Python
(une brève introduction à)
IUT GEII

intro_python1.5.odp

Licence

creative commons
COMMONS DEED

•Paternité - Pas d'Utilisation Commerciale -
• Partage des Conditions Initiales à l'Identique 2.0 France

•Vous êtes libres :
• * de reproduire, distribuer et communiquer cette création au public
• * de modifier cette création, selon les conditions suivantes :
•
•**Paternité**. Vous devez citer le nom de l'auteur original.
•
•**Pas d'Utilisation Commerciale**.
•Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.
•
•**Partage des Conditions Initiales à l'Identique**.
•Si vous modifiez, transformez ou adaptez cette création,
•vous n'avez le droit de distribuer la création qui en résulte
•que sous un contrat identiqueà celui-ci.
• * A chaque réutilisation ou distribution, vous devez faire apparaître clairement aux autres les
• conditions contractuelles de mise à disposition de cette création.
•* Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits
•Ce qui précède n'affecte en rien vos droits en tant qu'utilisateur (exceptions au droit d'auteur :
•copies réservées à l'usage privé du copiste, courtes citations, parodie...)
•voir le contrat complet sous : http://fr.creativecommons.org/contrats.htm

Python

Un langage libre inventé par Guido Van Rossum en 1998 :

Python est interprété (peut aussi être compilé)
Python est multi plate-forme
Python est fortement orienté objet

Deux versions majeures non compatibles :
<= 2.7 beaucoup de scripts existent **(*)**
>=3.0 casse la compatibilité ascendante mais apporte de nouvelles fonctionnalités

Python tire son nom d'une troupe de théâtre anglaise des années 1970 : "The Monty Phython"

Voir par exemple vidéo : "Monty python le pont de la mort"
Extrait du film "Monty Python and the Holy Grail"

(*) plus maintenue depuis le 01/01/2020 !

Python : l'interpréteur

Interpréteurs par défaut :

Python 2.6 :

```
philippe@jupiter:~$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python3 :

```
philippe@jupiter:~$ python3
Python 3.1.2 (r312:79147, Dec  9 2011, 20:54:36)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Très pratique pour tester une fonctionnalité.
Il existe d'autres interpréteurs python avec plus de fonctionnalités : bpython , ipython, ...

Documentation : https://docs.python.org

Installer python ?

Python (2 et 3) est installé de base sur la plupart des distributions **linux**.

Pour windows il faut installer l'interpréteur :

https://www.python.org/downloads/windows/

Un bon point de départ :

www.python.org/about/gettingstarted/

Installer des librairies

Avec la plupart des distributions linux, on peut installer les libraires avec le gestionnaire de paquets :
Pour exemple pour Debian/Ubuntu :

```
$sudo apt install python3-numpy python3-matplotlib
```

On peut aussi utiliser le python package index : **pip** ou **pip3**
Pour avoir la version courante :

```
$sudo pip3 install nom_du_package
```

Pour avoir la version 1.2.4 :

```
$ sudo pip3 install nom_du_package == 1.2.4
```

Pour avoir une version au moins égale à 1.2.4 :

```
$ sudo pip3 install nom_du_package >= 1.2.4
```

Voir :
https://docs.python.org/fr/3.5/installing/index.html

Vérifier l'installation de vos libraires

Nous aurons besoin des librairies ***numpy***, ***matplolib*** et ***scipy*** installées pour **python3**. Vérifiez la bonne installation et les versions dans l'interpréteur :

```
Python 3.5.2 (default, Nov 12 2018, 13:43:14)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import matplotlib
>>> matplotlib.__version__
'1.5.1'
>>> import numpy
>>> numpy.__version__
'1.16.0'
>>> import scipy
>>> scipy.__version__
'1.2.0'
>>>
```

IDE python ?

Un **interpréteur** (python , Ipython,...) et **un éditeur de texte qui peut afficher les caractères blancs** (espace,tab).
est tout ce qui est nécessaire pour programmer en python.

Cependant il existe des IDE spécifiques qui peuvent apporter plus de confort : complétion de code, terminal intégré, point d'arrêt,.... :

Geany , **Spyder3**, Eric, PyCharm, Atom,....

On peut aussi faire tourner python sur des microcontrôleurs :

projet MicroPython

Par exemple sur un ESP8266 (nodeMCU V2) :
4Mo Flash, 512ko de RAM, Wifi, SPI, I2C, GPIO, pour quelques euros.
Ce peut être très intéressant si on n'a pas de contrainte trop forte de performances ni de consommation. (Attention tout les périphériques ne sont pas supportés.)

Un script python (3)

Documentation : https://docs.python.org

Faire un fichier HelloPython.py contenant :

```
print ( "Hello Python!")
```

Puis lancer le par : $python HelloPython.py

Ou alors :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
print ( "Hello Python!")
```

Chemin vers python

Pour les accents (python2)

Puis :

```
$ chmod +x HelloPython.py
$ ./HelloPython.py
```

Variables

Pas de déclaration !

Le type est fixée par la valeur

Le type change si la valeur change de type

```
>> a = 8
>> b = 'bonjour'
>> type(a) , type(b)
(<class 'int'>, <class 'str'>)
>> a = "au revoir"
>>type(a)
<class 'str'>
>>
```

Affectation multiple :

```
  a , b = 3,4
  a , b  = b,a
```

Python 2 / Python 3

En python 2 print est une instruction :

```
a=4
print "bonjour" a
```

En python 3 print est une fonction (avec plus de possibilités):

```
print("bonjour",a)
# affichage sans retour ligne
print("bonjour",a,end='')
# rajoute XX en fin d'affichage
print("bonjour",end='XX\n')
```

Pour utiliser la fonction print en python 2 on peut ajouter :

```
from __future__ import print_function
```

Entrée clavier

On récupère toujours une chaîne (classe str)

Python 2

```
a=raw_input('Valeur de a ?')
```

Python 3

```
a=input('Valeur de a ?')
```

Que l'on peut transformer en un autre type si nécessaire :

```
a = int (a)
```

Ou

```
a = float(a)
```

try / except

On peut éviter de mettre fin au programme sur une erreur (exception) avec un bloc try except :

```
import sys
a=input("Entrez un nombre :")
try :
    a=float(a)
except ValueError :
  print("Un nombre svp !"),
  sys.exit()
print("Merci")
```

Ou attribuer une valeur par défaut et continuer :

```
try :
    a=float(a)
except ValueError :
    a=0
```

try / execpt

Si on ne précise pas le type d'exception alors toute erreur provoque l'exécution du bloc except (déconseillé)

On peut écrire un gestionnaire d'exception pour plusieurs types d'erreurs :

```
try :
  a=int(a)
  b = 1/a
except ValueError :
  b=a=1
  print("Erreur de conversion de a : a=1")
except ZeroDivisionError :
  b=10000
  print("a=0  donc b très grand (b=10000)")
print("Suite")
```

Types numériques

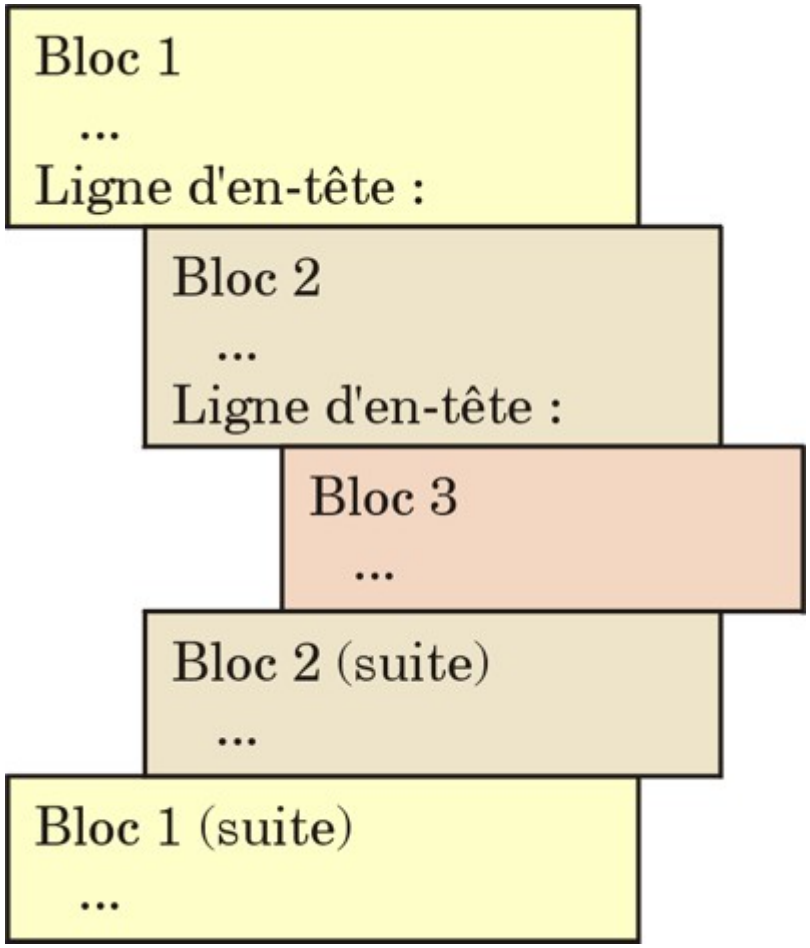
```
>>a=7 , b=4.5     a est un int b est un float
>>c=a+b
```

Opérateurs disponibles : +,-,*,/,% et en plus

```
   // division entière
  ** puissance
```

Syntaxe : la présentation compte !

Les blocs sont associés à une ligne d'entête se terminant par un double point :
Les blocs sont délimités par l'indentation



if/else

