Mémento Python

`python3` *program.py*	exécuter le programme *program.py*
`ipython3`	console Python: utile pour tester des bouts de code, faire des calculs, …
`jupyter notebook`	création de documents mélangeant code Python, images, graphiques et texte (markdown, html/css, latex)

Types usuels

Exemples: (Le type des variables est détecté automatiquement).

			objet modifiable
	`bool`	booléen	`True` `False`	
	`int`	entier (sans limite de taille)	`k = 1075`	•
	`float`	réel en virgule flottante	`12.4` `x = 3.2e18`	•
	`complex`	nombre complexe	`2 + 3j`	•
`' '`	`str`	chaîne de caractères	`s = 'bon' "C'est " + s` *texte =* `""" Je peux m'étendre sur plusieurs lignes et utiliser des caractères "spéciaux" comme " ' \n \t ... """`	•
`[ ]`	`list`	liste d'objets	`[1, 3, 3, 'plus', 5.1] + [8]`	✓
`( , )`	`tuple`	liste non modifiable d'objets	`(1, 3, 3, 'plus', 5.1)` `(8, )`	•
`{ }`	`set`	ensemble	`{2, 4, 'ABC', 5.1}`	✓
`{ }`	`dict`	dictionnaire (ensemble de clé → valeur)	`{'A': 1, 'B': 2}`	✓

*objets **itérables** (objet conteneur dont on peut parcourir les éléments)*

Identifiants (noms de variables, fonctions, ...), affectations (=)

Affectation: `x = 3e8/2e-5`

Identifiant:
- sensible min/Maj
- accents acceptés
- _ est un identifiant possible

```
x, y, z = 4.5, 2.2, 'yes'   # affectation multiple (via tuple)
i = j = 0
i += 1      ⇔ i = i + 1
j *= 2      ⇔ j = j * 2
```

Exemple:
```
L = [1, 2, 3]
M = L
L.append(4)
```
M et L pointent vers le **même** objet, la liste [1,2,3], qui peut être modifiée.

Boucles

```
for variable in sequence:
    Bloc d'instructions
```
*tout objet **itérable** (list, str, tuple, set, dict, ...)*

bloc défini par l'indentation

► **Boucle sur des entiers**

`range(debut, fin, pas)` crée un objet itérable pour parcourir une **séquence d'entiers**. ⚠ *fin n'est pas inclus.*

```
for i in range(1, 3):        # pour i = 1, 2
for i in range(4):           # pour i = 0, 1, 2, 3
for i in range(4, 0, -1):    # pour i = 4, 3, 2, 1
```
Pour convertir range en liste: `list(range(debut, fin))`

► **Boucle sur des objets itérables**

```
for lettre in "Mamma mia"
for element in ['k', 7]
for x in reversed(sequence)
for index, element in enumerate(ma_liste)
    print("L'élément à l'index", index, "est", element)
for v1, v2 in zip(sequence1,sequence2)
```
Parcourir plusieurs séquences en parallèle

```
while expression_logique:
    Bloc d'instructions
```

Exemple:
```
x = 500
while x > 1.0:
    x = x/2
    print(x)
```

`continue`	passage immédiat au tour suivant
`break`	sortie immédiate de la boucle

Tests

```
if x > 0:
    print('positif')
else:
    print('négatif ou nul')
```

```
if s == 'oui':
    instructions
elif 2 < x < 5:
    instructions
else:
    instructions
```
Optionnel: 1 ou plusieurs clauses **sinon-si** (`elif` signifie *else if*).

Blocs d'instructions définis par l'**indentation du code**, par exemple 4 espaces ou 1 tabulateur.

⚠ **Obligation** d'indenter correctement le programme.

Un bloc indenté est toujours précédé par le caractère `:` à la ligne précédente.

`==`	égalité
`!=`	différent de
`> <`	sup./inf.
`>= <=`	sup./inf. ou =
`in`	dans

On peut comparer non seulement des nombres, mais aussi des **séquences** (objets de type str, list, ...)

Logique booléenne:
```
a and b
a or b
not a
True, False
```

`y = expression1 if x > 2 else expression2`

Maths

Opérateurs

`+ - * /`	*La division / donne toujours un float*
`//`	Division entière
`**`	puissance ⚠ *a^b signifie a XOR b*
`%`	modulo (reste de la division entière)
`@`	*(entre tableaux numpy)* multiplication matricielle ou produit scalaire
`abs(x)`	valeur absolue
`round(x,n)`	arrondir à *n* décimales

```
from math import *
```
↗ Définit `pi` et `e` et les fonctions:

```
sin(pi/2)          # vaut 1.0
sqrt(2)            # √2
log(e**3)          # ln(e³)
log(100, 10)       # log10(100)
radians(90), degrees(pi)
```
etc...

