Faculté des Sciences & Techniques

Université de Limoges

Master 1ère année

Version 3

Mon'tit Python

—

P-F. Bonnefoi

Version du 15 septembre 2025

Table des matières

Parce qu'il est :

- **portable** : disponible sous toutes les plate-formes (de Unix à Windows, en passant par des systèmes embarqués avec MicroPython) ;
- **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 évoluées 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 utilisées comme argument d'une autre fonction, il dispose aussi de lambda expression (fonction anonyme), de générateur etc*.
  - **programmation asynchrone** : boucle d'événement, co-routines et E/S asynchrones ;
  - **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*.

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 est associée à 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 (WebSockets HTML5, templating avec Django), inter-opérabilité avec des BDs (SQL et NoSQL), des middlewares objets (SOAP/COM/CORBA/AJAX), de micro-services avec Flask/Bottle et Nginx, **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, ...).
- ▷ Il existe une version légère, MicroPython, pour l'embarqué et l'IoT, «*Internet of Things*» ;
- ▷ Et il permet de faire de la **programmation réseaux** !

Dans le cadre des différents modules réseaux

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.*
L'utilisation de bibliothèques pour résoudre les problèmes de TPs est formellement déconseillée pour débuter !

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 «`python3`» pour passer en mode interactif : entrer du code et en obtenir l'exécution, utiliser les fonctions intégrées (*builtins*), charger des bibliothèques *etc*

xterm
```
pef@borg:~$ python3
Python 3.12.3 (main, Jul 31 2024, 17:43:48) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 10+20
30
>>> _
30
>>> _*2
60
>>>
>>> help()
```

■→ *La variable «_» mémorise* ***automatiquement*** *le résultat obtenu précédemment.*
L'utilisation répétée de la touche «tabulation» *permet de* ***compléter*** *le nom des éléments dans l'interprète :*

xterm
```
>>> h[tab→|tab→]
hasattr(  hash(     help(     hex(
>>> h
```

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 :

```
xterm
$ python3 mon_source.py
```

prompt du shell

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

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

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

```
1 #!/usr/bin/python3
```

   - ◇ on lance directement le programme :

```
xterm
$ ./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 «`python`» à la place de «`python3`».

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 qui regroupent des instructions ensembles

Les blocs d'instruction sont matérialisés par des **indentations** : *plus de « { » et « } » comme en C !*

```
1 #!/usr/bin/python3
2
3 # les modules utilisés
4 import sys, socket
5 # le source utilisateur
6 if (a == 1) :
7     # sous bloc
8     # indenté (espaces ou tabulation)
```

le «:»

Le caractère « **:** » sert à introduire les blocs.

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.

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 = 'entité chaîne de caractères'
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 !*»

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

Il existe des valeurs prédéfinies :

`True`	valeur booléenne vraie
`False`	valeur booléenne faux
`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()` (ce qui permet de savoir si deux alias référencent 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>
    <instructions3>
elif <test2>:
    <instructions4>
else:
    <instructions5>
```

bloc d'instructions

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 <condition>:
    <instructions1>
    <instructions2>
else :
    <instructions3>
```

jamais utilisé

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

■→ *La boucle «`for`» sera introduite plus loin dans le support pour le parcours des **listes**.*

Les ruptures de contrôle

`continue`	**saute** 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** !

Dans l'exemple, `<condition2>` n'est évaluée que si `<condition1>` n'est pas vraie.

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)*

xterm

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

Arithmétique

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

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

xterm

```
>>> 2//3
0
>>> 2/3
0.6666666666666666
```

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 chaîne avec des 'quotes'" # ou 'une chaîne 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'` qui correspond au «*retour à la ligne*», «*newline*», 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 faire précéder la chaîne de `r` (pour *raw*) :

xterm

```
>>> une_chaine = r'pour passer à la ligne il faut utiliser \n dans une chaine'
>>> une_chaine
'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 chaines de caractères avec l'opérateur + :

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

Table de codage : concept et opérations

À chaque caractère ou symbole est associé un **rang** dans une table de codage :

- ▷ la fonction `ord()` permet d'obtenir le **rang** d'un caractère ou symbole dans la **table de codage** ;
- ▷ la fonction `chr()` permet d'obtenir le **caractère** ou **symbole** à partir de son rang dans la **table de codage** ;

Table de codage : composition

Deux **contraintes opposées** :

- □ la table de codage doit **être petite** : **consommer peu de place** en mémoire ou lors des transferts réseaux ;

  Exemple de table de codage pour les 26 lettres de l'alphabet latin :

a:0	b:1	c:2	d:3	e:4	f:5	g:6	h:7	i:8	j:9
k:10	l:11	m:12	n:13	o:14	p:15	q:16	r:17	s:18	t:19
u:20	v:21	w:22	x:23	y:24	z:25				

  26 symboles $\Longrightarrow 2^4 = 16 < 26 < 2^5 = 32$
  $\Longrightarrow$ il faut 5 bits pour coder en binaire tous les rangs de la table et il reste 6 symboles supplémentaires.

- □ la table de codage doit **être grande** : elle doit contenir **l'ensemble des symboles** pour exprimer :
  - ◇ des langues basées sur un **alphabet** et utilisant des diacritiques (français, grec, arabe, russe, sanskrit, *etc.*)
  - ◇ des langues basées sur des **idéogrammes** comme le chinois ou le japonais ;
  - ◇ d'autre symboles utilisées couramment : pictogrammes, © , ®, *etc.*

Table de codage : compromis entre occupation mémoire et nombre de symboles

Codage des rangs sur 8 bits, ou un octet $\Longrightarrow 2^8 = 256$ rangs possibles dans la table de codage : *de la place pour l'alphabet latin en minuscule, majuscule, des nombres, des symboles comme les parenthèses, les accolades, les opérateurs arithmétiques,* etc.
$\Longrightarrow$ codage **ANSI** ou **ASCII** : c'est le choix le plus courant pour les ordinateurs jusqu'à il y a quelques années.

Table de codage : l'avenir ? une table de codage universelle et un codage du rang extensible

- ▷ normaliser et internationaliser le classement des symboles : **Unicode**, «*Universal Coded Character Set*» ;
- ▷ utiliser un **rang extensible** sur :
  - ◇ un octet, ou 8 bits, pour les caractères les plus courants utilisés dans la programmation et dans les «*vieux*» ordinateurs ;
  - ◇ plusieurs octets pour les rangs suivants associés aux différents symboles de toutes les langues.

$\Longrightarrow$codage **UTF-8**, *Unicode (or Universal Coded Character Set) Transformation Format – 8-bit*.

Mais alors ? Un octet c'est quoi ?

Un **octet** ou 8 bits peut être :

- une valeur numérique :
  - entier non signé de 0 à 255 ;
  - entier signé de -128 à 127 ;
- un symbole dans un codage sur 8 bits comme l'ANSI ou l'ASCII ;
- un symbole dans le codage UTF-8 s'il correspond aux symboles usuels ;
- une portion de codage UTF-8 pour des symboles de langues tenant sur **plusieurs octets** ;

Qui décide ?

- ▷ les opérations du **langage de programmation** que l'on utilise : utilise-t-il le codage UTF-8 ?
- ▷ **l'ordinateur** que l'on utilise : sous Linux le codage est en UTF-8 ;
- ▷ le **programmeur** quand il décide de manipuler ces octets avec la connaissance de leur origine, de leur sémantique et des opérations de conversion qu'il applique dessus ;

Peut-on faire des erreurs ?

Oui !

Peut-on prendre «connaissance» d'un octet sans décider de ce qu'il est ?