```
if a>b :
   print ("a plus grand que b")
elif a<(b-2):
  print ("Cas 2")
else:
  print ( 'autre cas')
```

while et while/else

```
i=0
while i < 7:
    print( i )
    i +=1
```

```
a = input()
a = int(a)
i=2
while  i < a/2 :
    if a%i == 0 :
      print (a,'est
divisible par',i)
      break
    i = i+1
else:
     print (a,'est premier')
```

La clause else est exécutée si on sort normalement (pas par break)

Itération par boucle for

La boucle for permet d'itérer une chaîne (et tout type itérable) :

```
S='bonjour'
for c in S :
  print (c)   # c va prendre toutes les valeurs de s de 'b' à 'r'
```

Parfois il est utile d'itérer en récupérant l'index :

```
for i,c in enumerate(S):
 print (i,c)
```

0 b
1 o
2 n
.....

for avec zip

Il est possible d'itérer deux (ou plusieurs) itérables en prenant un élément dans chaque :

```
s1='abcd'
s2='12345678'
for e1,e2 in zip(s1,s2) :
    print (e1,e2)
```

→

```
a 1
b 2
c 3
d 4
```

Arrêt à l'épuisement de la liste la plus courte

Si nécessaire on peut accéder à l'indice :

```
for i,(e1,e2) in enumerate(zip(s1,s2)) :
    print (i, e1,e2)
```

→

```
0 a 1
1 b 2
2 c 3
3 d 4
```

Le type : string

String est utilisé pour représenter les chaînes de caractères:

```
s ='bonjour'   # ' et " sont interchangeables sauf si s contient un ' ou un " : s='bon"jo"ur' ou s="bo'jo'ou'r"
s = s + ' ' + 'monsieur'
print(s)       # bonjour monsieur
s = 3*s
print(s) #bonjour monsieurbonjour monsieurbonjour monsieur
print(s[0],s[-1]) # b r
```

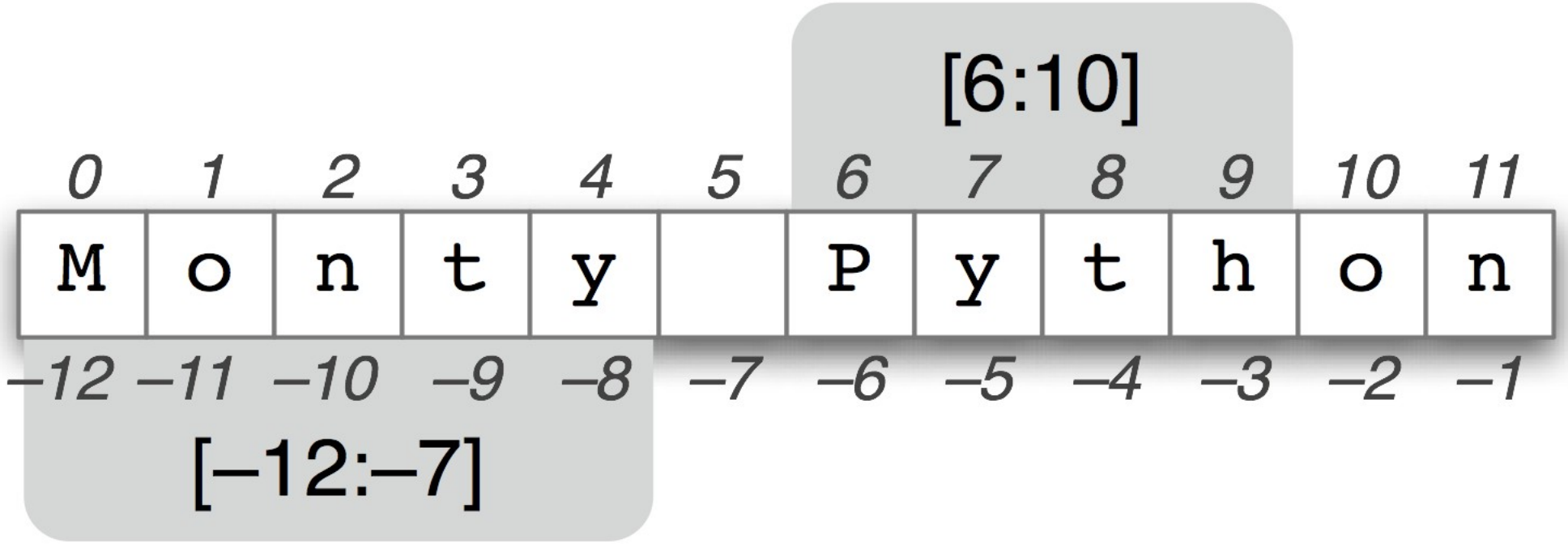
s[0] : premier élément , S[-1] dernier élément

```
s='allo'
for c in s:
   print (c,end=',')
```

a,l,l,o,

Les slices

s[i:j] est la portion de chaine comprise entre i et j-1



s='Monthy Python'
s[6:10] vaut 'Pyth' , s[-12:-7] vaut 'Monty'
s[2:5] .....
s[-4:-1] .....

* La portion est vide si i>=j

Les slices

Des "tranches" de chaînes :

S= 'bonjour'

S[i:j] est la partie de la chaîne comprise entre i et j-1

S[0:3] ↔ 'bon'
S[1:] ↔ 'onjour' de 1 à la fin
S[:4] ↔ 'bonj' du début à 3
S[:] ↔ 'bonjour'
S[:-1] ↔ 'bonjou' on coupe avant le dernier
S[3:-2] ↔ 'jo' du troisième jusqu'à avant l'avant dernier

S[1:-1:2] ↔ 'oju' par pas de 2 sans le premier ni le dernier
S[::-1] ↔ 'ruojnob' chaine à l'envers (pas de -1)

Modifier une chaîne

Attention **les chaînes sont "constantes"** (immutable en anglais):

```
S='bonjour'

S[0] = 'B'  ← erreur

S='B'+S[1:]  # remplace le premier caractère

S=S[::-1]      # passe la chaîne à l'envers

S=S[:2]+'N'+s[3:] # "remplace" n par N

S=S[:2]+'N'+s[2:] # insère N entre o et n
```

Quelques méthodes sur les chaînes

len(s) : longueur d'une chaine
int(s) , float(s) : passe une chaine en entier, en float
str(123) , str(1.234) : passe un entier , un float en chaine

s.upper(),s.lower() : chaine en majuscule,minuscule

s='bonjour monsieur'
s.replace('on','XXX')
remplace chaque occurrence de 'on' par 'XXX'
On obtient bXXXjour mXXXsieur

Attention ces méthodes ne modifient pas s
mais retournent une nouvelle chaine.
Si on veut passer une chaine en minuscule il faudra écrire
s=s.lower()

Modifier une chaîne

Attention **les chaînes sont "constantes"** (immutable en anglais):

```
S='bonjour'

S[0] = 'B'  ← erreur

S='B'+S[1:]  # remplace le premier caractère

S=S[::-1]      # passe la chaîne à l'envers

S=S[:2]+'N'+s[3:] # "remplace" n par N

S=S[:2]+'N'+s[2:] # insère N entre o et n
```

Formatage de chaîne (bientôt obsolète!)

En plus des formats par défaut de print, python accepte des format "à la printf" : %s,%d,%x,%c,%f,%e,%%?....

```
>>> n=3 ; data=254 ; name='toto'
>>> fmt='name=%s n=%d data=%x hex'
>>> fmt % (name,n,data)
'name=toto n=3 data=fe hex'
```

Crée une chaine de format

Crée une chaine au format fmt

Ou dans un script

```
print ( fmt % (name,n,data))
```

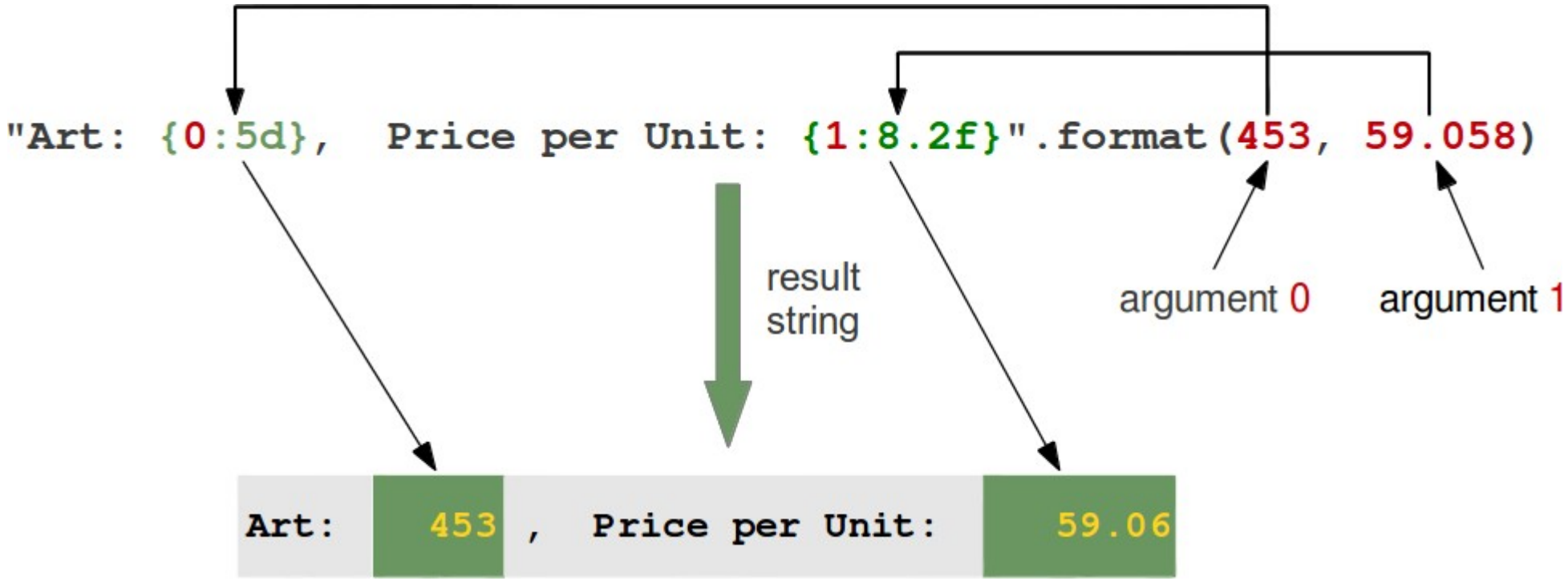
Ou directement :

