

Mémento Python 3 pour le calcul scientifique

©2019 – Éric Ducasse & Jean-Luc Charles

Licence Creative Commons Paternité 4

Version AM-1.52

Octobre 2024

Forme inspirée initialement du mémento de Laurent Pointal,
disponible ici : <https://perso.limsi.fr/pointal/python:memento>

dir(nom) liste des noms des méthodes
et attributs de **nom**

help(nom) aide sur l'objet **nom**

help("nom_module.nom") aide sur l'objet
nom du module **nom_module**

Aide
F1

Entier, décimal, complexe,
booléen, rien

Types de base

↳ objets non mutables

int 783 0 -192 0b010 0o642 0xF3
zéro binaire octal hexadécimal

float 9.23 0.0 -1.7e-6 (-1,7×10⁻⁶)

complex 1j 0j 2+3j 1.3-3.5e2j

bool True False

NoneType None (une seule valeur : « rien »)

Noms d'objets, de fonctions, de modules, de classes, etc.

a...zA...Z_ suivi de **a...zA...Z_0...9**

- accents possibles mais à éviter
- mots clés du langage interdits
- distinction casse min/MAJ

⌚ a toto x7 y_max BigOne
⌚ by and for

Symbol : = Affectation/nommage

↳ affectation ⇔ association d'un nom à un objet

nom_objet = <expression>

- 1) évaluation de l'expression de droite pour créer un objet
- 2) nommage de l'objet créé

x = 1.2 + 8 + sin(y)

Affectations multiples

<n noms> = <itérable de taille n>

u,v,w = 1j,"a",None

a,b = b,a échange de valeurs

Affectations combinée avec une opération ◊

x ♦= c équivaut à : **x = x ♦ c**

Suppression d'un nom

del x l'objet associé disparaît seulement s'il n'a plus de nom, par le mécanisme du « ramasse-miettes »

Conteneurs : opérations génériques

len(c) **min(c)** **max(c)** **sum(c)**
nom in c → booléen, test de présence dans **c**

d'un élément identique (comparaison ==) à **nom**

nom not in c → booléen, test d'absence

c1 + c2 → concaténation

c * 5 → 5 répétitions (**c+c+c+c+c**)

c.index(nom) → position du premier élément identique à **nom**

c.index(nom, idx) → position du premier élément identique à **nom** à partir de la position **idx**

c.count(nom) → nombre d'occurrences

Opérations sur listes

↳ modification « en place » de la liste **L** originale
ces méthodes ne renvoient rien en général

L.append(nom) ajout d'un élément à la fin

L.extend(itérable) ajout d'un itérable converti en liste à la fin

L.insert(idx, nom) insertion d'un élément à la position **idx**

L.remove(nom) suppression du premier élément identique (comparaison ==) à **nom**

L.pop() renvoie et supprime le dernier élément

L.pop(idx) renvoie et supprime l'élément à la position **idx**

L.sort() ordonne la liste (ordre croissant)

L.sort(reverse=True) ordonne la liste par ordre décroissant

L.reverse() renversement de la liste

L.clear() vide la liste

Conteneurs numérotés (listes, tuples, chaînes de caractères)

list [1,5,9]

["abc"]

[]

["x",-1j,["a",False]]

Conteneurs hétérogènes

tuple (1,5,9)

("abc" ,)

()

11,"y",[2-1j,True]

Objets non mutables

→ str

"abc"

"z"

