

Faculté
des Sciences
& Techniques



Université
de Limoges

Master 1^{ère} année



Mon'tit Python

—

P-F. Bonnefoi

Version du 14 septembre 2017

Table des matières

1	Pourquoi Python ?	5
	Pourquoi Python ? <i>Ses caractéristiques</i>	6
	Pourquoi Python ? <i>Ses usages</i>	7
2	Un programme Python	8
3	Structure d'un source Python	10
4	Les variables	11
5	Les valeurs et types de base	12
6	Les structures de contrôle – Instructions & Conditions	13
	Les structures de contrôle – Itérations	14
7	Les opérateurs	15
8	La gestion des caractères	16
9	Les chaînes de caractères	17
10	Les listes	20
	Les listes — Exemples d'utilisation	21
	Les listes — Utilisation comme «pile» et «file»	22
	L'accès aux éléments d'une liste ou d'un tuple	23
	Utilisation avancée des listes	24
	Utilisation avancée : création d'une liste à partir d'une autre	25
	Utilisation avancée : l'opérateur ternaire	26
	Tableau & liste modifiable	27

Un accès par indice aux éléments d'une chaîne	28
11 Les dictionnaires	29
12 Les modules et l'espace de nom	31
13 Les sorties écran	32
14 Les entrées clavier	33
15 Les conversions	34
16 Quelques remarques	37
17 Gestion des erreurs	38
Gestion des erreurs & Exceptions	39
18 Les fichiers : création	40
Les fichiers : lecture par ligne	41
Les fichiers : lecture spéciale et écriture	42
Manipulation des données structurées	43
19 Expressions régulières ou <i>expressions rationnelles</i>	45
Expressions régulières en Python	46
ER – Compléments : gestion du motif	47
ER – Compléments : éclatement et recomposition	48
20 Les fonctions : définition & arguments	49
21 Le contrôle d'erreur à l'exécution	51
22 Génération de valeurs aléatoires : le module <code>random</code>	52
23 L'intégration dans Unix : l'écriture de <i>Script système</i>	53
24 Gestion de processus : lancer une commande externe et récupérer son résultat	54
Gestion de processus : création d'un second processus	55

25	Programmation Socket: protocole TCP	57
	Programmation Socket: client TCP	60
	Programmation Socket: serveur TCP	61
	Programmation Socket: TCP & bufférisation	62
	Programmation Socket: lecture par ligne	63
	Programmation Socket: mode non bloquant	64
	Programmation socket: gestion par événement	65
	Programmation socket: le <code>select</code>	66
	Programmation socket: le protocole UDP	67
26	Multithreading – Threads	70
	Multithreading – Sémaphores	71
27	Manipulations avancées : système de fichier	72
	Réseaux : broadcast UDP, obtenir son adresse IP, Scapy	73
	La programmation objet : définition de classe et introspection	74
	Intégration Unix : les options en ligne de commande : le module <code>optparse</code>	75
28	Débogage : utilisation du mode interactif	77
	Débogage avec le module « <code>pdb</code> », «Python Debugger»	78
	Surveiller l'exécution, la «journalisation» : le module <code>logging</code>	79
29	Pour améliorer l'expérience de Python	81
30	Index	82



Parce qu'il est :

- ★ **portable** : disponible sous toutes les plate-formes (de Unix à Windows, en passant par des systèmes embarqués avec μ python) ;
- ★ **simple** : possède une **syntaxe claire**, privilégiant la lisibilité, libérée de celle de C/C++ ;
- ★ **riche** : incorpore de nombreuses capacités tirées de différents modèles de programmation :
 - ◊ **programmation impérative** : *structure de contrôle, manipulation de nombres comme les flottants, doubles, complexe, de structures complexes comme les tableaux, les dictionnaires, etc.*
 - ◊ **langages de script** : *accès au système, manipulation de processus, de l'arborescence fichier, d'expressions rationnelles, etc.*
 - ◊ **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 (fonction anonyme), de générateur etc.*
 - ◊ **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.*



Pourquoi Python ? Ses caractéristiques

6

Il est :

- * *dynamique* : il n'est pas nécessaire de déclarer le type d'une variable dans le source : elle sert à référencer une donnée dont le type est connu 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* : le source est compilé en *bytecode* à la manière de Java (pouvant être sauvegardé) puis exécuté sur une machine virtuelle ;
- * *interactif*: en exécutant l'interprète, on peut entrer des commandes, obtenir des résultats, *travailler* sans avoir à écrire un source au préalable.

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), de μ services avec Flask/Bottle, d'analyse réseau (SCAPY), manipulation d'XML, JSON 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.



- ▷ 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 !



Mode interactif

Sur tout Unix, Python est disponible, il est intégré dans les commandes de base du système.

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 obtenir l'exécution, utiliser les fonctions intégrées (*builtins*), charger des bibliothèques etc

```
xfce4-terminal xterm
pef@darkstar:~$ python
Python 2.7.12 (default, Jun 29 2016, 12:52:38)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 10+20
30
>>> _
30
>>> _*2
60
>>> help()
```

La variable «`_`» mémorise **automatiquement** le résultat obtenu précédemment.

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/>.



Écriture de code et exécution

L'extension par défaut d'un source Python est «.py» («.pyc» correspond à la version compilée).

Pour exécuter un source python (compilation et exécution sont simultanées), 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 sous Unix :

```
$ 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.

Remarque

Dans le cas où la commande «python» exécute Python v3, au lieu de v2, vous pouvez utiliser la commande «python2» à la place.



3 Structure d'un source Python

10

Les commentaires

Les commentaires vont du caractère # jusqu'à la fin de la ligne.

Il n'existe pas de commentaire en bloc comme en C / ... */.*

Les instructions

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 « } » !

```
#!/usr/bin/python
# coding= utf8
# les modules utilisés
import sys, socket
# le source utilisateur
if (a == 1) :
    # sous bloc
    # indenté (espaces ou tabulation)
```

Le caractère « : » sert à introduire les blocs.

La ligne ①, # coding= utf8, permet d'utiliser des accents dans le source Python si celui-ci est encodé en unicode lui-même codé sur 8bits, appelé «utf8».

La syntaxe est allégée, facile à lire et agréable (*si si !*).

Attention

Tout bloc d'instruction est :

- ▷ précédé par un « : » sur la ligne qui le précède ;
- ▷ indenté par rapport à la ligne précédente et de la même manière que **tous les autres blocs** de même niveau (même nombre de caractères espaces et/ou de tabulation) du même source.



4 Les variables

11

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* est levée comme nous le verrons plus loin).

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

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

La variable est une référence vers un élément du langage

```
a = 'entite chaine de caracteres'  
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	assert	break	class	continue
def	del	elif	else	except
exec	finally	for	from	global
if	import	in	is	lambda
not	or	pass	print	raise
return	try	while		



5 Les valeurs et types de base

12

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
()	<i>liste non modifiable ou tuple</i> 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()` (ce qui permet de savoir si deux alias réfèrent à la même donnée).