Appliquer une fonction à une liste / tableau:

① utiliser le module *numpy*
Exemple: `2*np.sin(T)` *T : tableau numpy*

② **générateur de séquences**
Ex: `[2*sin(x) for x in L]`

③ fonction **map()**:
Ex: `list(map(lambda x: 2*sin(x)), L)`
fonction x → 2 sin(x)

map() *crée un itérateur qui applique la fonction aux éléments de la liste.*

définit une fonction "en ligne" (fct anonyme, i.e. sans identificateur)

Entrée/Sortie: écran, fichiers

```
s = input('Votre réponse: ')
```
Chaîne de caractères
Convertir avec `int(s)` ou `float(s)`

```
print('texte', variable, ...)
```
Options suppl.: `sep=' '` `end='\n'` `file=f`

Formatage de chaînes de caractères: `'%format' % (variable)` ou `f'{variable:format}'`

Exemples:
```
'Variable %s = %8.3f' % ('x', x)          (codes de format du langage C)
'Variable {} = {} et y = {:.4f}.'.format('x', x, y)
f'Variable x = {x} et y = {y:^10.4f}.'
```
Python ≥ 3.6

alignement (optionnel): `<` *à gauche* `>` *à droite* `^` *centré*

(opt.) largeur = nb caractères — *(opt.) nb décimales* — *type:*
`d` *entier (*`b` *bin.,* `x` *hex.)*
`f` *flottant*
`e` *format scientifique*
`g` *général (sélection auto.)*
`s` *chaîne de caractères*

Fichiers: lecture

```
f = open('fichier.txt')
f.readline()    #retourne la ligne suivante
f.readlines()   #retourne une liste des lignes
f.read()        #chaîne avec contenu du fichier
```

Écriture

```
f = open('fichier.ext','w')
f.write('Ligne à écrire\n')
```
À la fin:
```
f.close()
```
`w` *write*
`a` *append*
`+` *read/write*
`r` *read*

```
with open('fichier.txt') as f:
    for ligne in f:
        print(ligne.rstrip())
```
*Avec un **bloc with**, le fichier est fermé automatiquement à la fin du bloc.*
← `ligne` contient le caractère "\n" à la fin. Remarque: Avec readline(), `ligne = ''` *si fin du fichier.*
← rstrip() supprime les caractères blancs (\n, \t et les espaces) à la fin.

Fonctions

```
def ma_fonction(arg1, arg2, arg_opt=9):
    '''Documentation opt. multi-ligne'''
    Bloc d'instructions
    return valeur
```
Valeur par défaut de l'argument optionnel

Appel: `v = ma_fonction(arg1=3, arg2='texte')`

Exemple:
```
def norme2(x, y):
    return x**2 + y**2
n = norme2(1.5, 3.0)
```

Pas de restriction sur le type d'objet retourné (→ tuple pour retourner plusieurs valeurs). Pas de 'return' ⇔ `return None`.

*Accès aux variables du prog. principal possible en **lecture**.*

Séquences (*seq* = list, tuple, str, ...)

`len(seq)`	nombre d'éléments
`seq[i]`	élément d'index i
`del seq[i]`	supprime l'élément d'index i
`min(seq), max(seq), sum(seq)`	

`seq + seq2`
`seq * 3`

L	i	s	t	e
0	1	−2	−1	

`seq[0]` 1er élément
`seq[-1]` dernier él.

Sous-liste: `seq[debut:fin:pas]`
⚠ `fin` *n'est **pas inclus**.*

Exemples:
```
L = [1,2,3,4]
L[3] = -4      # [1,2,3,-4]
L[:3]          # [1,2,3]  (début → élément 3 non inclus)
L[-2:]         # [3,-4]   (avant-dernier → fin)
L[:] = 1       # [1,1,1,1]
M = L[:]       crée une copie (superficielle) de la liste L
M = sorted(seq)   crée une liste triée
filter(x<0, seq)  Itérateur retenant les éléments selon la condition
                  Convertir, au besoin, avec list().
```

Générateur de séquence

```
expression for élément in itérateur if condition
```
Exemple: `[2*i for i in range(3)]  # liste [0,2,4]`

Méthodes spécifiques aux listes

```
L.append(élément)
L.insert(index, valeur)
L.remove(valeur)
L.index(valeur)
L.pop(index)   ⇔ del L[index] mais retourne la valeur de l'élément
L.sort()       trie la liste L en place (et retourne None)
```