On utilise la **notation hexadécimale** associée à un interprétation sous forme de **symbole en codage ANSI** (c'est l'interprétation minimale historique).

■→ *Sous shell Linux, on utilisera la commande* `xxd` *et sous Python la fonction d'affichage générique* `print()`.

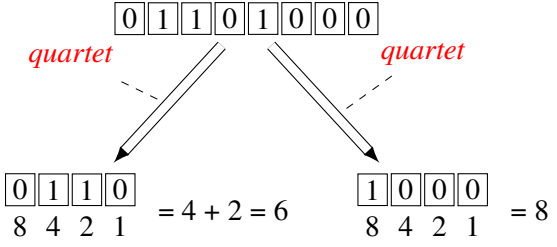
Notation et format

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 : 8 7 0 0 0

Et du côté d'un ordinateur ?

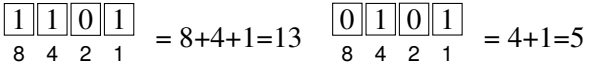
Qu'est-ce qu'un octet ? Le plus petit format utilisé et manipulé, soient 8bits, c-à-d ☐☐☐☐☐☐☐☐ rempli de `0` ou de `1`.
Comment le noter pour être facilement manipulable par un humain ? En utilisant la **notation hexadécimale** :



Ce qui donne `68` en notation hexadécimale.

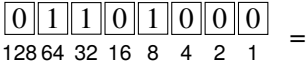
Comme la somme est comprise entre 0 et 15, on utilise les chiffres de 0 à 9 pour leur valeur respective, et les lettres de A à F pour les valeurs de 10 à 15.

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



soit la séquence binaire `11010101`.

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



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).

Codage étendu sur 8 bits «*à l'ancienne*» : un symbole tient sur un octet

Codage «ANSI» ou «ASCII» sur 7 bits de 0 à 127 et on complète celles de 128 à 255 sur 8 bits avec, au choix :

- ▷ codage «ISO-8859-1» ou «LATIN1» : permet de coder les caractères accentués des pays d'Europe de l'ouest. *Il ne permet pas de coder le symbole de l'euro «€», ni le «œ».*
- ▷ codage «Windows-1252» : permet de coder l'ensemble des symboles du français ainsi que le symbole de l'euro.

Codage UTF8 : économique et rétro-compatible : un symbole tient sur un octet *au minimum*

- ☐ un caractère **non accentué** dont la valeur associée suivant le code ASCII ou ANSI est entre 32 et 127 (soient 7 bits) ⟹ codé sur 1 octet ;
- ☐ un caractère **accentué** ⟹ codé sur 2 octets ;
- ☐ un caractère **non latin** ou un **idéogramme** ⟹ codé 2,3 ou 4 octets (6 au maximum).

⟹*L'utilisation de caractères accentués change la taille des données et peut pertuber la manipulation des données sous forme d'octets.*

Sous le shell, la commande `echo` peut 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                                    ..
```
chaque point correspond à un octet non représentable

Convertir le texte d'UTF-8 vers le codage 1252 à l'aide de la commande `iconv` :

xterm
```
$ echo -n "é" | iconv -f UTF8 -t WINDOWS-1252 | xxd
00000000: e9                                      .
```

On notera que 'e9' en hexadécimal donne 'e'→14, soit $14 * 16 + 9 = 233$ *en décimal, soit une valeur sur 8 bits > 127. La conversion inverse est également possible (elle est obligatoire pour un affichage correct dans un environnement Unix configuré pour traiter de l'UTF8).*

Gestion des caractères accentués sous Unix : utilisation du codage UTF-8 pour l'unicode

Le codage UTF-8, «*Universal Coded Character Set + Transformation Format – 8-bit*» permet de conserver un codage sur un octet pour les caractères latins non accentués et de passer sur un codage sur deux, voire trois ou quatre octets, pour les autres symboles.

Exemples de codage UTF-8

xterm

```
$ echo -n "e" | xxd
00000000: 65                                       e
```

l'octet valant 65 est représentable

Le caractère «e» est codé avec la valeur de l'octet 65.

xterm

```
$ echo -n "é" | xxd
00000000: c3a9                                     ..
```

ces octets sont non représentables

Le caractère «é» est codé sur deux octets.

xterm

```
$ echo -n " 電腦 " | xxd
00000000: e99b bbe8 85a6                           ......
```

L'idéogramme «電» est codé sur 3 octets.

Les deux symboles 電腦 signifient «ordinateur».

xterm

```
$ echo -n '€' | xxd
00000000: e282 ac                                  ...
```

Le symbole euro «€» est codé sur 3 octets également.

Attention

Sous Python 3 **toutes les chaînes de caractères** sont en **unicode**.

xterm
```
>>> a = 'été'
>>> len(a)
3
```

Et comment gérer des octets ? chaîne d'octets

On utilise une notation spéciale :

xterm
```
>>> c = b'ete'
>>> c
b'ete'
>>> len(c)
3
>>> c = b'été'
  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
```

La chaîne contient des accents mais pas d'indication de codage !

On utilise le préfixe `b` *pour désigner une chaîne d'octets ou* «bytes».

Conversion vers une représentation octets et vice-versa

On utilise l'opérateur `bytes()` et son inverse l'opérateur `decode()` :

xterm
```
>>> rep = bytes(a,'utf-8')
>>> rep
b'\xc3\xa9t\xc3\xa9'
>>> len(rep)
5
>>> b'\xc3\xa9t\xc3\xa9'.decode('utf-8')
'été'
```

5 caractères : `'\x c3' '\x a9' 't' '\x c3' '\x c9'`

La notation `'\x ..'` *fournit l'octet sous forme hexadécimal*

passer d'une chaîne d'octets à une chaîne en UTF-8 avec la méthode `decode()`

xterm

```
>>> ord('é') ❶
233

>>> len('e')
1
>>> len('é')
1

>>> hex('é')❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object cannot be
interpreted as an integer

>>> hex(ord('é'))❸
'0xe9'

>>> '{0:X}'.format(ord('é'))❹
'E9'

>>> print(bytes('é', 'utf-8'))❺
b'\xc3\xa9'

>>> bytes('é','utf-8').hex()
'c3a9'
```

Attention

❸⟹ différent de l'UTF8.

UTF8 est un codage, pas le rang du caractère en unicode.

Première méthode : utilisation du rang Unicode

❶⇒la fonction `ord` renvoie le rang du caractère unicode.

❷⇒on voudrait obtenir le rang du caractère en notation hexadécimale

⟹erreur !

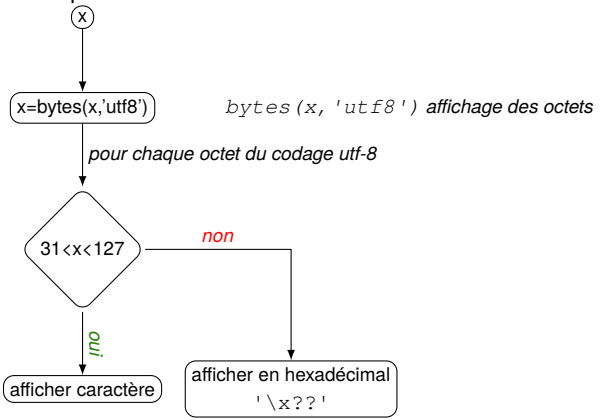
❸⇒il faut d'abord obtenir le rang, puis demander sa notation en hexadécimal !

❹⇒la fonction `format()` peut être utilisée pour obtenir la notation hexadécimale.

Seconde méthode : utilisation du codage UTF-8

❺⇒on convertit le symbole en octet, que l'on affiche avec `print()` :

⟹l'affichage des symboles d'une chaîne d'octets par «`print()`» suit la procédure de décision suivante :



Pourquoi obtient-on un «é» au final à l'écran alors qu'on a deux plusieurs octets de codage ?

Parce que le terminal sait traiter l'unicode, c-à-d le codage des deux octets vers un 'e' avec un accent aigu.

Chaîne d'octets vers chaîne de caractères