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

```
if <test1> :  
    <instructions1>  
    <instructions2>  
elif <test2>:  
    <instructions3>  
else:  
    <instructions4>
```

Attention

- ▷ Faire toujours attention au décalage : ils doivent être **identiques** (même nombre de tabulations ou d'espaces) !
- ▷ Ne pas oublier le « : » avant le bloc indenté !

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

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



Les itérations

L'exécution du bloc d'instructions de la boucle `while` dépend d'une condition.

```
while <test>:  
    <instructions1>  
    <instructions2>  
else :  
    <instructions3>
```

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

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*)

Boucle infinie & rupture de contrôle

Il est souvent pratique d'utiliser une boucle `while infinie` (dont la condition est toujours vraie), et d'utiliser les ruptures de contrôle pour la terminer :

```
while 1:  
    if <condition1> : break  
    if <condition2> : break
```

En effet, lorsqu'il existe :

- plusieurs conditions de sortie ;
- des conditions complexes et combinées ;

⇒ l'utilisation des ruptures de contrôle **simplifie** l'écriture, ce qui **évite les erreurs !**



Logique

or OU logique
and ET logique
not négation logique

Binaires (bit à bit)

| OU bits à bits
^ OU exclusif
& ET
>> 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.



8 La gestion des caractères

16

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, sans déclencher son interprétation, 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 que nous verrons plus loin.

Concaténation

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

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



9 Les chaînes de caractères

17

Il est possible d'insérer le contenu d'une variable dans une chaîne de caractères à l'aide de l'opérateur «%» qui s'applique sur une liste de valeur à substituer :

```
a = 120
b = 'La valeur de %s est %d' % ('a', a) # b <- la chaîne 'La valeur de a 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

`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.



Opérateur «%» vs fonction «format»

Affichage d'une chaîne composée de deux arguments :

```
□ — xterm —————
>>> print '%s %s' % ('un', 'deux')
un deux
```

Pour des nombres :

```
□ — xterm —————
>>> '%d' % 42
'42'
>>> '%f' % 3.1415926
'3.141593'
```

La même chose mais aussi avec une inversion :

```
□ — xterm —————
>>> print '{} {}'.format('un', 'deux')
un deux
>>> print '{1} {0}'.format('un', 'deux')
deux un
```

```
□ — xterm —————
>>> '{:d}'.format(42)
'42'
>>> '{:f}'.format(3.1415926)
'3.141593'
```

Formatage des données en notation hexadécimal

Ajouter ou non le préfixe '0x', ajouter des zéros devant pour obtenir une longueur fixe :

```
□ — xterm —————
>>> "{0:#0{1}x}".format(42, 6)
'0x002a'
>>> "{0:0{1}x}".format(42, 6)
'00002a'
>>> "{0:#0{1}X}".format(42, 6)
'0X002A'
>>> "{0:0{1}X}".format(42, 6)
'00002A'
>>> "{0:0{1}X}".format(42, 4)
'002A'
```

- ▷ { : identifiant de format
- ▷ 0 : : récupération du premier paramètre
- ▷ # : utiliser le préfixe "0x"
- ▷ 0 : compléter par des zéros...
- ▷ { 1 } : ...jusqu'à une longueur de *n* caractères (incluant le "0x"), défini par le second paramètre
- ▷ x: représentation sous forme hexadécimal, en utilisant les caractères minuscules
- ▷ } : identifiant de fin de format.

En utilisation directe :

```
□ — xterm —————
>>> format(255, '02X')
'FF'
>>> format(2, '02X')
'02'
```



Sous le shell, on peut utiliser la commande echo pour envoyer une chaîne contenant un «é», en entrée de la commande xxd pour l'afficher sous notation hexadécimale (le «00000000:» désigne le décalage par rapport au début de la chaîne) :

```
└── xterm ━
pef@darkstar:/Users/pef $ echo -n 'é' | xxd
00000000: c3a9
```

Sous Python :

```
└── xterm ━
>>> ord('é')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
>>> len('e')
1
>>> len('é') ①
2
>>> len(u'é') ②
1
>>> 'é'[0] ③
'\xc3'
>>> 'é'[1]
'\xa9'
>>> 'é'[1].encode('hex')
'a9'
>>> ord('é'[1])
169
>>> hex(ord('é'[1]))
'0xa9'
>>> '{0:X}'.format(ord('é'[1])) ④
'A9'
```

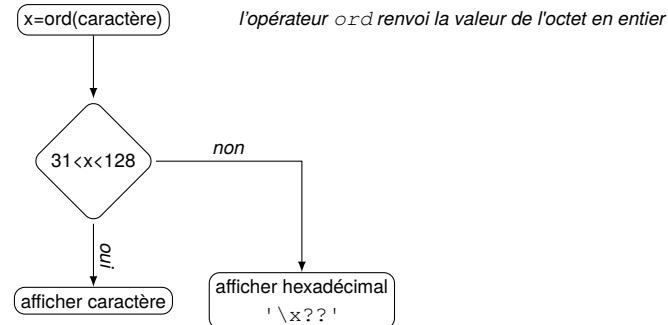
① ⇒ on saisit le caractère accentué dans une chaîne «normale» de Python où il est traité comme deux octets successifs.

② ⇒ on saisit le «é» dans une chaîne au format *unicode*, préfixée par un «u», c-à-d adaptée à gérer tout type d'alphabets ou d'idéogrammes. La chaîne ne contient donc qu'un seul symbole.

③ ⇒ on peut accéder au premier octet/caractère individuellement.

④ ⇒ la fonction format peut être utilisée pour obtenir la notation hexadécimale, une fois le caractère obtenu sous sa notation entière.

L'affichage des caractères suit le procédure de décision suivante :



Pourquoi obtient-on un «é» au final à l'écran ? Parce que le terminal sait traiter l'unicode, c-à-d le codage des deux octets vers un e avec un accent aigu.

Deux types de listes

- celles qui ne peuvent être modifiées, appelées *tuples* ;
- les autres, qui sont modifiables, appelées simplement liste !

Ces listes peuvent contenir n'importe quel type de données.

Les tuples (notation parenthèse)

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

```
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 (notation crochet)

Elles sont notées sous forme d'éléments entre crochets séparés par des virgules.