Nombre d'éléments

len(objet) donne : 3

1

0

3

Singleton

Objet vide

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

3

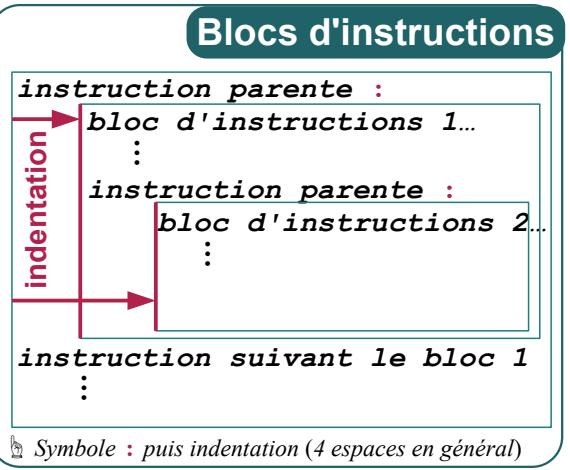
3

3

3

3

Mémento Python 3 pour le calcul scientifique



True/False Logique booléenne

- Opérations booléennes**
 - `not A` « non A »
 - `A and B` « A et B »
 - `A or B` « A ou B »
 - `(not A) and (B or C)` exemple
- Opérateurs renvoyant un booléen**
 - `nom1 is nom2` 2 noms du même objet ?
 - `nom1 == nom2` valeurs identiques ?
 - Autres comparateurs :
 - `< > <= >= != (=≠)`
 - `nom_objet in nom_iterable`
l'itérable `nom_iterable` contient-il un objet de valeur identique à celle de `nom_objet` ?

Conversions

```

bool(x) → False pour x: None,
0(int), 0.0(float), 0j(complex),
itérable vide
    → True pour x: valeur
    numérique non nulle, itérable non vide
int("15") → 15
int("15", 7) → 12 (base 7)
int(-15.56) → -15 (troncature)
round(-15.56) → -16 (arrondi)
float(-15) → -15.0
float("-2e-3") → -0.002
complex("2-3j") → (2-3j)
complex(2,-3) → (2-3j)
list(x) Conversion d'un itérable en liste
    exemple : list(range(12,-1,-1))
sorted(x) Conversion d'un itérable en
    liste ordonnée (ordre croissant)
sorted(x, reverse=True)
    Conversion d'un itérable en liste ordonnée
    (ordre décroissant)
tuple(x) Conversion en tuple
"{}".format(x) Conversion en
    chaîne de caractères
ord("A") → 65 ; chr(65) → 'A'

```

Mathématiques

```

Opérations
+ - * /
** puissance 2**10 → 1024
// quotient de la division euclidienne
% reste de la division euclidienne
Fonctions intrinsèques
abs(x) valeur absolue / module
round(x, n) arrondi du float x à n
    chiffres après la virgule
pow(a,b) équivalent à a**b
pow(a,b,p) reste de la division
    euclidienne de ab par p
z.real → partie réelle de z
z.imag → partie imaginaire de z
z.conjugate() → conjugué de z

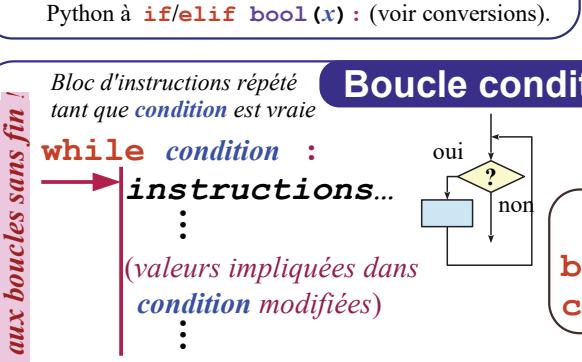
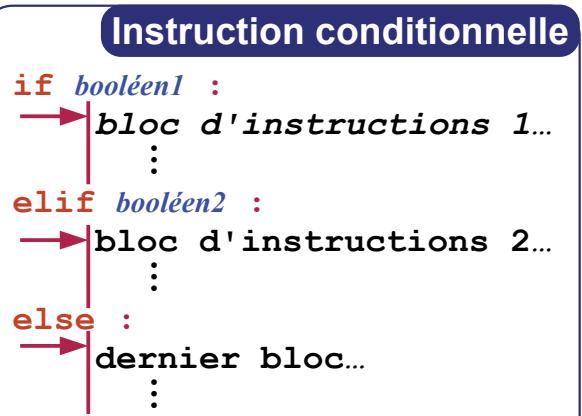
```

sys

```

sys.path → liste des chemins des dossiers
    contenant des modules Python
sys.path.append(chemin)
    Ajout du chemin absolu d'un
    dossier contenant des modules
sys.platform → nom du système
    d'exploitation