```
print ( '%d nom=%s %X' % (data,name,data) )
```

```
254 nom=toto FE
```

Cette méthode de formatage risque de disparaître des futures versions de python : utilisez plutôt **format** ou les **fString**

Python3 format



Voir :
http://www.python-course.eu/python3_formatted_output.php

Les f-String : Python >= 3.6

```
name = 'Casimir'
weight = 140

s=f"Il s'appelle {name} et pèse {weight} kilos"

print(s)

Il s'appelle Casimir et pèse 140 kilos
```

On peut également définir précisément le format :

```
pi=3.14159
print(f'pi={pi :.2f}')
```

⟶ 3.14

https://docs.python.org/3.6/library/string.html#formatspec

Caractères

Les caractères sont représentés par des motifs binaires conventionnels : encodage.

Encodages courants :

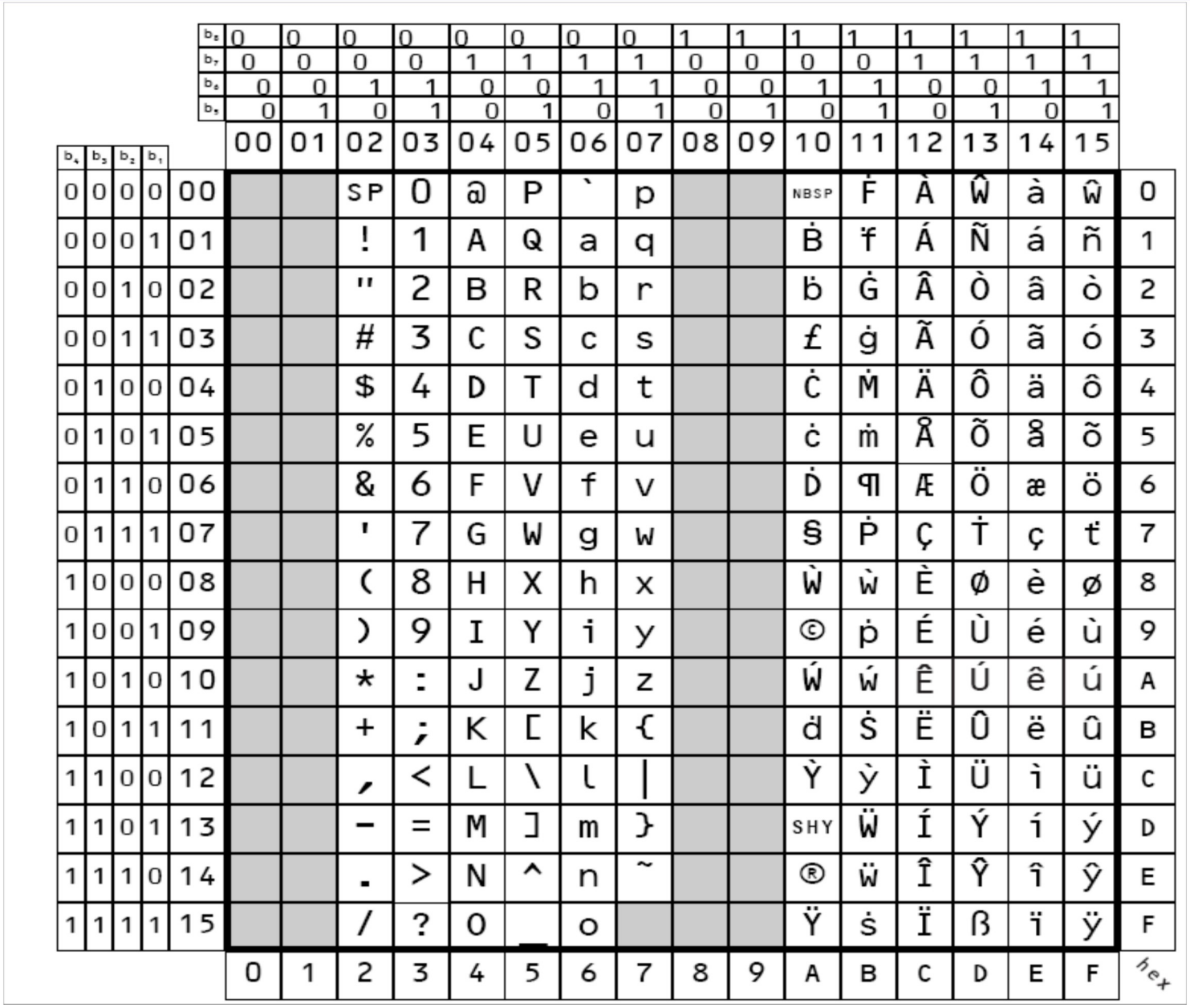
le code ASCII : chaque caractère est représenté sur un octet donc le poids faible est à 0 (0x00 à 0x7F).
128 caractères : majuscules, minuscules , chiffres, ponctuations, contrôles. **Mais pas d'accent !**

Code ASCII étendu à un octet : ASCII + 128 caractères supplémentaires (0x80 à 0xFF) → certains caractères accentués, symbole monétaire, ...
Plusieurs codages existent : jeux de caractères
Ex : latin1, ISO-8859-15, ...

ISO 8859-15 Latin alphabet n°9

b4 b3 b2 b1	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0 0 0 0	00		SP	0	@	P	`	p			NBSP	°	À	Ð	à	ð	0
0 0 0 1	01		!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0 0 1 0	02		"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0 0 1 1	03		#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0 1 0 0	04		$	4	D	T	d	t			€	Ž	Ä	Ô	ä	ô	4
0 1 0 1	05		%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0 1 1 0	06		&	6	F	V	f	v			Š	¶	Æ	Ö	æ	ö	6
0 1 1 1	07		'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1 0 0 0	08		(	8	H	X	h	x			š	ž	È	Ø	è	ø	8
1 0 0 1	09		)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1 0 1 0	10		*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1 0 1 1	11		+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1 1 0 0	12		,	<	L	\	l	|			¬	Œ	Ì	Ü	ì	ü	C
1 1 0 1	13		-	=	M	]	m	}			SHY	œ	Í	Ý	í	ý	D
1 1 1 0	14		.	>	N	^	n	~			®	Ÿ	Î	Þ	î	þ	E
1 1 1 1	15		/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ	F
			0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

ISO 8859-14 Latin alphabet n°8 (Celtic)



b4 b3 b2 b1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0 0 0 0	00			SP	0	@	P	`	p			NBSP	Ḟ	À	Ŵ	à	ŵ	0
0 0 0 1	01			!	1	A	Q	a	q			Ḃ	ḟ	Á	Ñ	á	ñ	1
0 0 1 0	02			"	2	B	R	b	r			ḃ	Ġ	Â	Ò	â	ò	2
0 0 1 1	03			#	3	C	S	c	s			£	ġ	Ã	Ó	ã	ó	3
0 1 0 0	04			$	4	D	T	d	t			Ċ	Ṁ	Ä	Ô	ä	ô	4
0 1 0 1	05			%	5	E	U	e	u			ċ	ṁ	Å	Õ	å	õ	5
0 1 1 0	06			&	6	F	V	f	v			Ḋ	¶	Æ	Ö	æ	ö	6
0 1 1 1	07			'	7	G	W	g	w			ß	Ṗ	Ç	Ṫ	ç	ṫ	7
1 0 0 0	08			(	8	H	X	h	x			Ẁ	ẁ	È	Ø	è	ø	8
1 0 0 1	09			)	9	I	Y	i	y			©	ṗ	É	Ù	é	ù	9
1 0 1 0	10			*	:	J	Z	j	z			Ẃ	ẃ	Ê	Ú	ê	ú	A
1 0 1 1	11			+	;	K	[	k	{			ḋ	Ṡ	Ë	Û	ë	û	B
1 1 0 0	12			,	<	L	\	l	|			Ỳ	ỳ	Ì	Ü	ì	ü	C
1 1 0 1	13			-	=	M	]	m	}			SHY	Ẅ	Í	Ý	í	ý	D
1 1 1 0	14			.	>	N	^	n	~			®	ẅ	Î	Ŷ	î	ŷ	E
1 1 1 1	15			/	?	O	_	o				Ÿ	ṡ	Ï	ß	ï	ÿ	F
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	hex

Unicode

En fait même avec plusieurs jeux de caractères, on ne parvient pas à coder tous les caractères utilisés dans toutes les langues.
La norme Unicode permet de coder de manière unique
tous les caractères utilisés par toutes les langues du monde :

Dans la version 3.1 de Unicode, près de 245000 caractères sont définis.

Il y a 17 "plans" de 16 bits.
Soit 17 x 65536 = **1 114 112 codes possibles**.

Un caractère Unicode est noté U+yyxxxx

Avec yy numéro hexa du plan (0 non significatifs omis)
xxxx code hexa du caractère dans le plan yy

Unicode

On retrouve nos jeux courants au début (plan 0).
0000-007F : le code ascii, 0080-FF : latin-1, etc...

Ex : A est
Codé 0041
(ascii étendu
à 16bits)

Sous-ensemble	Réservés		Décimal	
	Début	Fin	Début	Fin
latin basique	U+0000	U+007F	0	127
supplément latin-1	U+0080	U+00FF	128	255
latin étendu - A	U+0100	U+017F	256	383
latin étendu - B	U+0180	U+024F	384	591
extensions API	U+0250	U+02AF	592	685
lettres modificatives	U+02B0	U+02FF	688	767
diacritiques combinants	U+0300	U+036F	768	879
grec et copte	U+0370	U+03FF	880	1 023
cyrillique	U+0400	U+04FF	1 024	1 279
supplément cyrillique	U+0500	U+052F	1 280	1 327
arménien	U+0530	U+058F	1 328	1 423
hébreu	U+0590	U+05FF	1 424	1 535
arabe	U+0600	U+06FF	1 536	1 791
syriaque	U+0700	U+074F	1 792	1 871
supplément arabe	U+0750	U+077F	1 872	1 919
thâna	U+0780	U+07BF	1 920	1 983
n'ko	U+07C0	U+07FF	1 984	2 047
(samaritain)	U+0800	U+083F	2 048	2 111

Sous-ensemble	Réservés		Décimal	
	Début	Fin	Début	Fin
(mandaïque)	U+0840	U+085F	2 112	2 143
—	U+0860	U+087F	2 144	2 175
¿arabe étendu-A?	U+0880	U+08BF	2 176	2 239
—	U+08C0	U+08FF	2 240	2 303
dévanâgarî	U+0900	U+097F	2 304	2 431
bengalî	U+0980	U+09FF	2 432	2 559
gourmoukhî	U+0A00	U+0A7F	2 560	2 687
goudjarâtî (gujrâtî)	U+0A80	U+0AFF	2 688	2 815
oriyâ	U+0B00	U+0B7F	2 816	2 943
tamoul	U+0B80	U+0BFF	2 944	3 071
télougou	U+0C00	U+0C7F	3 072	3 199
kannara	U+0C80	U+0CFF	3 200	3 327
malayalam	U+0D00	U+0D7F	3 328	3 455
singhalais	U+0D80	U+0DFF	3 456	3 583
thaï	U+0E00	U+0E7F	3 584	3 711
laotien	U+0E80	U+0EFF	3 712	3 839
tibétain	U+0F00	U+0FFF	3 840	4 095

UTF-8

En pratique Unicode est peu utilisé directement.
Dans un texte la majorité des caractères font partis du code ascii de base (0-127).
L'utilisation directe d'Unicode doublerait donc la taille alors que la plupart du temps le poids fort reste à 00.

En UTF-8 :

Le numéro de chaque caractère est donné par le standard Unicode.

Les caractères de numéro 0 à 127 sont codés sur un octet dont le bit de poids fort est toujours nul.

Les caractères de numéro supérieur à 127 sont codés sur plusieurs octets. Dans ce cas, les bits de poids fort du premier octet forment une suite de 1 de longueur égale au nombre d'octets utilisés pour coder le caractère, les octets suivants ayant 10 comme bits de poids fort.

UTF-8

Représentation binaire UTF-8	Signification
0xxx xxxx	1octet , caractères de 0 à 127 (7 bits) ASCII-US
110x xxxx **10**xx xxxx	2 octets , codant 8 à 11 bits
1110 xxxx **10**xx xxxx **10**xx xxxx	3 octets, codant 12 à 16 bits
1111 0xxx **10**xx xxxx **10**xx xxxx **10**xx xxxx	4 octets, codant 17 à 21 bits

Avantages :

Efficace pour les langues utilisant une majorité de caractères ascii-us.
Permet de représenter une grande partie des caractères Unicode.
Compatible avec les codes C existants pour la fin de chaîne (caractère 0)

Inconvénients :

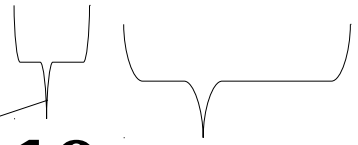
Plus difficile de compter les caractères.
Peu efficace pour les langues utilisant une majorité de caractères non ascii-us (chinois, arabe, ...)

UTF-8 exemples

Pour coder le caractère **A** pas de difficulté :
A : décimal 65 <127 ou hexa 0x41 <0x7F donc
Unicode: U+0041 et UTF-8: un octet 0x41 (**0**100 0001)

Pour coder le caractère **é** :
En ascii étendu il faut connaître le jeux utilisé (iso-latin1 =E9).
En Unicode U+00E9
En UTF-8 : >127 donc au moins deux octets

U+00E9 -> 0000 0000 1110 1001

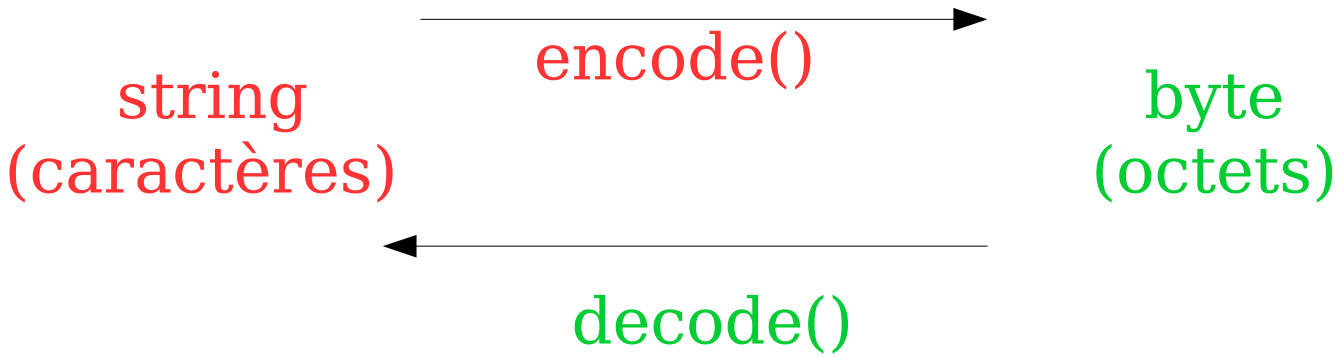


motif **110**x xxxx **10**xx xxxx
UTF-8 **110**0 0011 **10**10 1001

Donc **é** est codé **0xC3A9** en UTF-8

Python3 : string ↔ byte

Python 3 utilise **unicode** pour stocker les caractères.
L'encodage par défaut est l'UTF8.