Elles correspondent à des *objets* contrairement aux tuples :

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

Quelques méthodes supportées par une liste :

`append(e)` ajoute un élément e

`pop()` enlève le dernier élément

`pop(i)` retire le i^{ème} élément

`extend` concatène deux listes

`sort` trie les éléments

`reverse` inverse l'ordre des éléments

`index(e)` retourne la position de l'élément e



Ajout d'un élément à la fin

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

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

Retrait du dernier élément

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

quatre [1, 'deux', 3]

Tri des éléments

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

[1, 3, 'deux']

Inversion de l'ordre des éléments

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

['deux', 3, 1]

Retrait du premier élément

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

deux



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()`

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

La file

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

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

Attention à l'ajout d'une liste dans une autre

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



Parcourir les éléments d'une liste à l'aide de `for` :

```
for un_element in une_liste:  
    print un_element
```

Considérer les listes comme des **vecteurs ou tableau à une dimension** :

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

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

Extraire une «sous-liste» :

- une **tranche**, ou «*slice*», qui permet de récupérer une sous-liste

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

Créer une liste contenant des entiers compris dans un **intervalle** :

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

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

D'où le *fameux* accès indicé, commun au C, C++ ou Java : (*ici, on parcourera les valeurs 0,1,2,3 et 4*)

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



Affectations multiples & simultanées de variables

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

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

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

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

10 20 30

En particulier, cela est utile pour les fonctions retournant plusieurs valeurs.

Opérateur d'appartenance

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

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

True

C'est l'opérateur «`in`» que l'on utilisera pour tester l'existence d'un élément dans un objet.

Création automatique de liste

```
['a']*10
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
```



Il est possible d'obtenir une deuxième liste à partir d'une première, en appliquant une opération sur chaque élément de la première .

La deuxième liste contient le résultat de l'opération pour chacun des éléments de la première liste.

Une **notation particulière** permet en une instruction de combiner la création de cette deuxième liste et le parcours de la première.

Exemple :

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

```
[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']
```

Cette forme de création de liste à partir d'une liste s'appelle les «*lists comprehension*».

Création et initialisation d'un «tableau»

```
□ — xterm —
>>> [0 for i in range(0, 10)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

□ — xterm —
>>> [x for x in range(54, 78)]
[54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, ↵
 73, 74, 75, 76, 77]
```



Opérateur ternaire similaire à celle du C, avec les instructions conditionnelles :

En C :

```
a = (condition ? 1 : 0);
```

En Python :

```
x = true_value if condition else false_value  
contenu = ((doc + '\n') if doc else '')
```

Opérateur ternaire et les « lists comprehension »

- ▷ première utilisation de la «list comprehension» : on ajoute le caractère si le code ASCII qui lui correspond est impair (le modulo vaut 1 ce qui équivaut à vrai), sinon on ajoute une chaîne vide ''.

```
□ — xterm —  
>>> l=range(65,90)  
>>> l  
[65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89]  
>>> [chr(x) if (x % 2) else '' for x in l ]  
['A', '', 'C', '', 'E', '', 'G', '', 'I', '', 'K', '', 'M', '', 'O', '', 'Q', '', 'S', '', 'U', '', 'W', '', 'Y']
```

- ▷ seconde utilisation de la «list comprehension» : on ajoute le caractère suivant le même test que précédemment, mais dans le cas où la condition est fausse, on **n'ajoute rien** à la liste résultat !

```
□ — xterm —  
>>> l=range(65,90)  
>>> l  
[65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89]  
>>> [chr(x) for x in l if (x % 2)]  
['A', 'C', 'E', 'G', 'I', 'K', 'M', 'O', 'Q', 'S', 'U', 'W', 'Y']
```



Créer les éléments du tableau

Pour pouvoir traiter une liste à la manière d'un tableau, il faut en créer tous les éléments au préalable :

```
□ — xterm —
>>> t = [0]*10
>>> t
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Ici, on crée une liste de 10 éléments à zéro.

Accéder aux différentes «cases» du tableau

On peut ensuite modifier son contenu à l'aide de l'accès par indice :

```
□ — xterm —
>>> t[1]='un'
>>> t[3]=2
>>> t
[0, 'un', 0, 2, 0, 0, 0, 0, 0, 0]
```

On remarque que le «tableau» peut contenir des éléments de type quelconque.

Tableaux à deux dimensions : des listes imbriquées

```
□ — xterm —
>>> t=[[0 for x in range(0,4)] for x in range(0,5)]
>>> t
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Ici, on vient de créer un tableau à deux dimensions de 5 lignes de 4 colonnes (accès avec $t[a][b]$).



Rapport entre une liste et une chaîne de caractères ? Aucun !

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

```
a = 'une_chaine'  
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 :

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

Traceback (most recent call last) :

```
File "<stdin>", line 1, in <module>  
TypeError: 'str' object does not support item assignment
```

Chaîne ⇒ Liste ⇒ Accès tableau/Modification ⇒ Chaîne

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

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

Ce qui donne l'exécution suivante :

```
['l', 'a', ' ', 'v', 'o', 'i', 't', 'u', 'r', 'e']
```

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

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

```
la voiture
```



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') :
 if 'trois' in dico:`

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 avec un accès suivant les clés

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



Couples de valeurs obtenus par combinaison de deux listes

```
□ — xterm —  
>>> zip(['a', 'b', 'c'], [1, 2, 3])  
[('a', 1), ('b', 2), ('c', 3)]
```

Il est également possible d'obtenir un dictionnaire utilisant chacun des couples en tant que (clé, valeur) :

```
□ — xterm —  
>>> couples=zip(['a', 'b', 'c'], [1, 2, 3])  
>>> dict(couples)  
{'a': 1, 'c': 3, 'b': 2}
```

Ce qui permet d'utiliser l'accès direct du dictionnaire :

```
□ — xterm —  
>>> dico=dict(couples)  
>>> dico['b']  
2
```

Attention : les valeurs d'une liste qui ne sont associées à aucune valeur dans la seconde, sont supprimées :

