Université de Limoges | FACULTÉ DES SCIENCES ET TECHNIQUES

**Master 1 — UE Réseaux Avancés I**

# Mon'tit Python — P-F. Bonnefoi

*Version du 10 septembre 2012*



Table des matières

# 1 Pourquoi Python ?

Il est :

⋆ **portable**, disponible sous toutes les plate-formes (de Unix à Windows) ;

⋆ **simple**, avec une syntaxe claire, privilégiant la lisibilité, libérée de celle de C/C++ ;

⋆ **riche**. Il incorpore de nombreuses possibilités de langage
- ◇ tiré de la **programmation impérative** : *structure de contrôle, manipulation de nombres comme les flottants, doubles, complexe, de structures complexes comme les tableaux, les dictionnaires, etc.*
- ◇ *tiré des langages de script : accès au système, manipulation de processus, de l'arborescence fichier, d'expressions rationnelles, etc.*
- ◇ tiré de la **programmation fonctionnelle** : *les fonctions sont dites « fonction de première classe », car elles peuvent être fournies comme argument d'une autre fonction, il dispose aussi de lambda expression, de générateur etc.*
- ◇ tiré de la **programmation orienté objet** : *définition de classe, héritage multiple, introspection (consultation du type, des méthodes proposées), ajout/retrait dynamique de classes, de méthode, compilation dynamique de code, délégation ("duck typing"), passivation/activation, surcharge d'opérateurs, etc.*

# 1.1 Pourquoi Python ? *Ses caractéristiques*

Il est :

⋆ *dynamique* : il n'est pas nécessaire de déclarer le type d'une variable dans le source. Le type est associé lors de l'exécution du programme ;

⋆ *fortement typé* : les types sont toujours appliqués (un entier ne peut être considéré comme une chaîne sans conversion explicite, une variable possède un type lors de son affectation).

⋆ compilé/interprété à la manière de Java. Le source est compilé en bytecode (pouvant être sauvegardé) puis exécuté sur une machine virtuelle.

Il dispose d'une gestion automatique de la mémoire ("garbage collector").

Il dispose de nombreuses bibliothèques : interface graphique (`TkInter`), développement Web (le serveur d'application ZOPE, gestion de document avec Plone par exemple), inter-opérabilité avec des BDs, des middlewares ou intergiciels objets( SOAP/COM/CORBA/.NET), d'analyse réseau (SCAPY), manipulation d'XML, *etc*.
Il existe même des compilateurs vers C, CPython, vers la machine virtuelle Java (Jython), vers .NET (IronPython) !

Il est utilisé comme langage de script dans PaintShopPro, Blender3d, Autocad, Labview, *etc*.

# 1.2 Pourquoi Python ? *Ses usages*

▷ Il permet de faire du prototypage d'applications.

▷ C'est un langage « agile », adapté à l'eXtreme programming :
« Personnes et interaction plutôt que processus et outils »
« Logiciel fonctionnel plutôt que documentation complète »
« Collaboration avec le client plutôt que négociation de contrat »
« Réagir au changement plutôt que suivre un plan »
Intégrer de nombreux mécanismes de contrôle d'erreur (exception, assertion), de test (pour éviter les régressions, valider le code, ...).

▷ Et il permet de faire de la programmation réseaux !

**Dans le cadre du module Réseaux avancés I**

Les éléments combinés que sont : la gestion des expressions rationnelles, la programmation socket et l'utilisation de certaines classes d'objets nous permettrons de faire efficacement et rapidement des applications réseaux conforme à différents protocoles de communication.

Remarques

*La programmation objet ne sera pas obligatoire.*
*De même que l'utilisation de bibliothèques pour résoudre les problèmes de TPs est formellement déconseillée !*

# 2 Un programme Python

**Mode interactif**

Sur tout Unix, Python est intégré et disponible.
Sous la ligne de commande (*shell*), il suffit de lancer la commande « python » pour passer en mode interactif : on peut entrer du code et en demander l'exécution, utiliser les fonctions intégrées (*built-tins*), charger des bibliothèques *etc*

```
pef@darkstar:~$ python
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 10+20
30
>>> _
30
>>> _*2
60
>>> help()
```

*La variable « _ » mémorise **automatiquement** le résultat précédent.*

**Documentation**

Sous ce mode interactif, il est possible d'obtenir de la documentation en appelant la fonction `help()`, puis en entrant l'identifiant de la fonction ou de la méthode.
La documentation complète du langage est disponible sur le réseau à `http://docs.python.org/`.

# Un programme Python

**Écriture de code et exécution**

L'extension par défaut d'un source Python est « .py ».
Pour exécuter un source python (compilation et exécution sont simultanées), il existe deux méthodes :

1. en appelant l'interprète Python de l'extérieur du programme :

```
$ python mon_source.py
```

2. en appelant l'interprète Python de l'intérieur du programme :
   - ◇ on rend le source exécutable, comme un script :

```
$ chmod +x mon_source.py
```

   - ◇ on met **en première ligne du source** la ligne :

```
1 #!/usr/bin/python
```

   - ◇ on lance directement le programme :

```
$ ./mon_source.py
```

*Le « ./ » indique au système de rechercher le script dans le répertoire courant.*

# 4 Les variables

Une variable doit exister avant d'être référencée dans le programme Il faut l'instancier avant de s'en servir, sinon il y aura une erreur (une exception sera levée comme nous le verrons plus loin).

```
1 print a # provoque une erreur car a n'existe pas
```

```
1 a = 'bonjour'
2 print a # fonctionne car a est définie
```

**La variable est une référence vers une entité du langage**

```
1 a = 'entite chaine de caracteres'
2 b = a
```

*les variables* `a` *et* `b` *font références à la même chaîne de caractères.*
Une variable ne référençant rien, a pour valeur `None`.
Il n'existe pas de constante en Python (*pour signifier une constante, on utilise un nom tout en majuscule*).

**Choix du nom des variables**

⋆ Python est **sensible à la casse**, il fait la différence entre minuscules et majuscules.
⋆ Les noms des variables doivent être différents des mots réservés du langage.

**Les mots réservés « *Less is more !* »**

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

# 3 Structure d'un source Python

**Les instructions**

Les commentaires vont du caractère # jusqu'à la fin de la ligne.
*Il n'existe pas de commentaire en bloc comme en C (/* ... */).*
Chaque instruction s'écrit sur un ligne, il n'y a pas de séparateur d'instruction. *Si une ligne est trop grande, le caractère \permet de passer à la ligne suivante.*

**Les blocs d'instructions**

Les blocs d'instruction sont matérialisés par des indentations (plus de { et } !).

```
1 #!/usr/bin/python
2 # coding= latin1
3 # les modules utilises
4 import sys, socket
5 # le source utilisateur
6 if (a == 1) :
7    # sous bloc
8    # indente (1 ou 4 espaces)
```

Le caractère `:` sert à introduire les blocs.
La syntaxe est allégée, facile à lire et agréable (*si si !*).
*La ligne 2,* `# coding= latin1`*, permet d'utiliser des accents dans le source Python.*

# 5 Les valeurs et types de base

Il existe des valeurs prédéfinies :

| | |
|---|---|
| `True` | valeur booléenne vraie |
| `False` | valeur booléenne vide |
| `None` | objet vide retourné par certaines méthodes/fonctions |

Python interprète tout ce qui **n'est pas faux à vrai**.

Est considéré comme faux :

| | |
|---|---|
| `0 ou 0.0` | la valeur 0 |
| `''` | chaîne vide |
| `""` | chaîne vide |
| `()` | *liste non modifiable* ou *tuple* vide |
| `[]` | liste vide |
| `{}` | dictionnaire vide |

**Et les pointeurs ?**

Il n'existe pas de pointeur en Python : tous les éléments étant manipulés par **référence**, il n'y a donc pas besoin de pointeurs explicites !
◇ Quand deux variables référencent la même donnée, on parle « d'alias ».
◇ On peut obtenir l'adresse d'une donnée (par exemple pour comparaison) avec la fonction `id()`.