```
>>> s='mémé'
>>> s.encode('latin1')
b'm\xe9m\xe9'
>>> s.encode('UTF8')
b'm\xc3\xa9m\xc3\xa9'
```

```
>>> b=b'A\xC3\xA9BC'
>>> b.decode('UTF8')
'AéBC'
>>> b.decode('latin1')
'AÃ©BC'
```

string ↔ byte

Fichier ouvert en mode r (défaut) , input() :
On récupère des caractères : classe string
Fichier ouvert en mode rb , (ex lecture sur un port série) :
On récupère des octets : classe byte

```
>>> f=open('/tmp/testfile','w')
>>> f.write('ça va bien ?\n')
13
>>> f.close()
>>> f=open('/tmp/testfile','r')
>>> f.read()
'ça va bien ?\n'
>>> f=open('/tmp/testfile','rb')
>>> f.read()
b'\xc3\xa7a va bien ?\n'
```

Code ↔ caractère

chr() : retourne le caractère (class str) dont le code unicode est passé en paramètre :

```
>>> chr(0x41)
'A'
>>> chr(65)
'A'
>>> chr(0x2A4)
'ʤ'
```

ord() : retourne le code unicode (class int) dont le caractère est passé en paramètre :

```
>>> ord('A')
65
>>> ord('ʤ')
676
```

bytearray

Suite d'octets (bytes) modifiables :

```
>>> ba = bytearray('ABCDéF'.encode('UTF8'))
>>> ba
bytearray(b'ABCD\xc3\xa9F')
>>> ba[0]=66
>>> ba
bytearray(b'BBCD\xc3\xa9F')
>>> ba.decode()
'BBCDéF'
```

Little Endian / Big Endian

La valeur entière 109243=0x0001AABB est stockée à partir de l'adresse 0x00221234. Il y a deux possibilités :

Little Endian : le poids faible dans l'adresse de plus faible valeur (the little end comes first)

0x00221237	0x00
0x00221236	0x01
0x00221235	0xAA
0x00221234	0xBB

int i = 109243 ;

Big Endian : le poids faible dans l'adresse de valeur la plus forte (the big end comes first)

0x00221237	0xBB
0x00221236	0xAA
0x00221235	0x01
0x00221234	0x00

int i = 109243 ;

Conversions bytes ↔ int

Les bytes sont vus comme des suites d'octets. Pour pouvoir faire des calculs il faut les convertir en entier et donc fixer "*l'endianité* " big ou little.