Conversions de type

`type(variable)` #retourne le type de *variable*

`float` → `int`	`int(2.5)`, `round(2.5)`, `floor(-1.6)`, `ceil(-1.6)`
`str` → `int, float`	`int("5")`, `float("2.5")`
`int, float` → `str`	`str(2.5)`
`list` → `str`	`"-".join(['A', 'B', 'C'])` → `'A-B-C'`
`str` → `list`	`list(map(str, [.1,.5]))` → `['.1','.5']`
	`list("Ça va")` → `['Ç', 'a', ' ', 'v', 'a']`
	`"Ça va".split(" ")` → `['Ça', 'va']`

Dictionnaire

```
D = {'A': 1, 'B': 2}
D[new_key] = new_value
```
`D.keys()`
`D.values()`

```
for key, value in D.items():
    print('Clé:', key, 'Valeur:', value)
```

Bibliothèque standard

► Arguments du prog. `import sys;` `liste_args = sys.argv`

► Opérations sur fichiers/dossiers `chemin/acces/fichier.txt`

```
import os
import shutil
import glob
```
`os.path.basename(path)`
`os.path.dirname (path)`

`liste_fichiers = glob.glob("*.jpg")`

Fichiers / dossiers
```
shutil.copy(src, dest)
os.rename(src, dest)
os.remove(file_path)
```

Dossiers
```
os.getcwd()
os.listdir(path)
os.mkdir(path)
```

► Exécuter une commande shell
```
import subprocess as sproc
sproc.run('ls -l'.split(' '), \
stdout=sproc.PIPE).stdout.decode()
```

► Autres modules standard
`random, time`
`datetime, re`
etc...

Débogueur

```
import pdb; pdb.set_trace()
```
Entrer dans le débogguer

pdb>>>	**next**	(n)	**print** (p) *Affiche une expression*
	step in	(s)	**list** (l) *Affiche le code source*
	return	(r)	`!commande_python_à_exécuter`
	continue	(c)	↵ *répéter la dernière commande*
	quit	(q)	

Aide

`?` *module* (ou *type* ou autre) *raccourci ipython pour* **help**(...)
`dir(module)` *liste le contenu d'un module ou d'une classe d'objets.*

Cet aide-mémoire est un aperçu non exhaustif de commandes Python.
Documentation complète: http://docs.python.org
Auteur: Vincent Ballenegger (juillet 2018)

Module numpy (numerical python)

```
import numpy as np
```

- Tableaux d'éléments de même type
- Opérations mathématiques rapides sur des tableaux
  Exemple: `np.sin(tableau_ndarray)`
- Algèbre linéaire, nombres aléatoires, transf. de Fourier

Création de tableaux (type ndarray)

Tableaux à 1d : vecteur de *n* composantes

`np.zeros(n)`	vecteur de composantes 0
`np.ones(n)`	vecteur de composantes 1
`np.empty(n)`	vecteur non initialisé

Tableaux à 2d : *matrice de n lignes et m colonnes*

`np.zeros((n,m))`	matrice nulle
`np.identity(n)`	matrice identité
`np.eye(n, m, k)`	matrice avec 1 sur $k^{ième}$ diagonale
`np.diag([valeurs], k)`	matrice diagonale (cas k=0)

Tableaux à *d* dimensions : taille $N_1 \times N_2 \times ... \times N_d$ (cas général)

```
T = np.zeros(shape, dtype)
```

nb de valeurs dans chaque dimension: tuple (N1, N2, N3, ...)

(optionnel) ***type*** *des éléments:*

`float`	*réel* (`np.float16`, ..., 64)
`complex`	*nb complexe* (`np.complex64`, 128)
`int`	*entier* (`np.int8`, 16, 32, 64)
`np.uint`	*entier positif* (`np.uint8`, ..., 64)

```
T = np.random.rand(N1, N2, ...)
```

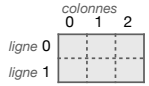
Crée un tableau de forme N_1, N_2, ... rempli de nombre aléatoires (loi de probabilité uniforme sur l'intervalle [0,1[ ).

```
T = np.fromfunction(function, shape)
```

Crée un tableau initialisé avec une fonction (voir help(np.fromfunction)).