# 6 Les structures de contrôle – Instructions & Conditions

**Les séquences d'instructions**

Une ligne contient une seule instruction. Mais il est possible de mettre plusieurs instructions sur une même ligne en les séparant par des `;` (syntaxe déconseillée).

```
a = 1; b = 2; c = a*b
```

**Les conditions**

*Faire toujours attention aux tabulations !*

```
if <test1> :
   <instructions1>
elif <test2>:
   <instructions2>
else:
   <instructions3>
```

Lorsqu'une seule instruction compose la condition, il est possible de l'écrire en une seule ligne :

```
if a > 3: b = 3 * a
```

# 7 Les opérateurs

**Logique**

| | |
|---|---|
| `or` | OU logique |
| `and` | ET logique |
| `not` | négation logique |

**Binaires (bit à bit)**

| | |
|---|---|
| `\|` | OU bits à bits |
| `^` | OU exclusif |
| `&` | ET |
| `«` | décalage à gauche |
| `»` | décalage à droite |

**Comparaison**

`<,>,<=,>=,==,!=` *inférieur, sup., inférieur ou égale, sup. ou égale, égale, différent*

`is, is not` *comparaison d'identité (même objet en mémoire)*

```
c1= 'toto'
c2 = 'toto'
print c1 is c2, c1 == c2 # teste l'identité et teste le contenu
```

**Arithmétique**

`+, -, *, /,//,%` *addition, soustraction, multiplication, division, division entière, modulo*

`+=,-=,...` *opération + affectation de la valeur modifiée*

> Python v3
>
> L'opérateur `<>` est remplacé définitivement par `!=`.
> L'opérateur `/` retourne **toujours un flottant**, et `//` est utilisé pour la division entière.

# 6.1 Les structures de contrôle – Itérations

**Les itérations**

La boucle `while` dépend d'une condition.

```
while <test>:
   <instructions1>
else :
   <instructions2>
```

**Les ruptures de contrôle**

| | |
|---|---|
| `continue` | continue directement à la prochaine itération de la boucle |
| `break` | sort de la boucle courante (la plus imbriquée) |
| `pass` | instruction vide (*ne rien faire*) |

*Le `else` de la structure de contrôle n'est exécuté que si la boucle n'a pas été interrompue par un `break`.*

**Boucle infinie**

Il est souvent pratique d'utiliser une boucle `while` *infinie* (dont la condition est toujours vraie), et d'utiliser les ruptures de séquences.

```
while 1:
   if <condition> : break
```

# 8 La gestion des caractères

Il n'existe pas de type caractère mais seulement des chaînes contenant un caractère unique. Une chaîne est délimitée par des `'` ou des `"` ce qui permet d'en utiliser dans une chaîne :

```
le_caractere = 'c'
a = "une chaine avec des 'quotes'" # ou 'une chaine avec des "doubles quotes"'
print len(a) # retourne 28
```

La fonction `len()` permet d'obtenir la longueur d'une chaîne. Il est possible d'écrire une chaîne contenant plusieurs lignes sans utiliser le caractère `'\n'`, en l'entourant de 3 guillemets :

```
texte ="""   premiere ligne
      deuxieme ligne"""
```

Pour pouvoir utiliser le caractère d'échappement dans une chaîne il faut la faire précéder de `r` (pour *raw*) :

```
une_chaine = r'pour passer à la ligne il faut utiliser \n dans une chaine'
```

*En particulier, ce sera important lors de l'entrée d'expressions régulières.*

**Concaténation**

Il est possible de concaténer deux chaines de caractères avec l'opérateur `+` :

```
a = "ma chaine"+" complete"
```

# 9 Les chaînes de caractères

Il est possible d'insérer le contenu d'une variable dans une chaîne de caractères à l'aide du `%`.

```
1 a = 120
2 b = 'La valeur est %d' % a      # b contient la chaîne 'La valeur est 120'
```

Les caractères de formatage sont :

- `%s` chaîne de caractères, en fait récupère le résultat de la commande `str()`
- `%f` valeur flottante, par ex. `%.2f` pour indiquer 2 chiffres après la virgule
- `%d` un entier
- `%x` entier sous forme hexadécimal

**Les chaînes sont des objets ⟶ un objet offre des méthodes**

Les chaînes proposent différentes méthodes :

- `rstrip` supprime les caractères en fin de chaîne (par ex. le retour à la ligne)
  Exemple : `chaine.rstrip('\n ')`
- `upper` passe en majuscule
  Exemple : `chaine.upper()`
- `splitlines` décompose une chaîne suivant les lignes et retourne une liste de « lignes », sous forme d'une liste de chaîne de caractères.
- *`etc.`*

# 10 Les listes

Ces listes peuvent contenir n'importe quel type de données.
Il existe deux types de listes :
1. celles qui ne peuvent être modifiées, appelées *tuples* ;
2. les autres, qui sont modifiables, appelées simplement liste !

**Les tuples**

Il sont notés sous forme d'éléments entre parenthèses séparés par des virgules.

```
1 a = ('un', 2, 'trois')
```

*Une liste d'un seul élément correspond à l'élément lui-même*

*La fonction `len()` renvoie le nombre d'éléments de la liste.*

**Les listes modifiables**

Elles sont notées sous forme d'éléments entre crochets séparés par des virgules.
Elles correspondent à des *objets* contrairement aux tuples.

```
1 a = [10, 'trois', 40]
```

Les méthodes sont :

- `append(e)` ajoute un élément `e`
- `pop(i)` retire le $i^{\text{ème}}$ élément
- `sort` trie les éléments
- `index(e)` retourne la position de l'élément `e`
- `pop()` enlève le dernier élément
- `extend` concaténe deux listes
- `reverse` inverse l'ordre des éléments

# 10.1 Les listes — Exemples d'utilisation

```
1 a = [1, 'deux', 3]
2 a.append('quatre')
3 print a
```

```
[1, 'deux', 3, 'quatre']
```

```
4 element = a.pop()
5 print element, a
```

```
quatre [1, 'deux', 3]
```

```
6 a.sort()
7 print a
```

```
[1, 3, 'deux']
```

```
8 a.reverse()
9 print a
```

```
['deux', 3, 1]
```

```
10 print a.pop(0)
```

```
deux
```

# 10.2 Les listes — Utilisation comme « pile » et « file »

Pour une approche « algorithmique » de la programmation, il est intéressant de pouvoir disposer des structures particulières que sont les **piles** et **files**.

**La pile**

empiler `ma_pile.append(element)`
dépiler `element = ma_pile.pop()`

```
>>> ma_pile = []
>>> ma_pile.append('sommet')
>>> ma_pile
['sommet']
>>> element = ma_pile.pop()
>>> element
'sommet'
```

**La file**

enfiler `ma_file.append(element)`
defiler `element = ma_file.pop(0)`

```
>>> ma_file = []
>>> ma_file.append('premier')
>>> ma_file.append('second')
>>> element = ma_file.pop(0)
>>> element
'premier'
```

Attention

Si « element » est une liste, alors il ne faut pas utiliser la méthode `append` mais `extend`.

## 10.3 Utilisation spéciale des listes

**Affectation multiples**

Il est possible d'affecter à une liste de variables, une liste de valeurs :

```
1 (a, b, c) = (10, 20, 30)
2 print a, b, c
```

*Les parenthèses ne sont pas nécessaires s'il n'y a pas d'ambiguïté.*

```
1 a, b, c = 10, 20, 30
2 print a, b, c
```

```
10 20 30
```

En particulier, se sera utile pour les fonctions retournant plusieurs valeurs.

**Opérateur d'appartenance**

L'opérateur `in` permet de savoir si un élément est présent dans une liste.

```
1 'a' in ['a', 'b', 'c']
```

```
True
```

## 10.4 L'accès aux éléments d'une liste ou d'un tuple

Il est possible de parcourir les éléments d'une liste à l'aide de `for` :

```
1 for un_element in une_liste:
2     print un_element
```

Il est possible de les considérer comme des vecteurs :

⋆ on peut y accéder à l'aide d'un indice positif à partir de zéro ou bien *négatif* (pour partir de la fin)

```
1 ma_liste[0] # le premier element
2 ma_liste[-2] # l'avant dernier element
```

Il est possible d'extraire une « sous-liste » :

⋆ une tranche qui permet de récupérer une sous-liste

```
1 ma_liste[1:4]   # du deuxieme element au 4ieme element
2                 # ou de l'indice 1 au 4 (4 non compris)
3 ma_liste[3:]      # de l'indice 3 à la fin de la liste
```

Il est possible de créer une liste contenant des entiers d'un intervalle :

```
1 l = range(1,4)    # de 1 à 4 non compris
2 m = range(0,10,2) # de 0 à 10 non compris par pas de 2
3 print l,m
```

```
[1, 2, 3] [0, 2, 4, 6, 8]
```

D'où le *fameux accès indicé*, commun au C, C++ ou Java :

```
1 for valeur in range(0,5) :
2     print vecteur[valeur]
```

## 10.5 Des listes comme tableau à 2 dimensions

Il n'existe pas de tableau à deux dimensions en Python comme dans d'autres langages de programmation.