```
b =b'\x00\x01\xaa\xbb'
```
ici premier ↔ MSB → donc Big

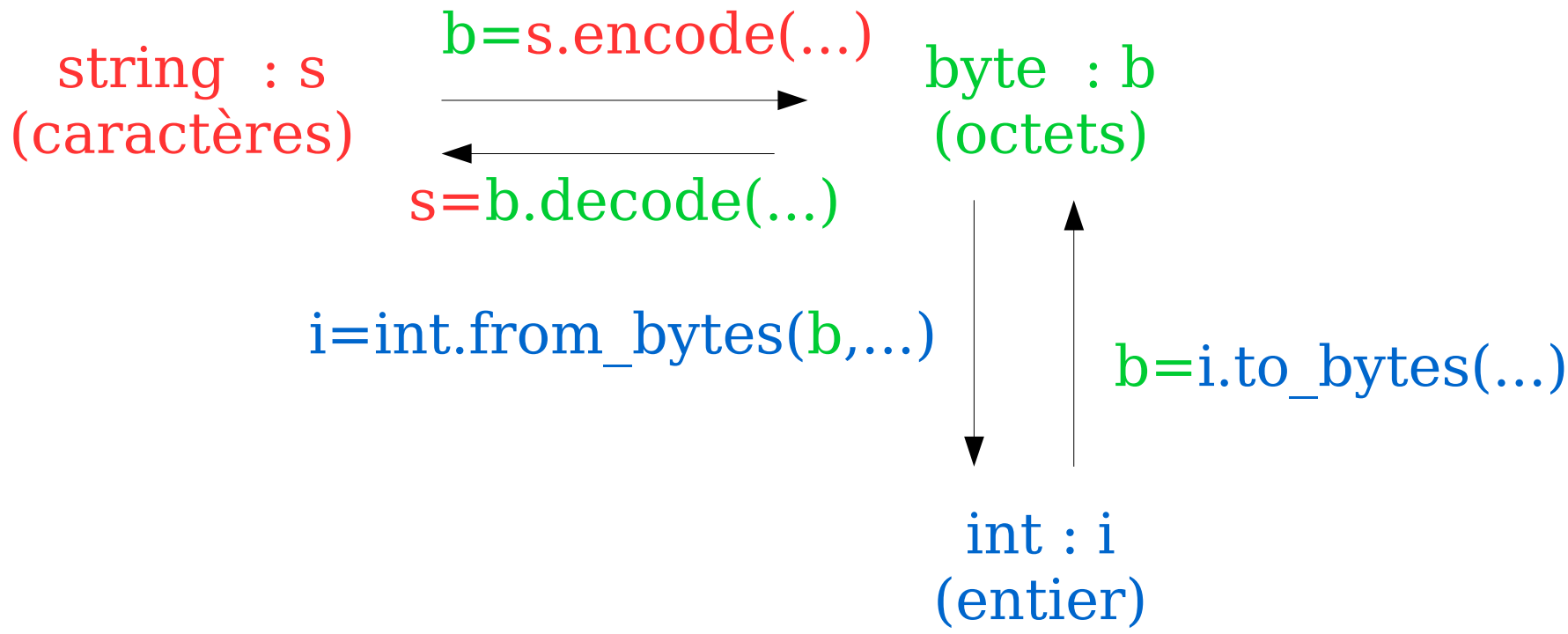
```
i = int.from_bytes(b,'big')
```
← i vaut 109243

```
i=i+1
```

```
b= i.to_bytes(4,'big')
```
← b vaut b'\x00\x01\xaa\xbc'

résultat sur 4 octets

string ↔ bytes ↔ int



string= caractère
affichage

bytes = octets
représentation mémoire

int = entier
valeur numérique

Le type "list"

Une liste est une collection ordonnée d'objets.
Une liste supporte les opérations d'indexation et de slice.
Contrairement au type string une liste est modifiable.

```
>>> L=[2,"bonjour",2.34,43,"stop"]
>>> L
[2, 'bonjour', 2.34, 43, 'stop']
>>> L[0]
2
>>> L[-1]
'stop'
>>> L[1]=345
>> L[2:4]=['toto',12,'titi']
>>> L[2:4]=[23,'toto','titi']
>>> L
[2, 345, 23, 'toto', 'titi', 'stop']
```

L[1] est modifié

remplacement des éléments 2 et 3 par 3 éléments

Liste : ajouter/retirer

Ajouter/retirer un élément à la fin ou au début :

```
>>> L.append('dernier')
>>> L
[2, 345, 23, 'toto', 'titi', 'stop', 'dernier']
>>> L.pop()
'dernier'
>>> L
[2, 345, 23, 'toto', 'titi', 'stop']
>>> L=['premier']+L[1:]
>>> L
['premier', 345, 23, 'toto', 'titi', 'stop']
>>> L.reverse()
>>> L
['stop', 'titi', 'toto', 23, 345, 'premier']
>>>
```

Opérations sur les listes

Ajout, Multiplication par un entier :

```
>>> L=[1,2,3]
>>> M=[4,5,6]
>>> L+M
[1, 2, 3, 4, 5, 6]
>>> 3*L
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Test d'appartenance:

```
>>> 2 in L
True
>>> 0 in L
False
>>>
```

Itérations

Que fait ce script ?

```
>>> L=['a','b','c','a','b','b','z','c']
>>> M=[]
>>> for x in L:
...  if not (x in M) :
...    M.append(x)
...
>>>M
??????????????????
```

Opérations sur les listes

Trier des listes :

```
>>> L=['ba', 'x', 'aab', 'ab', 'x', 'xy', 'x']
>>> L.sort()
>>> L
['aab', 'ab', 'ba', 'x', 'x', 'x', 'xy']
>>> L=[-6,-2,3,5,-7,-9,0,1]
>>> L.sort()
>>> L
[-6, -2, 3, 5, -7, -9, 0, 1]
>>> L=[-6,-2,3,5,-7,-9,0,1]
>>> L.sort(key=vabs)
>>> L
[0, 1, -2, 3, 5, -6, -7, -9]
>>>
```

utilise l'ordre par défaut

On choisit le critère de tri

Définit une fonction |x|

```
>>> def vabs(x):
...     if x<0: return -x
...     return x
...
```

Quelques méthodes du type list

Créer une liste à partir d'une chaine (d'un objet itérable) :

```
>>> L=list('bonjour') ; L
['b', 'o', 'n', 'j', 'o', 'u', 'r']
```

Compter les occurrences d'un élément

```
>>> L.count('o') ; L
2
```

Supprimer un élément (le premier rencontré, doit exister)

```
>>> L.remove('o') ; L
['b', 'n', 'j', 'o', 'u', 'r']
```

Insérer un élément

```
>>> L.insert(2,'A') ; L
['b', 'n', 'A', 'j', 'o', 'u', 'r']
```

Position d'un élément (doit exister sinon exception)

```
>>> L.index('j')
3
```

Liste ↔ Chaîne

Transformer une chaîne en liste :

```
>>> s='abcdef'
>>> list(s)
['a', 'b', 'c', 'd', 'e', 'f']
>>> s='bonjour monsieur du corbeau'
>>> s.split(' ')
['bonjour', 'monsieur', 'du', 'corbeau']
```

Transformer une liste en chaîne :

```
>>> l=['a','b','c']
>>> ''.join(l)
'abc'
>>> '+'.join(l)
'a+b+c'
```

Définir une liste en "Compréhension"

Définir une liste en compréhension est un moyen commode de générer ou de filtrer une liste. La syntaxe s'inspire des mathématiques.

```
new_list = [function(item) for item in list if condition(item)]
```

Exemples :
Les carrés des entiers de 0 à 9 :

```
[ x**2 for x in range(10)]
```

Les racines carrées des nombres impairs inférieurs à 20 :

```
[math.sqrt(x) for x in range(20) if x%2!=0]
```

L'intersection de deux listes l1,l2 sans doublon :

```
[e for e in l1 if e in l2]
```

Attention à la copie de liste !

```
>>> a=[1,2,3]
>>> b=a
>>> a.append(4)
>>> a , b
([1, 2, 3, 4], [1, 2, 3, 4])
```

a et b sont identiques ! b est simplement une nouvelle référence de la liste a

```
>>> a is b
True
```

Une solution :

```
>>> c=list(a)
```
ou alors `c=a[:]`

```
>>> a.append(5)
>>> a,b,c
([1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4])
```

Attention à la copie de liste !!

Autre solution :

```
>>> a=[1,2,3]
>>> import copy
>>> b=copy.copy(a)
>>> a.append(4)
>>> a,b
([1, 2, 3, 4], [1, 2, 3])
```

Mais si une liste contient une autre liste ça ne marche plus !

```
>>> a=[[1,2,3],'a','b']
>>> b=copy.copy(a)
>>> a[0].append(4) # on rajoute 4 à la liste élément de a
>>> a,b
([[1, 2, 3, 4], 'a', 'b'], [[1, 2, 3, 4], 'a', 'b'])
```

Attention à la copie de liste !!!

Lorsqu'une liste contient d'autres listes il faut utiliser deepcopy() :

```
>>> import copy
>>> a=[[1,2,3],'a','b']
>>> b=copy.deepcopy(a)
>>> a[0].append(4)
>>> a,b
([[1, 2, 3, 4], 'a', 'b'], [[1, 2, 3], 'a', 'b'])
>>>
```

Les files

Une file est une structure de données dynamique dans laquelle on insère des nouveaux éléments à la fin (queue) et où on enlève des éléments au début (tête de file).

Le premier élément inséré sera le dernier retiré.
On a un comportement du type : FIFO (First In First Out)

Les files sont utilisées :

- pour mettre en attente des traitements :
  - file d'attente de requêtes d'un serveur
  - file de stockage de messages ou de commandes en attente de traitement.
  - simulation : arrivée de clients, ...

