(v1) # 1'OBJET référencé de type builtins.int
23 id(v1)   # => 9361312 (adresse mémoire)
24
25 v2 = v1  # => nouv. variable pointant sur même DONNÉE
26 id(v2)   # => 9361312 => pointe bien sur MÊME donnée !
27
28 v1 = 'a' # => création donnée 'a', et chang. réf. v1
29 type(v1) # 1'OBJET référencé de type builtins.str
30 id(v1)   # => 9361332 => v1 pointe bien s'AUTRE donnée
31
32 del v2   # => la donnée 123 n'est plus référencée (ni
33          # par v1 ni v2) => le garbage collector
34          # pourra supprimer donnée de la mémoire
```

2.1.7 Interaction avec l'utilisateur (écran et clavier)

Affichage dans la console (sortie standard)

Pour afficher quelque-chose sur la console, on utilise la fonction `print` dont la syntaxe générale est :

```
print(arguments, sep=' ', end='\n', file=sys.stdout)
```

Seuls le(s) *argument(s)* sont nécessaires, les autres paramètres `sep`, `end` et `file` étant facultatifs. Par défaut `print` écrit les *argument(s)* sur la *sortie standard*, ajoutant un caractère *espace* entre chaque argument, et envoie finalement un *newline* (saut de ligne). Mais on peut librement modifier les valeurs de ces

paramètres.

Notez qu'on peut passer autant d'*argument(s)* que l'on veut (contrairement à la fonction `disp` de MATLAB/Octave qui est limitée à 1 argument).

```
1 print('Le produit de', a, 'et', b, 'est', a*b)
2 print('Total des %s : %.2f CHF' % ('Dépenses',123))
3
4 # Ci-dessous, résultats équivalents :
5 print('Hello world !')
6 print('Hello ',end=''); print('world !')
7
8
9 # Sous Python v2 on aurait écrit :
10 print 'Hello', # la virgule élimine le saut de ligne
11 print 'world !'
```

❗ Remarque : on a vu que le nommage des fonctions répond aux mêmes règles que les variables. Les noms de fonctions sont donc également sensibles aux majuscules/minuscules (`PRINT` ou `Print` retournant ainsi une erreur de type “... is not defined”).

P2 Sous Python ≤ 2 , `print` était une commande et non pas une fonction. Les arguments étaient donc passé à la suite de la commande sans parenthèses. Pour supprimer le saut de ligne, il fallait faire suivre le dernier argument d'une virgule.

Lecture au clavier (entrée standard)

À partir de Python v3, on utilise pour cela la fonction `input(prompt)` qui affiche le *prompt* puis retourne ce que l'utilisateur a frappé sous forme de chaîne (dans tous les cas !).

Notez, dans les exemples ci-contre, l'usage des fonctions de conversion `int` en entier et `float` en réel, ainsi que de `eval` qui évalue une expression Python et retourne son résultat.

```
21 nom = input('Votre nom ? ')
22 # retourne une chaîne
23 poids = float(input('Votre poids ? '))
24 # génère une erreur si on saisit une chaîne
25 annee = int(input('Votre année de naissance ? '))
26 # génère erreur si saisie chaîne ou flottant
27
28
29 res = eval(input('Entrer expression Python : '))
30 # => évaluation de l'expression (comme input MATLAB)
```

P2 Sous Python ≤ 2 , on disposait des fonctions `raw_input` pour saisir une chaîne, et `input` pour saisir un nombre, une variable ou une expression. L'ancien comportement de `input` (correspondant au `input` sans second paramètre `'s'` de MATLAB/Octave) est obtenu sous Python v3 avec `eval(input(...))`.

2.1.8 Opérateurs de base

Les opérateurs mathématiques de base sont :

- opérateurs classiques : `+` `-` `*` `/`

- division entière tronquée : `//`
- puissance : `**`
- modulo (reste de la division entière) : `%`

On verra que `+` et `*` s'appliquent aussi aux *séquences* (*chaînes*, *tuples* et *listes*), et `-` aux *ensembles* (*sets* et *frozensets*).

La fonction `divmod(a, b)` retourne un tuple contenant (`a//b`, `a % b`)

```

1 2*3    # => 6 entier, ou: int(2.)*3
2 2.*3   # => 6.0 réel, ou: float(2)*3
3 13/5   # => 2.6 (aurait retourné 2 sous Python v2)
4 13//5  # => 2   (div. int. tronquée comme / en Python v2)
5 10**2  # => 100
6 13%5   # => 3   (reste de la division entière)
7 divmod(13,5) # => (2, 3)
8
9 'oh' + 2*'là' # => 'ohlàlà'
10 (1,2) + 2*(4,) # => (1, 2, 4, 4)

```

Depuis Python v3, la division `/` de deux nombres entiers retourne toujours un flottant, et c'est donc à l'utilisateur de le convertir si nécessaire en entier avec `int` (troncature) ou `round` (arrondi).

P2 Sous Python ≤ 2 , la division `/` de deux nombres entiers retournait toujours un entier qui pouvait donc être tronqué, comme le fait la division `//` Python v3.

Les opérateurs de pré/post incrémentation/décrémentation `++` et `--` n'existant pas sous Python, on utilisera `+= 1` et `-= 1`

```

21 val = 10
22 val += 1    # val=val+1    => 11
23 val -= 1    # val=val-1    => 10
24 val *= 2.5  # val=val*2.5 => 25.0
25 val /= 10   # val=val/10   => 2.5
26 val %= 2    # val=val%2    => 0.5
27
28 fruit = 'pomme'
29 fruit += 's' # => 'pommes'
30 tpl = (1,2,3) # ici un tuple
31 tpl += (4,)   # => (1, 2, 3, 4)
32 tpl *= 2     # => (1, 2, 3, 4, 1, 2, 3, 4)

```

2.1.9 Opérateurs de comparaison et logiques

Python offre notamment les opérateurs suivants :

- opérateurs de comparaison : `==` (égal), `!=` (différent), `<` (inférieur), `<=` (inférieur ou égal), `>` (supérieur), `>=` (supérieur ou égal)
- comparaison d'objets : `is` (même objet), `in` (membre du *container*)
- opérateurs logiques (ci-après selon leur ordre de priorité) : `not` (négation logique), `and` (et logique), `or` (ou logique)

Ces opérateurs retournent les valeurs `True` ou `False` de type booléen. Notez qu'il

n'est pas nécessaire d'entourer les expressions logiques de parenthèses comme dans d'autres langages.

```
1 1 < 2 and 2 < 3 # => True
2 'oui' == 'OUI'.lower() # => True
3 False or not True and True # => False
4
5 [1,2,3] == [1,2,3] # => True
6 [1,3,2] == [1,2,3] # => False
7
8 a=[1,2] ; b=[1,2]; a is b # => False
9 a=[1,2] ; b=a; a is b # => True
10
11 2 in [1,2,3] # => True
12 'cd' in 'abcdef' # => True
```

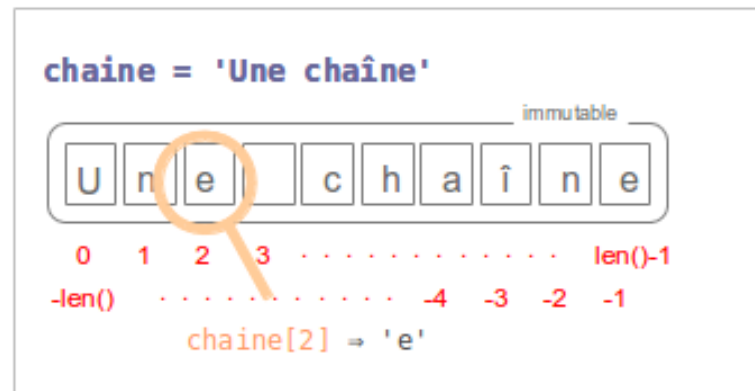
! Python permet de chaîner les comparaisons ! On peut par exemple définir la *condition* `10 < age < 20` qui est équivalente à `age>10 and age<20`.

2.2 Types containers

Les types *containers* permettent d'instancier des objets contenant plusieurs données, contrairement aux *types simples* (booléen, entier, flottant, complexe). On distingue fondamentalement **3 catégories de containers**, implémentés sous forme de **6 types de base** (*built-in*) Python :

- les *séquences* comprenant les types : *chaîne*, *liste* et *tuple*
- les *ensembles* comprenant les types : *set* et *frozenset*
- les *maps* (aussi appelés *hashs*) avec le type : *dictionnaire*

2.2.1 Chaînes de caractères





On définit une *chaîne* de caractères en la délimitant par des apostrophes `'` ou

guillemets `"`, caractères que l'on triple pour une chaîne littérale multi-ligne.

Le type *chaîne*, sous Python, est **immutable**. Les chaînes ne sont donc à proprement parler pas modifiables : les opérations de modification entraînent la création de nouvelles chaînes, et le *garbage collector* Python détruisant automatiquement les anciennes chaînes qui ne sont plus référencées.

Une chaîne étant une *séquence ordonnée* de caractères, on peut s'y référer par les indices de ces caractères. Les exemples ci-contre illustrent le fait que :

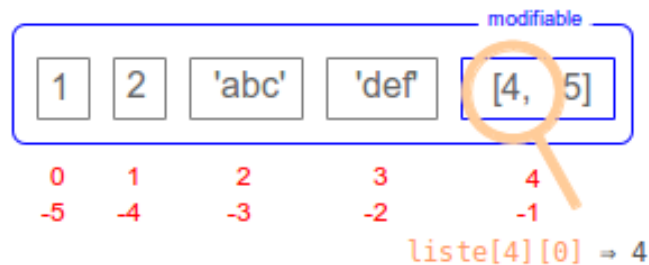
-  le premier caractère d'une chaîne a l'indice `0`, comme dans la plupart des langages (et non `1` comme sous MATLAB/Octave ou Fortran)
-  lorsque l'on se réfère à une partie de chaîne avec la notation `chaîne[debut:fin]` (technique du *slicing* décrite plus bas), cela désigne la portion de chaîne dont les caractères ont les indices : `debut ≤ indice ≤ fin-1` ; le nombre de caractères sélectionnés est ainsi égal à `fin - debut`

```
1  # Définition
2  metall = 'L'argent' # délimitation par apostrophes
3  metal2 = "l'or"     # ou par guillemets
4
5  # Concaténation
6  metal = metall + ' et ' + metal2 # => "L'argent et l'or"
7  type(metal) # => builtins.str
8  3.00 + '$'   # => erreur: faire: str(3.00)+'$' => "3.0$"
9  3.00 * '$'   # => erreur
10 3 * '$'      # pas d'erreur ! => "$$$"
11
12 # Adressage
13 len(metal)   # => 16
14 metal[:8]    # idem que metal[0:8] => "L'argent"
15 metal[12:]   # idem que metal[12:len(metal)] => "l'or"
16
17 # Modification
18 metal[9:0] = 'sale' # => erreur (chaînes immutables)
19 metal = metal[:9] + 'sale' => "L'argent sale"
20 # fonctionne, car la var. metal pointe sur une
21 # nouvelle donnée, et l'ancienne ("L'argent et l'or")
22 # sera supprimée par le garbage collector
23
24 # Chaînes multi-ligne (\n dans les 3 prem. expressions !)
25 str2 = '''chaîne
26 multi-ligne''' # => "chaîne\nmulti-ligne"
27
28 str2 = """chaîne
29 multi-ligne""" # => "chaîne\nmulti-ligne"
30
31 str2 = 'chaîne\nmulti-ligne' # => idem
32
33 str3 = 'chaîne \
34 multi-ligne' # => "chaîne mono-ligne" (sans \n)
```

Pour davantage d'information sur les traitements relatifs aux chaînes de caractères, voir le chapitre spécifique [plus bas](#).

2.2.2 Listes

```
liste = [1, 2, 'abc', 'def', [4,5]]
```



Une *liste* Python permet de stocker une collection **ordonnée** (*séquence*) et dynamiquement **modifiable** d'éléments **hétérogènes** (c-à-d. de n'importe quel type, y.c. listes ou dictionnaires imbriqués, fonctions, classes...). Ce type de donnée, qui correspondrait par exemple sous MATLAB/Octave au *tableau cellulaire*.

Syntaxiquement, on utilise les **crochets** `[elem,elem...]` pour définir le contenu d'une liste. Les virgules délimitant chaque élément sont obligatoires (contrairement à MATLAB/Octave).

Pour accéder au contenu d'une liste (adresser ses éléments), on indique également les indices entre **crochets** `[]` (et non parenthèses comme sous MATLAB/Octave). Comme pour les chaînes :

- les éléments d'une liste sont indicés par des entiers à partir de **0** (et non **1** comme sous MATLAB/Octave ou Fortran)
- lorsque l'on définit une séquence d'éléments avec `liste[debut:fin]` (technique du *slicing* décrite en détail plus bas), cela désigne l'ensemble contigu d'éléments d'indices : $debut \leq indice \leq fin-1$, ce qui correspond à un nombre d'éléments égal à $fin - debut$

Méthodes pour **ajouter** des éléments :

- un élément à la fois :
 - à la fin avec `.append(elem)`
 - n'importe où avec `.insert(indice,elem)`
- plusieurs éléments d'un coup :
 - à la fin avec `.extend([elem...])` ou `liste+= [elem...]`
 - au début avec `liste[:0]=[elem...]`

Pour **supprimer** des éléments :

```
1 # Définition
2 lst = [1,2,'abc','def'] # liste avec diff. types d'élém.
3 type(lst) # => type builtins.list (cf. %whos IPython)
4 type(lst[2]) # => type élément 'abc' => builtins.str
5
6 # Adressage et modification
7 lst[1:3] # => [2, 'abc']
8 lst[0,1,3] # => erreur (énumér. indices impossible)
9 lst[2:] = 3,4 # écrasement depuis pos. 2 => [1,2,3,4]
10 v1,v2,v3,v4 = lst # unpacking => v1=1, v2=2, v3=3, v4=4
11 # retournerait erreur si nombre de var != nb. élém.
12 # il faut dans ce cas procéder comme ci-dessous :
13 v1, *v, v4 = lst # => v1=1, v=[2, 'abc'], v4=4
14
15 # Ajout ou insertion d'éléments, concaténation de listes
16 lst[5] = 5 # => erreur "index out of range"
17 lst.append(5) # ajout en fin liste=> [1, 2, 3, 4, 5]
18 # ou faire: lst += [5]
19 lst.insert(0,'a') # insère élém. spécifié à pos. spécif.
20 # (ici 0) => ['a', 1, 2, 3, 4, 5]
21 # rem: append & insert n'insère qu'1 élém. à la fois !
22 lst.extend(['b','a']) # => ['a', 1, 2, 3, 4, 5, 'b', 'a']
23 # ou concat.: lst=lst+['b','a'] ou lst += ['b','a']
24 2 * [10, 20] # => [10,20,10,20] (et non [20,40] !)
25 [0]*10 # => [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
26
27 # Suppression d'éléments (selon contenu ou indice)
28 lst.remove('a') # 1er élém. 'a'=> [1,2,3,4,5,'b','a']
29 lst.remove('a') # occur. 'a' suiv. => [1,2,3,4,5,'b']
30 del(lst[4:]) # élém. pos. 4 -> fin => [1, 2, 3, 4]
31 # on aurait pu faire: lst[4:]=[]
32 lst.pop() # dern. élém. et le retourne => 4
33 lst.pop(1) # élém. pos. 1 et le retourne => 2
34
35 # Diverses méthodes et fonctions
36 lst = [4, 1, 3, 2]
37 len(lst) # nombre éléments de la liste => 4
38 min(lst) # le plus petit élément => 1
39 max(lst) # le plus grand élément => 4
40 # min & max utilisable que si collection mêmes types
41
42 2 in lst # test existence élément => True
43 lst.index(3) # position 1ère occurrence de 3 => 2
44 l2 = sorted(lst) # => [1, 2, 3, 4]
45 lst.sort() # modifierait direct. lst => [1,2,3,4]
46 lst.reverse() # modifierait direct. lst => [4,3,2,1]
47
```


- fonction `del(liste[...])`
- méthodes `.remove(elem)` et `.pop(indice)`