- Un tableau à une dimension correspond à une liste.
- Un tableau à deux dimensions correspond à une liste de liste.

**Création et utilisation du tableau à deux dimensions**

```
1 nb_lignes = 5
2 nb_colonnes = 4
3 tableau2D = []
4 for i in xrange(0, nb_lignes):
5     tableau2D.append([])
6     for j in xrange(0, nb_colonnes):
7         tableau2D[i].append(0)
8 tableau2D[2][3] = 'a'
9 tableau2D[4][3] = 'b'
0 print tableau2D
```

◇ *La fonction* `xrange()` *fonctionne comme la fonction* `range()` *mais elle est plus efficace dans ce contexte d'utilisation.*

◇ *L'accès aux différentes cases du tableau se fait suivant chaque dimension à partir de la position 0, suivant la notation :* `tableau2D[ligne][colonne]`

Ce qui donne :

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 'a'], [0, 0, 0, 0], [0, 0, 0, 'b']]
```

*Il est possible de généraliser à des tableaux de dimensions supérieures.*

## 10.6 Un accès par indice aux éléments d'une chaîne

**Et le rapport entre une liste et une chaîne de caractères ?**

Elles bénéficient de l'accès par indice, par tranche et du parcours avec `for` :

```
1 a = 'une_chaine'
2 b = a[4:7] # b reçoit 'cha'
```

Pour pouvoir modifier une chaîne de caractère, il n'est pas possible d'utiliser l'accès par indice :

```
1 a = 'le voiture'
2 a[1] = 'a'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Il faut d'abord convertir la chaîne de caractères en liste, avec la fonction `list()` :

```
1 a = list('le voiture')
2 a[1] = 'a'
```

Puis, recomposer la chaîne à partir de la liste de ses caractères :

```
1 b = ''.join(a)
2 print b
```

```
la voiture
```

## 11 Les dictionnaires

Ces objets permettent de conserver **l'association** entre une clé et une valeur.
Ce sont des *tables de hachage* pour un accès rapide aux données :
◇ La clé et la valeur peuvent être de n'importe quel type **non modifiable**.
◇ La fonction `len()` retourne le nombre d'associations du dictionnaire.

**Liste des opérations du dictionnaire**

| | |
|---|---|
| *Initialisation* | `dico = {}` |
| *définition* | `dico = {'un': 1, 'deux' : 2}` |
| *accès* | `b = dico['un'] # recupere 1` |
| *interrogation* | `if dico.has_key('trois') :`<br>`if 'trois' in dict:` |
| *ajout ou modification* | `dico['un'] = 1.0` |
| *suppression* | `del dico['deux']` |
| *récupère la liste des clés* | `les_cles = dico.keys()` |
| *récupère la liste des valeurs* | `les_valeurs = dico.values()` |

**Afficher le contenu d'un dictionnaire**

```
1 for cle in mon_dico.keys():
2    print "Association ", cle, " avec ", mon_dico[cle]
```

## 12 Les modules et l'espace de nom

Un **module** regroupe un ensemble cohérent de fonctions, classes objets, variables globales (pour définir par exemple des constantes).
Chaque module est nommé. Ce nom définit un *espace de nom*.

En effet, pour éviter des collisions dans le choix des noms utilisés dans un module avec ceux des autres modules, on utilise un accès préfixé par le nom du module :

```
1 nom_module.element_defini_dans_le_module
```

Il existe de nombreux modules pour Python capable de lui donner des possibilités très étendues.

**Accès à un module dans un autre**

Il se fait grâce à la commande `import`.

```
1 import os # pour accéder aux appels systèmes
2 import sys # pour la gestion du processus
3 import socket # pour la programmation socket
4
5 os.exit() # terminaison du processus
6 socket.SOCK_STREAM # une constante pour la programmation réseaux
```

## 13 Les sorties écran

La fonction `print` permet d'afficher de manière *générique* tout élément, que ce soit un objet, une chaîne de caractères, une valeur numérique *etc*.
Par défaut, elle ajoute un retour à la ligne après.
Le passage d'une liste permet de coller les affichages sur la même ligne.

```
1 a = 'bonjour'
2 c = 12
3 d = open('fichier.txt')
4 print a
5 print c,d, 'La valeur est %d' % c
```

On obtient :

```
bonjour
12 <open file 'fichier.txt', mode 'r' at 0x63c20> La valeur est 12
```

*Print affiche le contenu « affichable » de l'objet.*

Il est également possible d'utiliser `stdout` ou `stderr` :

```
1 import sys
2 sys.stdout.write('Hello\n')
```

## 14 Les entrées clavier

La fonction `input()` permet de saisir au clavier des valeurs.
Cette fonction retourne les données saisies comme si elles avaient été **entrées dans le source Python**.
*En fait, `input()` permet d'utiliser l'interprète ou parser Python dans un programme (combiné à la fonction `eval()`, elle permet de rentrer et d'évaluer du code dans un programme qui s'exécute !).*

```
1 a = input() # ici on entre [10,'bonjour',30]
2 print a
```

On obtient :

```
[10, 'bonjour', 30]
```

*Une* **exception** *est levée si les données entrées ne sont pas correctes en Python.*

**Ce que nous utiliserons**

Pour saisir des données en tant que chaîne de caractères uniquement, il faut utiliser la fonction `raw_input()` qui retourne un chaîne de caractères, que l'on convertira au besoin.

```
1 saisie = raw_input("Entrer ce que vous voulez") # retourne toujours une chaîne
```

Python v3

La fonction `input()` effectue le travail de la fonction `raw_input()` qui disparait.

# 15 Les conversions

Il existe un certain nombre de fonctions permettant de convertir les données d'un type à l'autre.
*La fonction* `type()` *permet de récupérer le type de la donnée sous forme d'une chaîne.*

| fonction | description | exemple |
|---|---|---|
| ord | retourne la valeur ASCII d'un caractère | ord('A') |
| chr | retourne le caractère à partir de sa valeur ASCII | char(65) |
| str | convertit en chaîne | str(10), str([10,20]) |
| int | interprète la chaîne en entier | int('45') |
| long | interprète la chaîne en entier long | long('56857657695476') |
| float | interprète la chaîne en flottant | float('23.56') |

**Conversion binaire**

Il est par exemple possible de convertir un nombre exprimé en format binaire dans une chaîne de caractères :

```
1 representation_binaire = bin(204) # à partir de Python v2.6, donne '0b11001100'
2 entier = int('11001100',2)   # on donne la base, ici 2, on obtient la valeur 204
3 entier = int('0b11001100',2) # donne le même résultat
```

*Le préfixe* `0b` *n'est pas obligatoire et peut être supprimé.*

# 16 Quelques remarques

**INTERDIT : Opérateur d'affectation**

L'opérateur d'affectation n'a pas de valeur de retour.
Il est interdit de faire :

```
1 if (a = ma_fonction()):
2    # opérations
```

**INTERDIT : Opérateur d'incrémentation**

L'opérateur ++ n'existe pas, mais il est possible de faire :

```
1 a += 1
```

**Pour convertir un caractère en sa représentation binaire sur 8 bits**

L'instruction `bin()` retourne une chaîne **sans les bits de gauche** égaux à zéro.
Exemple : `bin(5)` $\Longrightarrow$ `'0b101'`

```
1 representation_binaire = bin(ord(caractere))[2:] # en supprimant le '0b' du début
```