File (queue, fifo) en python

On peut utiliser append() pour ajouter un élément et et pop(0) pour en retirer.

```
queue = []              # la file est vide

queue.append(1)         # la file contient [1]
queue.append(2)         # la file contient [1, 2]
queue.append(3)         # la file contient [1, 2, 3]

result = queue.pop(0)  # la file contient [2, 3]

print(result)
print(queue)
```

Attention : il faut tester si la pile n'est pas vide avant de faire pop.

Les piles (stack)

Une pile est une structure de donnée qui a la propriété suivante :

Lorsqu'on ajoute un élément, on l'ajoute sur le dessus de la pile.
Lorsqu'on retire un élément, on retire l'élément situé sur le dessus de la pile.

On a donc un fonctionnement de type :
premier entré , dernier sorti.
ou **L**ast **I**n , **F**irst **O**ut : **LIFO**

Les structures de type piles sont utilisées par exemples :
pour évaluer des expressions mathématique
pour sauver les adresses de retour lors des appels de fonctions
Suppression des dernières actions : Control Z

Pile (lifo) en python

On peut utiliser append() pour ajouter un élément et et pop() pour en retirer.

```
stack = []              # la pile est vide
stack.append(1)         # la pile contient [1]
stack.append(2)         # la pile contient [1, 2]
stack.append(3)         # la pile contient [1, 2, 3]
result = stack.pop() # la pile contient [1, 2]

print(result)
print(stack)
```

Attention : il faut tester si la pile n'est pas vide avant de faire pop.

Dictionnaires

Un dictionnaire est une collection de données auxquelles on accède par une **clé**. Les dictionnaires sont modifiables mais ne sont pas itérables.(contrairement aux chaines et aux listes)

```
>>> D={'banane':3.5,'pain':2.2,'jambon':3.5}
>>> D['pain']
2.2
>>> 'pain' in D
True
>>> D['fromage']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'fromage'
>>> D['fromage']=4.2
>>> D
{'banane': 3.5, 'jambon': 3.5, 'pain': 2.2,
 'fromage': 4.2}
```

L'accès à une clé inexistante provoque une exception

Ajout d'une entrée

Dictionnaires

Avec la méthode get() l'accès à une clé inexistante ne provoque pas d'exception.

```
>>> D={'banane':3.5,'pain':2.2,'jambon':3.5}
>>> D.get('pain')
2.2
>>> D.get('pomme')
>>>
```

On peut aussi définir une valeur par défaut pour les clés inexistantes :

```
>>> D.get('pomme',0)
0
```

On peut supprimer une entrée :

```
>>> del D['banane']
```

Mais il faut vérifier l'existence avant (avec in par exemple) car provoque une exception Si la clé n'existe pas

Dictionnaires

La méthode update() met à jour un dictionnaire:
Les nouvelles entrées (clés) sont ajoutés, les valeurs des clés existantes sont remplacées :

```
>>> D1
{'saucisse': 4, 'jambon': 3.5, 'pain': 2.5}
>>> D2
{'pain': 3.7, 'frites': 4}
>>> D1.update(D2)
>>> D1
{'saucisse': 4, 'jambon': 3.5, 'pain': 3.7,
                                        'frites': 4}
```

On peut aussi supprimer une clé en récupérant sa valeur :

```
>>> D1.pop('jambon')
3.5
>>> D1
{'saucisse': 4, 'pain': 3.7, 'frites': 4}
```

La clé doit exister (vérifier avant)

Attention à la copie de dictionnaire !

Comme avec la copie de liste, il faut utiliser copy et deepcopy lorsqu'un dictionnaire contient des listes ou des dictionnaires.

```
>>> d={'d':{'x':-1,'y':-2},'a':1,'b':2}
>>> c=copy.copy(d)
>>> d['d']['z']=-3
>>> print(d) ; print(c)
{'a': 1, 'b': 2, 'd': {'y': -2, 'x': -1, 'z': -3}}
{'a': 1, 'b': 2, 'd': {'y': -2, 'x': -1, 'z': -3}}

>>> d={'d':{'x':-1,'y':-2},'a':1,'b':2}
>>> c=copy.deepcopy(d)
>>> d['d']['z']=-3
>>> print(d) ; print(c)
{'a': 1, 'b': 2, 'd': {'y': -2, 'x': -1, 'z': -3}}
{'a': 1, 'b': 2, 'd': {'y': -2, 'x': -1}}
```

Les fonctions

Forme générale :

```
 def nomdelafonction (param1,param2,...) :
      .....
      .....
      return  r1,r2,....
```

Exemple

```
 def   sommeproduit( a , b ) :
         s = a+b
         p = a*b
         return s,p
```

Appel

```
>>  x,y = sommeproduit(3,4)
>>  x,y
7,12
```

Les fonctions

On peut donner des valeurs par défauts aux paramètres. Les paramètres sans valeur par défaut doivent apparaître en premier.

```
def find ( s, c='a'):
    if c in s :
        return 1
    else:
        return 0
```

Appels possibles :

```
find('toto')
find('toto','t')
find('toto',c='t')
```

Accès aux fichiers

lire un fichier ligne par ligne :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

filename = 'texte.txt'
nl = 0
try :
 myfile = open (filename)
 for line in myfile:
     nl +=1
     print ('line :'+str(nl)+' : '+line,end='')
 print ('Il y a '+str(nl)+' lignes')
 myfile.close()
except :
 print ('impossible de lire ' + filename)
```

Accès aux fichiers par ligne

```
filename = 'texte.txt'
myfile = open (filename)

while 1:
    line = myfile.readline() # lecture d'une ligne
    if line == '' :
          Break # fin du fichier
    line = line.strip() # enlève les fin de ligne
    if line == '' :
        continue  # ne pas traiter les lignes vides
    # exploiter la chaîne line
myfile.close()
```

Préciser l'encodage à l'ouverture

Lorsqu'on ouvre un fichier avec open, l'encodage utilisée est celui utilisé par défaut sur le système. Pour un linux :

```
>>> import locale
>>> locale.getpreferredencoding()
'UTF-8'
```

Si le fichier a un encodage différent on peut le préciser lors de l'ouverture par le paramètre encoding :

```
file = open( 'fichier.txt', encoding='iso-8859-1' )
```

Accès aux fichiers par caractères

lire un fichier caractère par caractère :

```
filename = 'texte.txt'
fichier = open(filename,'r')
while 1:
  c = fichier.read(1)
  if c=='':
    break
  # exploiter c
fichier.close()
```

Accès aux fichiers par octets

Attention à cause des différents types d'encodage,
un caractères est parfois représenté par plusieurs octets.
Ex : 'é' est encodé 0xC3,0xA9 en UTF8 et 0xE9 en ascii latin-1

lire un fichier octets par octets :

```
filename = 'texte.txt'
fichier = open(filename,'rb')
while 1:
  b = fichier.read(1)
  if b==b'':                # b'' est un byte vide
    break
  # exploiter b
fichier.close()
```

A utiliser également pour lire des fichiers "binaire": image, exécutable,...

Accès aux fichiers

Une autre méthode : lire le fichier en une seule fois

```
filename = 'texte.txt'
fi = open(filename,'r')

for c in fi.read():
  #exploiter c
fi.close()
```

Ne pas utiliser pour un très gros fichier car la totalité du fichier doit tenir en mémoire : L'objet fi.read() est une chaîne ayant la taille du fichier !

Fermeture automatique

Le mot clé **with** permet de fermer le fichier "automatiquement " à la fin :

```
with open('monfichier.txt') as f :
    for c in f.read() :
      print(c, endl='')
```

Pour émettre un message d'erreur si le fichier est inaccessible :

```
try:
    with open("monfichier.txt") as f:
        for l in f :
            print(l)
except Exception as error:
    print('Error !!')
```

Ouvrir deux fichiers avec with

On peut ouvrir plusieurs fichiers en même temps :

```
with open('entree.txt') as fr ,\
     open('sortie.txt','w') as fw  :
   for c in fr.read() :
       fw.write(2*c)
```

Accès aux fichiers

Modes d'ouvertures : 'r' par défaut
'a' : append , 'w' write

```
try:
  myfile = open('/tmp/sortie.txt','a')
except:
  print ( "impossible d'écrire !")
  exit(-1)

myfile.write('quelque chose\n')
myfile.write('autre chose\n')
myfile.flush()
myfile.close()
```

Force une écriture

Ferme le fichier

import

import sert à "importer" des modules

Soit le fichier affiche.py ci-dessous :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
def multi_print ( s , n=1) :
  for i in range(n):    # ou bien print(n*s)
      print (s)

def reverse_print ( s ) :
  rv = s[::-1]
  print (rv)
```

import

Importe la totalité du module

```
>>>import affiche
>>>affiche.reverse_print('abc')
cba
>>>affiche.multi_print('ab',2)
ab
ab
```

Importe une fonction pour l'utiliser directement

```
>>> from affiche import reverse_print
>>> reverse_print('1234')
4321
```

Importe une fonction et la renomme

```
>>> from affiche import reverse_print as rvp
>>> rvp('345')
543
```

Importe la totalité du module pour utilisation directe

```
>>> from affiche import *
```

Test d'un module

On peut ajouter directement le test d'un module directement à l'intérieur de celui-ci :

```
if __name__ == '__main__':
    print('Test du module')
    print('multi_print :')
    multi_print('hi! ',3)
    print('reverse_print')
    reverse_print('hi!')
```

Les lignes ci-dessus ne sont exécutées que si le fichier est exécuté directement et pas lors d'une importation.

Modules standards

Très riche : voir python standard library :

os : une manière portable d'accéder à des fonctionnalités dépendant de l'os : chdir(),getlogin(), ...

math,random,time : fonctions mathématique, nombres aléatroire, temps

Fonctionnalités avancées :
socket, smtplib, email , gzip, audioop,etc....