```

48 lst.insert(2, ['ab', 'cd']) # => [1, 2, ['ab', 'cd'], 3, 4]
49 lst[2]                    # => ['ab', 'cd'] : élém. est liste !
50 lst[2][1]                 # adressage dans liste => 'cd'
51 lst[2][1][0]              # => 'c'

```

Le type *liste* se prête bien à l'implémentation de *pires (stacks)* :

- pour une pile *FIFO (first in, first out)* :
 - insérer *elem* avec `liste.insert(0, elem)` ou `liste[:0]=[elem]`
 - récupérer *elem* avec `elem=liste.pop()`
- pour une pile *LIFO (last in, first out)* :
 - empiler *elem* avec `liste.append(elem)`
 - dépiler *elem* avec `elem=liste.pop()` (le sommet de la pile étant ici la fin de la liste)

⚠ Remarque : on serait tenté d'utiliser des **listes imbriquées** pour manipuler des *matrices*, par exemple `mat_2D = [[1,2,3], [4,5,6]]`, puis accès aux éléments avec `mat_2D[1][0] => 4`. Mais ce n'est pas du tout efficace ni flexible (p.ex. difficile d'extraire une colonne). On a pour cela bien meilleur temps d'utiliser la bibliothèque [NumPy](#).

Pour itérer les éléments d'une liste, voir la [boucle for](#) décrite plus bas.

Pour construire des listes par itération avec une syntaxe très compacte, voir la [compréhension de listes, sets et dictionnaires](#).

Copie de listes

⚠ Si l'on souhaite dupliquer les données d'une liste `l1=[...]`, il est très important de comprendre qu'avec Python une simple assignation `l2=l1` n'effectue pas de copie des données : les 2 variables `l1` et `l2` référenceront en effet la même liste !

⚠ Pour **réellement recopier les données** d'une liste, il faut donc prendre des précautions particulières :

- on pourrait faire `l2 = l1[:]` (énumération par la technique du [slicing](#)) comme dans l'exemple ci-contre, mais cela ne duplique que les éléments de 1er niveau (les éléments de listes ou dictionnaires éventuellement imbriqués continuant d'être référencés et donc non dupliqués)
- une copie complète récursive (agissant dans toute la profondeur de la liste) doit plutôt être réalisée avec `l2 = copy.deepcopy(l1)` (nécessitant l'importation du module `copy`)

```

61 l1 = [1,2,3]
62 l2 = l1      # l2 pointe vers l1, ce qu'on peut vérifier
63             # avec id(l1) et id(l2)
64 l2[1] = 'a'  # donc on modifie ici données originales !
65 l2          # => [1, 'a', 3]
66 l1          # => également [1, 'a', 3] !!!
67
68 l1 = [1,2,3]
69 l2 = l1[:]   # on crée ici par copie un nouvel objet !
70 l2[1] = 'a'  # donc on agit dans ce cas sur nouvel objet
71 l2          # => [1, 'a', 3]
72 l1          # => [1, 2, 3] : données orig. intouchées !

```

Cette problématique de copie ne concerne pas les types immutables tels que les [tuples](#) et [frozensets](#), car il n'y a par définition aucun risque qu'on modifie leurs données.

2.2.3 Tuples

Le *tuple* Python est l'implémentation en lecture seule de la liste, c'est-à-dire une collection ordonnée **non modifiables** d'éléments hétérogènes. On parle ainsi *liste immutable*. Ce type de donnée possède les mêmes *méthodes* que la *liste*, à l'exception de celles permettant une modification.

Lorsque l'on définit par énumération les éléments du tuple, ceux-ci doivent être délimités par des virgules. On peut emballer le tout entre **parenthèses**

(*elem, elem...*), mais ce n'est pas obligatoire.

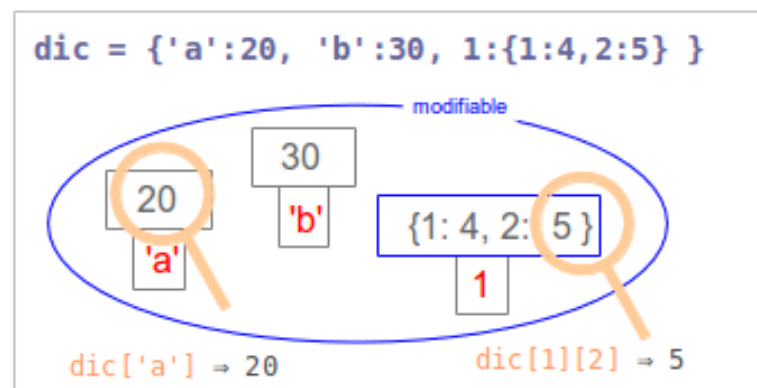
⚠ Lorsque le tuple ne contient qu'un seul élément, il est nécessaire de le faire suivre d'une virgule.

Pour accéder aux éléments du tuple, on spécifie les indices entre **crochets** [] (comme pour les *listes*).

```
1 # Définition
2 nb = (10) ; str = ('abc') # => resp. entier & chaîne !
3 tpl = (10,) ; tpl = 10, ; tpl=('abc',) # => tuples
4
5 tpl = (1, 'Linux', 2, 'Windows') # on peut omettre ( )
6 type(tpl) # => type builtins.tuple (cf. %whos IPython)
7 len(tpl) # => 4 (nombre d'éléments)
8 tpl[1::2] # => ('Linux', 'Windows')
9
10 tpl[3] = 'macOS' # => erreur (tuple non modifiable)
11 tpl += 3, 'macOS' # possible: crée nouv. objet tuple
12 'Linux' in tpl # => True
13
14 tpl2= tuple([10,11,12]) # copie liste->tuple=> (10,11,12)
15 tpl3= tuple('hello') # copie chaîne->tuple
16 # => ('h','e','l','l','o')
17
18 lst2= list(tpl2) # copie tuple->liste => [10,11,12]
```

Moins flexibles que les *listes* en raison de leur immutabilité, l'intérêt des *tuples* réside cependant dans le fait qu'ils occupent moins d'espace mémoire et que leur traitement est plus efficace.

2.2.4 Dictionnaires



Un *dictionnaire* est une liste **modifiable** d'éléments **hétérogènes** indicés par des **clés** (par opposition aux *listes* et *tuples* qui sont indicés par des séquences d'entiers). Le *dictionnaire* est donc **non ordonné**. Ce type de donnée Python correspond au *tableau associatif* (*hash*) sous Perl, et au *map* sous MATLAB/Octave.

Syntaxiquement, on utilise les **accolades** `{ }` pour définir les éléments du dictionnaire, c'est-à-dire les paires **clé: valeur** (notez bien le séparateur `:`). Les *clés* doivent être de *type simple immutable* (on utilise le plus souvent des chaînes ou des entiers) et **uniques** (i.e. plusieurs éléments ne peuvent pas partager la même clé). Les *valeurs*, quant à elles, peuvent être de n'importe quel type (y compris des *containers*).

```
1  # Définition
2  dic = {'a': 20, 'b': 30}
3  # ou élément après élément :
4  dic={} ; dic['a']=20 ; dic['b']=30
5  # ou en utilisant le constructeur dict :
6  dic = dict(a=20, b=30)
7  dic = dict( (('a', 20), ('b', 30)) )
8  dic = dict(zip(('a','b'), (20,30)))
9  type(dic)  # => type builtins.dict (cf. %whos s/IPython)
10 len(dic)   # => 2 paires clé:valeur
11
12 'a' in dic  # test existence clé => True
13 20 in dic   # => False, car 20 n'est pas une clé !
14
15 # Récupérer les valeurs
16 dic['a']    # => 20
17 dic['c']    # retourne erreur KeyError
18 dic.get('a') # => 20
19 dic.get('c','erreur blabla') # => 'erreur blabla'
20
21 # Ajouter un élément, modifier val., supprimer élém.
22 dic[1] = 10 # => {'a': 20, 'b': 30, 1: 10}
23 dic['b'] = 50 # on remplace la val. 30 par 50
24 del(dic['b']) # suppression élément 'b': 50
25 val=dic.pop('b') # récupère val. & suppr. élém.
26
27 # Ajouter plusieurs éléments, fusion de dict.
28 dic.update({2: 20, 3: 30})
29 # => dic = {'a': 20, 3: 30, 2: 20, 'b': 30}
30
31 # Objets itérables par boucle for
32 dic.keys()  # => objet dict_keys(...)
33 dic.values() # => objet dict_values(...)
34 dic.items() # => objet dict_items(...)
35 # ... et copie sur liste ou tuple
36 list(dic.keys()) # => liste ['a', 2, 3, 'b']
37 tuple(dic.values()) # => tuple (20, 20, 30, 30)
```

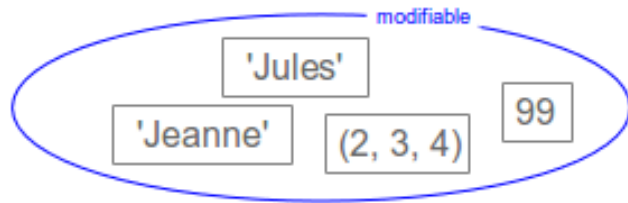
L'accès à une *valeur* du dictionnaire s'effectue en lui passant entre **crochets** `[]` la *clé* correspondante. Les méthodes `.keys()`, `.values()` et `.items()` retournent des objets permettant d'itérer, par une *boucle for*, respectivement les *clés*, les *valeurs* et les paires *clés/valeurs*.

Les dictionnaires n'étant pas ordonnés (non indicés par des entiers mais par des clés), la technique du *slicing* n'est bien entendu **pas** applicable.

❗ Si l'on veut **recopier les données** d'un *dictionnaire* `dic1`, le problème est le même qu'avec les *listes* : on ne fera pas `dic2=dic1` (les 2 variables se référant dans ce cas au même objet), mais une copie récursive (dans toute la profondeur du dictionnaire) avec `dic2 = copy.deepcopy(dic1)` (nécessitant l'importation du module `copy`).

2.2.5 Sets

```
st = {'Jules', 'Jeanne', 99, (2, 3, 4)}
```



Le *set* est un type Python permettant de créer des collections **non ordonnées** modifiables constituées d'éléments **uniques** de types **immuables**. Son grand intérêt réside dans le fait qu'on peut appliquer à ces objets des opérations propres aux ensembles (union, intersection, différence...), d'où le nom de *sets* (*ensembles*).

Notez bien que ce ne sont pas des *séquences* (les éléments ne sont pas ordonnés, donc non accessibles par des indices entiers), mais ils supportent l'itération (l'ordre n'étant cependant pas significatif). C'est un peu des *dictionnaires* (dont les clés sont aussi uniques) mais sans valeurs.

On crée un *set* avec la fonction `set(iterable)` ou en énumérant les éléments entre accolades `{elem, elem...}`. L'usage des accolades, comme pour les *dictionnaires*, est cohérent, ces deux types de données contenant des éléments uniques (les données elles-mêmes pour les *sets*, et les *clés* pour les *dictionnaires*).

A la place des opérateurs présentés dans les exemples ci-contre, on peut utiliser des méthodes : `.union(iterable)` pour `|`, `.intersection(iterable)` pour `&`, `.difference(iterable)` pour `-`, `.symmetric_difference(iterable)` pour `^`.

```
1 # Définition
2 set1 = {'a', 'b', 'a', 'c'}      # => {'a', 'c', 'b'}
3 type(set1)      # => type builtins.set (cf. %whos IPython)
4 len(set1)       # => 3
5
6 set2 = set( ('b','c','d','c') ) # => {'c', 'b', 'd'}
7     # seconde façon de définir un set, notez doubles ( ) !
8
9 # Ajout ou suppression d'éléments
10 set3 = set()      # set vide
11 set3 |= {1, 2}    # ajout plusieurs élém. => {1, 2}
12                 # identique à: set3 = set3 | {1,2}
13                 # ou à: set3.update({1,2})
14 set3.add(3)       # ajout 1 élém. seul. => {1, 2, 3}
15 set3.remove(2)    # détruit 1 élém. => {1, 3}
16 set3.discard(2)   # remove, mais pas err. si élém. absent
17 set3.pop()        # retourne élém. arbitraire et détruit
18 set3.clear()      # détruit tous les élém. => { }
19
20 # Opérations propres aux ensembles
21 set1 | set2      # union      => {'a', 'b', 'c', 'd'}
22 set1 & set2      # intersection => {'b', 'c'}
23 set1 - set2      # différence  => {'a'}
24 set2 - set1      # différence  => {'d'}
25 set1 ^ set2      # ou exclusif => {'a', 'd'}
26                 # élém. présents dans set1 ou set2 mais pas les deux
27
28 # Tests
29 {1,2} == {2, 1}   # sets identiques => True
30 1 in {1, 2, 3}    # présence élém. isolé => True
31 {1,2} <= {1,2,3,4} # présence de tous les élém. => True
```

⚠ La problématique de copie de sets est la même que pour les *listes* et *dictionnaires*. Pour réellement **copier les données** d'un set `s1`, on ne fera donc pas `s2=s1` (les 2 variables pointant dans ce cas sur le même objet), mais `s2=s1.copy()` ou `s2=set(s1)`.

2.2.6 Frozensets

Un *frozenset* est simplement un *set* **immutable**, c'est-à-dire non modifiable.

On crée un objet de type *frozenset* avec la fonction `frozenset(iterable)`.

```
1 fset = frozenset( {'x','y','z','x'} ) # ou...
2 fset = frozenset( ('x','y','z','x') ) # identique
3 type(fset)      # => type(builtins.frozenset)
```

Tout ce que l'on a vu concernant les *sets* est applicable, à l'exception de ce qui a trait à la modification. Les *frozensets* occupent moins d'espace en mémoire que les *sets*, et leur traitement est plus efficace.

2.2.7 Récapitulation des différents types de base

Les différents types de données de base (*built-in*) offerts par Python sont résumés dans le tableau ci-contre.

Types de base	modifiables	immutables
simples	<i>aucun</i>	<ul style="list-style-type: none">• booléen• entier• réel flottant• complexe
containers	<ul style="list-style-type: none">• liste (séquence)• set (ensemble)• dictionnaire (map)	<ul style="list-style-type: none">• tuple (séquence)• frozenset (ensemble)• chaîne (séquence)

2.2.8 Le slicing

Applicable aux *containers* indicés par des entiers, c'est à dire les *séquences* (*listes*, *tuples* et *chaînes* ; donc pas les *dictionnaires* ni les *sets* et *frozensets*), la technique du *slicing* est une forme avancée de l'indexation permettant d'accéder aux éléments par intervalles (*ranges*) et tranches (*slices*). Sa syntaxe générale est :

`sequ[debut:fin:pas]`, désignant dans la séquence `sequ` les éléments d'indices $debut \leq \text{indice} \leq fin-1$, avec une périodicité de `pas`.

On a déjà vu que les éléments d'une *séquence* sont numérotés positivement de gauche à droite de `0` à `len(sequ)-1` (et non de `1` à `len(sequ)` comme sous MATLAB/Octave ou Fortran).

⚠ Mais ce qu'il faut ajouter ici c'est qu'ils sont aussi numérotés négativement de droite à gauche de `-1`, `-2`, `-3` ... jusqu'à `-len(sequ)` ! Par conséquent, comme on le voit dans les exemples ci-contre, les paramètres `debut`, `fin` et `pas` peuvent aussi prendre des valeurs négatives. S'agissant du `pas`, une valeur positive signifie un déplacement vers les éléments suivants (à droite), et négative vers les éléments précédents (à gauche).