Accès aux éléments

Exemple:
```
for i in range(3):
    for j in range(3):
        T[i,j] = ...
```
no de ligne / *colonne*



En général: `T[i,j,k,...]`

Vue sur une portion de tableau: `T[début:fin:incrément]` *(pas de copie)*

`T[k,:]`	ligne *k* du tableau	`T[i:i+h, j:j+l]`	sous-matrice *h* × *l*
`T[:,k]`	colonne *k* du tableau		

Copier un tableau `T2 = T1.copy()`

Fichiers de données

- Format texte

```
T = np.loadtxt('data.txt',options)
```

Options:

`skiprows=n`	ignorer les *n* 1res lignes
`comments='#'`	symbole des commentaires (# par défaut)
`delimiter=str`	séparateur entre les valeurs (esp./tab par défaut)
`unpack=bool`	si True: x, y, z = loadtxt(...)
`usecols=(0,2)`	lit les colonnes *0* et *2* uniquement

Fichier data.txt:

```
# x      y      z
1900    3e3    4e3
1920    2876   3e9
```

```
np.savetxt('data.out', fmt='%.3e', autres_options)
```

- Format binaire .npy

```
np.save('data.out', T)
np.load('data.out')
```

numpy.fft: transformées de Fourier

Voir `help(np.fft)` dans une console ipython3 (ou dans Google).

Attributs d'un tableau T

`T.ndim`	nb de dimensions (axes)
`T.shape`	tuple avec nb d'éléments dans chaque dimension `T.shape[0]` : nombre de lignes
`T.size`	nb total d'éléments
`T.dtype`	type des éléments (data-type)
`T.itemsize`	taille d'un élément (octets)

Conversions

list→array: `np.array([[1,2],[3,4]])` → $\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}$

array→list: `T.tolist()` → `[[1,2],[3,4]]`

`T.astype(dtype)` Copie du tableau converti au type spécifié

Opérations mathématiques

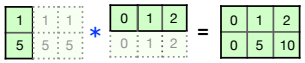
Les fonctions mathématiques de numpy s'appliquent à chaque élémen des tableaux. Ex:
`np.exp(tableau)` ✔ *fonctionne*
`math.exp(tableau)` ⚠ *erreur*

Somme, moyenne, min, max, écart-type (std):
`T.sum()`, `T.mean()`, `T.min()`, `T.max()`, `T.std()`

Somme le long d'un axe:
`T.sum(axis=0)` ⇔ T[0, :] + T[1, :] + ... *= somme des colonnes si 2d*
`T.sum(axis=1)` ⇔ T[:, 0] + T[:, 1] + ... *= somme des lignes si 2d*

Les opérations `+ - * / **` sont effectuées **terme-à-terme.**

Tableau `+` *nombre* ← *effectué sur chaque terme*



Calcul vectoriel et matriciel

- Produit matriciel (tableaux à 2d): `mat1 @ mat2`
- Produit scalaire entre 2 vecteurs (tableaux à 1d): `vec1 @ vec2`
- Produit vectoriel: `np.cross(vec1, vec2)`
- Produit extérieur de 2 vecteurs: `np.outer(vec1, vec2)`
- Norme d'un vecteur: `np.linalg.norm(vec)`
- Trace: `mat.trace()`
- Matrice transposée: `mat.T` ⚠ *sans effet sur un tableau à 1d*

En 1d: *pas de distinction entre vecteur ligne ou colonne.*
Les opérations mat@vec et vec@mat donnent le résultat escompté.

En 2d:
`vec.reshape((3,1))` *crée un vecteur-colonne (matrice de 3 lignes et 1 col.)*
`vec.reshape((1,3))` *crée un vecteur-ligne (matrice de 1 lignes et 3 col.)*

Autre notation : (Python < 3.5)

```
m1 @ m2 ⇔ np.dot(m1,m2) ⇔ m1.dot(m2)
v1 @ v2 ⇔ np.dot(v1,v2) ⇔ v1.dot(v2)
np.transpose(M) ⇔ m.transpose()
```

numpy.linalg: Méthodes d'algèbre linéaire

```
import numpy.linalg as la
```
← *pour abréger la notation*

`la.det(M)`	déterminant de la matrice *M*
`la.inv(M)`	matrice inverse
`la.eig(M)`	valeurs propres
`la.solve(A, B)`	renvoie *X* tel que $A \cdot X = B$
`la.matrix_rank(M)`	rang de M
`la.matrix_power(M, n)`	M^n

numpy.random: nombres aléatoires

`np.random.rand(N1, N2, ...)`
tableau de forme N_1, N_2, ... rempli de nombre aléatoires distribués selon la loi de probabilité uniforme sur l'intervalle [0,1[.