Et de très nombreux modules librement disponibles :
serial, cherrypy

Exercices

Avec l'interpréteur, entraînez vous à manipuler les slices avec des chaînes :

- Récupérer les deux derniers caractères,
- Insérer une chaîne à une position donnée,
- Prendre un caractère sur deux,....

Écrire un programme qui indique si une chaîne est un palindrome, qui compte les voyelles d'une chaîne
Écrire un jeu du pendu (vous pouvez prendre les mots au hasard dans un fichier)
Écrire un programme de tirage de loto sans doublon
Écrire un programme qui produit des statistiques sur un texte : % d'occurrence des lettres, des mots ....

numpy

Numpy est le module de calcul scientifique de python.

Voir : http://wiki.scipy.org/Tentative_NumPy_Tutorial

Installation : **`sudo apt-get install python3-numpy`**

Import classique : **`import numy as np`**

Type fondemantal : **numpy array**
représente vecteurs et matrices

Array creation

A partir d'une liste (ou d'un tuple) :

```
x = np.array([1,2,3]) ,   y = np.array((4,5,6))
```

Avec linspace : bornes respectées (pas forcément le pas)

```
np.linspace(0,1,10)
array([0.,0.11111111,0.22222222,0.33333333,0.44444444,
0.55555556,0.66666667,0.77777778,0.88888889,1.])
```

Avec arange : le pas est respecté
(pas forcément les deux bornes)

```
np.arange(0,1,0.1)
array([0.,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9])
```

Opérations sur les array

Les opérations sont "broadcastées" sur tous les éléments :

```
x = np.arange(0,1,0.1) # x : array([0, 1, 2, 3, 4])
y = 2*x                # y : array([0, 2, 4, 6, 8])
z = x + 2              # z : array([2, 3, 4, 5, 6])
w = x + y              # w : array([0, 3, 6, 9, 12])
t = x**2               # t : array([0, 1, 4, 9, 16])
```

Mais aussi pour les fonctions (de numpy) :

```
a = np.sin(x) # a : array([0.,0.84147098,0.90929743,
                           0.14112001, -0.7568025 ])
```

Array multidimmensionnels

```
>>> a=np.array([[1,2,3],[4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.shape
(2, 3)
>>> b=np.array([7,8,9,10,11,12])
>>> b
array([ 7,  8,  9, 10, 11, 12])
>>> b.shape
(6,)
>>> b=b.reshape(3,2)
>>> b
array([[ 7,  8],
       [ 9, 10],
       [11, 12]])
```

Attention : b.reshape retourne un nouvel array.

Calcul matriciel

Les sommes et les différences se font avec les opérateurs usuels + et - :

```
>>> a
array([[1, 2, 3],
       [4, 5, 6]])

>>> a+1
array([[2, 3, 4],
       [5, 6, 7]])

>>> 2*a
array([[ 2,  4,  6],
       [ 8, 10, 12]])

>>> b
array([[ 7,  8,  9],
       [10, 11, 12]])

>>> a+b
array([[ 8, 10, 12],
       [14, 16, 18]])
```

Le produit matriciel est **dot** :

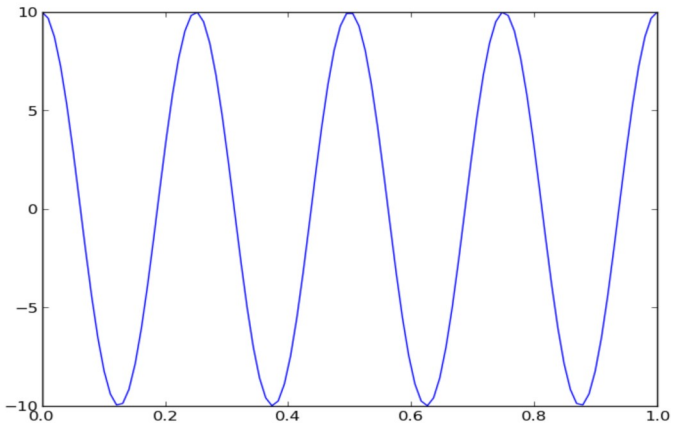
```
>>> b=b.reshape(3,2)
>>> b
array([[ 7,  8],
       [ 9, 10],
       [11, 12]])

>>> np.dot(a,b)
array([[ 58,  64],
       [139, 154]])

>>> np.dot(b,a)
array([[ 39,  54,  69],
       [ 49,  68,  87],
       [ 59,  82, 105]])
```

Matplotlib

Matplotlib est une bibliothèque qui permet de créer facilement des représentations graphiques de très bonnes qualités.
Voir : http://matplotlib.org/

Exemple :

```
import numpy as np
import matplotlib.pyplot as plt
f = 4.0 # 4Hz
t = np.linspace(0, 1,100)
v = 10*np.sin(2*np.pi*f*t + np.pi/2)
plt.plot(t,v)
plt.show()
```



On peut sauver l'image avec :

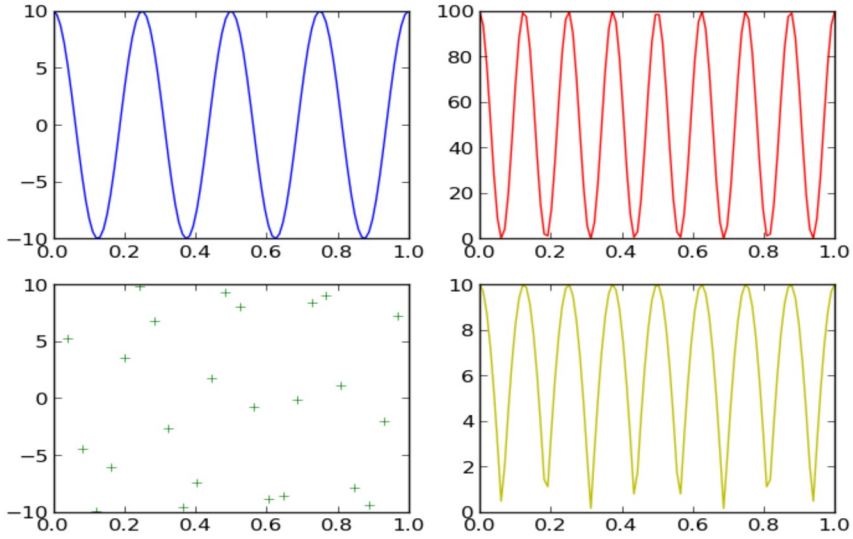
```
plt.savefig('image.jpg')
```

Plusieurs graphes sur une même figure

Avec subplot :

```
plt.subplot(221) # 2 lignes, 2 colonnes, plot 1
plt.plot(t,y)
plt.subplot(222) # 2 lignes , 2 colonnes, plot 2
plt.plot(t,y**2,'r') # y² en rouge
plt.subplot(223)
plt.plot(t[::4],y[::4],'g+')# un point sur 4
plt.subplot(224)
plt.plot(t,np.abs(y),'y')
plt.show()
```

221	222
223	224



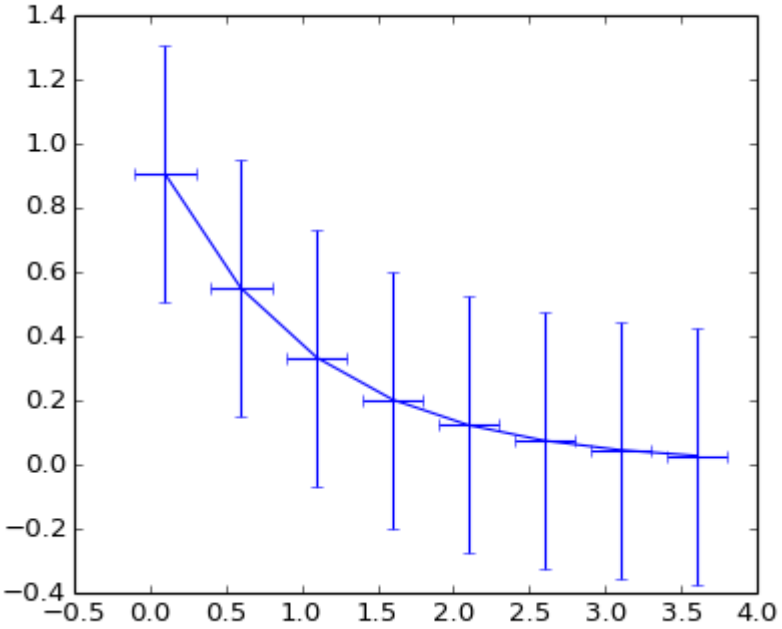
Quelques autres possibilités :