```
□ — xterm —  
>>> zip(['a', 'b', 'c'], [1, 2, 3, 4])  
[('a', 1), ('b', 2), ('c', 3)]  
>>> zip(['a', 'b', 'c', 'd'], [1, 2, 3])  
[('a', 1), ('b', 2), ('c', 3)]
```



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 :

```
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

Il se fait grâce à la commande `import`.

```
import os      # pour accéder aux appels systèmes
import sys    # pour la gestion du processus
import socket # pour la programmation socket

# quelques exemples
os.exit() # terminaison du processus
socket.SOCK_STREAM # une constante pour la programmation réseaux
```



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.

```
a = 'bonjour'  
c = 12  
d = open('fichier.txt')  
print a  
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 explicitement les canaux `stdout` ou `stderr`:

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



La fonction `raw_input()`

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

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

*C'est cette fonction que nous utiliserons **exclusivement**.*

La fonction `input()`

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 !).

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

On obtient :

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

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

Python v3

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



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
<code>ord</code>	retourne la valeur ASCII d'un caractère	<code>ord('A')</code>
<code>chr</code>	retourne le caractère à partir de sa valeur ASCII	<code>chr(65)</code>
<code>str</code>	convertit en chaîne	<code>str(10), str([10, 20])</code>
<code>int</code>	interprète la chaîne en entier	<code>int('45')</code>
<code>long</code>	interprète la chaîne en entier long	<code>long('56857657695476')</code>
<code>float</code>	interprète la chaîne en flottant	<code>float('23.56')</code>

Conversion en représentation binaire

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

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

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

```
representation_binaire = bin(204)[2:] # donne '11001100'
```

Ici, on utilise une «tranche» de la chaîne de caractères considérées comme un tableau depuis le troisième caractère jusqu'à la fin, ce qui supprime le «0b».



Vers la notation hexadécimale

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

```
□ — xterm —
>>> a = '\x20'
>>> ord(a)
32
```

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

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

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

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

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

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

Attention de bien faire la distinction entre ces différentes notations !



Notation et format

Quel rapport entre : ☀, III et 3 ? Ou entre ☀, V et 5 ? La notation !

La première représente la face d'un dé, la seconde utilise un chiffre romain et la dernière, un chiffre décimal.

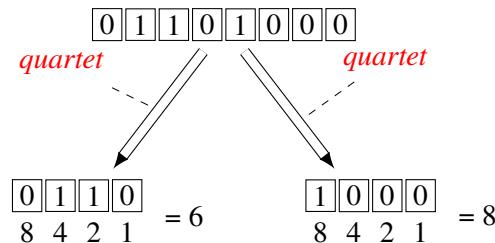
Est-ce que la valeur **change** suivant la notation utilisée ? **Non !**

Comment représente-t-on un **code postal** ? ☐☐☐☐☐, c-à-d exactement 5 chiffres, exemple : 87000

Et du côté d'un ordinateur ?

Qu'est-ce qu'un octet ? Le plus petit format utilisé et manipulé, soient 8bits, c-à-d ☐☐☐☐☐☐☐☐☐ rempli de 0 ou de 1.

Comment le noter pour être facilement manipulable par un humain ? En utilisant la **notation hexadécimale** :



Ce qui donne 68 en notation hexadécimale.

Comme la somme est comprise entre 0 et 15, on utilise les chiffres de 0 à 9 pour leur valeur respective, et les lettres de A à F pour les valeurs de 10 à 15.

Exemple : D5 correspond à 13 et 5 reste 5, soient :

$$\begin{array}{r} 1 \boxed{1} \boxed{0} \boxed{1} \\ 8 \quad 4 \quad 2 \quad 1 \end{array} = 13 \qquad \begin{array}{r} 0 \boxed{1} \boxed{0} \boxed{1} \\ 8 \quad 4 \quad 2 \quad 1 \end{array} = 5$$

soit la séquence binaire 11010101.

La notation décimale pour un octet (par exemple, pour la correspondance avec un caractère ASCII) :

$$\begin{array}{r} 0 \boxed{1} \boxed{1} \boxed{0} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \\ 128 \quad 64 \quad 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1 \end{array} = 104$$

La notation hexadécimale est la plus simple et la plus employée car elle permet de noter facilement des séquences d'octets (chaque octet est noté sous forme de deux digits hexadécimaux ou quartets).



INTERDIT : Opérateur d'affectation

L'opérateur d'affectation n'a pas de valeur de retour.

Il est interdit de faire :

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

INTERDIT : Opérateur d'incrémantation

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

```
a += 1
```

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

L'instruction bin() retourne une chaîne **sans les bits de gauche égaux à zéro**.

Exemple: bin(5) \Rightarrow '0b101'

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

La séquence binaire rentrée commence au premier bit à 1 en partant de la gauche.

Pour obtenir une représentation binaire sur 8bits, il faut la préfixer avec les '0' manquants:

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



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. Ce processus est répété jusqu'à un contexte gérant cette exception, ou jusqu'à l'arrêt du programme s'il n'y en a pas.

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
```

Gestion des exceptions

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

```
try:
    #travail susceptible d'échouer
except:
    #travail à faire en cas d'échec
```

Attention

Toutes les opérations susceptibles d'entraîner une erreur ne lève pas d'exception.



Traiter une exception

Elles sont traiter, «*intercepter*», suivant leur nature :

```
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 ?

Il existe de nombreux types d'exception correspondant à des classes objet héritant de la classe racine `Exception`.

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

Générer une exception

Il est possible de générer des exceptions à l'aide de la commande `raise`.

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



Ouverture, création et ajout

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

"r" lecture *lève une exception en cas d'erreur*

"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**.

On peut également obtenir 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'
```

On utilisera le type de la classe racine `Exception` pour intercepter l'exception, car on attend ici qu'une seule erreur.

Dans le cas où l'on veut gérer plusieurs exceptions de types différents pouvant être levées dans un bloc d'instruction, il faut indiquer leur type respectif.



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
```

Si on veut supprimer le \n à la fin de la ligne après la lecture :

```
ligne = ligne.rstrip('\n')
```



Lecture caractère par caractère : `read` vs `readline`

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 ajoutée 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



Les **données structurées** correspondent à une séquence d'octets : composée de groupes d'octets dont le nombre correspond au type de la variable.

En C ou C++ :

```
struct exemple {  
    int un_entier;          # 4 octets  
    float un_flaotant[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 à manipuler, on utilise une *chaîne de format* où des caractères spéciaux expriment les différents types qui se succèdent et permettent de regrouper les octets entre eux.