```

1 lst = [1,2,3,4,5,6,7]
2
3 lst[-1]      # => 7
4 lst[-3:]     # => [5, 6, 7]
5 lst[-1:-3:-1] # => [7, 6] (et non pas [7, 6, 5] !)
6 lst[-1:-3]   # => [] car on ne peut pas aller de -1
7             # à -3 avec le pas par défaut de 1
8 lst[2:-2]    # => [3, 4, 5]
9 lst[:2]      # => [1, 3, 5, 7]
10 lst[::-1]   # => [7, 6, 5, 4, 3, 2, 1]
11 lst[2:]     # => [1, 2]
12 lst[:-2]    # => [1, 2, 3, 4, 5]
13
14 lst[2:-2] = 'a' # remplacement d'éléments
15 lst          # => [1, 2, 'a', 6, 7]
16 lst[1:-1] = [] # suppression d'éléments
17 lst          # => [1, 7]
18
19 chaine = 'abcdef'
20 chaine[::-1] # du dern. au 1er car. (miroir)=> 'fedcba'
21 chaine[::-2] # idem en sautant 1 car. sur 2 => 'fdb'

```

Il faut encore préciser que chacun de ces 3 paramètres `debut`, `fin` et `pas` peut être omis :

- si l'on ne spécifie pas la valeur `debut` et que le `pas` est positif ou non spécifié, cela correspond à `0` (premier élém. de la séquence) ; mais si le `pas` est négatif, cela correspond alors à `len(sequ)+1` (dernier élém. de la séquence)
- si l'on ne spécifie pas la valeur `fin` et que le `pas` est positif ou non spécifié, cela correspond à `len(sequ)+1` (dernier élém. de la séquence) ; mais si le `pas` est négatif, cela correspond alors à `0` (premier élém. de la séquence)
- si l'on ne spécifie pas la valeur de `pas`, c'est la valeur `1` qui est utilisée par défaut (déplacement vers l'élém. suivant)

❗ Il faut finalement relever que, contrairement à MATLAB/Octave, Python ne permet pas d'adresser des éléments en énumérant leurs indices séparés par des virgules. L'expression `lst[0,3,4]` retourne donc une erreur.

2.2.9 Les fonctions `range` et `enumerate`

La fonction `range(debut, fin, pas)` crée un *objet itérable* (notamment utilisable par `boucle for`) correspondant à la suite de nombre entiers :

- `debut ≤ nombre ≤ fin-1` si le `pas` est positif,
- `debut ≥ nombre ≥ fin+1` si le `pas` est négatif.

S'ils ne sont pas fournis, les paramètres `debut` et `pas` prennent respectivement les valeurs par défaut `0` et `1`.

❗ Pour créer une *liste* d'entiers à partir de l'objet `range`, on utilisera la fonction `list(objet)`, et pour créer un *tuple*, la fonction `tuple(objet)`.

```
1 rg = range(5) # => range(0,5)
2 type(rg)      # => type builtins.range (%whos s/IPython)
3
4 list(rg)       # => [0, 1, 2, 3, 4]
5 tuple(rg)      # => (0, 1, 2, 3, 4)
6
7 list(range(-4,10,3)) # => [-4, -1, 2, 5, 8]
8 list(range(5,0,-1))  # => [5, 4, 3, 2, 1]
```

P2 Sous Python ≤ 2, la fonction `range` fabriquait directement une liste, et il existait une fonction `xrange` (qui a désormais disparu) qui se comportait comme la fonction `range` actuelle (création d'un itérateur).

La fonction `enumerate(sequence, debut)` crée un *objet itérable* retournant l'indice et la valeur des éléments d'une *séquence* (*liste*, *tuple* et *chaîne*) ou d'un *objet itérable*. Cette fonction est également très utilisée dans les *boucles for*.

Si l'on passe le paramètre `debut` (valeur entière), les indices retournés démarreront à cette valeur et non pas 0.

```
21 lst = list(range(3,7)) # => [3, 4, 5, 6]
22
23 en0 = enumerate(lst)
24 type(en0) # => type builtins.enumerate
25 list(en0) # => [(0, 3), (1, 4), (2, 5), (3, 6)]
26
27 en5 = enumerate(lst,5)
28 list(en5) # => [(5, 3), (6, 4), (7, 5), (8, 6)]
29
30 str = 'abc'
31 ens = enumerate(str)
32 list(ens) # => [(0, 'a'), (1, 'b'), (2, 'c')]
```

Sachez encore qu'il existe un module standard `itertools` permettant de créer différents types d'itérateurs.

2.3 Structures de contrôle

2.3.1 Indentation des blocs de code

❗ En Python, l'indentation du code est significative et donc fondamentale !!! C'est elle qui permet de définir les *blocs* de code, remplaçant en cela les accolades utilisées dans d'autres langages. Cette **obligation d'indenter son code** pour le structurer entraîne en outre une grande lisibilité et légèreté du code (absence d'accolades, points-virgules...).

❗ Pour des raisons de portabilité, le “[Style Guide for Python Code](#)” recommande d'utiliser 4 **<espace>** par niveau d'indentation et pas de caractère `<tab>`. En outre Python v3 ne permet plus de mixer les `<espace>` et `<tab>` dans un même bloc. Nous vous recommandons donc vivement de **configurer votre éditeur** ou votre IDE de façon que l'usage de la touche `<tab>` du clavier insère 4 caractères `<espace>` (et non pas 1 caractère `<tab>` ou 8 `<espace>`) !

❗ Notez encore qu'un *bloc de code* (ceci est notamment valable pour les *fonctions*, les structures `if`, `for`, `while`...) doit contenir au minimum une instruction. S'il n'en a pas,

on peut utiliser l'instruction `pass` qui n'effectue aucune action (*placeholder*).

2.3.2 Exécution conditionnelle if - elif - else

L'exemple ci-contre illustre la forme complète de cette structure. Les parties `elif...` et `else...` sont facultatives. Pour des tests multiples, on peut bien entendu cascader plusieurs parties `elif...`.

Notez bien la présence du caractère `:` (double point) précédant le début de chaque bloc !

```
1 a, b = 4, 5
2 if a > b:
3     print("%f est supérieur à %f" % (a,b) )
4 elif a == b:
5     print("%f est égal à %f" % (a,b) )
6 else:
7     print("%f est inférieur à %f" % (a,b) )
```

⚠ Sachez que si, à la place d'une *condition*, on teste directement un objet :

- sera assimilé à **faux** : entier ou flottant de valeur `0`, `None`, objet vide (chaîne, liste, tuple ou dictionnaire sans élément)
- sera assimilé à **vrai** : entier ou flottant différent de `0`, objet non vide (chaîne, liste, tuple, dictionnaire)

Mais comme une règle de bonne conduite Python dit qu'il faut programmer les choses *explicitement*, il est préférable de tester si un *objet* n'est pas vide en faisant `if len(objet) != 0` plutôt que `if objet` !

Expressions conditionnelles

⚠ Pour les structures `if... else...` dont les blocs ne contiendraient qu'une instruction, Python propose, comme d'autres langages, une forme compacte tenant sur une seule ligne appelée *expressions conditionnelles* (*ternary selection*). Sa syntaxe est :

```
expressionT if condition else expressionF
```

où *expressionT* est évalué si la *condition* est vraie, sinon *expressionF*.

```
21 # L'instruction :
22 print('Paire') if val%2 == 0 else print('Impaire')
23
24 # est une forme abrégée de :
25 if val%2 == 0:
26     print('Paire')
27 else:
28     print('Impaire')
```

Quelques fonctions logiques

On peut tester une *séquence*, un *ensemble* ou un *objet iterable* avec les fonctions built-in suivantes :

- `any(objet)` : retourne `True` si l'un au moins des éléments est `True` (ou assimilé à vrai)

- `all(objet)` : retourne `True` si tous les éléments sont `True` (ou assimilés à vrai)

```
41 any(range(0,3))      # => True
42     # car éléments de valeurs 1 et 2 assimilés à vrai
43 all(range(0,3))      # => False
44     # car élément de valeur 0 assimilé à faux
```

2.3.3 Boucle for

La boucle `for` permet d'itérer les valeurs d'une *liste*, d'un *tuple*, d'une *chaîne* ou de tout *objet itérable*. Comme dans les autres structures de contrôle, le caractère `:` (double point) définit le début du bloc d'instruction contrôlé par `for`.

Pour itérer sur une suite de nombres entiers, on utilise souvent la fonction `range` (*objet itérable*) présentée plus haut.

Ci-contre, notez aussi l'utilisation de la fonction `enumerate` retournant donc un objet permettant d'itérer sur l'*indice* et la *valeur* d'une *séquence*, d'où les 2 variables qui suivent le mot-clé `for` !

De même, la méthode `dictionnaire.items()` retourne un objet permettant d'itérer sur la *clé* et la *valeur* de chaque élément d'un dictionnaire. Les méthodes `.keys()` et `.values()` retournent quant à elles respectivement les *clés* et les *valeurs* du dictionnaire.

⚠ Après le bloc d'instruction subordonné au `for`, on pourrait (mais cela n'est pas courant !) ajouter une clause `else` suivie d'un bloc d'instructions, ces dernières s'exécutant 1 fois lorsque l'on a terminé d'itérer.

```

1  # Sur listes ou tuples
2  lst = [10, 20, 30]
3  for n in lst:
4      print(n, end=' ')    # => 10 20 30
5
6  for index in range(len(lst)):
7      print(index, lst[index])
8      # => affiche:  0 10
9      #               1 20
10     #               3 30
11
12     for index, val in enumerate(lst):
13         print(index, val)
14         # => même affichage que ci-dessus
15
16     # Sur chaînes
17     voyelles = 'aeiouy'
18     for car in 'chaine de caracteres':
19         if car not in voyelles:
20             print(car, end='')
21             # => affiche les consonnes: chn d crctrs
22
23     # Sur dictionnaires
24     carres = {}
25     for n in range(1,4):
26         carres[n] = n**2    # => {1: 1, 2: 4, 3: 9}
27
28     for k in carres:    # itère par défaut sur la clé !
29         # identique à: for k in carres.keys():
30             print(k, end=' ')    # => 1 2 3
31
32     for n in sorted(carres):
33         # identique à: for n in sorted(carres.keys()):
34             print("Carré de %u = %u" % (n, carres[n]))
35             # => affiche:  Carré de 1 = 1
36             #               Carré de 2 = 4
37             #               Carré de 3 = 9
38
39     for cle, val in carres.items():
40         print('Clé: %s, Valeur: %s' % (cle, val))
41         # => affiche:  Clé: 1, Valeur: 1
42         #               Clé: 2, Valeur: 4
43         #               Clé: 3, Valeur: 9

```

Compréhension de listes, sets et dictionnaires

❗ Lorsqu'il s'agit de construire un container de type *liste*, *set* ou *dictionnaire* à l'aide d'une boucle **for** assortie éventuellement d'une condition, Python offre une construction compacte appelée *comprehension expression*. Sa syntaxe est :

```
liste = [ expression for expr in iterable if cond ]
```

```
set = { expression for expr in iterable if cond }
```

```
dict = { expr1:expr2 for expr in iterable if cond }
```

S'agissant d'une *liste* ou d'un *set*, l'*expression*, évaluée à chaque itération de la boucle, constitue la valeur de l'élément inséré dans la liste ou le set.

Concernant le dictionnaire *dict*, les expressions *expr1* et *expr2* constituent respectivement les clés et valeurs des paires insérées.

Le test **if cond** est facultatif. Il peut aussi prendre la forme d'une *expression conditionnelle* telle que présentée plus haut :

```
[exprT if cond else exprF for expr in iterable]
```

```
51 # L'expression ci-dessous fabrique la liste des carrés
52 # des nombres pairs de 1 à 10 => [4, 16, 36, 64, 100]
53 carres = [ nb*nb for nb in range(1,11) if nb%2==0 ]
54
55 # ce qui est équivalent à :
56 carres = []
57 for nb in range(1,11):
58     if nb%2 == 0:
59         carres.append(nb*nb)
60
61 # L'expression ci-dessous retourne, dans l'ordre alpha-
62 # bétique, la liste des car. utilisés dans une chaîne
63 sorted({ car for car in 'abracadabra' })
64 # => ['a', 'b', 'c', 'd', 'r']
65
66 # L'expression ci-dessous récupère, dans le dictionnaire
67 # dic, tous les éléments dont la valeur est supérieure
68 # à 10, et les insère dans le nouveau dictionnaire sup10
69 dic = { 'a': 12.50, 'b': 3.50, 'c': 11.00, 'd': 6.00 }
70 sup10 = { cle:val for cle,val in dic.items() if val>10 }
71
72 # L'expression ci-dessous parcourt les entiers de 1 à 9
73 # et affiche les valeurs impaires, sinon 0
74 [ x if x%2 else 0 for x in range(1,10) ]
75 # => [1, 0, 3, 0, 5, 0, 7, 0, 9]
```

2.3.4 Boucle while

La boucle **while** permet d'exécuter un bloc d'instruction aussi longtemps qu'une *condition* (expression logique) est vraie.

Notez aussi la présence du caractère **:** (double point) définissant le début du bloc d'instruction contrôlé par **while**.

```
1 nb = 1 ; stop = 5
2 # Affiche le carré des nombres de nb à stop
3 while nb <= stop:
4     print(nb, nb**2)
5     nb += 1
```

Comme pour la *boucle for*, après le bloc d'instruction subordonné à **while**, on pourrait aussi ajouter une clause **else** suivie d'un bloc d'instructions, ces dernières s'exécutant 1 fois lorsque la *condition* sera fausse.

EXERCICE 3

2.3.5 Instructions continue et break

Dans une `boucle for` ou une `boucle while` :

- l'instruction `continue` passe à l'itération suivante de la boucle courante (i.e. sans poursuivre l'exécution des instructions du bloc)
- l'instruction `break` sort de la boucle courante ; si la boucle comporte un bloc `else`, celui-ci n'est pas exécuté

```
1 # Affiche les nombres impairs et leur carré jusqu'à ce
2 # que le carré atteigne 25 => 1 1, 3 9, 5 25
3 for n in range(0, 10):
4     if (n % 2) == 0: # nombre pair
5         continue   # => re-boucler
6     carre = n**2
7     if carre > 25:
8         break       # => sortir de la boucle
9     print(n, carre)
```

2.4 Fonctions, modules, packages, scripts

2.4.1 Fonctions

De façon générale, on implémente une *fonction* lorsqu'un ensemble d'instructions est susceptible d'être utilisé plusieurs fois dans un programme. Cette décomposition en petites unités conduit à du code plus compact, plus lisible et plus efficace.

L'exemple ci-contre illustre les principes de base de définition d'une *fonction* en Python :

- la première ligne `def nomFonction(arguments...):` définit le **nom** avec lequel on invoquera la fonction, suivi entre parenthèses de son(s) éventuel(s) **arguments** (paramètres d'entrée) séparés par des virgules ; cette ligne doit se terminer par `:`
- les instructions de la fonction (**corps** de la fonction) constituent ensuite un bloc qui doit donc être indenté à droite
- au début du corps on peut définir, sous forme de chaîne/commentaire multi-ligne, le texte d'**aide en-ligne** (appelé *docstrings*) qui sera affiché avec `help(fonction)`
- avec `return expression` on sort de la fonction en renvoyant optionnellement des **données de retour** sous forme d'un objet de n'importe quel type ; si l'on ne passe pas d'argument à `return`, la fonction retournera alors `None` (objet nul) ; dans l'exemple ci-contre, on retourne 2 valeurs que l'on a choisi d'emballer sous forme de *tuple*

```
1 def somProd(n1, n2):
2     """Fonction calculant somme et produit de n1 et n2
3     Résultat retourné dans un tuple (somme, produit)"""
4     return (n1+n2, n1*n2)
5
6
7 help(somProd) # => affiche :
8 **somProd** (n1, n2)
9     Fonction calculant somme et produit de n1 et n2
10    Résultat retourné dans un tuple (somme, produit)
11
12 somProd(3,10) # => (13, 30)
13 somProd()    # => erreur "somProd() takes exactly 2 args"
14             # et même erreur si on passe 1 ou >2 args.
15
16 # Une fonction étant un objet, on peut l'assigner
17 # à une variable, puis utiliser celle-ci comme un
18 # "alias" de la fonction !
19 sp = somProd
20 sp(3,10)     # => (13, 30)
```

S'agissant du nom de la fonction, il est de coutume, en Python, de le faire débiter par un caractère minuscule. En outre s'il est composé de plusieurs mots, on concatène ceux-ci et les faisant débiter chacun par une majuscule. Exemple : `uneFonctionBienNommee` .

Lors de l'appel à la fonction, il est aussi possible de passer les paramètres **de façon nommée** avec `paramètre=valeur` . Dans ce cas, l'ordre dans lequel on passe ces paramètres n'est pas significatif !

On peut en outre définir, dans la déclaration `def` , des **paramètres optionnels** . Si l'argument n'est pas fourni lors de l'appel de la fonction, c'est la *valeur par défaut* indiquées dans la définition qui sera utilisée par la fonction.