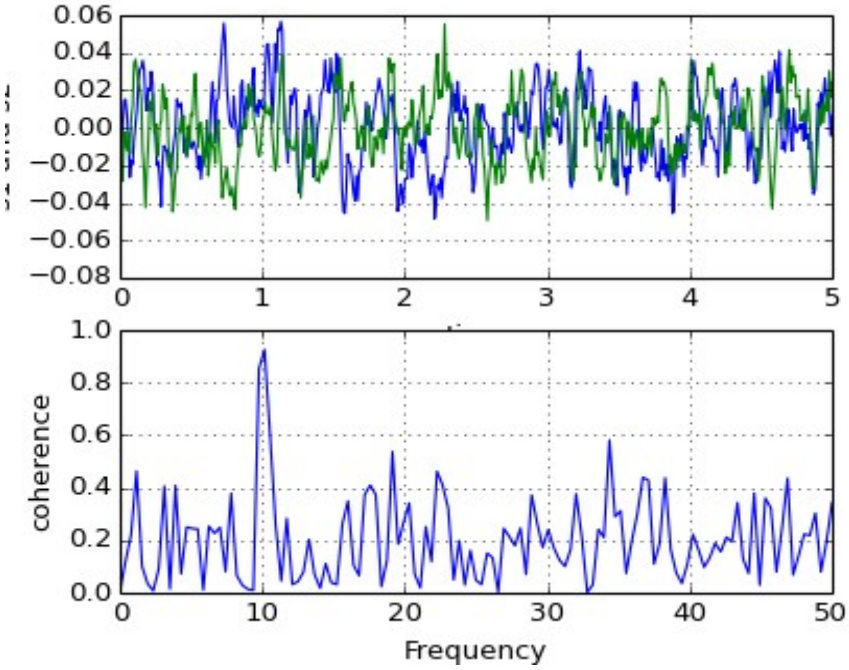
211	211212		231	232	233
212			234	235	236

Matplotlib

Un aperçu des nombreuses possibilités de matplotlib :
http://matplotlib.org/gallery.html

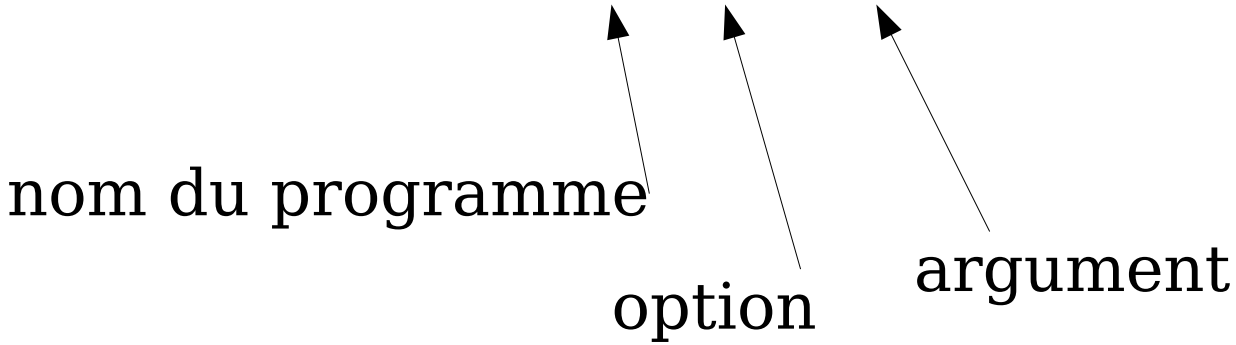
Le code est donné pour chaque exemple. C'est un moyen très rapide d'apprendre les bases de matplotlib.





Command line parsing

La ligne de commande peut être utilisée pour entrer des données dans un programme.
Exemple le programme linux wc avec la ligne de commande : **`$ wc -c fichier.txt`**



Il existe des bibliothèques qui permettent d'analyser facilement la ligne de commande dans la plupart des langages.
En python on dispose (entre autre) du module **argparse**.
La description suivant est très succincte.
Voir les très nombreuses possibilités à :
https://docs.python.org/3.6/library/argparse.html
(pour le C voir getopt)

Module argparse

On crée d'abord l'objet *parser* en renseignant la "description" et "l'épilogue". Ces deux chaînes apparaîtront respectivement avant et après l'aide.

```
#!/usr/bin/env python3
import argparse
# create parser
parser = argparse.ArgumentParser(
    description='A simple argument parsing demo',
    epilog = 'please report comments/bugs to\
              arlotto@univ-tln.fr')
```

Module argparse

On peut ensuite ajouter des options. Quelques exemples :

```
#optional boolean argument
parser.add_argument('-v',
           action='store_true',help='verbose mode')
#optional integer argument
parser.add_argument('-s','--speed',
                    type=int,default=9600)
#optional string argument
parser.add_argument('-o','--output',
                    default='out.data')
#optional multiple choice
parser.add_argument('-p', '--parity',
                    choices=['odd', 'even', 'none'],
                    default='none')
```

Module argparse

On peut ensuite ajouter des arguments positionnels. Ils seront repéré par leur ordre d'apparition sur la ligne de commande.

```
#add positionnal arguments

#position 1 (required no default)
parser.add_argument('filename')

#position 2 (not required)
parser.add_argument('configfile',
                    nargs='?',default='conf.data')
```

Module argparse

Quand on va lancer le programme, la ligne de commande va être analysée et les attributs de args pourront être utilisés pour modifier le comportement du programme

```
# parse command line arguments
args = parser.parse_args()
#use in your program
print(args)
print('speed :',args.speed)
print('verbose :',args.v)
print('output :',args.output)
print('config :',args.configfile)
print('filenmane :',args.filename)
print('parity :',args.parity)
```

Module argparse

L'analyse est robuste et reprend quasiment toutes les les possibilités des programmes linux.

```
$./parse1.py --help
usage: parse1.py [-h] [-v] [-s SPEED] [-o OUTPUT] [-p {odd,even,none}]
                 filename [configfile]

A simple argument parsing demo

positional arguments:
  filename
  configfile

optional arguments:
  -h, --help            show this help message and exit
  -v                    verbose mode
  -s SPEED, --speed SPEED
  -o OUTPUT, --output OUTPUT
  -p {odd,even,none}, --parity {odd,even,none}

please report comments/bugs to arlotto@univ-tln.fr
$
```

Module argparse

Exemple :

```
$ ./parse1.py fichier.txt config.ini -v -s 19200 -p even
Namespace(configfile='config.ini', filename='fichier.txt', output='out.data',
parity='even', speed=19200, v=True)
speed : 19200
verbose : True
output : out.data
config : config.ini
filenmane : fichier.txt
parity : even
```

Et la gestion des erreurs :

```
$ ./parse1.py fichier.txt config.ini -v -s 19200 -p paire
usage: parse1.py [-h] [-v] [-s SPEED] [-o OUTPUT] [-p {odd,even,none}]
                 filename [configfile]
parse1.py: error: argument -p/--parity: invalid choice: 'paire' (choose from
'odd', 'even', 'none')
```

Exécuter des commandes linux depuis python

Un moyen simple d'exécuter des commandes bash depuis Python :

```
import os
f = os.popen('ls -l')  # exécuter la commande ls -t
s=f.read()             # on lit le résultat renvoyé dans s
```

La fonction revient immédiatement mais le résultat n'est disponible qu'à la fin de la commande :

```
f = os.popen('pwd  ; sleep 10  ; echo fin')
print(f.read())
```

Application web avec cherrypy

Le module cherrypy (http://www.cherrypy.org) permet d'ajouter facile une interface web à une application python.

Fichier de conf minimal 'serveur.conf':

```
[global]
server.socket_host = "127.0.0.1"
server.socket_port = 8080
server.thread_pool = 5
tools.sessions.on = True
tools.encode.encoding = "Utf-8"
[/annexes]
tools.staticdir.on = True
tools.staticdir.dir = "annexes"
```

Un serveur web en quelques lignes

```
import cherrypy

class MonSiteWeb(object):
      # Classe maîtresse de l'application
  def index(self):# Méthode invoquée comme URL racine (/)
      return "<h1>Hello World, this is cherrypy !</h1>"
  index.exposed = True  # rend la page visible

# Programme principal du serveur
cherrypy.quickstart(MonSiteWeb(), config="serveur.conf")
```

Le serveur renvoie la valeur de retour de la fonction index (ici du html).

Ajouter des pages

Pour ajouter des pages, il suffit d'ajouter des méthodes à la classe principale.

```
def autre_page:
        Instructions python
        ....
        return 'du html'
autre_page.exposed = True
```

On peut créer un lien html vers cette nouvelle page dans index par exemple :

```
<a href="/autre_page">Cliquez pour voir l'autre page</a>
```

On peut utiliser un chemin relatif pour revenir

```
 <a href="../">Retour</a>
```

Récupérer des données du client

On peut utiliser la balise html form avec la méthode GET:

```
<form action="page_reponse" method="GET">

    Quel est votre nom ?

    <input type="text" name="nom" />

    <input type="submit" value="OK" />

    </form>
```

Page à afficher après la saisie du formulaire

Type de bouton

Nom de la variable qui sera passé à la méthode de la page action

Récupérer des données du client

```
def page_reponse(self, nom =None):

    if nom:       # un nom est bien fourni
       html = 'Bonjour :' + nom
       return  html
    else:         # aucun nom n'a été fourni :
       return 'Veuillez fournir votre nom\\
                                  <a href="/">ici</a>.'

  salutations.exposed = True
```

Dans le cas où il y plusieurs champ dans le formulaire, on a plusieurs paramètres passés à la méthode de la page :

```
def page_reponse(self,nom=None,prenom=None,age=18):
```

Utiliser les cookies

Le serveur envoie une suite aléatoire d'octet au client, le coockie, à la première connexion. Le client renvoie la valeur du cookie lorsqu'il visite chaque page. Ce cookie permet au serveur de retrouver les paramètres propres à ce client.

```
def page_lire_cookie(self):
    ma_variable = cherrypy.session.get('var', 2)
    s = str(ma_variable)
    return s
 page_cookie.exposed = True
```

Nom dans le dictionnaire de session

Valeur par défaut si var absent

Utiliser les cookie

Lire une variable, la modifier et la sauver dans le dictionnaire de session :

```
def page_lire_et_modifier_cookie(self):
    v = cherrypy.session.get('var', 2)
    new_v = v + 1
    cherrypy.session['var'] = v
    return ''' var valait %d <br>
               var vaut maintenant %d''' % (v,new_v)
```