Le module fournit 3 fonctions :

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



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		1 caractère
b	signed char		integer
h	short		integer
i	int		integer
l	long		integer
f	float		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 : iffcccc ou i2f5c.

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

Attention : le format '5c' renvoie 5 caractères alors que '5s' renvoie une chaîne de 5 caractères.



19 Expressions régulières ou *expressions rationnelles*

45

Une ER permet de faire de l'appariement de motif, *pattern matching* :

- ▷ savoir si un motif est **présent** dans une chaîne,
- ▷ **comment** il est présent dans la chaîne (en mémorisant la séquence correspondante).

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

- ▷ une *position* pour le motif
- ▷ une alternative
 - ^ : 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
- ▷ 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



Le module `re` permet la gestion des expressions régulières :

- i. composition de l'expression, avec des caractères spéciaux (comme `\d` pour *digit*) ;
- ii. compilation, pour rendre rapide le traitement ;
- iii. recherche dans une chaîne de caractères ;
- iv. 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 à l'aide la méthode «`group`» : (`resultat.group(indice)`).



Différence majuscule/minuscule

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

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

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

Motifs mémorisés

Il est possible de récupérer la liste des motifs mémorisés :

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

On obtient :

```
10
56
38
```



Décomposer une ligne

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

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

```
liste = ['Mon', 'chat', "s'appelle", 'Neko']
print liste
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 de la liste.



20 Les fonctions : définition & arguments

49

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

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

Les paramètres de la fonction peuvent être :

- ▷ nommés et recevoir des valeurs par défaut :
- ▷ être donnés dans le *désordre* et/ou pas en totalité.

Ceci est très utile pour les objets d'interface graphique comportant de nombreux paramètres dont seulement certains sont à changer par rapport à leur valeur par défaut.

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

Ce qui donne :

5.0
3.33333333333
6.75

Pour conserver les modifications d'une variable définie à l'extérieur de la fonction, il faut utiliser «`global`» :

```
variable_a_modifier = "toto"
def modif(nom):
    variable_a_modifier = nom
modif("titi")
print variable_a_modifier
```

Ce qui donne :

toto

```
variable_a_modifier = "toto"
def modif(nom):
    global variable_a_modifier
    variable_a_modifier = nom
modif("titi")
print variable_a_modifier
```

Ce qui donne :

titi



Plusieurs valeurs de retour

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

6 5
(30, 11)

Une *lambda expression* ou un objet fonction anonyme

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

10

En combinaison avec la fonction `filter` qui applique une fonction anonyme sur chaque élément d'une liste et retourne la liste des résultats :

```
└── xterm └──
>>> filter(lambda x: x, [0,1,1,1,0,0])
[1, 1, 1]
```



21 Le contrôle d'erreur à l'exécution

51

Une forme simple de contrôle d'erreur est introduite grâce à la fonction assert :

```
assert un_test, une_chaine_de_description
```

Fonctionnement de l'assert

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

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

Exception: mauvaise valeur pour a

Ce mécanisme permet de faire de la programmation par contrat :

- ▷ la fonction s'engage à fournir un résultat si les conditions sur les variables d'entrées sont respectées : plus de contrôle ⇒ un programme plus sûr ;
- ▷ les conditions des «assert» écrites dans le source fournissent des informations sur les données : le programme est mieux documenté ;
- ▷ une fois le programme testé, le mécanisme d'assert peut être désactivé afin de supprimer les tests et leur impact sur le temps d'exécution du programme.



Choix d'une valeur numérique aléatoire dans un intervalle

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

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

2769369623

Choix aléatoire d'une valeur depuis un ensemble de valeurs

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

'B'



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)
```

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



24 Gestion de processus : lancer une commande externe et récupérer son résultat 54

Exécution et récupération de la sortie d'une commande : le module subprocess

```
1 import subprocess  
2 l = subprocess.check_output('ls *.py', shell=True) # Recupere la sortie d'une commande
```

Communication en Entrée/Sortie avec une commande :

```
2 cmd_ext = subprocess.Popen('wc -l', stdin=subprocess.PIPE, stdout=subprocess.PIPE, shell=True)  
3 cmd_ext.stdin.write("Bonjour tout le monde")  
4 cmd_ext.stdin.close() # important pour demarrer le travail de la commande  
5 print cmd_ext.stdout.read()
```

- ▷ 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 ;
- ▷ ligne 3, on envoie une ligne de texte à la commande qui s'exécute en multi-tâche ;
- ▷ 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 ;
- ▷ ligne 5, on récupère le résultat de son travail et on l'affiche.

La méthode «communicate»

```
2 cmd_ext = subprocess.Popen('wc -w', stdin=subprocess.PIPE, stdout=subprocess.PIPE, shell=True)  
3 (sortie_standard,sortie_erreur) = cmd_ext.communicate("Bonjour tout le monde")  
4 if (cmd_ext.returncode != 0): # on teste la valeur de succes de la commande  
5     print sortie_erreur  
6 else :  
7     print sortie_standard
```

Attention

La méthode `communicate`:

- transmets les données à la commande externe sur `stdin` et ferme `stdin` de la commande externe ;
- lit toutes les sorties de la commande externe et attend la fin de l'exécution de la commande ;
- retourne la sortie complète de la commande externe et sa valeur de succès.



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.

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

Gestion des arguments du processus

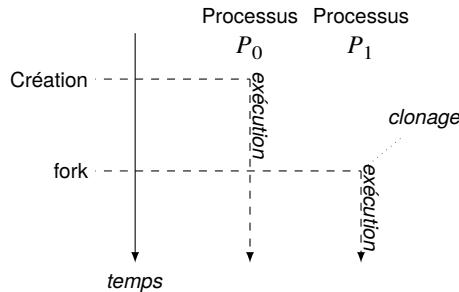
Pour récupérer la liste des arguments du script (nom du script compris) :

```
import sys
print sys.argv
```

Si on lance ce script :

```
□ — xterm —
$ ./mon_script.py -nom toto
['mon_script.py', '-nom', 'toto']
```