```
>>> ma_chaîne_octets = b'toto'
>>> ma_chaîne = str(b'toto', encoding = 'utf8')❶
>>> ma_chaîne = str(b'toto', 'utf8')
>>> print(ma_chaîne)
toto
```

❶ *Le nom du paramètre* `encoding` *peut être omis.*

⟹ *Ici, pas de différence car les octets restent codés en UTF8 (c'est son intérêt).*

Chaîne d'octets vers chaîne de caractères

```
>>> ma_chaîne = 'été'
>>> ma_chaîne_octets = bytes('été', encoding = 'utf8')❷
>>> ma_chaîne_octets = bytes('été', 'utf8')
>>> print(ma_chaîne_octets)
b'\xc3\xa9t\xc3\xa9'
```

❷ *Le nom du paramètre* `encoding` *peut être omis.*

⟹ *Ici, on voit que le l'encodage UTF8 utilise plus d'octets*

xterm

```
>>> b'ù'
  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
>>> ord('é')
233
>>> ord('ù')
249
>>> ord(b'a')
97
>>> ord('a')
97
>>> ord('ù')
249
```

Seule une chaîne de caractères peut contenir des accents.

`ord()`

- fonctionne sur un caractère :
  - de chaîne de caractère
  - de chaîne d'octets
- retourne un entier.

`chr()` **ne retourne qu**'une chaîne de caractères.

Pour obtenir une **chaîne d'octet**, au choix :

- `bytes([val])`.
- `b'%c'%val`

xterm

```
>>> chr(233)
'é'
>>> chr(61)
'='
>>> bytes([61])
b'='
>>> bytes([233])
b'\xe9'
>>> b'%c'%61
b'='
```

xterm

```
>>> 'a'.hex()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'hex'
>>> b'a'.hex()
'61'
>>> bytes('é','utf8')
b'\xc3\xa9'
>>> bytes.fromhex('61')
b'a'
>>> bytes('é','utf8').hex()
'c3a9'
```

`.hex()` ne fonctionne que sur une chaîne d'octets.

La conversion d'une chaîne de caractères en une chaîne d'octets nécessite la donnée du codage à utiliser (ici, `'utf8'`).

`hex()` retourne une chaîne de caractères contenant la représentation hexadécimale.

Conversion chaîne vers valeur numérique : **recommendations**

Caractère unicode vers valeur numérique ⟹ déconseiller mais possible

```
xterm
>>> ord('A')
65
>>> ord('œ')
339
>>> ord('€')
8364
>>>
```

>254 ⟹ ne tient pas sur un octet

>254 ⟹ ne tient pas sur un octet

```
xterm
>>> chr(8364)
'€'
>>> chr(339)
'œ'
>>> chr(65)
'A'
```

Caractère octet vers valeur numérique ⟹ facile et «*à l'ancienne*»

```
xterm
>>> b'A'[0]
65
>>> b'%'[0]
37
>>> b'z'[0]
122
```

```
xterm
>>> bytes([122])
b'z'
>>> bytes([37])
b'%'
>>> bytes([65])
b'A'
```

La stratégie pour gérer les données quelconques par octet ? deux possibilités

chaîne unicode ou chaîne d'octets → notation hexadécimale → *la plus portable*

- ☐ manipulation décimale
- ☐ manipulation par groupe d'octets

2 digits hexa⟹1 octet

chaîne unicode → chaîne d'octets codage UTF8 → *la plus directe*

- ☐ manipulation décimale
- ☐ manipulation par groupe d'octets

1 symbole⟹1 octet

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

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

format	**affiche**	**argument**
`%s`	chaîne de caractères, en fait récupère le résultat de la fonction `str()` appliquée sur l'argument	quelconque
`%f`	valeur flottante, par ex. `%.2f` pour indiquer 2 chiffres après la virgule	flottant
`%d`	un entier	entier
`%x`	entier sous forme hexadécimal	entier
`%c`	caractère	rang du caractère

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) s'ils sont présents
Exemple : `chaine.rstrip('\n ')`

`upper` passe en majuscule
Exemple : `chaine.upper()`

`splitlines` décompose une chaîne suivant les `\n` et retourne une liste des «lignes», c-à-d une liste de chaîne (caractères ou octets).

etc.

Remarque : ces méthodes fonctionnent aussi sur des chaînes d'octets : `b'la ligne\n'.splitlines()`.

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'
```

Formattage des données, exemple pour sur la notation hexadecimal

Ajouter des zéros devant pour obtenir une longueur fixe :

```
xterm
>>> "{:06x}".format(42)
'00002a'
>>> "{:06X}".format(42)
'00002A'
>>> "{:04X}".format(42)
'002A'
```

En utilisation directe :

```
xterm
>>> format(255, '02X')
'FF'
>>> format(2, '02X')
'02'
>>> format(10,'08b')
'00001010'
```

- ▷ `{` identifiant de début de format
- ▷ `:` indication de formatage
- ▷ `0` compléter par des zéros...
- ▷ `6` compléter par autant de zéros qu'il faut pour une valeur sur 6 digits
- ▷ `x` hexadécimal avec caractères minuscules
- ▷ `X` hexadécimal avec caractères majuscules
- ▷ `b` permet d'obtenir la **notation binaire** d'une valeur
- ▷ `}` identifiant de fin de format.

Remarques : «`format()`» ne fonctionne que sur des **chaîne de caractères** (unicode).

Deux types de listes

- celles qui ne peuvent être modifiées après leur définition, 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 :

xterm

```
>>> a = ('un', 2, 'trois')
>>> a
('un', 2, 'trois')
>>> len(a)
3
```

Une liste d'un seul élément, `('un')`, correspond à l'élément lui-même, `'un'`.

■→ *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]
```

Quelque méthodes supportées par une liste :

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

Listes — Exemples d'utilisation

Ajout d'un élément à la fin

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

Retrait du dernier élément

xterm
```
>>> element = a.pop()
>>> print (element, a)
quatre [1, 'deux', 3]
```

Tri des éléments

Attention :

ils doivent être mutuellement comparables

xterm
```
>>> b = [1, 4, 3]
>>> b.sort()
>>> c = a.sort()
>>> c
>>> b
[1, 3, 4]
```

Attention : `sort()` ne retourne pas la liste

Inversion de l'ordre des éléments

xterm
```
>>> a.reverse()
>>> a
[3, 'deux', 1]
```

Retrait du premier élément

xterm
```
>>> print (a.pop(0))
3
>>> a
['deux', 1]
```


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`.

Accès aux éléments d'une liste ou d'un tuple : accès direct ou parcours

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

```
for un_élément in une_liste:
    print (un_élément)
```

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 élément
ma_liste[-2] # l'avant dernier élément
```

Extraire une «**sous-liste**» :

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

```
ma_liste[1:4]  # du deuxieme élément au 4ème élément ou de l'indice 1 au 3 (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** :

xterm

```
>>> l = list(range(1,4))       # de 1 à 4 non compris
>>> m = list(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]
```

l'utilisation de la fontion `list()` permet d'obtenir le contenu complet

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

xterm

```
>>> a = ['un','deux','trois']
>>> for i in range(0,len(a)):
...     print(a[i])
...
un
deux
trois
```

Affectations multiples & simultanées de variables

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

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

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

xterm
```
>>> 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.

xterm
```
>> '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

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

Utilisation avancée : création d'une liste à partir d'une autre

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') :

xterm
```
>>> [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]
```


En C :

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

En Python :

```
contenu = ((doc + '\n') if doc else '')
```

x = true_value if condition else false_value

Opérateur ternaire et les «*lists comprehension*»

▷ **Choix de l'élément à ajouter** : on ajoute le caractère si le code ANSI 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)
>>> list(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']
```

Pourquoi appliquer la fonction `list()` ? Pour forcer la création de la liste !