*La séquence binaire retournée commence au premier bit à 1 en partant de la gauche.*
Pour obtenir une représentation binaire sur 8bits, il faut la préfixer avec des `'0'` :

```
1 rep_binaire = '0'*(8-len(representation_binaire))+representation_binaire
```

## Autres conversions

**Vers la notation hexadécimale**

- un caractère donné sous sa notation hexadécimale dans une chaîne :

```
>>> a = '\x20'
>>> ord(a)
32
```

- avec le module `binascii`, on obtient la représentation hexa de chaque caractère :

```
>>> import binascii
>>> binascii.hexlify('ABC123...\x01\x02\x03')
'4142433132332e2e2e010203'
```

- avec la méthode `encode` et `décode` d'une chaîne de caractères :

```
>>> s='ABCD1234'
>>> s.encode("hex")
'4142434431323334'
>>> '4142434431323334'.decode('hex')
'ABCD1234'
```

- Il est possible de revenir à la valeur décimale codée par ces valeurs :

```
>>> int(s.encode('hex'),16)
4702394921090429748
>>> bin(int(s.encode('hex'),16))
'0b100000101000010010000110100010000110001001100100011001100110100'
```

**Attention de bien faire la distinction entre ces différentes notations !**

# 17 Gestion des erreurs

Python utilise le mécanisme des exceptions : lorsqu'une opération ne se déroule pas correctement, une **exception est levée** ce qui interrompt le contexte d'exécution, pour revenir à un environnement d'exécution supérieur, jusqu'à celui gérant cette exception.
*Par défaut, l'environnement supérieur est le shell de commande depuis lequel l'interprète Python a été lancé, et le comportement de gestion par défaut est d'afficher l'exception :*

```
Traceback (most recent call last):
  File "test.py", line 1, in ?
    3/0
ZeroDivisionError: integer division or modulo by zero
```

Pour gérer l'exception, *et éviter la fin du programme*, il faut utiliser la structure `try` et `except` :

```
1 try:
2    #travail susceptible d'échouer
3 except:
4    #travail à faire en cas d'échec
```

## 17.1 Gestion des erreurs & Exceptions

Il est possible de générer des exceptions à l'aide de la commande `raise`.
Ces exceptions correspondent à des classes objet héritant de la classe racine `Exception`.

```
raise NameError('Oups une erreur !') #NameError indique que le nom n'existe pas
```

Il existe de nombreux types d'exception (différentes classes héritant de `Exception`).
Elles peuvent également transmettre des paramètres.

```
nombre = raw_input( "Entrer valeur: " )
try:
      nombre = float( nombre )
      resultat = 20.0 / nombre
except ValueError:
      print "Vous devez entrer un nombre"
except ZeroDivisionError:
      print "Essai de division par zéro"
print "%.3f / %.3f = %.3f" % ( 20.0, nombre, resultat )
```

Question : est-ce que toutes les exceptions sont gérées dans cette exemple ?

*La définition de ses propres exceptions est en dehors du domaine d'application de ce cours.*

## 18.1 Les fichiers : lecture par ligne

**Lecture d'un fichier**

L'objet de type `file` peut être utilisé de différentes manières pour effectuer la lecture d'un fichier. C'est un objet qui peut se comporter comme une liste, ce qui permet d'utiliser le `for` :

```
for une_ligne in fichier:
   print une_ligne
```

Mais également comme un « itérateur » (qui lève une exception à la fin) :

```
while 1:
   try:
      une_ligne = fichier.next() #renvoie l'exception StopIteration en cas d'échec
      print une_ligne
   except: break
```

Ou bien simplement à travers la méthode `readline()` :

```
while 1:
   une_ligne = fichier.readline() #renvoie une ligne avec le \n à la fin
   if not une_ligne: break
   print une_ligne
```

## 18 Les fichiers : création

**Ouverture, création et ajout**

La fonction "open" renvoie un objet de type `file` et sert à ouvrir les fichiers en :

| | | |
|---|---|---|
| "r" | lecture | |
| "w" | écriture | *le fichier est créé s'il n'existe pas, sinon il est écrasé* |
| "a" | ajout | *le fichier est créé s'il n'existe pas, sinon il est ouvert et l'écriture se fait à la fin* |

Pour vérifier que l'ouverture du fichier se fait correctement, il faut **traiter une exception** de type `Exception` (elle peut fournir une description de l'erreur).

```
try:
      fichier = open("lecture_fichier.txt","r")
except Exception, message:
      print message
```

Ce qui peut produire :

```
[Errno 2] No such file or directory: 'lecture_fichier.txt'
```

*Pour simplifier, on utilisera le type de la classe racine `Exception`, car on attend ici qu'une seule erreur.*
*Dans le cas où l'on veut gérer plusieurs exceptions de types différents, il faut indiquer leur type respectif.*

## 18.2 Les fichiers : lecture spéciale et écriture

**Lecture caractère par caractère**

Sans argument, la méthode `readline` renvoie la prochaine ligne du fichier.
Avec l'argument `n`, cette méthode renvoie `n` caractères au plus (jusqu'à la fin de la ligne).
*Pour lire exactement `n` caractères, il faut utiliser la méthode `read`.*
Avec un argument de 1, on peut lire un fichier caractère par caractère.
À la fin du fichier, elle renvoie une chaîne vide (pas d'exception).

```
while 1:
   caractere = fichier.read(1)
   if not caractere : break
fichier.close() # ne pas oublier de fermer le fichier
```

**Écriture dans un fichier**

```
fichier = open("lecture_fichier.txt","a")    # ouverture en ajout
fichier.write('Ceci est une ligne ajoutee en fin de fichier\n')
fichier.close()
```

**Autres méthodes**

| | |
|---|---|
| `read(n)` | lit n caractères quelconques (même les `\n`) dans le fichier |
| `fileno()` | retourne le descripteur de fichier numérique |
| `readlines()` | lit et renvoie toutes les lignes du fichier |
| `tell()` | renvoie la position courante, en octets depuis le début du fichier |
| `seek(déc,réf)` | positionne la position courante en décalage par rapport à la référence indiquée par 0 : début, 1 : relative, 2 : fin du fichier |

## 18.3 Manipulation des données structurées

Les données structurées correspondent à une séquence d'octets. Cette séquence est composée de groupes d'octets correspondant chacun à un type. En C ou C++ :

```
struct exemple {
   int un_entier;          # 4 octets
   float un_flottant[2];   # 2*4 = 8 octets
   char une_chaine[5];   # 5 octets
   }               # la structure complète fait 17 octets
```

En Python, les structures de cette forme **n'existent pas** et une chaîne de caractère sert à manipuler cette séquence d'octets.
Le module spécialisé `struct` permet de décomposer ou composer cette séquence d'octets suivant les types contenus. Exemple : un entier sur 32bits correspond à une chaîne de 4 caractères.
Pour décrire la structure, on utilise une *chaîne de format* où des caractères spéciaux expriment les différents types.

Le module fournit 3 fonctions :

| | |
|---|---|
| `calcsize(chaine_fmt)` | retourne la taille de la séquence complète |
| `pack(chaine_fmt,...)` | construit la séquence à partir de la chaîne de format et d'une liste de valeurs |
| `unpack(chaine_fmt,c)` | retourne une liste de valeurs en décomposant suivant la chaîne de format |

## 19 Expressions régulières ou *expressions rationnelles*

Une ER permet de faire de l'appariement de motif, *pattern matching* : il est possible de savoir si un motif est **présent** dans une chaîne, mais également **comment** il est présent dans la chaine (en mémorisant la séquence correspondante).