```
21 def fct(p1, p2=9, p3='abc'):  
22     # 1 param. obligatoire, et 2 param. optionnels  
23     return (p1, p2, p3)  
24  
25 print(fct())           # => erreur (1 param. oblig.)  
26 print(fct(1))          # => (1, 9, 'abc')  
27 print(fct(1, 2))       # => (1, 2, 'abc')  
28 print(fct(1, 2, 3))    # => (1, 2, 3)  
29 print(fct(p3=3, p1='xyz')) # => ('xyz', 9, 3)
```

Si le nombre de paramètres n'est pas fixé d'avance, on peut utiliser les techniques suivantes :

- récupérer les *paramètres positionnels* sur un *tuple* en définissant une variable précédée de `*`
- récupérer les *paramètres nommés* sur un *dictionnaire* en définissant une variable précédée de `**` ; il sera dans ce cas nécessaire d'invoquer la fonction en passant les paramètres sous la forme `paramètre=valeur`

```
41 def fct1(*params_tpl):  
42     print(params_tpl)  
43  
44 fct1(1, 'a')           # => (1, 'a')  
45  
46  
47 def fct2(**params_dic):  
48     print(params_dic)  
49  
50 fct2(p1=1, p2='a')     # => {'p2': 'a', 'p1': 1}
```

On peut finalement combiner ces différentes techniques de définition de paramètres (paramètres optionnels, tuple, dictionnaire) ! Par exemple la fonction `fct(n1, n2=4, *autres)` a 1 paramètres obligatoire, 1 paramètre facultatif avec valeur par défaut, et 0 ou plusieurs paramètres positionnels supplémentaires facultatifs.

❗ Il est important de noter que sous Python les objets transmis aux fonctions sont **passés par référence** (adresses vers ces objets), contrairement à MATLAB/Octave (passage par valeur) ! S'ils sont modifiés par les fonctions, ces objets le seront donc également dans le programme appelant !

Il y a cependant une exception à cette règle qui concerne les variables de type non modifiable (*immutables*), celles-ci étant alors passées **par valeur** (copie des données). Cela concerne donc les *types simples* (entier, flottant, complexe) et les *containers* de type *tuples*, *chaînes* et *frozensets*.

```
61 def ajoutElem(liste, elem):  
62     liste.append(elem)  
63     elem = 'xyz'  
64     # notez que cette fonction n'a pas de return !  
65  
66 liste=['un']  
67 elem='deux'  
68 ajoutElem(liste, elem)  
69 print(liste) # => ['un', 'deux'] : liste a été modifiée!  
70 print(elem)  # => 'deux' : chaîne n'a PAS été modifiée
```


Portée des variables (et autres objets)

La *portée* est le périmètre dans lequel un nom (de variable, fonction...) est connu (*visible*) et utilisable. Sous Python, cela est lié au concept d' *espaces de noms* (*namespaces*) où il faut distinguer :

- l'espace de noms global : il est créé au début de l'exécution du programme Python, respectivement lorsque l'on lance un interpréteur en mode interactif ; la fonction `globals()` retourne le *dictionnaire* des objets globaux, la valeur d'un *objet* spécifié étant donc `globals()["objet"]`
- les espaces de noms locaux des fonctions : ils sont créés lors des appels aux fonctions, à raison d'un espace local pour chaque imbrication de fonction ; la liste des objets de cet espace est donnée par `locals()`
- les espaces de noms des modules et des classes...

La fonction `dir()` retourne la liste des noms connus dans l'espace de noms courant (et `dir(objet)` la liste des attributs associés à l' *objet* spécifié).

❗ La recherche d'un nom débute dans l'espace de noms le plus intérieur, puis s'étend jusqu'à l'espace de nom global. C'est ainsi que :

- les variables du programme principal (variables dites *globales*) sont visibles dans les fonctions appelées, mais ne sont par défaut pas modifiables ; si l'on définit dans une fonction une variable de même nom qu'une variable de niveau supérieur, cette dernière est en effet masquée
- les variables définies dans le corps d'une fonction (variables *locales*) ne sont pas visibles dans les niveaux supérieurs (fonction appelante, programme principal)
- les fonctions peuvent cependant modifier des variables globales en utilisant l'instruction `global var1, var2, ...` (forçant ces noms à être liés à l'espace de nom global plutôt qu'à l'espace local courant)

```
81 # Exemple 1
82
83 def afficheVar():      # fonction sans param. d'entrée
84     var2 = 999         # variable locale
85     print('var1 =', var1, '\nvar2 =', var2)
86
87
88 var1 = 111 ; var2 = 222    # variables globales
89 afficheVar() # => affiche :
90     # var1 = 111 => variable globale vue par la fct
91     # var2 = 999 => var. locale masquant var. globale
92
93
94
95
96 # Exemple 2
97
98 def incremCompt():     # fonction sans param. d'entrée
99     global compteur    # définition var. globale
100     if 'compteur' not in globals(): # test exist. var.
101         compteur = 0   # initialisation du compteur
102     compteur += 1
103     print('Appelé', compteur, 'fois')
104
105
106 incremCompt() # => affiche: Appelé 1 fois
107 incremCompt() # => affiche: Appelé 2 fois
108 incremCompt() # => affiche: Appelé 3 fois
```

Lambda fonctions

❗ Pour les fonctions donc le corps ne contiendrait qu'une seule instruction `return expression`, Python offre la possibilité de définir une *fonction anonyme* aussi appelée *lambda fonction* (ou *lambda expression*). Sa syntaxe est :

```
lambda arguments... : expression
```

Dans l'exemple ci-contre, nous l'assignons la lambda fonction à une variable pour

pouvoir l'utiliser comme une fonction classique. Dans la pratique, les lambda fonctions sont utiles pour définir une fonction simple comme argument à une autre fonction, pour retourner des valeurs, etc...

```
121 # La lambda fonction ci-dessous :
122 somProd = lambda n1, n2: (n1+n2, n1*n2)
123
124 # est équivalente à :
125 def somProd(n1, n2):
126     return (n1+n2, n1*n2)
127
128 # Dans les 2 cas :
129 somProd(3, 10)    # => (13, 30)
```

2.4.2 Modules

Utilisation de modules

Un *module* Python (parfois appelé *bibliothèque* ou *librairie*) est un fichier rassemblant des *fonctions* et *classes* relatives à un certain domaine. On implémente un module lorsque ces objets sont susceptibles d'être utilisés par plusieurs programmes.

Pour avoir accès aux fonctions d'un module existant, il faut charger le module avec la commande `import`, ce qui peut se faire de différentes manières, notamment :

- A. `from module import *` : on obtient l'accès direct à l'ensemble des fonctions du *module* indiqué sans devoir les préfixer par le nom du module
- B. `from module import fct1, fct2...` : on ne souhaite l'accès qu'aux fonctions *fct1*, *fct2*... spécifiées
- C. `import module1, module2...` : toutes les fonctions de(s) module(s) spécifié(s) seront accessibles, mais seulement en les préfixant du nom du *module*
- D. `import module as nomLocal` : toutes les fonctions du *module* sont accessible en les préfixant du *nomLocal* que l'on s'est défini

Les méthodes (C) et (D) sont les plus pratiquées. La technique (A) ne devrait en principe **pas être utilisée**, car elle présente le risque d'écrasement d'objets si les différents modules chargés et/ou votre programme implémentent des objets de noms identiques. Elle rend en outre le code moins lisible (on ne voit pas d'où proviennent les fonctions utilisées).

Explication technique : quand on importe un module avec (C) ou (D), un "espace de nom" spécifique (que l'on peut examiner avec `dir(module)`) est créé pour tous les objets du module, puis un objet de module est créé pour fournir l'accès (avec la notation `module.fonction()`) à cet espace de nom.

On obtient la liste des modules couramment chargés avec l'attribut `sys.modules`.

Les fonctions et classes Python de base (*built-in*) sont définis dans un module préchargé nommé `builtins`.

S'agissant de l'installation de modules et packages ne faisant pas partie de la *bibliothèque standard*, voyez [ce chapitre](#).

```
1  # Admettons qu'on doive utiliser la constante 'pi' et la
2  # fonction 'sin', tous deux définis dans le module 'math'
3  # (Rem: %who et %whos ci-dessous sont propre à IPython)
4
5  # A)
6  from math import *
7  %who      # ou %whos => on voit toutes fcts importées !
8  dir()     # => objets dans namespace, notamment ces fcts
9  sin(pi/2) # => 1.0
10 cos(pi)   # => -1.0
11
12 # B)
13 from math import pi, sin
14 %who      # ou %whos => seules 'pi', 'sin' accessibles
15 dir()     # => objets dans namespace, notamment ces fcts
16 sin(pi/2) # => 1.0
17 cos(pi)   # => erreur "name 'cos' is not defined"
18
19 # C)
20 import math
21 %who      # ou %whos => module 'math' importé
22 math.<tab> # => liste les fonctions du module
23 dir(math) # => objets dans namespace
24 help(math) # => affiche l'aide sur ces fonctions
25 help(math.sin) # => aide sur la fct spécifiée (sin)
26 math.sin(math.pi/2) # => 1.0
27 cos(pi)   # => erreur (non préfixés par module)
28 math.cos(math.pi) # => -1.0
29
30 # D)
31 import math as mt
32 %who      # (ou %whos) => module math importé s/nom mt
33 mt.<tab>   # => liste les fonctions du module
34 dir(mt)   # => objets dans namespace
35 help(mt)  # => affiche l'aide sur ces fonctions
36 mt.sin(mt.pi/2) # => 1.0
37 math.sin(math.pi/2) # => erreur "name math not defined"
38 mt.cos(mt.pi) # => -1.0
```

Écriture de modules

Créer un *module* revient simplement à créer un fichier nommé `module.py` dans

lequel seront définies les différents objets (fonctions, classes...) du module. Le nom du module est en général défini en caractères minuscules seulement (**a-z** , avec éventuellement des **_**).

Outre les fonctions, on peut également ajouter dans le module des instructions à exécution immédiate, celles-ci étant alors exécutées au moment de l'import. **!** Si l'on souhaite qu'elles ne soient exécutées que lorsqu'on exécute le module en tant que *script*, on les définira dans un bloc précédé de `if __name__ == '__main__':`

Lors d'un `import`, Python commence par rechercher le module spécifié dans le répertoire courant, puis dans le(s) répertoire(s) défini(s) par la variable d'environnement `PYTHONPATH` (si celle-ci est définie, par exemple par votre prologue `~/.profile`), puis finalement dans les répertoires systèmes des modules Python (`/usr/lib/python<version>`). La liste de tous ces répertoires (chemins de recherche) est donnée par l'attribut `sys.path`. Python considère, dans l'ordre, les extensions suivantes :

- `.pyd` et `.dll` (Windows), `.so` (Unix) : modules d'extension Python
- `.py` : modules source Python
- `.pyc` : modules Python compilés en *bytecode* (automatiquement généré et mis en *cache* lorsque l'on importe le module)

```
61 # Fichier mon_module.py
62
63 def carre(nb):
64     return nb*nb
65
66 def cube(nb):
67     return nb*nb*nb
68
69 if __name__ == '__main__':
    # module exécuté en tant que script
    print('Exemple de la fonction carre()')
    print('  carre(4) => ', carre(4))
    print('Exemple de la fonction cube()')
    print('  cube(3) => ', cube(3))

# Utilisation du module

import mon_module      # => n'exécute pas code
mon_module.carre(3)     # => 9
mon_module.cube(2)      # => 8

# Exécution du module en tant que script

$ python mon_module.py  # => affiche ce qui suit :
Exemple de la fonction carre()
  carre(4) =>  16
Exemple de la fonction cube()
  cube(3) =>  27
```

! Lorsque l'on développe interactivement, Python ne charge le module que la première fois qu'on l'importe. Si l'on veut recharger un module que l'on vient de modifier, il faut faire : `imp.reload(module)`. Il faut cependant être attentif au fait que les objets créés avant le rechargement du module ne sont pas mis à jour par cette opération !

2.4.3 Packages

Un package (paquetage) permet de réunir plusieurs *modules* sous un seul nom et pouvant ainsi être chargés par une seule instruction `import package`. C'est donc un ensemble de modules physiquement rassemblés dans un répertoire ou une arborescence. Nous n'entrerons ici pas davantage dans les détails d'implémentation d'un package.

2.4.4 Scripts

Un script (ou programme) Python est un fichier de code Python que l'on peut exécuter dans un interpréteur Python. Son nom se termine en principe par l'extension `.py`.

On a déjà dit quelques mots sur les 2 lignes d'*en-têtes* d'un script (définition de l'interpréteur, encodage des caractères). Suivent alors les instructions d'importation des *modules* utilisés. Puis le script se poursuit par la définition d'éventuelles *fonctions*. On trouve finalement le corps principal du programme, entité connue par l'interpréteur Python sous le nom `__main__`

On peut exécuter un script nommé `script.py` de différentes manières :

- depuis un **shell** (fenêtre de terminal/commande) en frappant : `python script.py` (ou éventuellement `python3 script.py` selon votre installation, afin d'utiliser Python v3) ; ajoutez l'option `-i` si vous désirez que l'interpréteur Python passe en **mode interactif** juste après l'exécution du script (par exemple pour examiner des variables globales où tracer le stack d'erreurs...)
- depuis un shell en frappant directement : `./script.py`, mais pour autant que le script ait l'attribut de permission execute (que l'on peut définir, sous Linux et macOS, avec : `chmod +x script.py`)
- depuis un **explorateur de fichier** en double-cliquant sur l'icône du script, pour autant également que le script ait la permission execute
- depuis **IPython** en frappant : `%run script` (pas besoin de saisir l'extension `.py`)
- interactivement depuis le **shell Python** avec : `exec(open('script.py').read())` ; noter que dans ce cas le script s'exécute dans le contexte de la session IPython et "voit" toutes les objets courants, ce qui peut être intéressant... mais peut aussi constituer un danger !

```
1  # Fichier monScript.py
2
3  #!/usr/bin/env python
4  # -*- coding: utf- 8  -*-
5
6  import sys
7
8  def pause(prompt): # exemple de fonction dans script
9      print()
10     input(prompt)
11
12     print('- nom de fichier du script :', sys.argv[0])
13     print('- invoqué avec', len(sys.argv)-1, \
14           'arguments :', sys.argv[1:])
15
16     pause('Frapper <enter> pour refermer la fenêtre ')
17     # Ligne ci-dessus: pour voir qqch si on lance le
18     # script par double-clic depuis un explorateur
19
20
21
22     # Exécution du script
23
24     $ python monScript.py un deux # => affiche :
25
26     - nom de fichier du script : monScript.py
27     - invoqué avec 2 arguments : ['un', 'deux']
28
29     Frapper <enter> pour refermer la fenêtre
```

Comme on le voit dans l'exemple ci-dessus, on peut récupérer sur l'attribut `sys.argv` le nom du script et les **arguments** qui lui ont été passés à l'exécution. Mais sachez qu'il existe des modules spécifiquement dédiés au parsing d'arguments (notamment `argparse`) qui vous seront bien utiles si vous souhaitez invoquer le script à la façon des commandes Unix (avec des arguments du type : `-o`, `-option`, `-option=valeur`, etc...).

2.5 Opérations sur les chaînes

Après avoir brièvement présenté *plus haut* comment on définit et manipule des chaînes de caractères sous Python, nous présentons ici plus en détail d'autres traitements possibles.

2.5.1 Formatage

Opérateur de conversion %

Le formatage permet de composer des chaînes de caractères incorporant des données de types entier, flottant et chaîne. Cela correspond, sous C ou MATLAB/Octave, à l'usage de la fonction `sprintf`. Sous Python, on utilise la syntaxe suivante :

```
'chaîne avec spécifications de conversion %s %d %o %x %f %e' % ( valeurs, variables ou expressions )
```

À la suite de la chaîne et de l'opérateur de conversion `%`, les objets à insérer (valeurs, variables, expressions) sont passés entre parenthèses, c'est-à-dire sous forme de *tuple*. Les valeurs seront alors insérées dans la chaîne aux emplacements balisés par des spécifications de conversion qui doivent exactement correspondre, en nombre et type, avec les éléments insérés !

Les principales spécifications de conversion sont les suivantes (issues du langage C) :

- `%s` pour une chaîne
- `%d`, `%o` ou `%x` pour un entier, respectivement en base décimale, octale ou hexadécimale
- `%f` ou `%e` pour un flottant, respectivement en notation fixe ou notation scientifique avec exposant