▷ **Ajout conditionnel d'un élément** : 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)
>>> list(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']
```

Remarque

Certaines fonctions Python comme `range()` font de l'évaluation paresseuse, «*lazy evaluation*», c-à-d qu'elles ne calculent leur résultat que lorsqu'il est nécessaire, d'où le `list(range())`.
Sinon `print(range(0,10))` *ne renverrait que* `range(0,10)`.

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.

Accèder 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]).

Accès par indice aux éléments d'une chaîne

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 :

xterm

```
>>> 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()` :

xterm

```
>>> a = list('le voiture')
>>> a[1] = 'a'
>>> print(a)
['l', 'a', ' ', 'v', 'o', 'i', 't', 'u', 'r', 'e']
```

Puis, recomposer la chaîne à partir de la liste de ses caractères avec `join()` :

xterm

```
>>> b = ''.join(a)
>>> print (b)
la voiture
```

indique le séparateur à insérer entre les caractères, ici il est vide ''

Et pour les chaînes d'octets ?

xterm
```
>>> chaine_octets = b'ABC'
>>> print(chaine_octets)
b'ABC'
>>> chaine_octets[0]
65
```

L'accès par indice dans une chaîne d'octets retourne le **rang du caractère** !

C'est équivalent à :

xterm
```
>>> ord(b'A')
65
```

Et pour la boucle `for` ?

xterm
```
>>> chaine_octets = b'ABC'
>>> print(chaine_octets)
b'ABC'
>>> for b in chaine_octets:
...     print(b)
...
65
66
67
>>>
```

Chaque élément parcouru par la boucle `for` est retourné sous forme de son **rang**.

Et les tranches de chaîne d'octets ?

xterm
```
>>> b'abcdef'[2:3]
b'c'
>>> b'abcdef'[2:5]
b'cde'
```

On obtient...une **chaîne d'octets**, y compris si la tranche ne contient qu'un **seul caractère** !

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 'trois' in dico:`
ajout ou modification	`dico['un'] = 1.0`
suppression	`del dico['deux']`
récupère la liste des clés	`les_cles = list(dico.keys())`
récupère la liste des valeurs	`les_valeurs = list(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])
```

Trier les éléments d'un dictionnaire

Attention

Les éléments d'un dictionnaire ne sont **pas triés** ! (sauf à partir de Python v3.7)
Ils ne sont **pas triés** dans l'ordre d'ajout.

$\Longrightarrow$ *Ils sont organisés pour un* ***accès rapide*** *par la clé.*

Tri suivant les clés

xterm
```
>>> dico={'a':1, 'f':10, 'e':3, 'd':5}
>>> list(dico.keys())
['a', 'f', 'e', 'd']
>>> les_clés = list(dico.keys())
>>> les_clés.sort()
>>> les_clés
['a', 'd', 'e', 'f']
```

Une fois les clés triées, on peut accéder à la valeur :

`dico[les_clés[1]]`

$\Longrightarrow$ accès à la deuxième valeur.

Tri suivant les valeurs

On récupère les «*tuples*» du dictionnaire avec la méthode `items` :

xterm
```
>>> liste_des_éléments_du_dico = list(dico.items())
>>> liste_des_éléments_du_dico
[('a', 1), ('f', 10), ('e', 3), ('d', 5)]
```

On utilise la fonction `sorted` en lui donnant une fonction d'accès à la clé de tri :

xterm
```
>>> sorted(liste_des_éléments_du_dico,key=lambda t:t[1])
[('a', 1), ('e', 3), ('d', 5), ('f', 10)]
```

La fonction d'accès est donnée par une «*lambda fonction*» ou une fonction anonyme définie en une ligne.

Couples de valeurs obtenus par combinaison de deux listes

xterm
```
>>> list(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, 'b': 2, 'c': 3}
```

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 pas associées à une valeur dans la seconde, sont supprimées :

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

`list()` permet d'obtenir le contenu complet pour l'affichage (évaluation paresseuse de `zip()`)

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
```


Installation de nouveaux modules et utilisation simplifiée

Pour installer un module, on utilise pip, «*package installer for python*» :

```
xterm
$ python3 -m pip install requests
```

Pour obtenir la liste des modules installés :

```
xterm
$ python3 -m pip list
```

Pour utiliser un «*alias*» pour l'espace de nom :

```
import plotly.graph_objects as go
fig = go.Figure(data=go.Bar(y=[2, 3, 1]))
fig.show()
```

l'alias «go» *remplace* «plotly.graph_objects»

Des modules qui peuvent être «*exécutés*» de manière autonome :

```
xterm
$ python3 -m http.server 8000 --directory ./a_partager
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

accéder par http://adresse_machine:8000/ au contenu du répertoire «a_partager»

Des «*one-liner*», c-à-d des programmes en une ligne de commande :

```
xterm
$ python3 -c "import sys, urllib.parse as ul; print(ul.unquote(sys.argv[1]))"
```

Permet de décoder les URLS contenant du «*percent-encoding*» (`%20` pour un espace, *etc*)

Utilisation de l'outil `pipx` : installation d'application écrite en Python

xterm
```
$ sudo apt install pipx
$ pipx install mon_application
```

Installation de bibliothèque dans des environnements virtuels : «*venv*», «*virtual environment*»

- ▷ éviter des conflits de version entre bibliothèques Python installées globalement dans l'OS ;
- ▷ disposer de différents contextes d'exécution : une version de Python + l'ensemble des bibliothèques ;

⟹ utilisation de `venv`

Depuis le shell :

xterm
```
$ mkdir MON_ENVIRONNEMENT
$ cd MON_ENVIRONNEMENT
$ python3 -m venv .venv
$ ls -l
drwxrwxr-x    5 pef pef 4.0K Sep  3 17:14 .venv/
```

Un répertoire `.venv` est créé : son nom commence par un «.», il sera caché par défaut.

Ensuite, **à chaque fois** que l'on veut utiliser l'environnement :

xterm
```
$ source .venv/bin/activate
```

Ainsi, toute installation de bibliothèques sera faite dans le répertoire caché « `.venv` ».

La fonction `print()` permet d'afficher de manière ***générique*** tout élément, que ce soit un objet, une chaîne de caractères, une valeur numérique, *etc* :

▷ le passage d'une liste permet de «*coller*» les affichages sur la même ligne :

xterm
```
>>> a = 'bonjour'
>>> c = 12
>>> d = open('fichier.txt', 'w')
>>> print (a)
bonjour
>>> print (c,d, 'la valeur est %d' % c)
12 <_io.TextIOWrapper name='fichier.txt' mode='w' encoding='UTF-8'> la valeur est 12
```
❶

`print()` affiche le contenu «affichable» *de l'objet* ❶.

▷ par défaut, elle ajoute un retour à la ligne après l'affichage des paramètres.
Si on ne veut pas de retour à la ligne après :
```
print('.', end='')
```

▷ si on veut forcer la sortie après l'affichage (utile lors de l'utilisation en programmation système avec un «*pipe*») :
```
print('.', end='', flush=True)
```

Utiliser explicitement les canaux `stdout` ou `stderr`

```
import sys

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

La fonction `input()`

Pour la saisie des données au clavier, la fonction `input()` retourne un **chaîne de caractères au format UTF-8**, que l'on convertira au besoin.

```
1 saisie = input("Entrer ce que vous voulez :") # retourne une chaîne de caractères
```

xterm

```
>>> saisie = input("Entrer ce que vous voulez :")
Entrer ce que vous voulez :Python 3 c'est super !
>>> saisie
"Python 3 c'est super !"
```

Pour obtenir une **valeur entière**, il est nécessaire de **convertir** la valeur saisie :

xterm

```
>>> saisie = input("Entrer une valeur :")
Entrer une valeur :10
>>> saisie
'10'
>>> int(saisie)
10
```

Si la chaîne de caractères rentrées contient autre chose que la valeur à convertir, une exception est levée :

xterm

```
>>> saisie = input("Entrer une valeur :")
Entrer une valeur :la valeur est 10 dans cet exemple
>>> int(saisie)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'la valeur est 10 dans cet exemple'
```

Attention

La chaîne retournée par la fonction « `input()` » ne contient pas le `'\n'`.