Une expression régulière est exprimée par une suite de *meta-caractères*, exprimant :

▷ une *position* pour le motif
- `^` : début de chaîne
- `$` : fin de chaîne

▷ un caractère
- `.` : n'importe quel caractère
- `[ ]` : un caractère au choix parmi une liste, exemple : `[ABC]`
- `[^ ]` : tous les caractères sauf..., exemple : `[^@]` tout sauf le « @ »
- `[a-zA-Z]` : toutes les lettres minuscules et majuscules

▷ une alternative
- `|` : *ceci* ou *cela*, exemple : `a|b`

▷ des quantificateurs, qui permettent de répéter le caractère qui les précédent :
- `*` : zéro, une ou plusieurs fois
- `+` : **une** ou plusieurs fois
- `?` : zéro ou une fois
- `{ n }` : *n* fois
- `{ n, m }` : entre *n* et *m* fois

▷ des familles de caractères :
`\d` : un chiffre — `\D` : tout sauf un chiffre — `\n` newline
`\s` : un espace — `\w` : un caractère alphanumérique — `\r` retour-chariot

## Manipulation de données structurées : formats

**La chaîne de format**

Elle est composée d'une suite de caractères spéciaux :

| | type C | Python | | type C | Python |
|---|---|---|---|---|---|
| `x` | pad byte | pas de valeur | | | |
| `c` | char | chaîne de 1 car. | `s` | string | char[ ] |
| `b` | signed char | integer | `B` | unsigned char | integer |
| `h` | short | integer | `H` | unsigned short | integer |
| `i` | int | integer | `I` | unsigned int | long |
| `l` | long | integer | `L` | unsigned long | long |
| `f` | float | float | `d` | double | float |

*Un `!` devant le caractère permet l'interprétation dans le sens réseau (Big-Endian).*
Pour répéter un caractère il suffit de le faire précéder du nombre d'occurrences (obligatoire pour le `s` où il indique le nombre de caractères de la chaîne).
**Il faut mettre un `'='` devant le format pour garantir l'alignement des données.**

Sur l'exemple précédent, la chaîne de format est : `iffccccc` ou `i2f5c'`.

```
import struct
donnees_compactes = struct.pack('=iffccccc', 10, 45.67, 98.3, 'a', 'b', 'c', 'd', 'e')
liste_valeurs = struct.unpack('=i2f5c', donnees_compactes)
```

## 19.1 Expressions régulières en Python

Le module `re` permet la gestion des expressions régulières :
- ⋆ la composition de l'expression, avec des caractères spéciaux (comme `\d` pour *digit*) ;
- ⋆ la compilation, pour rendre rapide le traitement ;
- ⋆ la recherche dans une chaîne de caractères ;
- ⋆ la récupération des séquences de caractères correspondantes dans la chaîne.

```
import re
une_chaine = '12 est un nombre'
re_nombre = re.compile(r"(\d+)") # on exprime, on compile l'expression régulière
resultat = re_nombre.search(une_chaine) #renvoie l'objet None en cas d'échec
if resultat :
   print 'trouvé !'
   print resultat
   print resultat.group(1)
```

```
trouvé !
<_sre.SRE_Match object at 0x63de0>
12
```

L'ajout de parenthèses dans l'ER permet de mémoriser une partie du motif trouvé, accessible comme un groupe indicé de caractères (méthode `group(`*`indice`*`)`).

## 19.2 ER – Compléments : gestion du motif

**Différence majuscule/minuscule**

Pour ne pas en tenir compte, il faut l'indiquer avec une constante de module.

```
1 re_mon_expression = re.compile(r"Valeur\s*=\s*(\d+)", re.I)
```

*Ici, `re.I` est l'abréviation de `re.IGNORECASE`.*

**Motifs memorisés**

Il est possible de récupérer la liste des motifs mémorisés :

```
1 import re
2 chaine = 'Les valeurs sont 10, 56 et enfin 38.'
3 re_mon_expression = re.compile(r"\D*(\d+)\D*(\d+)\D*(\d+)", re.I)
4 resultat = re_mon_expression.search(chaine)
5 if resultat :
6    liste = resultat.groups()
7    for une_valeur in liste:
8       print une_valeur
```

On obtient :

```
10
56
38
```

## 20 Génération de valeurs aléatoires : le module `random`

**Choix d'une valeur numérique dans un intervalle**

```
1 #!/usr/bin/python
2 # coding= utf8
3 import random

5 random.seed() # initializer le générateur aléatoire
6 # Pour choisir aléatoirement une valeur entre 1 et 2^32
7 isn = random.randrange(1, 2**32) # on peut aussi utiliser random.randint(1,2**32)
8 print isn
```

```
2769369623
```

**Choix aléatoire d'une valeur depuis un ensemble de valeurs**

```
1 # retourne une valeur parmi celles données dans la chaîne
2 caractere = random.choice('ABC123')
3 print caractere
```

```
'B'
```

## 19.3 ER – Compléments : éclatement et recomposition

**Décomposer une ligne**

Il est possible "d'éclater" une ligne suivant l'ER représentant les séparateurs :

```
1 import re
2 chaine = 'Arthur:/::Bob:Alice/Oscar'
3 re_separateur = re.compile( r"[:/]+" )
4 liste = re_separateur.split(chaine)
5 print liste
```

```
['Arthur', 'Bob', 'Alice', 'Oscar']
```

**Composer une ligne**

Il est possible de composer une ligne en « joignant » les éléments d'une liste à l'aide de la méthode `join` d'une chaîne de caractère :

```
1 liste = ['Mon', 'chat',"s'appelle",'Neko']
2 print liste
3 print "_".join(liste)
```

```
['Mon', 'chat', "s'appelle", 'Neko']
Mon_chat_s'appelle_Neko
```

*Ici, la chaîne contient le séparateur qui sera ajouté entre chaque élément.*

## 21 Les options en ligne de commande : le module `optparse`

Le module « `optparse` » permet de :

- définir les options du programme et leur documentation ainsi que leur traitement :
  - chaque option possède une version courte ou longue, plus explicite ou « verbose » :

```
$ ./ma_commande.py -l mon_fichier.txt
$ ./ma_commande.py --lire-fichier=mon_fichier.txt
```

  - lors de la demande d'aide avec « `-h` », chaque option dispose d'une description :

```
$ ./ma_commande.py -h
Usage: ma_commande.py [options]

Options:
  -h, --help            show this help message and exit
  -l NOM_FICHIER, --lire-fichier=NOM_FICHIER
                        lit un fichier
  -c, --convertir       convertit le fichier
```

  - une option peut être associée :
    - ⋆ à la valeur qui la suit :

```
$ ./ma_commande.py -l mon_fichier.txt
```

    - ⋆ à un booléen :

```
$ ./ma_commande.py -c
```

  - les options peuvent être combinées :

```
$ ./ma_commande.py -l mon_fichier.txt -c
```

## Les options en ligne de commande : le module `optparse`

Le module « `optparse` » permet de :

○ décomposer les options passées au programme sur la ligne de commande :

```
1 #!/usr/bin/python
2 import optparse
3 parseur = optparse.OptionParser() # crée un parseur que l'on configure ensuite
4 parseur.add_option('-l', '--lire-fichier', help='lit un fichier',dest='nom_fichier')
5 parseur.add_option('-c', '--convertir', help='convertit le fichier',dest='conversion',
6                   default=False,action='store_true') # store_true stocke True si l'option est présente
7 (options, args) = parseur.parse_args() # args sert pour des options à multiples valeurs
8 dico_options = vars(options) # fournit un dictionnaire d'options
9 print dico_options # affiche simplement le dictionnaire
```

Les fonctions :

▷ `optparse.OptionParser()` sert à créer un « parseur » pour l'analyse des options ;

▷ `parseur.add_option` sert à ajouter une option :

◇ l'argument « `dest` » permet d'associer une clé à la valeur dans le dictionnaire résultat ;

◇ l'argument « `default` » définit une valeur par défaut que l'option soit ou non présente ;

◇ l'argument « `action` » définit une opération à réaliser avec l'option présente :

⋆ si rien n'est précisé, la valeur de l'option est stockée sous forme de chaîne ;

⋆ si on précise « `store_true` » on associe la valeur True en cas de présence de l'option.