```
1 data = [ 100, 'stylos', 2.5,  
2         20, 'plumes', 25,  
3         2, 'calculettes', 105 ]  
4 n = 0  
5 while n < len(data):  
6     print(' %4d %-12s %7.2f ' %  
7         ( data[n], data[n+1], data[n+2] ) )  
8     n += 3  
9  
10 # L'exécution du code ci-dessus affiche :  
11 100 stylos          2.50  
12 20  plumes         25.00  
13 2   calculettes    105.00
```

On peut en outre intercaler, entre l'opérateur `%` et les codes de conversion `s d o x f` et `e` :

- un tiret `-` pour que la donnée soit justifiée à gauche de son champ plutôt qu'à droite (alignement par défaut)
- un nombre définissant la largeur du champ (nombre de caractères) occupé par la donnée insérée

Le nombre doit être entier pour les spécifications `%s`, `%d`, `%o` et `%x`. Il peut être réel (sans exposant) pour `%f` et `%e`, la valeur après le point décimal indiquant alors la précision d'affichage, c'est-à-dire le nombre de chiffres après la virgule. Par exemple `%7.2f` insère, dans un champ de 7 caractères, le nombre formaté avec 2 chiffres après la virgule. Sans indication de nombre, la largeur du champ est dynamique, correspondant au nombre de chiffres effectif de l'entier inséré ou au nombre de caractères de la chaîne insérée. S'agissant d'un flottant, il sera alors affiché avec une précision de 6 chiffres après la virgule.

Méthode de formatage format

Sous Python v3, le type chaîne s'enrichit d'une nouvelle méthode `.format` :

```
'chaîne avec indicateurs de format {}'.format( valeurs, clé=valeur, variables, expressions )
```

Définis dans la chaîne entre accolades `{ }`, les indicateurs de format permettent de spécifier notamment :

- un nombre entier : c'est l'indice de l'élément du tuple à insérer (passage par position)
- un nom : c'est la clé de l'élément à insérer (passage par nom)
- `:s :d :o :x :f :e` : spécifications analogues à celles vues plus haut (notez que le caractère `:` remplace `%`)
- caractère `<` `>` ou `^` (juste après le `:`) : pour justifier respectivement à gauche, à droite ou centrer dans le champ
- possibilité d'intercaler aussi un nombre fixant la largeur du champ

S'agissant des données à formater, si l'on passe une variable référant une séquence, on peut faire un unpack des valeurs en faisant précéder le nom de variable du caractère `*`. Dans le cas où la variable se réfère à un dictionnaire, on la précédera de `**`.

Pour davantage de détails, voyez la [documentation Python](#).

```
21 # Pour obtenir le même affichage que l'ex. ci-dessus :
22 ...
23     print('  {:4d}  {:<12s}  {:7.2f} '.format(
24         data[n], data[n+1], data[n+2]) )
25 ...
26
27 # Passage par position :
28 '{ } { } {}'.format(11, 'abc', 3.5)    # => '11 abc 3.5'
29 '{0}{1}{0}'.format('abra', 'cad')     # => 'abracadabra'
30 '{1:.4f} {0}'.format('abc', 3.5)      # => '3.5000 abc'
31
32 # Passage d'un objet tuple :
33 coord=(12.3, 56.7)
34 '{0[0]}, {0[1]}'.format(coord)         # => '12.3, 56.7'
35 'X= {:.2f} Y= {:.2f}'.format(*coord)   # => 'X= 12.30 Y= 56.70'
36
37
38 # Passage par nom, ou passage d'un objet dictionnaire :
39 'Coord: {x:.2f}, {y:.2f}'.format(y=56.7, x=12.3)
40                                     # => 'Coord: 12.30, 56.70'
41
42 dic = {'y': 56.7, 'x': 12.3}
43 'Coord: {x:.2f}, {y:.2f}'.format(**dic) # => 'Coord: 12.30, 56.70'
```

Template de chaîne

P2 Sous Python v2 (depuis 2.4), le module `string` offrait une méthode `.Template()` permettant de définir des modèles de chaînes puis en faire usage par un mécanisme de substitution. Mais cette technique, bien que toujours disponible sous Python v3, n'est plus tellement utile, étant donné que la méthode `.format()`, avec le passage par nom, fait la même chose sans nécessiter de module spécifique.

```
61 from string import Template
62
63 coords = [ (11, 22), (33, 44) ]
64 tpl = Template('X = $x, Y = $y')
65
66 for point in coords:
67     print( tpl.substitute(x=point[0], y=point[1]) )
68 # => affiche :
69 X = 11, Y = 22
70 X = 33, Y = 44
```

2.5.2 Fonctions et méthodes de chaînes

Quelques fonctions :

- `len(chaine)` : nombre de caractères de chaîne
- `str(nombre)` : conversion de nombre en chaîne
- `ord(caractère)` : code-ASCII du caractère
- `chr(code-ASCII)` : caractère correspondant au code-ASCII
- `min(chaine)` , `max(chaine)` : respectivement le plus petit ou le plus grand caractère de chaîne
- `ascii(objet)` : affiche objet (pas seulement chaîne) en échappant les caractères non-ASCII par des séquences `\x`, `\u` ou `\U`

Rappelons que `+` et `*` ainsi que les opérateurs de comparaison (`==` `<` `>` etc...) peuvent être utilisés sur les chaînes.

```
1 len('123')          # => 3 caractères
2 str(123.456)        # => '123.456'
3
4 [ord(car) for car in 'abcdef']
5 # => [97, 98, 99, 100, 101, 102]
6
7 [chr(code) for code in range(97,103)]
8 # => ['a', 'b', 'c', 'd', 'e', 'f']
9
10 min('abcABC')       # => 'A'
11 max('abcABC')       # => 'c'
12 ascii('Élève')      # => '\\xc9l\\xe8ve'
13
14 'oh' + 2*'là'        # => 'ohlàlà'
15 'Ceci'[0:2]=='Cela'[0:2] # => True
16 'Ceci' > 'Cela'      # => False
17
18 def isMinuscule(ch):
19     if 'a' <= ch <= 'z': # condition double !
20         return True
21     else:
22         return False
23 isMinuscule('e')     # => True
24 isMinuscule('E')     # => False
```

Dans les **méthodes** ci-après, les paramètres *debut* et *fin* sont optionnels. Ils permettent de limiter la portée de la méthode sur la portion de chaîne comprise entre les indices *debut* et *fin*.

- `.find(sch,debut,fin)` ou `.index(...)` : retourne l'indice du début de la sous-chaîne *sch* ; ces 2 fonctions diffèrent seulement si *sch* n'est pas trouvée (`.find` retournant `-1`, et `.index` une erreur)
- `.rfind(sch,debut,fin)` ou `.rindex(...)` : idem, sauf que la recherche s'effectue à partir de la fin de la chaîne
- `.count(sch,debut,fin)` : retourne le nombre d'occurrence(s) de la sous-chaîne *sch*
- `.startswith(sch)` et `.endswith(sch)` : teste respectivement si la chaîne débute ou se termine par *sch*
- `.replace(s1,s2,nb)` : remplace toutes (ou seulement *nb*) occurrences de *s1* par *s2*
- `.maketrans(table)` : remplacement, caractère par caractère, selon la table de translation *table* définie avec `.maketrans(avant,après)` (où

```
31 # Recherche et décompte
32 st = '123 abc 456 abc 789'
33 st.find('abc')       # ou st.index('abc')    => 4
34 st.find('abc',5)     # ou st.index('abc',5) => 12
35 st.rfind('abc')      # ou st.rindex('abc')  => 12
36 st.find('xyz')       # => -1
37 st.index('xyz')      # => erreur "substring not found"
38
39 st.count('abc')       # => 2 occurrences
40 st.count('a',0,4)    # => 0 occurrences
41 'abc' in st          # => True
42 'Oui'.lower().startswith('o') # => True
43 'fichier.py'.endswith('.py') # => True
44
45 # Substitution
46 st.replace('abc','xxx',1) # => '123 xxx 456 abc 789'
47
48 table=st.maketrans('àâä', 'aaa') # table de substit
```

avant et après sont 2 chaînes de longueur identique)

- `.strip(cars)`, `.lstrip(cars)`, `.rstrip(cars)` : enlève les caractères espace (ou les caractères `cars` spécifiés) respectivement au début, à la fin ou autour de la chaîne
- `.split(delim,max)` : découpe la chaîne là où se trouvent un(des) caractère(s) espace, tab et newline (ou la chaîne `delim`) ; limite à `max` le nombre de découpage
- `.splitlines()` : découpe une chaîne multi-lignes aux emplacements des sauts de ligne newline
- `'delim'.join(iterable)` : concatène tous les éléments de l'objet `iterable` en intercalant la chaîne `delim` entre chaque élément
- `.upper()`, `.lower()` et `.swapcase()` : convertit tous les caractères respectivement en majuscules, minuscules, ou inverse la casse
- `.capitalize()` et `.title()` : convertit en majuscule respectivement le 1er caractère de la chaîne ou le premier caractère de chaque mot de la chaîne
- `.ljust(largeur,car)`, `.rjust(largeur,car)` et `.center(largeur,car)` : aligne la chaîne respectivement à gauche, à droite ou de façon centrée dans un champ de largeur caractères, et complétant les espaces vides par des espaces ou par le `car` spécifié

Pour plus de précisions sur ces méthodes, voir `help(str.methode)`.

```
48 table = str.maketrans('àèé', 'ae', # table de substitution
49 'accentués àèé'.translate(table) # => 'accentues aee'
50
51 # Nettoyage
52 ' * bla * '.strip() # => '* bla *'
53 ' bla '.rstrip() # => ' bla'
54 '* -bla--'.strip(' *-') # => 'bla'
55 '> commande'.lstrip('> ') # => 'commande'
56
57 # Découpage
58 'Une p\`tit phrase'.split() #=> ['Une','p`tit','phrase']
59 # <espace>, <tab>, <newline>
60 '2013-11-08'.split('-') # => ['2013', '11', '08']
61
62 '''Parag.
63 multi-
64 lignes'''.splitlines() # => ['Parag.','multi-','lignes']
65
66 # Concaténation
67 mots = ['Une', 'petite', 'phrase']
68
69 concat = ''
70 for mot in mots:
71     concat += mot # => 'Unepetitephrase'
72
73 ''.join(mots) # => 'Unepetitephrase'
74 ' '.join(mots) # => 'Une petite phrase'
75
76 # Changement de casse
77 'Attention'.upper() # => 'ATTENTION'
78 'Oui'[0].lower() == 'o' # => True
79 'Inv. Maj/Min'.swapcase() # => 'iNV. mAJ/mIN'
80 'une phrase'.capitalize() # => 'Une phrase'
81 'un titre'.title() # => 'Un Titre'
82
83 # Alignement/justification
84 'Bla'.ljust(6) # => 'Bla   '
85 '123'.rjust(6,'0') # => '000123'
86 ' xxxx '.center(14,'-') # => '---- xxxx ----'
```

En outre des méthodes booléennes permettent de tester les caractères d'une chaîne : `.isalnum()` (alphanumérique), `.isalpha()` (alphabétique), `.isdecimal()`, `.isdigit()` et `.isnumeric()` (numérique).

2.5.3 Expressions régulières



Rédaction de ce chapitre en cours, merci de patienter !

2.6 Manipulation de fichiers

Nous décrivons ici les techniques standards de manipulation de fichiers, c'est-à-dire les méthodes et attributs de l'objet-fichier. Sachez que d'autres modules offrent des méthodes spécifiques de lecture/écriture de fichiers qui peuvent être utiles, notamment :

- certains modules de la librairie standard : `fileinput`, `csv`, `json` ...
- les modules `NumPy` (fonctions `.loadtxt`, `.savetxt`) et `SciPy` (fonctions `.io.loadmat`, `io.savemat`).

2.6.1 Ouverture et fermeture

Pour manipuler un fichier, il faut d'abord l'**ouvrir**, ce qui se fait avec le constructeur **open** dont les paramètres principaux sont les suivants :

```
fd = open(fichier, mode, buffering, encoding=encodage)
```

- le **fd** (descripteur de fichier) retourné est un objet-fichier, les diverses manipulations décrites dans les chapitres qui suivent étant donc des méthodes relatives à cet objet
- **fichier** est une chaîne définissant le nom du fichier, incluant son chemin d'accès (*path*) s'il n'est pas dans le répertoire courant (à moins de changer préalablement de répertoire courant avec **os.chdir(chemin)**)
- **mode** (voir ci-après) est une chaîne définissant le mode d'accès ; en l'absence de ce paramètre, fichier sera ouvert en mode **'rt'**
- **buffering** est un entier définissant si un buffer mémoire (accélérant les lecture/écriture) est associé au fichier :
 - absence de ce paramètre ou valeur **1** : buffer standard
 - **0** : pas de buffer (uniquement pour fichier binaire)
 - nombre : buffer de la taille spécifiée par nombre
- il est possible de définir le type d'encodage ; en l'absence de ce paramètre, l'encodage par défaut dépend de l'OS, mais c'est généralement **'utf-8'**, l'encodage **'latin1'** tendant à disparaître

```
1 fd = open('fich.txt', 'w')
2     # ouverture nouveau fichier en écriture
3
4 type(fd)          # => io.TextIOWrapper
5 print(fd)         # => affiche :
6     <... name='fich.txt' mode='w' encoding='UTF-8'>
7 help(fd)          # => méthodes associée à objet-fichier
8 fd.<tab>           # => ~idem sous IPython
9 fd.closed         # => False
10 fd.name           # => 'fich.txt'
11 fd.mode           # => 'w'
12 fd.encoding       # => 'UTF-8'
13 fd.readable()    # => False (serait True si 'r')
14
15
16 # On doit le cas échéant gérer soi-même les
17 # erreurs d'ouverture, p.ex. ici avec l'implé-
18 # mentation de notre propre fonction 'error' :
19
20 import sys
21 def error(msg): # fct d'affich. message & exit
22     sys.stderr.write("Error: %s\n" % msg)
23     sys.exit('Aborting...')
24
25 try:
26     fd = open('inexistant.txt', 'r')
27 except IOError as err:
28     error(err.strerror) # erreur => avorter
29 # ... suite du programme si OK
30
31 # Si fichier inexistant => cela affichera :
32 Error: No such file or directory
33 An exception has occurred, use %tb to
34 see the full traceback.
35 SystemExit: Aborting...
```

Les modes d'accès sont les suivants :

- **'r'** : accès en lecture seule ; erreur si le fichier n'existe pas
- **'w'** : accès en écriture ; si le fichier n'existe pas, il est créé ; s'il existe, son contenu est écrasé
- **'r+'** : accès en lecture et écriture, permettant de lire et intercaler des données dans le fichier
- **'a'** : accès en mode ajout (*append*) ; si le fichier n'existe pas, il est créé ; s'il existe, son contenu est laissé intact, et les écritures s'effectuent à la fin du fichier

et l'on peut ajouter les codes suivants :

- **'t'** : mode texte (par défaut)

- `'b'` : mode binaire

La fermeture du fichier s'effectue logiquement avec `fd.close()`. En cas d'oubli, tous les fichiers ouverts sont automatiquement fermés à la sortie du programme.

```
41 # ... suite de l'exemple ci-dessus :  
42 fd.close()    # fermeture du fichier  
43 fd.closed    # => True
```

2.6.2 Lecture

Plusieurs méthodes peuvent être utilisées, selon que l'on souhaite lire le fichier :

- A. d'une traite
- B. par paquets d'un certain nombre de caractères
- C. ligne après ligne

Dans les exemples qui suivent, on utilisera le fichier `essai.txt` contenant les 3 lignes suivantes :

```
1ère ligne  
2e ligne  
fin
```

A) Pour la **lecture d'une traite** en mémoire (convenant à des fichiers pas trop volumineux !), on dispose de deux méthodes :

- `.read()` : déverse le fichier sur une seule chaîne, y compris les `newline` (`\n`) ; on pourrait ensuite faire un découpage en lignes avec `chaîne.split('\n')`, mais on a dans ce cas meilleur temps d'utiliser la méthode ci-après
- `.readlines()` : déverse le fichier sur une liste qui contiendra autant d'éléments que de lignes du fichier, chaque élément étant terminé par un `newline` (`\n`)

```
1 # Ouverture
2 fd = open('essai.txt', 'r')
3
4 # Lecture intégrale sur une chaîne
5 chaîne = fd.read()
6 # => chaîne = '1ère ligne\n2e ligne\nfin\n'
7
8 fd.seek(0) # repositionnement au début fichier
9 # on aurait pu faire un close puis un open
10
11 # Seconde lecture intégrale, sur une liste cette fois
12 liste = fd.readlines()
13 # => liste = ['1ère ligne\n', '2e ligne\n', 'fin\n']
14
15 # Fermeture
16 fd.close()
17
18
19 # Les 3 instructions open, read/readline, close
20 # pourraient être réunies en une seule, respectivement :
21
22 chaîne = open('essai.txt', 'r').read()
23 liste = open('essai.txt', 'r').readlines()
24
25 # et c'est alors le garbage collector Python qui
26 # refermera les fichiers !
```

B) Si le fichier est très volumineux, la lecture d'une seule traite n'est pas appropriée (saturation mémoire). On peut alors utiliser la méthode


`.read(nbcar)` de **lecture par paquets** de `nbcar` caractères.

La taille de paquet de l'exemple ci-contre est bien entendu trop petite ; ce n'est que pour les besoins de la démonstration.