Un certain nombre de fonctions permettent 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 le rang utf8 d'un caractère non accentué	`ord('A')`
chr	retourne le caractère non accentué à partir de son rang utf8	`chr(65)`
str	convertit en chaîne	`str(10), str([10,20])`
int	interprète la chaîne en entier	`int('45')`
float	interprète la chaîne en flottant	`float('23.56')`
bytes	convertit une chaîne UTF-8 en chaîne d'octets	`bytes('toto','utf8')`

Conversion utf8 vers chaîne d'octet puis vers rang de chaque caractère

```
>>> bytes('é','utf8')
b'\xc3\xa9'
>>> list(bytes('é','utf8'))
[195, 169]
>>> [hex(x) for x in list(bytes('é','utf8'))]   # on vérifie que les valeurs sont les bonnes
['0xc3', '0xa9']
```

Conversion en représentation binaire

Convertir un nombre exprimé en format binaire dans une chaîne de caractères :

```
representation_binaire = format(204,'b') # retourne '11001100'
entier = int('11001100',2)    # on donne la base, ici 2, on obtient la valeur 204
representation_binaire = bin(204)[2:] # la fonction bin() donne '0b11001100' et on enlève '0b' avec la tranche
```

Vers la notation hexadécimale

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

```
xterm
>>> a='\x21'
>>> a
'!'
>>> ord(a)
33
```

```
xterm
>>> a=b'\x21'
>>> a
b'!'
>>> list(a)
[33]
```

- avec le type `bytes` et sa méthode `hex()`, on obtient la représentation hexadécimale de chaque caractère :

```
xterm
>>> bytes('été','UTF-8').hex()
'c3a974c3a9'
>>> b'abcABC\n\t'.hex()
'6162634142430a09'
```

Depuis la notation hexadécimale

- avec la méthode `fromhex()` du type `bytes` :

```
xterm
>>> bytes.fromhex('6162634142430a09')
b'abcABC\n\t'
```

- passer d'une valeur exprimée en notation hexadécimale vers un entier, puis en binaire avec `bin()` :

```
xterm
>>> int('ef6263c142430a09',16)
17249459204473948681
>>> bin(int('ef6263c142430a09',16))
'0b1110111101100010011000111100000101000010010000110000101000001001'
>>> int('0b1110111101100010011000111100000101000010010000110000101000001001',2)
17249459204473948681
```

Attention de bien faire la distinction entre ces différentes notations !

Mélange chaîne d'octet, «b' ' » et chaîne de caractères, « ' ' »

La conversion vers une représentation «*chaîne de caractères*»

xterm

```
>>> liste = [2, 3]
>>> str(liste)
'[2, 3]'
>>> bytes(liste)
b'\x02\x03'
>>> liste = [2, 'yop']
>>> bytes(liste)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object cannot be interpreted as an integer
```

représentation de la liste en chaîne de caractère

conversion de chaque entier en chr(entier)

ne marche pas car un élément de la liste n'est pas entier

xterm

```
>>> f=open('.bashrc','r')
>>> str(f)
"<_io.TextIOWrapper name='.bashrc' mode='r' encoding='UTF-8'>"
>>> bytes(f)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object cannot be interpreted as an integer
```

*convertir tout objet vers une chaîne de caractère **mais pas vers une chaîne d'octets***

Conversion vers un représentation «*chaîne d'octets*» pour la programmation réseau

```
desc.sendall(bytes(str(un_evenement.getpeername()),encoding='utf8')+b': '+ligne)
```

Ici, on a besoin pour le `sendall()` d'une chaîne d'octets :

❶⇒*conversion vers une chaîne de caractères ;*

❷⇒*conversion vers une chaîne d'octets ;*

❸⇒*la concaténation n'est possible qu'entre chaîne de caractères ou chaîne d'octets respectivement.*

Manipulation de données de taille fixe en octets

xterm

```
>>> chaine = b'ABcDEfghIJK\x03\x04'
>>> chaine
b'ABcDEfghIJK\x03\x04'
>>>
>>> [hex(c) for c in chaine]
['0x41', '0x42', '0x63', '0x44', '0x45', '0x66', '0x67', '0x68', '0x49',
'0x4a', '0x4b', '0x3', '0x4']
>>>
>>> [format(c,'02x') for c in chaine]
['41', '42', '63', '44', '45', '66', '67', '68', '49', '4a', '4b', '03', '04']
>>> hexes = [format(c,'02x') for c in chaine]
>>>
>>> ''.join(hexes)
'41426344456667684 94a4b0304'
>>>
>>>
>>> chaine.hex()
'41426344456667684 94a4b0304'
>>>
>>> int('41426344456667684 94a4b0304',16)
5170376580132657723804780004100
>>>
>>> format(5170376580132657723804780004100, '02X')
'41426344456667684 94A4B0304'
>>>
```

Soit une donnée sur 13 octets exprimée sous forme de caractères et aussi en hexa

la conversion avec `hex()` n'est pas intéressante

la conversion avec `format()` est parfaite

on peut facilement regrouper les octets en notation hexadécimale

une façon alternative plus simple et directe depuis `chaine`

et obtenir la valeur combinée sous forme d'entier par exemple

l'opération inverse étant possible

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'incrémentation

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)` ⟹ `'0b101'`

■→ La commande `format` :

```
representation_binaire = format(5,'b')     # ici, on obtient '101'
```

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 les `'0'` manquants :

```
rep_binaire = format(5,'08b')      # ce qui donne '00000101'
```

Pour la conversion en octet d'une séquence binaire :

```
caractere = bytes([int(rep_binaire,2)])    # attention il y a une liste [ ]
```

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 (celui qui a appelé le précédent).
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 (on retourne au contexte du shell qui a appelé le programme).

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 :

xterm
```
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 $\Longrightarrow$ pas universel.

Traiter une exception

Elles sont traitées, «*interceptées*», suivant leur nature :

```
nombre = 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 cet 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
```


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*

UTF-8 vs octets

Par défaut, l'ouverture ou la création d'un fichier est faite en codage UTF-8 : une lettre accentuée lue ou écrite est un symbole **unicode**.
Pour ouvrir ou créer le fichier au **format octet**, «*bytes*», on utilise le suffixe «`b`» : `"rb"`, `"wb"` ou `"ab"` : une lettre accentuée lue ou écrite donne au moins deux octets.

■→ Pour vérifier que l'ouverture du fichier se fait correctement, il faut **traiter une exception**.

Gérer l'exception et obtenir une description de l'erreur :

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

Ce qui peut produire : *Description de l'erreur*

```
[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.