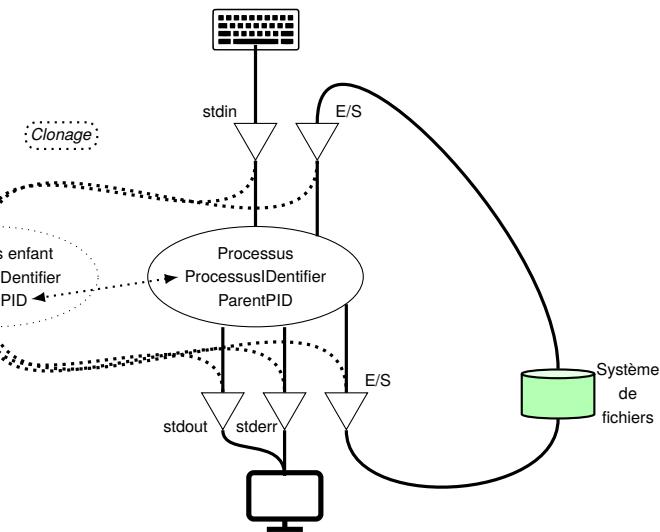




Les deux processus P_0 et P_1 :

- sont **identiques** : ils partagent le même code, leurs entrées sorties, les fichiers ouverts **avant le fork** ;
- sont **différenciés** par la valeur de retour de la fonction `fork` :
 - ◊ le processus P_0 , père, reçoit le PID du fils ;
 - ◊ le processus P_1 , enfant, reçoit la valeur 0 (il peut consulter son PPID pour connaître le PID de son père) ;
- sont **indépendants** : **après le fork**, toute entrée/sortie créée n'est **pas partagée**.

- Le processus P_0 est créé et s'exécute.
- Lors de l'exécution de l'instruction `fork` : il est «**clonné**» en un nouveau processus P_1 .
- P_1 s'exécute en concurrence de P_0 .



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 en lecture (SHUT_RD), en écriture (SHUT_WR) 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 retourne le nombre d'octets effectivement envoyés
sendall (data)	bloque jusqu'à l'envoi effectif de data ⇒ <i>C'est cette opération qu'il faut utiliser car elle envoi immédiatement les données !</i>



- On appelle :
- «**client**» le programme qui **demande** à établir la communication (connexion) ;
 - «**serveur**» le programme qui **attend** la demande de connexion du client.

L'élément logiciel qui représente la communication est une «socket» ou prise.

Le système d'exploitation doit allouer un SAP, «Service Access Point», ou *point d'accès au service* pour chaque socket utilisée pour une communication (le *service* correspond au processus associé).

Le **SAP** est l'association de l'adresse IP de la machine et d'un «numéro de port».

Il est ainsi possible de faire du «multiplexage» : une seule adresse IP mais plusieurs communications simultanées.

Du point de vue du serveur ce numéro de port doit être :

- ◊ libre (non déjà utilisé par un autre processus) : *il identifie le processus lié à la communication* ;
- ◊ connu du client : *pour que le client sache quel port demander*.

Les différentes étapes pour l'établissement de la communication :

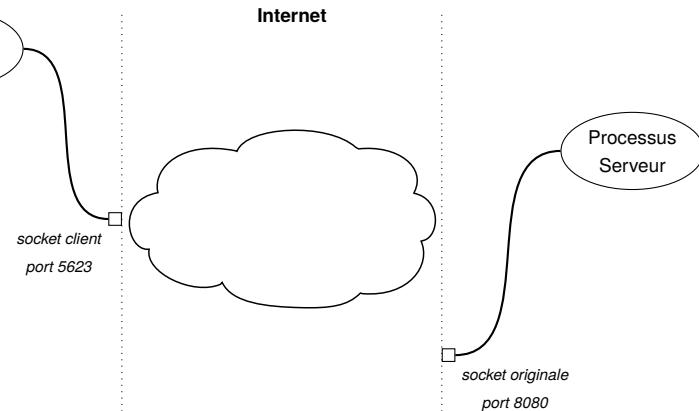
1. Le serveur attend sur le SAP :

[@IP serveur, numéro de port]
 `socket, bind, listen, accept`

Ici, le port choisi par le serveur est le 8080.

Ce port devra être connu du client.

Le client obtient automatiquement un numéro de port libre (par ex. 5623) `socket`.



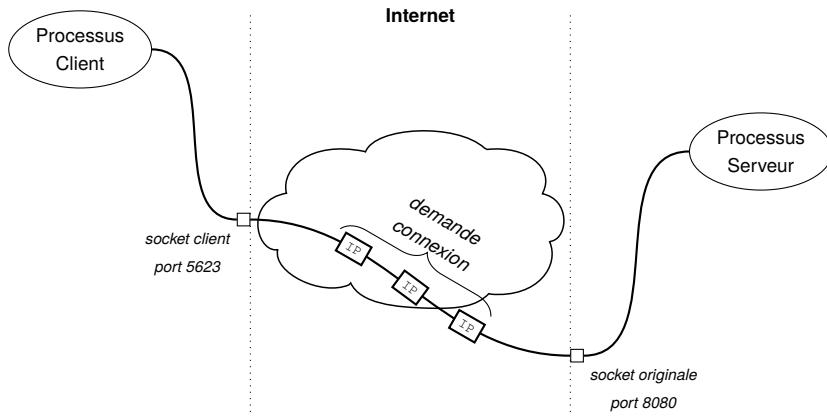
Description des instructions :

- ▷ `socket` : demande au système d'exploitation d'autoriser une communication ;
- ▷ `bind` : accroche la socket à un numéro de port (inutile sur le client) ;
- ▷ `listen` : configure le nombre de communications simultanées sur le serveur.



Le modèle «client/serveur» avec TCP

59

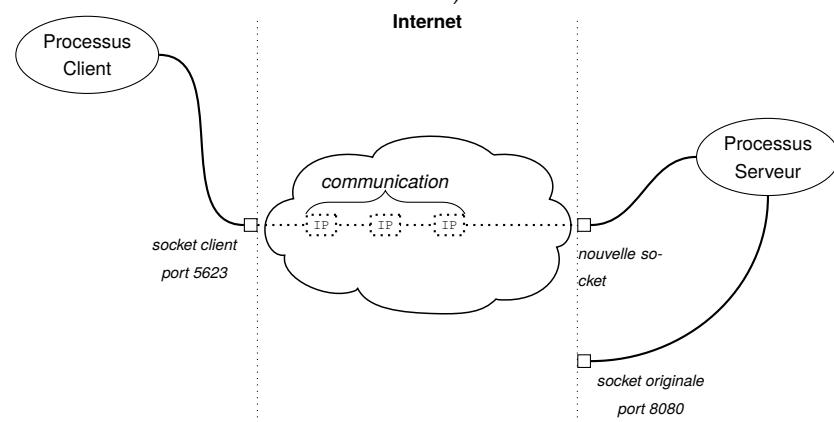


2. Le client se connecte au serveur `connect`

Le système d'exploitation du client et du serveur, mémorise la communication par un couple (SAP client,SAP serveur) :

SAP client	↔	SAP serveur
(@IP client: 5623)	↔	(@IP serveur: 8080)

Cette communication peut être affichée avec la commande Linux «`ss -tna`» (état ESTABLISHED).