À l'exécution :

```
$ ./ma_commande.py -l mon_fichier.txt -c
{'conversion': True, 'nom_fichier': 'mon_fichier.txt'}
```

## 22.1 Débogage avec le module « `pdb` », « Python Debugger »

On peut lancer un programme Python en activant le débogueur :

```
$ python -m pdb mon_programme.py
```

Les commandes sont les suivantes :

| | |
|---|---|
| `n next` | passe à l'instruction suivante |
| `l list` | affiche la liste des instructions |
| `b break` | positionne un *breakpoint* |
| | ex : break tester_dbg.py:6 |
| `c continue` | va jusqu'au prochain breakpoint |
| `r return` | continue l'exécution |
| | jusqu'au retour de la fonction |

*Lors du débogage, il est possible d'afficher le contenu des variables.*

Il est également possible d'insérer la ligne suivante dans un programme à un endroit particulier où on aimerait déclencher le débogage :

```
1 import pdb; pdb.set_trace()
```

```
pef@darkstar:~$ python -m pdb tester_dbg.py
> /home/pef/tester_dbg.py(3)<module>()
-> compteur = 0
(Pdb) next
> /home/pef/tester_dbg.py(4)<module>()
-> for i in range(1,10):
(Pdb) n
> /home/pef/tester_dbg.py(5)<module>()
-> compteur += 1
(Pdb) n
> /home/pef/tester_dbg.py(4)<module>()
-> for i in range(1,10):
(Pdb) i
1
(Pdb) n
> /home/pef/tester_dbg.py(5)<module>()
-> compteur += 1
(Pdb) i
2
(Pdb) l
  1     #!/usr/bin/python
  2
  3     compteur = 0
  4     for i in range(1,10):
  5  ->     compteur += 1
  6     print compteur
[EOF]
(Pdb)
```

# 22 Débogage : utilisation du mode interactif

**Le mode interactif pour « dialoguer » avec le programme**

On peut déclencher l'exécution d'un programme Python, puis basculer en mode interactif dans le contexte de ce programme, avec l'option « `-i` » :

```
$ python -i mon_programme.py
```

Sur le programme de génération de valeurs aléatoires :

```
pef@darkstar:~$ python -i test.py
184863612
>>> isn
184863612
>>>
```

On peut également passer en mode interactif depuis le programme lui-même :

```
1 #!/usr/bin/python
2 import code
3 ...
4 # on bascule en mode interactif
5 code.interact(local=locals())
```

*Il est alors possible de consulter la valeur des variables ou d'appeler des fonctions etc.*

## 22.2 Surveiller l'exécution, la « journalisation » : le module `logging`

Le « jogging » permet d'organiser les sorties de suivi et d'erreur d'un programme :

◇ plus efficace qu'un « `print` » : on peut rediriger les sorties vers un fichier ;

◇ plus facile à désactiver dans le cas où le debogage n'est plus nécessaire ;

◇ contrôlable suivant un niveau plus ou moins détaillé :

`logging.CRITICAL`
`logging.ERROR`
`logging.WARNING`
`logging.INFO`
`logging.DEBUG`

Lorsqu'un niveau est activé, automatiquement ceux de niveau inférieur sont également activés : le niveau `WARNING` active également ceux `INFO` et `DEBUG`.

*Dans le cours nous nous limiterons au seul niveau DEBUG.*

◇ contrôlable par une ligne dans le source :

```
1 #!/usr/bin/python
2 import logging

4 logging.basicConfig(level=logging.DEBUG)
5 ...
6 logging.debug('Ceci est un message de debogage')
```

```
$ python debogage.py
DEBUG:root:Ceci est un message de debogage
```

Pour désactiver le débogage, il suffit de modifier le programme en changeant la ligne 4 :

```
4 logging.basicConfig()
```

◇ possibilité de renvoyer les sorties de débogage vers un fichier :

```
4 logging.basicConfig(level=logging.DEBUG, filename='debug.log')
```

## Surveiller l'exécution, la « journalisation » : le module `logging`

◇ ajout de l'heure et de la date courante à chaque sortie :

```
4 logging.basicConfig(level=logging.DEBUG, filename='debug.log',
5                     format='%(asctime)s %(levelname)s: %(message)s',
6                     datefmt='%Y-%m-%d %H:%M:%S')
```

◇ activation par option du débogage et choix de son stockage dans un fichier :

```
 1 #!/usr/bin/python
 2 import logging,optparse

 4 parser = optparse.OptionParser()
 5 parser.add_option('-l', '--logging', dest='logging', default=False, action='store_true')
 6 parser.add_option('-f', '--logging-file', dest='logging-file',help='Logging file name')
 7 (options, args) = parser.parse_args()
 8 dico_options = vars(options)
 9 if dico_options['logging'] :
10    logging.basicConfig(level=logging.DEBUG, filename=dico_options['logging-file'],
11                format='%(asctime)s %(levelname)s: %(message)s',
12                datefmt='%Y-%m-%d %H:%M:%S')
13 logging.debug('Ceci est un message de debogage')
```

À l'exécution :

```
$ python debogage.py -l
2012-09-02 16:25:02 DEBUG: Ceci est un message de debogage
```

## 24 Le contrôle d'erreur

Une forme simple de *programmation par contrat* est introduite grâce à la fonction `assert` :

```
1 assert un_test, une_chaine_de_description
```

Si la condition n'est pas vérifiée alors le programme lève une exception.

```
1 try:
2         a = 10
3         assert a<5, "mauvaise valeur pour a"
4 except AssertionError,m:
5         print "Exception: ",m
```

```
Exception:  mauvaise valeur pour a
```

Ainsi, il est possible d'intégrer des tests pour être sûr des entrées d'une fonction par exemple et ce, afin de faire des programmables **maintenables** !

## 23 Les objets

Il est possible de définir des **classes d'objets.**
Une classe peut être définie à tout moment dans un source, et on peut en définir plusieurs dans le même source (contrairement à Java)

```
1 class ma_classe(object):    #herite de la classe object
2    variable_classe = 10
3    __init__(self):
4       self.variable_instance=10
5    def une_methode(self):
6       print self.variable_instance
```

- La fonction « `__init__()` » permet de définir les variables d'instance de l'objet.
- Le mot clé `self` permet d'avoir accès à l'objet lui-même et à ses *attributs*.
- Les attributs sont des méthodes et des variables.

*Les attributs d'un objet peuvent varier au cours du programme (comme en Javascript).*
Il est possible de parcourir les attributs d'un objet avec la fonction `dir()`.

```
1 print dir([])
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__',
 '__str__', 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
```

## 25 Gestion de processus : lancer une commande

**Exécuter une commande**

Il est possible de lancer une commande shell pour en obtenir le résultat :

```
1 import commands
2 resultat_ls = commands.getoutput('ls *.py') # récupère la liste des fichiers
```

**Se connecter à une commande**

Il est possible de lancer une commande shell en multitâche et :
– de lui envoyer des lignes en entrée (sur le `stdin` de la commande) ;
– de récupérer des lignes en sortie (depuis le `stdout`).

```
1 import os
2 sortie_commande = os.popen('ma_commande')
3 while 1:
4    line = sortie_commande.readline()
```

équivalent à

```
2 sortie_commande = os.popen('ma_commande','r') # pour lire uniquement
```

Pour envoyer des lignes à la commande :

```
2 entree_commande = os.popen('ma_commande','w') # pour écrire uniquement
```

## 25.1 Gestion de processus. . . *Nouvelle Version !*

Dans les dernières versions de Python, vous obtiendrez un avertissement d'obsolescence *DeprecationWarning* à l'utilisation de `os.popen`.

La nouvelle façon de procéder est la suivante, on utilise le module `subprocess` :

```
1 import subprocess
2 mon_programme = subprocess.Popen(['wc','-l'], stdin=subprocess.PIPE,
3                                  stdout=subprocess.PIPE)
4 mon_programme.stdin.write("Bonjour tout le monde")
5 mon_programme.stdin.close()
6 print mon_programme.stdout.read()
```

- ▷ à la ligne 2, on lance la commande, *ici wc*, avec l'argument *-l*, et on indique que l'on veut récupérer les canaux `stdin`, *pour l'entrée*, et `stdout`, *pour la sortie*, de cette commande ;
- ▷ à la ligne 3 on envoie une ligne de texte à la commande qui s'exécute en multi-tâche ;
- ▷ à la ligne 4, on ferme le fichier d'entrée ce qui indique à la commande qu'elle ne recevra plus d'entrée et donc qu'elle peut commencer à travailler ;
- ▷ en ligne 5, on récupère le résultat de son travail ;
- ▷ *ça marche !*

## 25.2 Gestion de processus : création d'un second processus

**Scinder le processus en deux**

La commande `fork` permet de scinder le processus courant en deux avec la création d'un nouveau processus. L'un est désigné comme étant le père et l'autre, le fils.

```
1 import os, sys
2 pid = os.fork()
3 if not pid :
4    # je suis l'enfant
5 else:
6    # je suis le père
7    os.kill(pid, 9) # terminaison du processus fils
```