Création d'un fichier en codage UTF-8 et lecture sous forme d'octets

```
#!/usr/bin/python3
import sys

try:
    f = open("fichier_test", "w")
except Exception as e:
    print(e.args)
    sys.exit(1)

f.write('été')
f.close()

f = open("fichier_test", "rb")
ligne = f.readline()
print(ligne)
```

❶⇒ouverture en écriture au format UTF-8 par défaut ;

❷⇒ouverture en lecture au format binaire.

Attention

La chaîne retournée par la méthode «`readline()`» contient le `'\n'`.

Lors de l'exécution, une chaîne au fomat «*bytes*» est bien affichée :

xterm

```
$ python3 exemple_fichier.py
b'\xc3\xa9t\xc3\xa9'
```

Si on regarde le contenu de fichier avec la commande «`xxd`» :

xterm

```
$ xxd fichier_test
00000000: c3a9 74c3 a9                             ..t..
```

Le fichier contient 5 octets : 2 octets pour chaque `'é'` et 1 octet pour le `'t'`.

Fichiers : lecture par ligne

L'objet de type `file` peut être utilisé de différentes manières pour effectuer la lecture d'un fichier.

- comme **une liste**, ce qui permet d'utiliser le `for` :

```
for une_ligne in fichier:
   print (une_ligne)
```

⟹ ***Déconseillé, on ne l'utilisera pas...***

- comme un «itérateur» sur lequel on applique la fonction `next()` (qui lève une exception à la fin) :

```
while 1:
   try:
   # renvoie l'exception StopIteration
   # à la fin
      une_ligne = next(fichier)
      print (une_ligne)
   except:
      break
```

⟹ ***Déconseillé, on ne l'utilisera pas...***

- à travers la méthode `readline()` :

C'est cette façon que l'on utilisera

```
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 :

```
   une_ligne = fichier.readline()
   if not une_ligne:
      break
   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 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 ⟹ fonctionne sur les fichiers UTF-8 (`"r"` ou `"a"`) ou octets (`"rb"` ou `"ab"`)
- ☐ 1 : relative ⟹ **ne fonctionne que** sur les fichiers octets (`"rb"` ou `"ab"`)
- ☐ 2 : fin du fichier ⟹ fonctionne sur les fichiers UTF-8 ou octets

Utilisation du «*with*» : le fichier est associé à un bloc d'instructions

```
with open("lecture_fichier.txt", "r") as fichier:
   while 1:
      caractere = fichier.read(1)
      if not caractere :
         break
```

Ici, la fermeture du fichier est automatique en dehors du bloc associé au «with»*, l'instruction* `fichier.close()` *n'est pas nécessaire*.

Gestion de l'exception

```
#!/usr/bin/python3

try:
    with open("lecture_avec_with.py", "w") as f:
        f.write('ajout de contenu')
except Exception as e:
    print(e.args)
```

L'exception qui peut survenir lors de l'ouverture du fichier doit être gérée.

Ici, si l'erreur est liée à l'impossibilité d'écrire dans le fichier (par exemple, par manque de place) le fichier sera automatiquement fermé $\Longrightarrow$ *plus de sûreté.*

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_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 d'octets» sert à manipuler cette séquence d'octets.

Le module spécialisé `struct` permet de composer ou de décomposer 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 :

`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

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 octet	n`s`	string	chaîne de n caractères
`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`.

xterm

```
>>> import struct
>>> données_compactées = struct.pack('=iffccccc', 10, 45.67, 98.3, b'a', b'b', b'c',
b'd', b'e')
>>> données_compactées
b'\n\x00\x00\x00\x14\xae6B\x9a\x99\xc4Babcde'
>>> liste_valeurs = struct.unpack('=i2f5c', données_compactées)
>>> liste_valeurs
(10, 45.66999816894531, 98.30000305175781, b'a', b'b', b'c', b'd', b'e')
```

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

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 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
- ▷ 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
  - `\s` : un espace
  - `\w` : un caractère alphanumérique
  - `\n` newline
  - `\r` retour-chariot
- ▷ une alternative
  - `|` : *ceci* ou *cela*, exemple : `a|b`

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

i. **composition du motif ou exp. rég.**, avec des caractères spéciaux (comme `\d` pour *digit*) ;
ii. **compilation**, pour rendre rapide le traitement ;
iii. **recherche** dans une chaîne de caractères ;
iv. **mémorisation** des séquences de caractères correspondant à la recherche dans la chaîne.

le préfixe `r` permet de bloquer l'interprétation des « \ »

```
import re
une_chaine = 'La valeur 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))
```

⇓

xterm

```
trouvé !
<_sre.SRE_Match object; span=(0, 2), match='12'>
12
```

Récupération de la séquence de caractères correspondant au motif

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`*`)`).

Unicode et octets

- Pour rechercher dans une **chaîne de caractères** : le motif ou ER doit être une **chaîne de caractères**.
- Pour rechercher dans une **chaîne d'octets** : le motif doit être une **chaîne d'octets**.

Exemple avec de l'unicode

```
import re
une_chaine = "Ma saison préférée est l'été, mais le printemps est pas mal non plus"
re_mot = re.compile(r"(\w*é\w*)")
resultat = re_mot.findall(une_chaine)
if resultat :
    print (resultat)
```

La méthode `findall` renvoie une liste de toutes les occurences, ici, des mots contenant un 'é'

xterm

```
$ python3 er2.py
['préférée', 'été']
```

Exemple avec des octets

```
import re
une_chaine = b'La valeur 12 est un nombre'
re_nombre = re.compile(rb"(\d+)")
resultat = re_nombre.search(une_chaine)
if resultat :
    print (resultat.group(1))
```

*on utilise le préfixe `'rb'` pour indiquer une chaîne d'octets et bloquer l'interprétation des *

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)
```

⇓

```
xterm
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)
```

⇓

```
xterm
['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))
```

⇓

```
xterm
['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.

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

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

Paramètres

Les paramètres de la fonction peuvent être :

- ▷ nommés et recevoir des valeurs par défaut ;
- ▷ être donnés dans le *désordre* et/ou partiellement.

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))
```

⇒

```
xterm
5.0
3.33333333333
6.75
```

Variables locales *vs* globales

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

```
variable_a_modifier = "Alice"
def modif(nom):
    variable_a_modifier = nom
modif("Bob")
print (variable_a_modifier)
```

⇓

```
xterm
Alice
```

```
variable_a_modifier = "Alice"
def modif(nom):
    global variable_a_modifier
    variable_a_modifier = nom
modif("Bob")
print (variable_a_modifier)
```

⇓

```
xterm
Bob
```

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)
```

⇓

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

Une *lambda expression* ou un objet fonction anonyme

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

⇒

```
xterm
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
>>> list(filter(lambda x: x > 3, [1,4,2,5]))
[4, 5]
```

Ici, sont filtrés les éléments pour lesquels la valeur de retour de la lambda fonction donne `False` *ou 0.*

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 as e:
        print ("Exception: ",e.args)
```

⇓

```
xterm
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

```
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.ran
dint(1,2**32)
print (isn)
```

⇓

```
xterm
2769369623
```

Choix aléatoire d'une valeur depuis un ensemble de valeurs

```
import random

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

⇓

```
xterm
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/python3
2  import sys
3
4  #configuration
5  nom_defaut = "document.txt"
6
7  #programme
8  saisie = input("Entrer le nom du fichier [%s]" % nom_defaut)
9  nom_fichier = saisie or nom_defaut
10
11 try:
12    entree = open(nom_fichier, "r")
13 except Exception as e:
14    print (e.args)
15    sys.exit(1)
```

- ▷ ligne 4 : on définie une valeur par défaut pour le nom du fichier à ouvrir ;
- ▷ ligne 6 : on saisie 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 et l'opérateur «`or`» affecte la valeur par défaut, qui, elle, est considérée comme *vraie*.

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