```
41 fd = open('essai.txt', 'r') # ouverture
42 chaîne = ''
43 while True:
44     paquet = fd.read(10) # paquet 10 car.
45     if paquet: # paquet non vide
46         chaîne += paquet
47     else: # paquet vide => fin du fichier
48         break
49 fd.close() # fermeture
50 # => chaîne = '1ère ligne\n2e ligne\nfin\n'
```

C) Une lecture **ligne par ligne** peut s'avérer judicieuse si l'on veut analyser chaque ligne au fur et à mesure de la lecture, ou que le fichier est très volumineux (i.e. qu'on ne veut/peut pas le charger en mémoire).

Dans tous les exemples ci-contre, on illustre leur fonctionnement en affichant les lignes au cours de la lecture, raison pour laquelle on supprime le `\n` terminant chaque ligne lue avec la méthode de chaîne `.rstrip()`.

- La solution 1 utilise la méthode `.readline()`. Elle est explicite mais verbeuse : on lit jusqu'à ce que `ligne` soit vide, signe que l'on est alors arrivé à la fin du fichier. Note : en cas de ligne vide dans le fichier, la variable `ligne` n'est pas vide mais contient `\n`.
- La solution 2 utilise la méthode `.readlines()` vue plus haut et non pas `.readline()`.
- La solution 3, encore plus simple, démontre le fait qu'un fichier est un objet itérable fournissant une ligne à chaque itération !
-  La solution 3bis (dérivée de la précédente) s'appuie sur l'instruction `with` dont le rôle est de définir un contexte au sortir duquel certaines actions sont automatiquement exécutées ; l'objet-fichier supportant le contexte manager, la fermeture du fichier est donc automatiquement prise en charge lorsque l'on sort du bloc `with`, et il n'y a donc pas besoin de faire explicitement un `close` !

```
61 # Solution 1
62 fd = open('essai.txt', 'r')
63 while True: # boucle sans fin
64     ligne = fd.readline()
65     if not ligne: break # sortie boucle
66     # identique à: if len(ligne)==0: break
67     print(ligne.rstrip())
68 fd.close()
69
70 # Solution 2, plus légère
71 fd = open('essai.txt', 'r')
72 for ligne in fd.readlines():
73     print(ligne.rstrip())
74 fd.close()
75
76 # Solution 3, plus élégante
77 fd = open('essai.txt', 'r')
78 for ligne in fd:
79     print(ligne.rstrip())
80 fd.close()
81
82 # Solution 3bis, encore plus compacte
83 with open('essai.txt', 'r') as fd:
84     for ligne in fd:
85         print(ligne.rstrip())
```

2.6.3 Écriture ou ajout

En premier lieu il faut rappeler où débutera l'écriture après l'ouverture du fichier. Pour cela, il faut distinguer les différents modes d'accès :

- `'w'` : l'écriture débute au début du fichier
- `'a'` : l'écriture débute à la fin du fichier
- `'r+'` : l'écriture s'effectue à la position courante suite à d'éventuelles opérations de lecture ou de positionnement

Pour garantir de bonnes performances, l'écriture est par défaut bufferisée en mémoire. Cette mémoire tampon est bien entendu entièrement déversée dans le fichier lorsqu'on referme celui-ci avec `fd.close()`. Mais sachez que l'on peut forcer à tout instant l'écriture du buffer avec `fd.flush()`.

A) La méthode `.write(chaine)` écrit dans le fichier la chaîne spécifiée. Elle n'envoie pas de newline après la chaîne, donc on est appelé à écrire soi-même des `'\n'` là où l'on veut des sauts de ligne.

B) La méthode `.writelines(sequence ou ensemble)` écrit dans le fichier une séquence (*tuple* ou *liste*) ou un ensemble (*set* ou *frozenset*) contenant exclusivement des chaînes. Tous les éléments sont écrits de façon concaténée. Tout comme `.write`, elle n'envoie pas non plus de newline. Si l'on veut des sauts de ligne après chaque élément, le caractère `'\n'` devrait être inclu à la fin de chaque élément. Si ce n'est pas le cas, on peut utiliser `print` (voir ci-après).

C) La fonction `print(...)`, en lui passant l'argument `file = fd`, peut aussi être utilisée pour écrire dans un fichier ! Elle se comporte de façon standard, donc envoie par défaut un newline après l'écriture, à moins d'ajouter le paramètre `end = ''` (chaîne vide).

On montre ci-contre comment on peut simplifier la technique **B)** en faisant usage de l'instruction `with` qui nous dispense de fermer le fichier. Mais on pourrait simplifier de façon analogue les techniques **A)** et **C)** !

```
1 liste = ['1ère ligne', '2e ligne', 'fin']
2   # ici éléments non terminés par \n
3
4 fd = open('ecrit.txt', 'w')
5 for ligne in liste:
6     fd.write(ligne + '\n') # on ajoute les \n
7 fd.close()
```

```
21 liste = ['1ère ligne\n', '2e ligne\n', 'fin\n']
22   # les éléments doivent se terminer
23   # par \n si l'on veut des sauts de ligne
24
25 fd = open('ecrit.txt', 'w')
26 fd.writelines(liste)
27 fd.close()
```

```
41 liste = ['1ère ligne', '2e ligne', 'fin']
42   # ici éléments non terminés par \n
43
44 fd = open('ecrit.txt', 'w')
45 for ligne in liste:
46     print(ligne, file=fd)
47 fd.close()
```

```
61 with open('ecrit.txt', 'w') as fd:
62     fd.writelines(liste)
```

2.6.4 Autres méthodes sur les fichiers

Les méthodes de fichiers suivantes sont encore intéressantes :

- `.tell()` : indique la position courante dans le fichier, exprimée en nombre de caractères depuis le début du fichier
- `.seek(nb, depuis)` : déplace le curseur (position courante) dans le fichier :
 - si `depuis` est ignoré ou vaut `0` : on se déplace de façon absolue au `nb`ème caractère du fichier
 - si `depuis` vaut `1` : on se déplace de `nb` caractères par rapport à la position courante, en avant ou en arrière selon que ce nombre est positif ou négatif
 - si `depuis` vaut `2` : on se déplace de `nb` caractères par rapport à la fin du fichier (nombre qui doit être négatif)

❗ Sous Python 3.2 `.seek` semble bugué lorsque l'on utilise l'option `depuis = 1` ou `2` sur des fichiers ouverts en mode texte. Le problème ne se pose pas si on les ouvre en binaire, mais on ne manipule alors plus des caractères Unicode mais des bytes.

```
1 fd = open('essai.txt', 'r')
2 fd.tell()      # => 0 (position à l'ouverture)
3
4 fd.seek(6)     # aller au 6e car. depuis déb. fichier
5 fd.read(5)     # => lecture 'ligne'
6 fd.tell()     # => 11 (6 + 5)
7
8 fd.seek(0,2)   # aller à la fin du fichier
9 fd.tell()     # => 25 (nb de car. total fichier)
10
11 fd.seek(-4,2)  # aller au 4e car avant fin fichier
12 fd.read(3)    # => lecture 'fin'
13
14 fd.seek(0)     # "rewind" au début du fichier
15 fd.tell()     # => 0
16
17 fd.close()
```

2.6.5 Sérialisation et stockage d'objets avec le module pickle

Si vous souhaitez sérialiser des objets et les stocker sur fichier, penchez-vous sur le module standard `pickle`.



Rédaction de ce chapitre en cours, merci de patienter !

2.7 Quelques fonctions built-in

Les fonctions built-in sont des fonctions de base définies dans le module `builtins` (anciennement nommé `__builtin__` sous Python v2). Ce module étant préchargé (contrairement aux autres modules de la [bibliothèque standard](#)), elles sont directement utilisables sans devoir importer de modules. Nous nous contentons de présenter ici les plus courantes qui n'auraient pas été présentées plus haut. Pour la liste complète, voyez la [documentation Python](#).

- `resultat = eval(expression)` : évaluation, par l'interpréteur Python et dans l'environnement courant, de l'expression spécifiée (chaîne) et retourne le resultat
- `exec(instructions)` : évalue et exécute les instructions spécifiées (une chaîne)

```
1 a, b = 2, 3
2 eval('a * b')      # => retourne 6
3
4 exec("print(a, '*', b, '=>', end='') ; print(a * b)")
5                  # => affiche:  2 * 3 => 6
```



La suite de ce chapitre est en cours de rédaction, merci de patienter !

2.8 Quelques modules de la librairie standard

La librairie standard Python est constituée des modules faisant partie de la distribution de base Python. Pour les utiliser, il est nécessaire de les importer avec l'instruction `import` (voir [plus haut](#)). Les modules standards principaux sont les suivants :

- `os` : interface portable au système d'exploitation (système de fichier, processus...)
- `glob`, `shutil`, `filecmp` : expansion de noms de fichiers, copie et archivage de fichiers et arborescences, comparaison de fichiers et répertoires
- `fileinput`, `csv`, `json`, `configparser` : lecture/écriture de fichiers spécifiques
- `tempfile` : gestion de fichiers et dossiers temporaires
- `sys`, `platform` : interface à l'environnement de l'interpréteur Python et à la plateforme d'exécution
- `re` : fonctionnalités relatives au pattern-matching par expressions régulières
- `math`, `cmath` : interface aux outils de la librairie mathématique standard C, opérations sur nombres complexes
- `random` : générateurs de nombres aléatoires
- `time` : interface aux fonctions temporelles du système
- `datetime`, `calendar` : manipulation de dates et calendriers
- `argparse` (et plus anciens modules `optparse` et `getopt`) : processing de commandes, arguments et options
- `base64`, `uu`, `binhex`, `binascii`, `quopri` : encodage et décodage
- `gzip`, `zipfile`, `tarfile`, `zlib`, `bz2`, `lzma` : gestion d'archives compressées
- `pickle`, `copyreg`, `dbm`, `shelve`, `marshal` : gestion d'objets persistants
- `tkinter` : construction d'interfaces utilisateur graphiques (GUI) basé sur la librairie portable Tk
- `sqlite3` : API au système de base de données SQLite ; les interfaces vers d'autres SGBD sont à installer séparément
- `urllib.request`, `urllib.parse`, `http.client`, `http.server`, `cgi`, `html`, `xml`, `xmlrpc` : modules web
- `email`, `imaplib`, `poplib`, `smtplib` : modules email
- `telnetlib`, `ftplib` : autres services Internet
- `subprocess`, `multiprocessing`, `_thread`, `threading`, `queue` : gestion de processus et threads
- `socket`, `socketserver`, `ssl`, `xdrlib`, `select` : communication réseau de bas niveau TCP & UDP
- `signal` : gestion de timers et signaux

⚠ On ne présentera ci-après que quelques-uns de ces modules, tout en nous limitant aux fonctions les plus utiles. Pour davantage d'information, souvenez-vous qu'une fois un module importé, vous pouvez accéder à sa documentation complète (description, liste des classes, fonctions et données) avec `help(module)`. En frappant `help(module.fonction)`, vous obtiendrez directement l'aide sur la fonction spécifiée. Sous IPython finalement, en frappant `module.<tab>` vous faites apparaître toutes les fonctions relatives à un module, et respectivement avec `objet.<tab>` les méthodes relatives à un objet ou classe.

2.8.1 os

Le module `os` offre une interface d'accès aux services du système d'exploitation, et ceci de façon uniforme et portable (c-à-d. fonctionnant sur tous les OS). Il contient un sous-module `os.path` donnant accès aux objets du système de fichiers (répertoires, fichiers, liens...).

Nous ne présentons ci-après que les méthodes les plus courantes. Sachez que `os` implémente en outre des fonctions de gestion de processus.

Fichiers et répertoires (ci-après, chemin représente, selon la méthode, un nom de fichier ou de répertoire, précédé ou non d'un chemin relatif ou absolu) :

- `.getcwd()` : chemin du répertoire courant
- `.chdir(chemin)` : changement de répertoire courant
- `.listdir(chemin)` : liste contenant toutes les entrées (fichiers et répertoires) du répertoire chemin
- `.rename(ancien,nouveau)` : renomme ancien fichier ou répertoire en nouveau
- `.mkdir(chemin)` : crée répertoire chemin
- `.makedirs(chemin)` : idem en créant répertoires parents s'ils n'existent pas
- `.rmdir(chemin)` : détruit répertoire chemin (si vide)
- `.remove(chemin)` ou `.unlink(chemin)` : détruit fichier chemin
- `.chmod(chemin,mode)` : changement de permission
- `.chown(chemin,uid,gid)` : changement de propriétaire
- `.walk(chemin)` : parcourt récursivement l'arborescence chemin, retournant pour chaque répertoire un tuple à 3 éléments: chemin, liste des sous-répertoires, liste des fichiers
- `.path.abspath(chemin)` : retourne le chemin absolu correspondant à chemin relatif
- `.path.dirname(chemin)` : retourne le chemin contenu dans chemin
- `.path.basename(chemin)` : retourne le nom de fichier ou de répertoire terminant le chemin
- `.path.split(chemin)` : découpe chemin en un tuple composé des 2 éléments ci-dessus : `.path.dirname(...)` et `.path.basename(...)`
- `.path.splitext(chemin)` : retourne l'extension contenue dans chemin
- `.path.splitdrive(chemin)` (Windows seulement) : découpe chemin en une tuple contenant le drive: et le reste de chemin
- `.path.join(ch1,ch2,...)` : concatène intelligemment les chemins ch1, ch2...
- `.path.exists(chemin)` : teste si chemin existe (fichier ou répertoire)
- `.path.isfile(chemin)`, `.path.isdir(chemin)` : teste respectivement si chemin est un fichier ou un répertoire
- `.path.getsize(chemin)` : retourne la taille en octets de chemin (fichier ou répertoire)