Programmation d'un client en TCP

```
import os,socket,sys

adresse_serveur = socket.gethostname() # realise une requête DNS
numero_port = 6688
ma_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try: ma_socket.connect((adresse_serveur, numero_port))
except Exception, description:
    print "Probleme de connexion", description
    sys.exit(1)
while 1:
    ligne = ma_socket.recv(1024) #reception d'une ligne d'au plus 1024 caracteres
    if not ligne: break
    else:
        print ligne
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.



Programmation d'un serveur en TCP

```
import os,socket,sys

numero_port = 6688
ma_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP)
ma_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
ma_socket.bind(('', numero_port)) # équivalent à INADDR_ANY
ma_socket.listen(socket.SOMAXCONN)
while 1:
    (nouvelle_connexion, TSAP_depuis) = ma_socket.accept()
    print "Nouvelle connexion depuis ", TSAP_depuis
    nouvelle_connexion.sendall('Bienvenu\n')
    nouvelle_connexion.close()
ma_socket.close()
```

Utilisation de la méthode sendall

Pour envoyer **immédiatement** une ligne :

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

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

Attention

Si vous échangez entre un **client** et un **serveur**, et que l'un **attend de recevoir avant de répondre**, vous pouvez bloquer ces deux processus si l'envoi est retardé (bufférisé) et non immédiat.



Programmation Socket : TCP & bufférisation

62

- ▷ les échanges sur Internet sont **asynchrones** : les données échangées sont reçues et envoyées sous forme de datagramme IP, qui sont acheminés sans garanties d'ordre et de temps ;
- ▷ le système d'exploitation améliore la situation en introduisant un buffer à la réception et à l'envoi ;
- ▷ la programmation réseau est **synchrones**, mais **sur ces buffers** d'envoi et de réception.

Pour la réception

`donnees = ma_socket.recv(1024)` peut retourner moins d'octets mais pas plus de 1024.

Exemple :

- * la machine A envoie à la machine B, 30 lignes de texte pour un total de 920 octets ;
- * lors du transfert dans le réseau, ces 920 octets sont décomposés en un datagramme de 500 octets et un autre de 420 octets ;
- * lorsque B reçoit les 500 octets, le buffer de réception se remplit et les données sont mises à disposition du programme ;
- * la méthode `recv` ne retourne que 500 octets au programme.

Solution : tester le nombre d'octets obtenus en retour de l'appel à la méthode `recv`.

Pour l'envoi

`nb_octets = ma_socket.send(data)` les données sont mises dans le buffer d'envoi, et lorsqu'il y en aura suffisamment pour remplir un datagramme, ce datagramme sera réellement envoyé sur le réseau.

Il se peut que l'utilisation de la méthode `send` ne transmette rien sur le réseau et donc, le récepteur ne recevra rien et ne réagira pas.

Solution : utiliser l'instruction `sendall()` au lieu de `send()`, pour forcer l'envoi immédiat des données.

Les protocoles basés sur TCP utilisent le concept de ligne

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

Il faut vérifier les données reçues et éventuellement les concaténer aux suivantes pour retrouver une ligne complète.

Chaque ligne doit être envoyée immédiatement sans bufférisation.



Lecture d'une ligne de protocole orienté texte comme HTTP, SMTP, POP etc.

Lorsque l'on lit les informations reçues sur la socket TCP, on récupère des données découpées suivant des blocs de taille quelconque, on ne récupère pas ces données ligne par ligne depuis la socket.

Il est nécessaire de définir une fonction renvoyant une ligne lue caractère par caractère :

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

C'est la norme des protocoles sur Internet

Attention

Ce sera cette version que vous utiliserez dans les TPs.



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 !



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 concerne à une socket serveur ;
- ★ la présence de données à lire ;
- ★ la possibilité d'écrire sans être bloqué (le récepteur ne bloquera pas l'émetteur).

Il faut fournir en argument de `select` trois listes de descripteurs ou socket, correspondant à la catégorie des événements :

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

La fonction `select` retourne les trois listes modifiées, 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 survenu.

Au retour de la fonction, il suffit de parcourir le contenu de ces listes pour trouver les descripteurs/sockets à traiter (par exemple, trouver une socket où des données sont lisibles).

Pour détecter une demande de connexion, il faut comparer chaque socket de la liste `evnt_entree` à la socket utilisée pour faire l'opération «`accept`».



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):
            # il y a une demande de connexion
            nouvelle_connexion, depuis = ma_socket.accept()
            print "Nouvelle connexion depuis ",depuis
            nouvelle_connexion.sendall('Bienvenu')
            surveillance.append(nouvelle_connexion)
            continue
        # sinon cela concerne une socket connectee a un client
        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, etc
```



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 pour IPv4.



Serveur Web

Le serveur Web va servir le contenu du répertoire courant depuis lequel on lance la commande :

```
└── xterm ━━ pef@darkstar:/Users/pef $ python -m SimpleHTTPServer
  Serving HTTP on 0.0.0.0 port 8000 ...
  127.0.0.1 - - [06/Sep/2017 22:01:21] "GET / HTTP/1.1" 200 -
```

En utilisant la commande «curl», on peut lancer une requête vers le serveur :

```
└── xterm ━━ pef@darkstar:/Users/pef $ curl -v http://127.0.0.1:8000/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 8000 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1:8000
> User-Agent: curl/7.54.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: SimpleHTTP/0.6 Python/2.7.13
< Date: Wed, 06 Sep 2017 20:01:21 GMT
< Content-type: text/html; charset=utf-8
< Content-Length: 20596
<
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
```

Pour disposer d'un serveur Web avec exécution de scripts CGI dans le sous-répertoire `cgi-bin/`:

```
└── xterm ━━ python -m CGIHTTPServer
```

Il faut que les scripts soient exécutables (`chmod +x`) et qu'ils produisent en sortie un contenu adapté (HTML, JSON).