```
1 import subprocess
2 r  = subprocess.run('ls *.py',shell=True,stdout=subprocess.PIPE) # Récupère la sortie d'une commande
3 print(r.stdout)
```

Communication en Entrée/Sortie avec une commande :

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

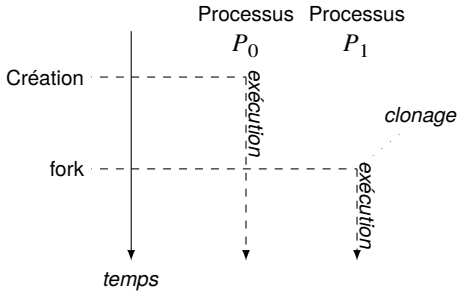
```
import sys

print (sys.argv)
```

Si on lance ce script :

xterm

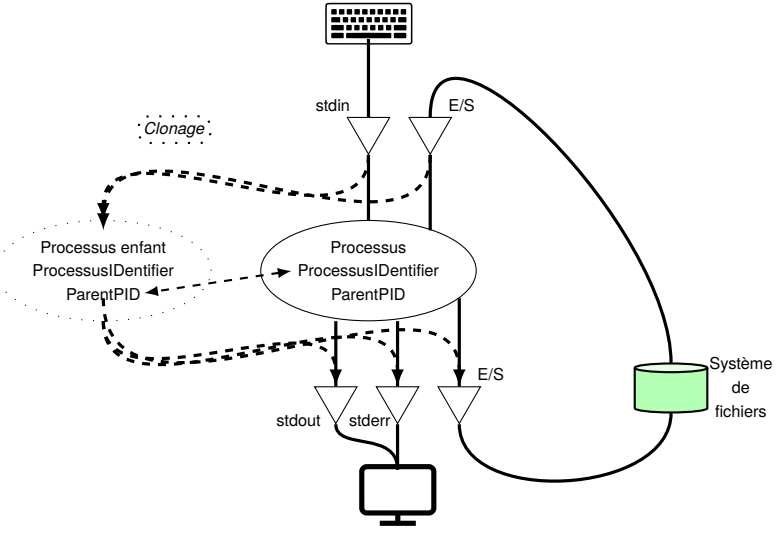
```
$ ./mon_script.py -nom toto
['mon_script.py', '-nom','toto']
```



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 connaitre le PID de son père) ;
- sont **indépendants** : **après le fork**, toute entrée/sortie créée n'est **pas partagées**.

- Le processus P_0 est créé et s'exécute.
- Lors de l'exécution de l'instruction `fork` : il est «cloné» 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` $\Longrightarrow$ *C'est cette opération qu'il faut utiliser car elle envoi immédiatement les données !*

Attention : *Les opérations de lecture et d'écriture utilisent des chaînes d'octets* `b''`.

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 TSAP, «*Transport Service Access Point*», ou *point d'accès au service* pour chaque socket utilisée pour une communication (le *service* correspond au processus associé).

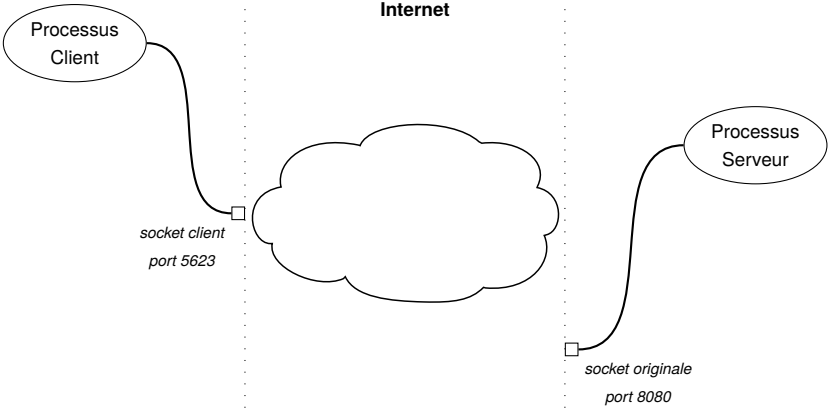
■→ Le **TSAP** est l'association de **l'adresse IP** de la machine et d'un «**numéro de port**».
Il est ainsi possible de faire du «multiplexage» : une seule adresse IP mais plusieurs communications simultanées.

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*.

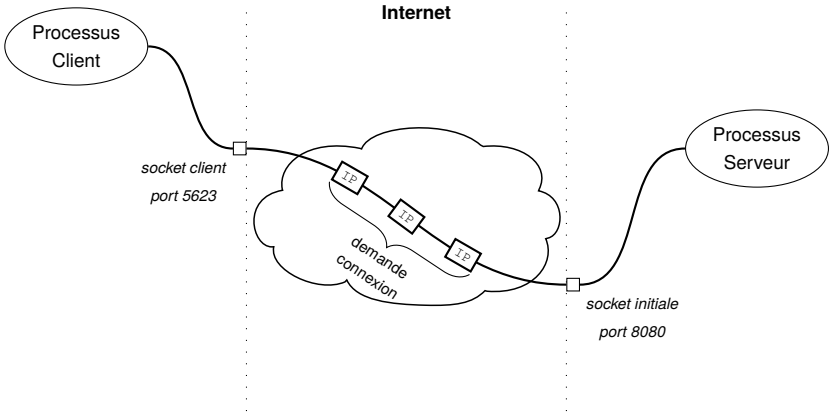
Les différentes étapes pour l'établissement de la communication :

1. Le serveur attend sur le TSAP :
   `[@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 choisi (inutile sur le client) ;
- ▷ `listen` : configure le nombre possible de communications simultanées sur le serveur.



2. Le client se connecte au serveur `connect`

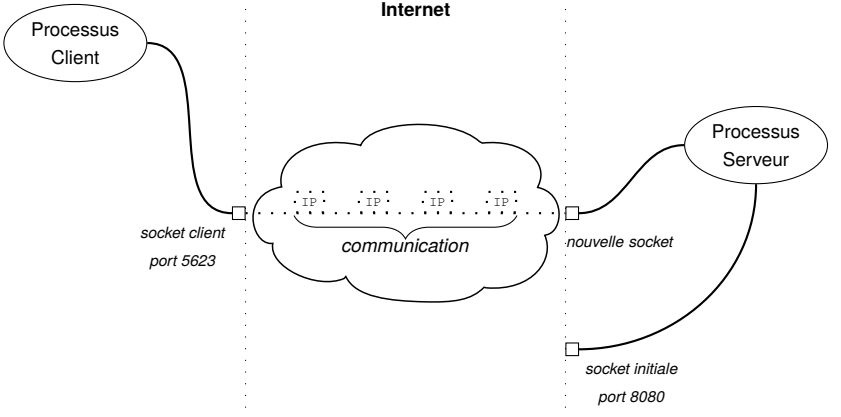
   Le système d'exploitation du client et du serveur, mémorise la communication par un couple (TSAP client,TSAP serveur) :

TSAP client		*TSAP serveur*
`(@IP client: 5623)`	⟺	`(@IP serveur: 8080)`

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

Remarques :

- L'instruction `accept` donne au serveur une **nouvelle** *socket* qui correspond à la communication avec le client.
  C'est par cette socket que l'on peut communiquer avec ce client.
- Le serveur peut recevoir la connexion de nouveaux clients sur la socket initiale.
  Un serveur peut avoir plusieurs communications simultanées avec différents clients.
  Chacune de ces communications correspond à un couple différent de TSAP : pour chaque communication, le même TSAP du côté du serveur est associé à un TSAP côté client différent.



Programmation d'un client en TCP