`np.random.randn(N1, N2, ...)` idem pour la loi normale

`np.random.randint(min, max, nb)`
nb nombres entiers aléatoires dans l'intervalle [min, max[(max est exclu).

`np.random.random_integers(min, max, nb)`
nb nombres entiers aléatoires dans l'intervalle [min, max] (max est inclus).

Module matplotlib: création de graphiques

```
import matplotlib.pyplot as plt
```

Notebook jupyter: - utiliser `%matplotlib inline` (ou notebook) pour l'importation
- instructions `plt.figure()` et `plt.show()` non obligatoires

Création d'un graphique

```
plt.figure(figsize=(8,5))
plt.plot(X, Y, label='mes_données')
plt.xlabel('légende axe x')
plt.ylabel('légende axe y')
plt.legend()
plt.show()
```

création d'un nouveau graphique
X, Y: ***listes ou tableaux numpy*** *contenant les coordonnées x, resp. y, des points.*
légendes des axes
affiche la légende des courbes (voir l'option 'label' de plot)
affiche le graphique

```
plt.xscale('log')
plt.axis('equal')
plt.xlim(xmin, xmax)
plt.savefig('fichier.pdf')
```

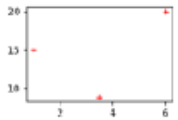
axe x en échelle log
même échelle pour les axes x et y
bornes de l'axe des x
enregistre le graphique

Création de tableaux de valeurs régulièrement espacées

`np.linspace(a, b, n)`	tableau (numpy) contenant *n* valeurs dans l'intervalle [*a*,*b*]
`np.logspace(a, b, n)`	*n* valeurs de 10^{-a} à 10^{-b} régulièrement espacées en échelle log
`np.arange(a, b, dx)`	*n* valeurs de *a* (inclus) à *b* (exclus) par pas *dx*

Graphique de points ou d'une fonction

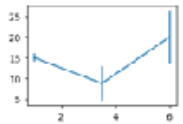
listes ou tableaux numpy



```
X = [1, 3.5, 6]
Y = [15, 8.8, 20]
plt.plot(X, Y, marker='+', ls='', c='r' )
```

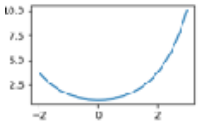
Options de plot():

`marker=`	`'o'`, `'s'`, `'*'`, `'+'`, `'x'`, `'^'`
`linestyle=`	`'-'`, `'--'`, `'_'`, `':'`
`color=`	`'r'`, `'g'`, `'b'`, `'DarkGreen'`, ...



```
X = [1, 3.5, 6]
Y = [15, 8.8, 20]
dY= [1., 4.2, 6.4]
plt.errorbar(X, Y, dY)
```

Graphique d'une fonction

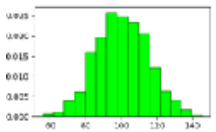


```
X = np.linspace(-2, 3, 100)
plt.plot(X, np.cosh(X))
```

⚠ *utiliser des tableaux numpy et les fonctions mathématiques de numpy*

`np.exp(tableau)`	✔ *correct*
`math.exp(tableau)`	⚠ *erreur*
`math.exp(liste)`	⚠ *erreur*

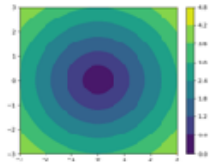
Histogramme



```
X = 100 + 15*np.random.randn(1000)
plt.hist(X, 15, normed=1, facecolor = 'lime', edgecolor = 'green')
```

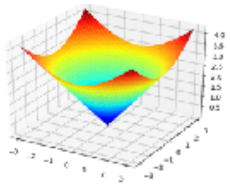
nombre de barres

Graphique en 3d: z = f(x,y)



```
X = np.linspace(-3, 3, 200)
Y = X
XX, YY = np.meshgrid(X, Y)
Z = np.sqrt(XX**2 + YY**2)
plt.contourf(XX, YY, Z)
plt.colorbar()
```

coordonnées selon axe x
coordonnées selon axe y
coordonnées x et y de ***tous les points*** *de la grille*
hauteurs: $z = \sqrt{x^2 + y^2}$
graphique de ***courbes de niveaux*** *(f = filled)*
barre avec légende des couleurs



```
from mpl_toolkits.mplot3d import Axes3D
ax = plt.subplot(111, projection='3d')
ax.plot_surface(XX, YY, Z, cmap='jet')
ax.set_aspect(1)
```

111 signifie graphique numéro 1 dans une grille de 1 x 1 graphiques.
cmap signifie colormap
même échelle pour les axe x, y et z