```
1 import os
2
3 help(os)          # => documentation du module os
4 os.<tab>           # (s/IPython) => liste des fonctions
5 help(os.getcwd)   # => aide sur fct spécifiée
6
7 os.getcwd()       # => p.ex. '/home/bonjour'
8 os.curdir         # => '.'
9 os.listdir()      # => liste fichiers répertoire courant
10 os.mkdir('tes')   # création sous-répertoire
11 os.rename('tes', 'test') # renommage
12 'test' in os.listdir() # => True
13 os.chdir('test')  # changement répertoire courant
14 open('essai.txt','a').close() # comme 'touch' Unix
15 # => crée fichier s'il n'existe pas et le referme
16 os.path.exists('essai.txt') # => True
17 os.path.isfile('essai.txt') # fichier ? => True
18 fich = os.path.abspath('essai.txt')
19 # => /home/bonjour/test/essai.txt
20 os.path.dirname(fich) # path => /home/bonjour/test
21 nomf = os.path.basename(fich) # fichier => essai.txt
22 os.path.splitext(nomf) # => ('essai', '.txt')
23 os.remove('essai.txt') # destruction fichier
24 os.chdir('..')       # on remonte d'un niveau
25 os.path.isdir('test') # répertoire ? => True
26 os.rmdir('test')     # destruction répertoire
27
28 for dirpath, dirnames, filenames in os.walk(os.curdir):
29     for fp in filenames:
30         print(os.path.join(os.path.abspath(dirpath),fp))
31 # => affiche récursivement tous les fichiers
32 #   de l'arborescence avec leur path absolu
33
34 os.environ         # => dict. avec toutes var. env.
35 os.environ['USER']  # => 'bonjour'
36 os.environ['HOME']  # => '/home/bonjour'
37
38 os.system('ls -a')  # passe la commande au shell...
39 # et affiche résultat sur la sortie standard
40
41 list_dir = os.popen('ls -l').read()
42 # idem mais récupère résultat sur chaîne
43 print(list_dir)
```

Environnement du système d'exploitation :

- `.environ` : dictionnaire contenant toutes les variables d'environnement
- `.uname()` : tuple renseignant sur : OS, nom machine, release, version, architecture

Exécution de **commandes** shell et **programmes** externes :

- `.system(comande)` : envoie (spawn) la commande à l'interpréteur du système d'exploitation (shell), et retourne le statut de terminaison (`0` si ok) ; le résultat s'affiche sur la sortie standard ; sous Unix, on ajoutera `&` à la fin de la commande si elle doit s'exécuter en background
- `chaine = os.popen(comande).read()` : passe aussi la commande au shell, mais récupère son résultat sur `chaine`
- `.startfile(chemin)` (Windows seulement) : ouvre le fichier `chemin` comme si l'on double-cliquait sur celui-ci dans l'explorateur de fichier ; équivalent à `.system('start chemin')`
- voir aussi `.exec(...)` et `.spawnv(...)` , plus complexes à utiliser mais offrant davantage de possibilités

2.8.2 glob

Le module `glob` a pour seul objectif de récupérer, pour le répertoire courant ou spécifié, la liste des fichiers dont le nom correspond à certains critères (pattern matching) définis au moyen des caractères suivants :

- `?` : correspond à 1 caractère quelconque
- `*` : correspond à 0 à N caractères quelconques
- `[séquence]` : correspond à l'un des caractères de la séquence spécifiée, p.ex. `[abc]` , `[a-z]` , `[0-9]`
- `[!séquence]` : correspond à un caractères ne faisant pas partie de la séquence spécifiée

La fonction `.glob(pattern)` retourne une liste de noms de fichiers/répertoires, et la fonction `.iglob(pattern)` retourne un itérateur.

Si la pattern inclu un chemin, la recherche s'effectue dans le répertoire correspondant, et les noms de fichiers retournés inclueront ce chemin.

Soit le répertoire contenant les fichiers suivants :

ess1.dat	ess2.dat	fil.txt	fi2.txt
fich1.txt	z2.txt		

Quelques exemples de matching :

```
1 import glob
2
3 glob.glob('*.dat')      # => ['ess1.dat', 'ess2.dat']
4 glob.glob('f??.*')     # => ['fil.txt', 'fi2.txt']
5 glob.glob('[ef]*2.*')  # => ['fi2.txt', 'ess2.dat']
6 glob.glob('[!f]*.*')   # => ['ess1.dat', 'ess2.dat']
7
8 glob.glob('../r*.pdf') # => retournerait p.ex.
9     # ['../r1.pdf', '../r2.pdf'] , donc avec chemin !
10
11 # Si l'on fait un glob dans un autre répertoire et
12 # qu'on veut récupérer les noms de fichiers sans leurs
13 # chemins, on peut faire une compréhension list
14 # utilisant os.path.basename pour enlever le chemin :
15 [os.path.basename(f) for f in glob.glob('path/*.txt')]
```

2.8.3 shutil

Le module `shutil` , abréviation de shell utilities, offre des fonctions intéressantes de copie et archivage de fichiers et arborescences de répertoires.

- `.copy(fichsource, destination)` : copie le fichier `fichsource` sous le nom `destination` ; si `destination` est un répertoire, copie le fichier dans celui-ci sous le même nom que le fichier source
- `.copyfile(fichsource, fichdestination)` : copie le fichier `fichsource` sous le nom `fichdestination` ; retourne une erreur si `fichdestination` est un répertoire
- `.copymode(souce, destination)` : copie les permissions (pas les données) de `souce` sur `destination` (qui doit donc exister)
- `.move(souce, destination)` : déplace récursivement toute l'arborescence `souce` vers `destination` (comme la commande `mv` Unix)
- `.copytree(repsource, repdestination)` : copie toute l'arborescence de répertoires/fichiers de racine `repsource` vers `repdestination` ; le répertoire `repdestination` ne doit pas pré-exister
- `.rmtree(repertoire)` : détruit toute l'arborescence `repertoire` (i.e. détruite récursivement tous ses fichiers et répertoires)
- `.make_archive(...)` et `.unpack_archive(...)` : crée et déballe une archive de type ZIP, TAR, BZTAR ou GZTAR...

2.8.4 filecmp

Le module `filecmp` permet de tester si des fichiers sont identiques, par comparaison individuelle ou par contenu de répertoire :

- `.cmp(fichier1, fichier2, listefichiers)` : compare les **fichiers** `fichier1` et `fichier2` ; retourne `True` s'ils sont identiques, et `False` sinon
- `.cmpfiles(dir1, dir2)` : compare, dans les 2 **répertoires** `dir1` et `dir2` (sans descendre dans éventuels sous-répertoires), les fichiers de noms définis dans `listefichiers` ; retourne une liste comprenant 3 sous-listes : la première avec les noms des fichiers identiques, la seconde avec les noms des fichiers différents, la troisième avec les noms des fichiers présents seulement dans l'un ou l'autre des 2 répertoires

Si l'on est intéressé à afficher les différences de **contenu** entre des fichiers, on se tournera vers le module `difflib`.

```
1  # Comparaison de 2 fichiers
2
3  import filecmp
4
5  if filecmp.cmp('f1.dat', 'f2.dat'):
6      print('Fichiers identiques')
7  else:
8      print('Fichiers différents')
9
10
11 # Comparaison de 2 répertoires
12 #
13 # Soient 2 répertoires contenant les fichiers suivants
14 # - Répertoire :   dir1       dir2       Remarques :
15 # - Fichiers :    a.txt      a.txt      identiques
16 #                b.txt      b.txt      différents
17 #                c.txt      --         unique
18 #                --        d.txt      unique
19
20 import filecmp, os
21
22 set1 = { fn for fn in os.listdir(dir1)
23          if os.path.isfile(os.path.join(dir1,fn)) }
24 # set contenant le nom de tous les fichiers de dir1
25 # (if exclut les noms des éventuels sous-répertoires)
26 set2 = { fn for fn in os.listdir(dir2)
27          if os.path.isfile(os.path.join(dir2,fn)) }
28 # set contenant le nom de tous les fichiers de dir2
29
30 res = filecmp.cmpfiles(dir1, dir2, set1 | set2)
31 # on considère les fichiers de noms correspondant
32 # à l'union des 2 sets set1 et set2 =>
33 # res[0] = ['a.txt'] : liste fich. identiques
34 # res[1] = ['b.txt'] : liste fich. différents
35 # res[2] = ['c.txt', 'd.txt'] : fichiers uniques
```

2.8.5 sys, platform

Le module `sys` fournit une interface à l'environnement de l'interpréteur Python.

- `.argv` : liste contenant le nom du script et les arguments qui lui ont été passés (décrit précédemment)
- `.exit(n)` ou `.exit(chaine)` : termine le processus Python avec le status `n` (par défaut `0`) ou en affichant `chaine`
- `.stdin` : canal de standard input (pré-ouvert, initialisé à la valeur `__stdin__`), utilisé par `input` ... mais pouvant être utilisé par un objet avec `read`
- `.stdout` : canal de standard output (pré-ouvert, initialisé à la valeur `__stdout__`), utilisé par `print` ... mais pouvant être utilisé par un objet avec `write`
- `.stderr` : canal de standard error (pré-ouvert, initialisé à la valeur `__stderr__`), utilisé pour l'affichage des erreurs... mais pouvant être utilisé par un objet avec `write`

```

1 import sys
2
3 sys.platform # => 'linux2'
4 sys.version
5 # => '3.2.3 (Sep 25 2013, 18:22:43) \n[GCC 4.6.3]'
6 sys.executable # => '/usr/bin/python'
7 sys.path # => path de recherche des modules
8 sys.getrecursionlimit() # => 1000
9 sys.dont_write_bytecode # => False
10
11
12 # Ex. d'utilisation .stdin .stdout .stderr
13 chaine = sys.stdin.readline()
14 sys.stdout.write(chaine)
15 sys.stderr.write('erreur: ...')
16
17
18 # Pour sortir d'un script
19 sys.exit('Sortie du script')

```

Les méthodes et attributs ci-dessous sont beaucoup plus techniques et probablement moins utiles pour vous :

- `.platform` : type de la plateforme sur laquelle s'exécute le programme (linux2, win32, darwin, cygwin...)
- `.version` : version de l'interpréteur Python utilisé
- `.executable` : chemin de l'interpréteur Python utilisé
- `.path` : liste des répertoires dans lesquels s'effectue la recherche de modules lors d'un `import` ; incorpore les répertoires définis par la variable d'environnement `PYTHONPATH`
- `.builtin_module_names` : tuple indiquant les modules C compilés dans l'interpréteur Python
- `.modules` : dictionnaire des modules chargés
- `.ps1` (pas valable sous IPython) : prompt de l'interpréteur Python (par défaut `>>>`), peut être changé avec `sys.ps1=chaine`
- `.ps2` (pas valable sous IPython) : prompt secondaire pour les lignes de continuation (par défaut `...`), pouvant également être changé
- `.setrecursionlimit(nombre)` : pour modifier la profondeur max de récursion (par défaut 1000, que l'on peut voir avec `.getrecursionlimit()`)
- `.dont_write_bytecode` : par défaut à `False`, cette variable permet de demander de ne pas écrire de fichiers bytecode (`.pyc` ou `.pyo`) lors de l'importation de modules
- `.getsizeof(objet)` : taille mémoire d'un objet en octets

Le module `platform` fournit des informations assez techniques sur la plateforme matérielle sur laquelle s'exécute le programme.

- `.architecture()` et `.machine()` : architecture sur laquelle tourne le programme
- `.node()` : nom de la machine
- `.platform()`, `.system()`, `.version()` et `.release()` : OS et version
- `.processor()` : processeur
- `.uname()` : les différentes infos ci-dessus sous forme de tuple
- `.win32_ver()`, `.mac_ver()`, `.dist()` : spécifique aux OS respectifs
- `.python_version()` et `.python_build()` : version Python

```

21 import platform
22
23 platform.uname()    # => OS, nom de machine...
24 platform.system()   # => 'Linux'
25 platform.release()  # => '3.2.0-56-generic'
26 platform.dist()     # => ('Ubuntu', '12.04', 'precise')
27 platform.architecture() # => ('64bit', 'ELF')
28 platform.python_version() # => '3.2.3'

```

2.8.6 math, random

Le module `math` donne accès aux fonctions de la librairie mathématique standard C. Pour le support des nombres complexes, voir les fonctions de même nom du module `cmath`. Les différentes catégories de fonctions sont :

- constantes : `.pi`, `.e`
- arrondi, troncature : `.ceil`, `.floor`, `.trunc`
- puissance, racine : `.pow`, `.sqrt`
- trigonométriques : `.cos`, `.sin`, `.tan`, `.acos`, `.asin`, `.atan`, `.atan2`, `.cosh`, `.sinh`, `.tanh`, `.acosh`, `.asinh`, `.atanh`
- logarithmiques, exponentielles : `.log`, `.log10`, `.log1p`, `.ldexp`, `.exp`, `.expm1`, `.frexp`
- conversion d'angles : `.degrees`, `.radians`
- gamma : `.gamma`, `.lgamma`
- autres : `.fabs` (val abs), `.factorial`, `.fsum` (somme iterable), `.hypot` (distance euclidienne), `.fmod` (modulo), `.modf`, `.erf`, `.erfc`, `.copysign`
- test : `.isfinite`, `.isinf`, `.isnan`

Le module `random` fournit des générateurs aléatoires.

Citons notamment : `.random()`, `.uniform(debut, fin)`, `.randint(debut, fin)`, `.randrange(debut, fin, pas)`, `.gauss(mu, sigma)`, etc...

2.8.7 time

Le module `time` offre une interface à l'horloge système (date/heure courante, pause...) ainsi que des fonctions de conversion de format.

⚠ Contrairement au module `datetime` (qui gère les dates sous forme d'**objets** depuis le début de notre ère), celui-ci gère en interne les dates/heures sous forme de **réel flottant** (dates numériques) qui expriment le temps écoulé en **secondes** depuis l'origine ici définie au 1.1.1970 à 00:00 UTC (GMT) (dénommée *epoch* sous Linux). Ce module ne permet donc pas de manipuler des dates antérieures à cette origine, mais il se prête mieux que `datetime` au stockage de séries chronologiques (sous forme de liste de réels).

Les dates/heures peuvent se présenter sous différentes formes :

- `datetime` : date numérique (selon description ci-dessus)
- `datestruct` : objet de type `time.struct_time` offrant les attributs suivants :
 - `date`: `tm_year` (année), `tm_mon` (mois), `tm_mday` (jour), `tm_wday` (jour semaine: 0=lundi...)
 - `heure`: `tm_hour` (heure), `tm_min` (minute), `tm_sec` (seconde)
 - `autres`: `tm_yday` (numéro du jour dans l'année), `tm_isdst` (daylight saving time: 0=heure hiver, 1=heure été)
- `datetuple` : date sous forme de tuple de 9 entiers (`année, mois, jour, heure, min, sec, 0, 0, -1`)
- `datechaîne` : date convertie en chaîne avec `.strftime(format, date)`
- les différents objets du module `datetime` (voir chapitre suivant)

Date/heure courante et conversions :

- `datetime=time.time()` : retourne la date/heure **courante** sous forme numérique
- `datetime=time.mktime(datetuple ou datestruct)` : conversion numérique de la date/heure fournie au format `datetuple` ou `datestruct`
- `datestruct=time.localtime(datetime)` : conversion de la date/heure numérique `datetime` (sans paramètre: de la date/heure courante) en heure **locale** au format `datestruct` ;
la fonction `time.gmtime(datetime)` ferait de même mais en heure UTC
- `datechaîne=time.strftime(format, datetuple ou datestruct)` : conversion en **chaîne**, à l'aide du format spécifié (voir codes ci-dessous), de la date/heure fournie au format `datetuple` ou `datestruct`
- conversion en chaîne au format 'Mon Nov 18 00:27:35 2013' :
`time.ctime(datetime)` : depuis date/heure `datetime`
`time.asctime(datestruct)` : depuis date/heure `datestruct`

Autres fonctions :

- `time.sleep(sec)` : effectue une **pause** de `sec` secondes
- `time.clock()` : sous Linux, retourne le temps CPU consommé ; sous Windows, le temps écoulé ; voir aussi la fonction `os.times()`
- `time.timezone` : retourne le décalage horaire "heure UTC" moins "heure locale", en secondes
- `time.tzname` : retourne le nom de l'heure locale

```

1 import time
2
3 # Instant présent
4 nownum = time.time() # => date/heure courante (nb. réel)
5 nowstruct = time.localtime(nownum)
6 type(nowstruct)      # => objet de type time.struct_time
7
8 # Définition d'une date/heure, attributs
9 dhnum = time.mktime( (2013,11,18,0,27,35,0,0,-1) )
10
11 dhstruct = time.localtime(dhnum) # => conversion objet
12 dhstruct.tm_year   # => 2013
13 dhstruct.tm_mon    # => 11
14 dhstruct.tm_mday   # => 18
15 dhstruct.tm_hour   # => 0
16 dhstruct.tm_min    # => 27
17 dhstruct.tm_sec    # => 35
18 dhstruct.tm_wday   # => 0 (lundi)
19 dhstruct.tm_yday   # => 322 ème jour de l'année
20 dhstruct.tm_isdst  # => 0 (heure d'hiver)
21
22 # Formatage
23 time.strftime('It\'s %A %d %B %Y at %H:%M:%S', dhstruct)
24 # => "It's Monday 18 November 2013 at 00:27:35"
25 time.strftime('%je jour de %Y, %Uème semaine', dhstruct)
26 # => '322e jour de 2013, 46ème semaine'
27 time.ctime(dhnum)   # => 'Mon Nov 18 00:27:35 2013'
28 time.asctime(dhstruct) # => 'Mon Nov 18 00:27:35 2013'
29
30 # Autres
31 time.sleep(2.5)     # => effectue pause 2.5 sec
32 time.timezone       # => -3600
33 time.tzname         # => ('CET', 'CEST')

```

En guise d'illustration de ce que l'on vient de voir, construisons une série chronologique affichant ce qui suit (dates/heures toutes les 6 heures entre 2 dates) :