```
import os,socket,sys

adresse_serveur = socket.gethostbyname('localhost') # réalise 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 as e:
    print ("Probleme de connexion", e.args)
    sys.exit(1)

while 1:
   ligne = ma_socket.recv(1024) # réception d'au plus 1024 caracteres
   if not ligne:
      break
   print (ligne)

ma_socket.close()
```

l'adresse symbolique du serveur

SOCK_STREAM désigne TCP

retourne une chaîne d'octets `b''`

Attention

- les «*sockets*» échangent des données au **format chaîne d'octets** : `b''`
- Le paramètre donné à «`ma_socket.recv(1024)`» indique la **taille maximale** que l'on peut recevoir, mais **ne garantie pas** qu'elle retournera 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 en C
ma_socket.listen(socket.SOMAXCONN)

while 1:
    (nouvelle_connexion, TSAP_depuis) = ma_socket.accept()
    print ('Nouvelle connexion depuis ', TSAP_depuis)
    nouvelle_connexion.sendall(b'Bienvenu\n')
    nouvelle_connexion.close()

ma_socket.close()
```

socket retournée par le `accept()`

chaîne d'octets !

Utilisation de la méthode `sendall`

Pour envoyer **immédiatement** une ligne :

```
nouvelle_connexion.sendall(b'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**, les deux processus peuvent **se bloquer** si l'envoi est retardé (bufférisé) et non immédiat.

- ▷ 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 = b''

    while 1:
        caractere_courant = ma_socket.recv(1)
        if not caractere_courant :
            break
        ligne += caractere_courant
        if caractere_courant == b'\n':
            break

    return ligne
```

*La ligne retournée par la fonction **inclut** le retour à la ligne :* `b'\r\n'` *ou* `b'\n'`.

■→ *La norme des* «retour à la ligne» *pour les protocoles sur Internet est* `b'\r\n'` .

Attention

Ce sera cette version que vous utiliserez dans les TPs.

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)
```

$\Longrightarrow$ 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.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
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(b'Bienvenu\n')
         surveillance.append(nouvelle_connexion)
         continue
      # sinon cela concerne une socket connectée à un client
      ligne = un_evenement.recv(1024)
      if not ligne :
         surveillance.remove(un_evenement) # le client s'est déconnecté
      else :
         print (un_evenement.getpeername(),':',ligne)
         # envoyer la ligne a tous les clients, etc
```

Utilisation du protocole UDP :

```
import socket

TSAP_local = ('', 7777)
TSAP_distant = (socket.gethostbyname("p-fb.net"), 8900)

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

ma_socket.sendto(b'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.
- Les données échangées dans les méthodes `sendto()` et `recvfrom()` sont des chaînes d'octets `b''`.

26 Multithreading – Threads

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
    print('Travail de la thread')
    return

# Déclenchement de la thread
ma_thread = threading.Thread(target = ma_fonction)
ma_thread.start()

# Thread principale
print('Travail de la thread principale')
```

⇓

```
xterm
Travail de la thread
Travail de la thread principale
```

Multithreading – Sémaphores

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

```
import threading

def ma_fonction():
   # travail
   print("Travail de la thread")
   s.release()
   return

s = threading.Semaphore(0)

# Déclenchement de la thread
ma_thread = threading.Thread(target = ma_fonction)
ma_thread.start()

# Thread principale
print('Travail de la thread principale')
s.acquire()
```

Serveur Web

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

xterm
```
pef@darkstar:/Users/pef $ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
127.0.0.1 - - [12/Aug/2018 21:33:47] "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://localhost:8000/
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET / HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.61.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: SimpleHTTP/0.6 Python/3.6.4
< Date: Sun, 12 Aug 2018 19:33:47 GMT
< Content-type: text/html; charset=utf-8
< Content-Length: 1713
<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
```

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

xterm
```
$ python3 -m http.server --cgi
```

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 | python3 -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
from urllib.request import urlopen

url = 'http://api.icndb.com/jokes/random?li
mitTo=[nerdy]'
response = 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,

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", "rb")
mon_fichier.seek(2,0)    #On se place en fin, soit à zéro en partant de la fin
taille = mon_fichier.tell()
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
   print('fichier existe')
if os.path.isfile("chemin_fichier"):
   # c'est un fichier
   print("c'est un fichier et non un répertoire")
if os.path.isdir("chemin_fichier") :
   # c'est un répertoire
   print("c'est un répertoire et non un fichier")
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 :

Attention : c'est la lettre l, comme «linux»...

```
import socket,struct

ma_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
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.run(['uname','-a'],stdout=subprocess.PIPE).stdout.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.

```
from scapy.all import *

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

Manipulations avancées : programmation objet, classe et introspection

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):      #hérite de la classe object
    variable_classe = 10
    def __init__(self):         # deux caractères underscore _
        self.variable_instance = 2
    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).

Introspection

Obtenir les attributs et méthodes d'un objet avec la fonction `dir()` :

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

⇓

xterm

```
>>> print(dir([]))
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

Intégration Unix : 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» :

```
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/python3
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 : True si l'option est présente
(options, args) = parseur.parse_args() # args : 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
{'nom_fichier': 'mon_fichier.txt', 'conversion': True}
```

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
```
$ python3 -i mon_programme.py
```

Exemple sur le programme de génération de valeurs aléatoires :

xterm
```
$ python3 -i test.py
184863612
>>> isn
184863612
>>>
```

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

▷ avec le module «`code`» :

```
#!/usr/bin/python3
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.

▷ avec en plus la complétion des variables et méthodes :

```
#!/usr/bin/python3
import code,readline,rlcompleter
...
vars = locals()
readline.set_completer(rlcompleter.Completer(vars).complete)
readline.parse_and_bind("tab: complete")
code.interact(local=vars)
```

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

On peut lancer un programme Python en activant le débogueur :

```
xterm
$ python3 -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
$ python3 -m pdb tester_dbg.py
> /home/pef/PYTHON3/tester_dbg.py(3)<module>()
-> compteur = 0
(Pdb) next
> /home/pef/PYTHON3/tester_dbg.py(4)<module>()
-> for i in range(1,10):
(Pdb) n
> /home/pef/PYTHON3/tester_dbg.py(5)<module>()
-> compteur += 1
(Pdb)
> /home/pef/PYTHON3/tester_dbg.py(4)<module>()
-> for i in range(1,10):
(Pdb) i
1
(Pdb) n
> /home/pef/PYTHON3/tester_dbg.py(5)<module>()
-> compteur += 1
(Pdb) i
2
(Pdb) n
> /home/pef/PYTHON3/tester_dbg.py(4)<module>()
-> for i in range(1,10):
(Pdb) l
  1     #!/usr/bin/python3
  2
  3     compteur = 0
  4  ->    for i in range(1,10):
  5         compteur += 1
  6     print (compteur)
[EOF]
(Pdb) n
> /home/pef/PYTHON3/tester_dbg.py(5)<module>()
-> compteur += 1
(Pdb) compteur
2
```

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

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.

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

- contrôlable par une ligne dans le source :

```
1 #!/usr/bin/python3
2 import logging
3
4 logging.basicConfig(level=logging.DEBUG)
5 ...
6 logging.debug('Ceci est un message de debogage')
```

```
xterm
$ 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 :

```
logging.basicConfig()
```

- possibilité d'enregistrer 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/python3
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
2018-08-13 14:57:05 DEBUG: Ceci est un message de debogage
```