**Gestion des arguments du processus**

Pour récupérer la liste des arguments du script (nom du script compris) :

```
1 import sys
2 print sys.argv
```

```
['mon_script.py', '-nom','toto']
```

## 26 Les fonctions : définition & arguments

La définition d'une fonction se fait à l'aide de `def` :

```
1 def ma_fonction():
2    #instructions
```

Les paramètres de la fonction peuvent être nommés et recevoir des valeurs par défaut. Ils peuvent ainsi être donné dans le *désordre* et/ou pas en totalité (très utile pour les objets d'interface comportant de nombreux paramètres dont seulement certains sont à changer par rapport à leur valeur par défaut).

```
1 def ma_fonction(nombre1 = 10, valeur = 2):
2    return nombre1 / valeur
3 print ma_fonction()
4 print ma_fonction(valeur = 3)
5 print ma_fonction(27.0, 4)
```

```
5.0
3.33333333333
6.75
```

## Les fonctions : valeur de retour & $\lambda$-fonction

**Plusieurs valeurs de retour**

```
1 def ma_fonction(x,y):
2    return x*y,x+y
3 # code principal
4 produit, somme = ma_fonction(2,3)
5 liste = ma_fonction(5,6)
6 print produit, somme
7 print liste
```

```
6 5
(30,11)
```

**Une *lambda expression* ou un objet fonction**

```
1 une_fonction = lambda x, y: x * y
2 print une_fonction(2,5)
```

```
10
```

# 27 Programmation Socket : protocole TCP

**Utilisation du protocole TCP**

Une connexion TCP correspond à un tube contenant deux canaux, un pour chaque direction de communication (A vers B, et B vers A).
Les échanges sont **bufférisés** : les données sont stockées dans une mémoire tampon jusqu'à ce que le système d'exploitation les envoie dans un ou plusieurs datagrammes IP.

Les primitives de connexion pour le protocole TCP : `socket`, `bind`, `listen`, `accept`, `connect`, `close`, `shutdown`.

| | |
|---|---|
| `shutdown(flag)` | ferme la connexion<br>en lecture (`SHUT_RD`), en écriture (`SHUT_WR`)<br>en lecture et écriture (`SHUT_RDWR`) |
| `close()` | ferme la connexion dans les deux sens |
| `recv(max)` | reçoit **au plus** `max` octets, mais peut en recevoir **moins** suivant le débit de la communication (ATTENTION !) |
| `send(data)` | envoie `data`<br>retourne le nombre d'octets effectivement envoyés |
| `sendall(data)` | bloque jusqu'à l'envoi effectif de `data` |

## 27.1 Programmation Socket : client TCP

**Programmation d'un client en TCP**

```
1  import os,socket,sys
2
3  adresse_serveur = socket.gethostbyname('localhost') # realise une requête DNS
4  numero_port = 6688
5  ma_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6
7  try: ma_socket.connect((adresse_serveur, numero_port))
8  except Exception, description:
9      print "Probleme de connexion", description
10     sys.exit(1)
11 while 1:
12    ligne = ma_socket.recv(1024) #reception d'une ligne d'au plus 1024 caracteres
13    if not ligne: break
14    else:
15       print ligne
16 ma_socket.close()
```

Attention

Le paramètre donné à « `ma_socket.recv(1024)` » indique la taille maximale que l'on voudrait recevoir, mais **ne garantie pas** qu'elle retournera forcément 1024 caractères.

## 27.2 Programmation Socket : serveur TCP

**Programmation d'un serveur en TCP**

```
1  import os,socket,sys
2
3  numero_port = 6688
4  ma_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP)
5  ma_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
6  ma_socket.bind(('', numero_port)) # équivalent à INADDR_ANY
7  ma_socket.listen(socket.SOMAXCONN)
8  while 1:
9     (nouvelle_connexion, TSAP_depuis) = ma_socket.accept()
10    print "Nouvelle connexion depuis ", TSAP_depuis
11    nouvelle_connexion.sendall('Bienvenu\n')
12    nouvelle_connexion.close()
13 ma_socket.close()
```

**Remarques**

Pour envoyer une ligne :

```
1  nouvelle_connexion.sendall('Ceci est une ligne\n')
```

*L'envoi se fait sans attendre que le « buffer d'envoi » soit plein.*

## 27.3 Programmation Socket : TCP & gestion par ligne

**Les protocoles basés sur TCP utilisant le concept de ligne**

Certains protocoles échangent des données structurées sous forme de lignes (séparées par des '`\r\n`').

**Pour la réception**

`donnees = ma_socket.recv(1024)` peut retourner plus qu'une seule ligne.
Exemple : A envoie à B, 30 lignes de texte pour un total de 920 octets.
Lors du transfert dans le réseau ces 920 octets sont décomposés en un paquet de 500 octets et un autre de 420 octets.
Lorsque B reçoit les 500 octets alors il est probable qu'ils ne contiennent pas un nombre entier de lignes (le dernier octet reçu est situé au milieu d'une ligne).
Il est nécessaire de vérifier les données reçues et éventuellement de les concaténer aux suivantes pour retrouver l'ensemble de lignes.

**Pour l'envoi**

`nb_octets = ma_socket.send(data)` peut ne pas envoyer toutes les octets.
Il est nécessaire de vérifier que le système d'exploitation a bien transmis l'intégralité des données avant de passer à l'envoi des données suivantes.
Solution : utiliser l'instruction `sendall()` au lieu de `send()`.

## 27.4 Programmation Socket : TCP & utilisation spéciale

**Utilisation d'une socket à la manière d'un fichier**

Une fois la `socket` créée, il est possible de la manipuler à la manière d'un fichier à l'aide de l'instruction `makefile()`.

```
mon_fichier_socket = ma_socket.makefile()
while 1:
   ligne = mon_fichier_socket.readline()
   if not ligne : break
   else:
      print ligne
```

Il est également possible de lire *caractère par caractère* les données.

```
mon_fichier_socket = ma_socket.makefile()
while 1:
   caractere = mon_fichier_socket.readline(1)
```

Attention

La fermeture du fichier obtenu est indépendante de la socket.
Autrement dit, il faut fermer les deux pour fermer la communication !

## 27.5 Programmation Socket : lecture par ligne

**Lecture d'une ligne de protocole**

On peut définir une fonction renvoyant une ligne séparée par '`\r\n`' lue depuis la socket caractère par caractère :

```
def lecture_ligne(ma_socket):
   ligne=""
   while 1:
      caractere = ma_socket.recv(1)
      if not caractere :
         break
      ligne += caractere
      if caractere == '\r':
         caractere = ma_socket.recv(1)
         ligne += caractere
         if caractere == '\n':
            break
         ligne += caractere
   return ligne
```

Attention

*Ce sera cette version que vous utiliserez dans les TPs.*

## 27.6 Programmation Socket : mode non bloquant

**Utilisation d'une socket en mode non bloquant**

Une fois la `socket` créée, il est possible de ne plus être bloqué en lecture lorsqu'il n'y a pas de données disponibles sur la socket.

```
ma_socket.setblocking(0)
while 1:
   try :
      donnees = ma_socket.recv(1024)
   except :
      pass
   else :
      print donnees
```

Ainsi, s'il n'y a pas de données à recevoir, une exception est levée.

Attention

Dans ce cas là, le programme attend de manière **active** des données !
Vous gaspillez inutilement les ressources de la machine !

## 27.7 Programmation socket : gestion par événement

Le module `select` et sa fonction `select()` permet d'être *averti* de l'arrivée **d'événements** sur des descripteurs de fichier ou des sockets.
Ainsi, il est possible de ne plus se *bloquer* **en lecture**, voire en écriture, sur tel ou tel descripteur ou socket.
Ces événements sont :

- une demande de connexion, lorsque cela correspond à une socket serveur ;
- la présence de données à lire ;
- la possibilité d'écrire sans être bloqué.

Il faut lui fournir en argument trois listes de descripteurs ou socket, correspondant à des événements :

1. en entrée (lecture ou connexion),
2. en sortie,
3. exceptionnels.