Pour traiter des résultats au format JSON

Dans le shell :

La commande «curl» va récupérer les données au format JSON et ces données vont être formatées par le module «json.tool» :

```
└── xterm
pef@darkstar:/Users/pef $ curl -sH 'Accept: application/json'
http://api.icndb.com/jokes/random | python -m json.tool
{
    "type": "success",
    "value": {
        "categories": [
            "nerdy"
        ],
        "id": 543,
        "joke": "Chuck Norris's programs can pass the Turing Test by staring at the interrogator."
    }
}
    "tx_index": 1939114,
    "ver": 1,
    "vin_sz": 3,
    "vout_sz": 2,
    "weight": 2468
}
],
"ver": 1
}
```

Dans un programme :

```
└── xterm
import json
import urllib2

url = 'http://api.icndb.com/jokes/random?limitTo=[nerdy]'
response = urllib2.urlopen(url)
data = response.read()
values = json.loads(data)
print values['value']['joke']
```

Le format JSON

Le format JSON est un format très utile pour utiliser des APIs de services Web et le développement de μservices basés sur l'architecture REST.

Cette architecture REST peut être définie par les 5 règles suivantes :

1. l'URI comme identifiant des ressources (URL explicite).
2. les verbes HTTP comme identifiant des opérations (GET, POST, PUT DELETE).
3. les réponses HTTP comme représentation des ressources (JSON).
4. les liens comme relation entre ressources (définition de l'attribut «rel» en accord avec l'IANA).
5. un paramètre comme jeton d'authentification,



26 Multithreading – Threads

70

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
```



Multithreading – Sémaphores

71

La classe semaphore, pour la protection des accès concurrents

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 séaphore, bloque si la séaphore est à zéro
release ()	augmente la valeur de la séaphore

Exemple d'utilisation

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



27 Manipulations avancées : système de fichier

72

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) :

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

Pour connaître la nature d'une entrée du répertoire :

```
import os.path
if os.path.exists("chemin_fichier") : # l'entrée existe
if os.path.isfile("chemin_fichier") : # c'est un fichier
if os.path.isdir("chemin_fichier") : # c'est un répertoire
taille = os.path.getsize("chemin_fichier") # pour obtenir la taille d'un fichier
```



La réception de datagramme UDP en multicast

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 :

```
import struct
mcast = struct.pack("4sl", socket.inet_aton("224.0.0.127"), socket.INADDR_ANY)
ma_socket.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mcast)
# configurer le nombre de routeurs traversables
ma_socket.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 2)
# si on veut recevoir son propre envoi
ma_socket.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_LOOP, 1)
```

Obtenir l'adresse IP de la machine que l'on utilise

```
mon_nom_symbolique = subprocess.check_output(['uname', '-a']).split()[1]
mon_adresse_ip = socket.gethostbyname(socket.gethostname(mon_nom_symbolique))
```

Scapy

Le module `scapy` dispose de capacités à traiter le contenu des paquets reçus suivant le protocole associé.

Cette bibliothèque d'injection de *paquets forgés* dispose de fonctions d'analyse et d'affichage de données de différents protocoles : DNS(), IP(), UDP(), etc.

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



La programmation objet : définition de classe et introspection

74

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)

```
class ma_classe(object):      #herite de la classe object
    variable_classe = 10
    __init__(self):           # deux caractères underscore _
        self.variable_instance=10
    def une_methode(self):
        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()`.

```
print dir([])

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



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» :

```
□ — xterm —
$ ./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 :

```
□ — xterm —
$ ./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 :

```
□ — xterm —
$ ./ma_commande.py -l mon_fichier.txt
```

- ★ à un booléen :

```
□ — xterm —
$ ./ma_commande.py -c
```

- ◊ les options peuvent être combinées :

```
□ — xterm —
$ ./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 :

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

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



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`» :

```
□ — xterm —  
$ python -i mon_programme.py
```

Sur le programme de génération de valeurs aléatoires :

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

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

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

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



Débogage avec le module «pdb», «Python Debugger»

On peut lancer un programme Python en activant le débogueur :

```
xterm
$ 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 :

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

```
xterm
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)
```

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

79

Le «logging» 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 débogage n'est plus nécessaire ;
- ◊ contrôlable suivant un niveau plus ou moins détaillé :

logging.CRITICAL

logging.ERROR

logging.WARNING

logging.INFO

logging.DEBUG

Dans le cours nous nous limiterons au seul niveau DEBUG.

- ◊ contrôlable par une ligne dans le source :

```
1#!/usr/bin/python
2import logging
3
4logging.basicConfig(level=logging.DEBUG)
5...
6logging.debug('Ceci est un message de debogage')
```

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

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

Pour désactiver le débogage, il suffit de modifier le programme en changeant la ligne 4 :
logging.basicConfig()

- ◊ possibilité de renvoyer les sorties de débogage vers un fichier :

```
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 :

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

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

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

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

À l'exécution :

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



29 Pour améliorer l'expérience de Python

81

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 :

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

Ainsi le script sera exécuté à chaque démarrage de Python.

a	logging 79–80 dictionnaire 29 données structurées 43 chaîne de format 44	i
accents 10, 19		itération for 23 for(;;) 23 rupture 14 while 14
aide 8		
aléatoire 52		
c	e	l
caractère 16 chaîne 17, 28 chaîne&liste 28	entrée/sortie input 33 print 32 raw_input 33	liste 20 construction avancée 25
commentaire 10	espace de nom 31	file 22
condition 13 assert 51 expression logique 15	exécuter code 9 commande 54 processus (fork) 55–56	pile 22
contrôle d'erreur assert 51	expression régulière 45 motif 46 split 48	tableau 27
exception raise 39 try 38		
conversion ascii 34 binaire 34 hexadécimale 35–36	f fichier lecture 41–42 ouverture 40	o objet classe 74
d	taille 72 type 72	opérateur affectation 37 incrémentation 37
débogage débogueur 78 interactif 77	fonction 49–50	options ligne commande 75–76



p
pointeur **12**

programmation socket

select **65–66**

TCP **57, 62**

TCP (client) **60**

TCP (lecture ligne) **63**

TCP (non bloquant) **64**

TCP (serveur) **61**

UDP **67**

UDP&multicast **73**

t

tableau **27**

thread **70**

semaphore **71**

type **12**

v

variable **11**

w

Web **68**

JSON **69**