```
20-11-2013 00:00:00
20-11-2013 06:00:00
20-11-2013 12:00:00
20-11-2013 18:00:00
21-11-2013 00:00:00
21-11-2013 06:00:00
21-11-2013 12:00:00
21-11-2013 18:00:00
22-11-2013 00:00:00
```

```
41 import time
42
43 # Série chronologique de 'deb' -> 'fin' avec pas 'step'
44 deb = time.mktime((2013,11,20,0,0,0,0,0,-1)) # date num.
45 fin = time.mktime((2013,11,22,0,0,0,0,0,-1)) # date num.
46 step = 6*60*60 # 6 heures exprimées en secondes
47
48 datenum = deb
49 while datenum <= fin:
50     jourstruct = time.localtime(datenum)
51     print(time.strftime('%d-%m-%Y %H:%M:%S',jourstruct))
52     datenum += step
```

Formatage de dates et heures avec strftime

La fonction `.strftime` du module `time` et la méthode du même nom du module `datetime` admettent notamment les codes de **formatage** suivants :

- année : `%Y` (à 4 chiffres: 2013...), `%y` (à 2 chiffre: 13...)
- mois : `%m` (numéro), `%B` (nom: January...), `%b` (nom abrégé: Jan...)
- semaine : `%U` (numéro de la semaine dans l'année)
- jour : `%d` (numéro), `%j` (numéro du jour dans l'année)
- jour de la semaine : `%A` (nom: Monday...), `%a` (nom abrégé: Mon...), `%w` (numéro: 0=dimanche...)
- heure : `%H` (heure), `%M` (minute), `%S` (seconde), `%f` (microseconde)

2.8.8 datetime

Le module `datetime` est totalement orienté-objet (contrairement au module `time`). Il implémente un certain nombre de classes (avec attributs et méthodes associés) permettant de manipuler : dates avec heures (classe `.datetime`), dates seules (classe `.date`), heures seules (classe `.time`), différences de temps (classe `.timedelta`). Les dates se réfèrent au calendrier Grégorien et peuvent être manipulées dans la plage d'années 0001 à 9999 de notre ère.

A) La classe `.datetime` et ses objets `dateheure` :

- ses attributs sont : `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, `tzinfo` (time zone)
- `dateheure = datetime.datetime.today()` ou `dateheure = datetime.datetime.now(tz=timezone)` : retourne la date/heure **courante locale** sur l'objet `dateheure` (l'argument `timezone` étant optionnel) ; `utcnow()` retournerait l'heure UTC
- `dateheure = datetime.datetime(year, month, day, hour=0, minute=0, second=0)` : retourne la date/heure spécifiée sur l'objet `dateheure`
- `dateheure = dateheure.replace(champ=val...)` : modifie l'objet `dateheure` en remplaçant la val du champ spécifié (le nom des champs correspondant aux attributs vus plus haut, donc : `year`, `month`, `day`, etc...)
- `datestruct = dateheure.timetuple()` : retourne date/heure sur un objet de type `datestruct` (voir module `time`)
- `print(datestruct)` : affiche la date/heure au format '2013-11-19 17:32:36.780077'
- `datechaine = dateheure.strftime(format)` : conversion en **chaîne** à l'aide du format spécifié (voir codes de formatage au chapitre précédent) ; la fonction inverse serait `dateheure.strptime(datechaine, format)`
- `dateheure.ctime()` : retourne la date sous forme de chaîne au format 'Mon Nov 18 00:27:35 2013'
- `dateheure.isoformat()` : retourne la date sous forme de chaîne au format ISO '2013-11-18T00:27:35'
- `dateheure.date()` : retourne la date sur un objet de type `.date`
- `dateheure.time()` : retourne l'heure sur un objet de type `.time`

On peut finalement comparer des dates/heures en comparant des objets

`.datetime` ou `.date` avec les opérateurs `<` et `>`

```
1 import datetime as dt # nom abrégé 'dt' pour simplifier
2
3 # Instant présent
4 now = dt.datetime.today() # => (2013, 11, 19, 17, 32, 36)
5 type(now) # => objet de type datetime.datetime
6 print(now) # => affiche: 2013-11-19 17:32:36.780077
7
8 # Définition d'une date/heure, attributs
9 dh = dt.datetime(2013,11,18,0,27,35)
10 dh.year # => 2013
11 dh.month # => 11
12 dh.day # => 18
13 dh.hour # => 0
14 dh.minute # => 27
15 dh.second # => 35
16
17 # Formatage
18 dh.strftime('It\'s %A %d %B %Y at %H:%M:%S')
19 # => "It's Monday 18 November 2013 at 00:27:35"
20 dh.strftime('%jème jour de %Y, dans la %Uème semaine')
21 # => '322ème jour de 2013, dans la 46ème semaine'
22 dh.ctime() # => 'Mon Nov 18 00:27:35 2013'
23 dh.isoformat() # => '2013-11-18T00:27:35'
24
25 # Manipulations
26 now > dh # => True
27 delta = now - dh # => (1, 61501) : jours et secs
28 type(delta) # => datetime.timedelta
29 dh - now # => (-2, 24899)
30 # i.e. reculer de 2 jours et avancer de 24899 sec.
31
32 # Conversions de types
33 dh.date() # => objet date (2013, 11, 18)
34 dh.time() # => objet time (0, 27, 35)
```

B) La classe `.date` n'est qu'une simplification de la classe `.datetime` en ce sens qu'elle ne contient que la date (sans heure) :

- ses attributs sont donc uniquement : `year`, `month`, `day`
- `date=datetime.date.today()` : retourne la date *courante locale* sur l'objet `date`
- `date=datetime.date(year, month, day)` : retourne la date spécifiée sur l'objet `date`

Pour le reste, les méthodes de l'objet `.datetime` vues plus haut s'appliquent également.

```
41 import datetime as dt # nom abrégé 'dt' pour simplifier
42
43 # Jour courant
44 auj = dt.date.today() # => objet (2013, 11, 19)
45 type(auj)             # => objet de type datetime.date
46 print(auj)            # => affiche: 2013-11-19
47
48 # Attributs
49 auj.year               # => 2013
50 auj.month              # => 11
51 auj.day                # => 19
52
53 # Manipulations
54 proch_anni = dt.date(auj.year, 2, 8)
55 if proch_anni < auj:
56     proch_anni = proch_anni.replace(year=auj.year+1)
57 nb_jours = proch_anni - auj
58 print(proch_anni.strftime('Next birthday: %A %d %B %Y'))
59 print('In: %d days' % (nb_jours.days) ) # => affiche :
60     # Next birthday: Saturday 08 February 2014
61     # In: 81 days
```

C) La classe `.time` est également une simplification de la classe `.datetime` en ce sens qu'elle ne contient que l'heure (sans date) et que ses méthodes sont très limitées :

- ses attributs sont : `hour`, `minute`, `second`, `microsecond`, `tzinfo` (time zone)
- `heure=datetime.time(hour, minute, second)` : retourne l'heure spécifiée sur l'objet `heure`

```
71 import datetime as dt # nom abrégé 'dt' pour simplifier
72
73 # Définition d'heures
74 heure = dt.time(9,45,10) # => objet (9, 45, 10)
75 type(heure)             # => objet de type datetime.time
76 print(heure)            # => affiche: 09:45:10
77
78 # Attributs
79 heure.hour              # => 9
80 heure.minute            # => 45
81 heure.second            # => 10
82 heure.microsecond       # => 0
```

D) La classe `.timedelta` permet d'exprimer une différence de temps entre des objets `.datetime` ou `.date`

- ses attributs sont : `days`, `seconds`, `microseconds`
- la soustraction d'objets `.datetime` ou `.date` retourne un objet `.timedelta`, et l'on peut aussi ajouter ou soustraire un `.timedelta` à une `.datetime` ou `.date`
- pour définir manuellement un objet `.timedelta` on utilise :
`delta = datetime.timedelta(days=val, seconds=val, microseconds=val, milliseconds=val, minutes=val, hours=val, weeks=val)` chaque argument étant optionnel, les val (entier ou flottant) par défaut étant 0
- `secs=delta.total_seconds()` : retourne un réel secs exprimant le nombre de secondes de l'objet delta

Nous reprenons ci-contre l'idée de la série chronologique du chapitre précédent (dates/heures toutes les 6 heures entre 2 dates), en l'implémentant ici avec les objets `.datetime` et `.timedelta`. Elle affiche la série ci-dessous :

```
20-11-2013 00:00:00
20-11-2013 06:00:00
20-11-2013 12:00:00
20-11-2013 18:00:00
21-11-2013 00:00:00
21-11-2013 06:00:00
21-11-2013 12:00:00
21-11-2013 18:00:00
22-11-2013 00:00:00
```

2.8.9 calendar

Le module orienté-objet `calendar` permet de manipuler des calendriers, un peu à la façon de la commande `cal` Unix.

```
91 import datetime as dt # nom abrégé 'dt' pour simplifier
92
93 # Définition de dates/heures
94 dh0 = dt.datetime(2013,11,20) # le 20.11.2013 à 0:00:00
95 dh1 = dt.datetime(2013,11,21,2,15,30,444)
96 # le 21.11.2013 à 2:15:30 et 444 microsec
97
98 # Calcul de timedelta, attributs
99 delta = dh1 - dh0 # => (1, 8130, 444) : jour,sec,micro
100 # mais: dh0 -dh1 => (-2, 78269, 999556) , donc
101 # reculer de 2 jours et avancer de 78269 sec...
102 type(delta) # => objet de type datetime.timedelta
103 print(delta) # => affiche: 1 day, 2:15:30.000444
104
105 # Attributs et méthodes
106 delta.days # => 1
107 delta.seconds # => 8130
108 delta.microseconds # => 444
109 delta.total_seconds() # 94530.000444 (secondes)
```

```
121 import datetime as dt # nom abrégé 'dt' pour simplifier
122
123 # Série chronologique
124 deb = dt.datetime(2013,11,20) # objet datetime
125 fin = dt.datetime(2013,11,22) # objet datetime
126 step = dt.timedelta(hours=6) # objet timedelta
127
128 dateheure = deb
129 while dateheure <= fin:
130     print(dateheure.strftime('%d-%m-%Y %H:%M:%S'))
131     dateheure += step
```

Quelques fonctions :

- `.weekday(annee,mois,jour)` : retourne le numéro de jour dans la semaine (0=lundi ... 6=dimanche) de la date spécifiée
- `.monthcalendar(annee,mois)` : retourne une matrice (c-à-d. liste imbriquée) des jours du mois spécifié
- `.monthrange(annee,mois)` : retourne un tuple avec : numéro de jour du premier jour du mois, nombre de jours dans le mois
- `.isleap(annee)` : teste si l'année est bissextile

Ci-dessous, `1erjour` définit le numéro de jour de la semaine auquel débute l'affichage des semaines. Si l'on omet ce paramètre, c'est par défaut 0=lundi.

Les classes `.TextCalendar` et `.LocaleTextCalendar` permettent de créer des calendriers mensuels/annuels sous forme **texte** :

- `.TextCalendar(1erjour).formatmonth(annee,mois,w=2,l=1)` : retourne une chaîne contenant le calendrier du mois spécifié ; par défaut les jours occupent w=2 caractères, et les semaine l=1 ligne ; la méthode `.prmonth(...)` fonctionne de la même façon mais affiche le calendrier du mois sur la console
- `.TextCalendar(1erjour).formatyear(annee,mois,w=2,l=1,m=3)` : retourne une chaîne contenant le calendrier de l'année spécifiée ; il y aura par défaut m=3 mois côte-à-côte ; la méthode `.pryear(...)` fonctionne de la même façon mais affiche le calendrier de l'année sur la console
- la classe `.LocaleTextCalendar(locale)` fonctionne de la même manière, sauf qu'on peut lui passer un paramètre définissant la locale (langue), par exemple : `locale='fr_FR.UTF-8'`

Les classes `.HTMLCalendar` et `.LocaleHTMLCalendar` permettent de créer des calendriers mensuels/annuels **HTML** :

- `.HTMLCalendar(1erjour).formatmonth(annee,mois)` : retourne une chaîne HTML contenant le calendrier du mois spécifié
- `.HTMLCalendar(1erjour).formatyear(annee,width=3)` : retourne une chaîne HTML contenant le calendrier de l'année spécifiée ; il y aura par défaut width=3 mois côte-à-côte
- la classe `.LocaleHTMLCalendar(locale)` fonctionne de la même manière, sauf qu'on peut lui passer un paramètre définissant la locale (langue), par exemple : `locale='fr_FR.UTF-8'`

Finalement, la classe `.Calendar` permet de créer des listes de dates ou créer des itérateurs. Nous vous renvoyons à la documentation pour les détails.

```
1 import calendar as cal
2
3 # Fonctions
4 cal.weekday(2013,11,19)      # => 1 (lundi)
5 cal.monthcalendar(2013,11)  # => liste/matrice :
6     # [[ 0,  0,  0,  0,  1,  2,  3],
7     #   [ 4,  5,  6,  7,  8,  9, 10],
8     #   [11, 12, 13, 14, 15, 16, 17],
9     #   [18, 19, 20, 21, 22, 23, 24],
10    #   [25, 26, 27, 28, 29, 30,  0]]
11 cal.monthrange(2013,11)     # => (4, 30)
12 cal.isleap(2012)            # => True
13
14 # Classes TextCalendar et LocaleTextCalendar
15 catext = cal.LocaleTextCalendar(locale='fr_FR.UTF-8')
16 catext.prmonth(2013,11)     # => affiche :
17     #      novembre 2013
18     #  lu ma me je ve sa di
19     #
20     #           1  2  3
21     #    4  5  6  7  8  9 10
22     #   11 12 13 14 15 16 17
23     #   18 19 20 21 22 23 24
24     #   25 26 27 28 29 30
25 catext.pryear(2013)        # => voir lien 1. ci-dessous
26
27 # Classes HTMLCalendar et LocaleHTMLCalendar
28 # (on écrit ci-dessous directement dans fich. HTML)
29 cahtml=cal.LocaleHTMLCalendar(locale='fr_FR.UTF-8')
30
31 with open('cal-nov-2013.html', 'w') as fd:
32     fd.write(cahtml.formatmonth(2013,11))
33     # => voir lien 2. ci-dessous
34
35 with open('cal-2013.html', 'w') as fd:
36     fd.write(cahtml.formatyear(2013))
37     # => voir lien 3. ci-dessous
38
```

Illustrations des exemples ci-dessus (il resterait bien entendu à styler les deux sorties HTML avec du CSS) :

1. [LocaleTextCalendar.pryear](#)
2. [LocaleHTMLCalendar.formatmonth](#)
3. [LocaleHTMLCalendar.formatyear](#)

2.8.10 *this*

Pour conclure sur une touche humoristico-philosophique cette présentation des modules les plus importants de la librairie standard, voici le module `this` qui n'a pas d'autre but que de résumer la philosophie Python. Il suffit de l'importer... et vous verrez alors apparaître le texte ci-dessous d'un gourou Python !

The Zen of Python, by Tim Peters :

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Flat is better than nested
- Sparse is better than dense
- Readability counts
- Special cases aren't special enough to break the rules - Although practicality beats purity
- Errors should never pass silently - Unless explicitly silenced
- In the face of ambiguity, refuse the temptation to guess
- There should be one –and preferably only one– obvious way to do it - Although that way may not be obvious at first unless you're Dutch
- Now is better than never - Although never is often better than right now
- If the implementation is hard to explain, it's a bad idea
- If the implementation is easy to explain, it may be a good idea
- Namespaces are one honking great idea – let's do more of those !

```
1 import this
2
3 # <= cela affiche le texte ci-contre !
```

Programmation objet avec Python

Voir le [support de cours](#) de Samuel Bancal.

Scientific Python

Voir les supports de cours de Samuel Bancal relatifs à l'usage des librairies suivantes :

- [NumPy](#)
- [Matplotlib](#)
- [Pandas](#)

Installation et utilisation de Python et outils associés

Voir [ce document](#)



Jean-Daniel Bonjour, puis *Samuel Bancal*, EPFL, ENAC-IT (2013-2020)