Elle fournit en sortie trois listes mise à jour, c-à-d ne contenant que les descripteurs pour lesquels un événement est survenu.

```
import select
(evnt_entree,evnt_sortie,evnt_exception) = select.select(surveil_entree,[],[])
```

*L'appel à la méthode* `select` *bloque tant qu'aucun événement n'est pas survenu.*
La méthode renvoie 3 listes contenant les descripteurs pour chaque événement.

## 27.8 Programmation socket : le `select`

Exemple : lancer un `accept` uniquement lorsqu'un client essaye un `connect`.

```
import sys,os,socket,select
adresse_hote = ''
numero_port = 6688
ma_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM,socket.IPPROTO_TCP)
ma_socket.bind((adresse_hote, numero_port))
ma_socket.listen(socket.SOMAXCONN)
surveillance = [ma_socket]
while 1:
   (evnt_entree,evnt_sortie,evnt_exception) = select.select(surveillance,[],[])
   for un_evenement in evnt_entree:
      if (un_evenement == ma_socket):
         nouvelle_connexion, depuis = ma_socket.accept()
         print "Nouvelle connexion depuis ",depuis
         nouvelle_connexion.sendall('Bienvenu')
         surveillance.append(nouvelle_connexion)
         continue
      ligne = un_evenement.recv(1024)
      if not ligne :
         surveillance.remove(un_evenement) # le client s'est déconnecté
      else : # envoyer la ligne a tous les clients
```

## 27.9 Programmation socket : le protocole UDP

Utilisation du protocole UDP :

```
import socket

TSAP_local = ('', 7777)
TSAP_distant = (socket.gethostbyname("ishtar.msi.unilim.fr"), 8900)

ma_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM,socket.IPPROTO_UDP)
ma_socket.bind(TSAP_local)

ma_socket.sendto("Hello", TSAP_distant)
donnees,TSAP_emetteur = ma_socket.recvfrom(1000)
```

On utilise les méthodes suivantes de l'objet `socket` :

- la méthode « `sendto` » reçoit en paramètre les données et le TSAP du destinataire.
- la méthode « `recvfrom` » :
  - reçoit en paramètre la taille maximale des données que l'on peut recevoir (s'il y a plus de données reçues elles seront ignorées) ;
  - retourne ces données et le TSAP de l'émetteur.

> **Attention**
>
> En UDP, on échange uniquement un datagramme à la fois, d'au plus 1500 octets environ.

# 28 Multithreading – Threads

Il est possible d'utiliser directement des fonctions :

```
import thread
def fonction():
    # travail
thread.start_new(fonction,())
```

**Mieux : utiliser la classe « threading »**

Cette classe permet d'exécuter une fonction en tant que *thread*.
Ses méthodes sont :

| | |
|---|---|
| `Thread(target=func)` | permet de définir la fonction à transformer en thread, retourne un objet thread. |
| `start()` | permet de déclencher la thread |

```
import threading
def ma_fonction:
   # travail
   return
ma_thread = threading.Thread(target = ma_fonction)
ma_thread.start()
# Thread principale
```

## 28.1 Multithreading – Sémaphores

**La classe semaphore**

Les méthodes sont :

| | |
|---|---|
| `Semaphore([val])` | fonction de classe retournant un objet Semaphore initialisé à la valeur optionnelle 'value' |
| BoundedSemaphore([v]) | fonction de classe retournant un objet Semaphore initialisé à la valeur optionnelle 'value' qui ne pourra jamais dépasser cette valeur |
| `acquire()` | essaye de diminuer la sémaphore, bloque si la sémaphore est à zéro |
| `release()` | augmente la valeur de la sémaphore |

**Exemple d'utilisation**

```
import threading
class semaphore(Lock):
   int compteur = 0
   def up():
      self.acquire()
      compteur += 1
      return self.release()
   def down():
      self.acquire()
```

# 29 Manipulations avancées : système de fichier

**Informations sur les fichiers**

Pour calculer la taille d'un fichier, il est possible de l'ouvrir, de se placer en fin et d'obtenir la position par rapport au début (ce qui indique la taille) :

```
1 mon_fichier = open("chemin_fichier", "r")
2 mon_fichier.fseek(2,0)   #On se place en fin, soit à zéro en partant de la fin
3 taille = mon_fichier.ftell()
4 mon_fichier.fseek(0,0)   # Pour se mettre au début si on veut lire le contenu
```

Pour connaître la nature d'un fichier :

```
1 import os.path
2 if os.path.exists("chemin_fichier") : # etc
3 if os.path.isfile("chemin_fichier") : # etc
4 if os.path.isdir("chemin_fichier") : # etc
5 taille = os.path.getsize("chemin_fichier") # pour obtenir la taille d'un fichier
```

# 29.1 Manipulations avancées : UDP & Broadcast, Scapy

**Les sockets**

Pour la réception de paquets UDP en *multicast*, il est nécessaire d'informer l'OS de la prise en charge d'un groupe par l'application :

```
1 import struct
2 mcast = struct.pack("4sl", socket.inet_aton("224.0.0.127"), socket.INADDR_ANY)
3 ma_socket.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mcast)
```

Pour obtenir l'adresse IP de la machine que l'on utilise :

```
1 mon_adresse_ip = socket.gethostbyname(socket.gethostname())
```

**Scapy**

Le module `scapy` dispose de capacités à traiter le contenu des paquets reçus.
Cette bibliothèque d'injection de *paquets forgés* dispose de fonctions d'analyse et d'affichage de données brutes, *raw*, comme `DNS()`, `IP()`, `UDP()`, etc.

```
1 import scapy
2
3 # la variable donnees contient le contenu d'un paquet DNS
4 analyse = scapy.DNS(donnees)
5 analyse.show()
```

# 29.2 Manipulation avancées : l'écriture de *Script système*

Lorsque l'on écrit un programme Python destiné à être utilisé en tant que « script système », c-à-d comme une commande, il est important de soigner l'interface avec l'utilisateur, en lui proposant des *choix par défaut* lors de la saisie de paramètres :

```
1  #!/usr/bin/python
2  import sys
3  #configuration
4  nom_defaut = "document.txt"
5  #programme
6  saisie = raw_input("Entrer le nom du fichier [%s]" % nom_defaut)
7  nom_fichier = saisie or nom_defaut
8  try:
9     entree = open(nom_fichier, "r")
10 except Exception, message:
11    print message
12    sys.exit(1)
```

- ▷ en ligne 4, on définie une valeur par défaut pour le nom du fichier à ouvrir ;
- ▷ en ligne 6, on saisie le nom de fichier avec, entre crochets, le nom par défaut ;
- ▷ en ligne 7, si l'utilisateur tape « entrée », `saisie` est vide, c-à-d *fausse* ;
  résultat : l'opérateur « or » affecte la valeur par défaut, qui est *vraie*.

# 29.3 Manipulations avancées : construction de listes

Il est possible d'obtenir une deuxième liste à partir d'une première en appliquant sur chaque élément de la première une opération.
*La deuxième liste contient le résultat de l'opération pour chacun des éléments de la première liste.*

Il est possible d'obtenir le résultat en une instruction unique pour la construction de cette deuxième liste.

Exemple : on cherche à obtenir la liste des lettres en majuscule à partir des valeurs ASCII de 65 ('A') à 91 ('B') :

```
1 [chr(x) for x in range(65, 91)]
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

## 30 Pour améliorer l'expérience de Python

**Des interprètes alternatifs**

Ils proposent un historique des commandes déjà entrées dans une session précédente, la complétion des commandes, de l'aide contextuelle, *etc*.

- ⋆ iPython, `http://ipython.org/`
- ⋆ bpython, `http://bpython-interpreter.org/`

**Une configuration particulière de l'interprète courant**

L'intérêt de cette configuration est de fournir les avantages de la complétion et de l'historique dans l'interprète Python courant et déjà installé dans le système.

1. télécharger le script à l'adresse suivante :
   `http://www.digitalprognosis.com/opensource/scripts/pythonrc`
2. le sauvegarder sous le nom : `~/.pythonrc`
3. ajouter dans son environnement shell la variable suivante :

```
$ export PYTHONSTARTUP=~/.pythonrc
```

Ainsi le script sera exécuté à chaque démarrage de Python.