```



Exemple

```

from random import randint
somme, nombre = 0, 0
while somme < 100 :
    nombre += 1
    somme += randint(1, 10)
print(nombre, ";", somme)

```

Le nombre d'itérations n'est pas connu à l'avance

Liste en compréhension

- Inconditionnelle / conditionnelle
- `L = [f(e) for e in itérable]`
- `L = [f(e) for e in itérable if b(e)]`

Fichiers texte

- N'est indiquée ici que l'ouverture avec fermeture automatique, au format normalisé UTF-8.
- Le « `chemin` » d'un fichier est une chaîne de caractères (voir module `os` ci-dessous)
- Lecture intégrale d'un seul bloc**

```

with open(chemin, "r", encoding="utf8") as f:
    texte = f.read()

```

- Lecture ligne par ligne**

```

with open(chemin, "r", encoding="utf8") as f:
    lignes = f.readlines()
    (Nettoyage éventuel des débuts et fins de lignes)
    lignes = [c.strip() for c in lignes]

```

- Écriture dans un fichier**

```

with open(chemin, "w", encoding="utf8") as f:
    f.write(début) ...
    :
    f.write(suite) ...
    :
    f.write(fin)

```

Quelques modules internes de Python (The Python Standard Library)

import os

`os.getcwd()` → Chemin absolu du « répertoire de travail » (working directory), à partir duquel on peut donner des chemins relatifs.

Chemin absolu : chaîne commençant par une lettre majuscule suivie de ":" (Windows), ou par "/" (autre)

Chemin relatif par rapport au répertoire de travail `wd` :

- `nom de fichier` ↔ fichier dans `wd`
- `..` ↔ `wd` ; `..` ↔ père de `wd`
- `.../...` ↔ grand-père de `wd`
- `"sous-dossier/image.png"`

`os.listdir(chemin)` → liste des sous-dossiers et fichiers du dossier désigné par `chemin`.

`os.path.isfile(chemin)` → Booléen : est-ce un fichier ?

`os.path.isdir(chemin)` → Booléen : est-ce un dossier ?

`for sdp, Lsd, Lnf in os.walk(chemin) :`

Parcourt récursivement chaque sous-dossier, de chemin relatif `sdp`, dont la liste des sous-dossiers est `Lsd` et celle des fichiers est `Lnf`

Une fonction fait des actions et renvoie un ou plusieurs objets, ou ne renvoie rien.

def nom_fct(x,y,z=0,a=None) :

```

bloc d'instructions...
    :
    if a is None :
        :
    else :
        :
return r0,r1,...,rk

```

↓ Autant de noms que d'objets renvoyés

`a0,a1,...,ak = nom_fct(-1,2)`

`b0,b1,...,bk = nom_fct(3.2,-1.5,a="spline")`

Définition de fonction

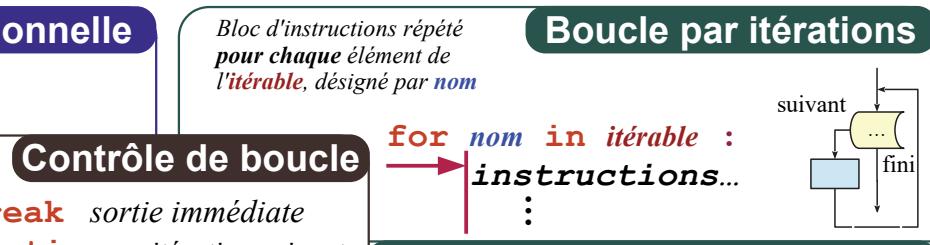
`x et y` : arguments positionnels, obligatoires
`z et a` : arguments optionnels avec des valeurs par défaut, nommés

↓ Plusieurs `return` possibles (interruptions)
 ↓ Une absence de `return` signifie qu'à la fin, `return None` (rien n'est renvoyé)

Appel(s) de la fonction

`a0,a1,...,ak = nom_fct(-1,2)`

`b0,b1,...,bk = nom_fct(3.2,-1.5,a="spline")`



Variantes avec parcours en parallèle

for a,b in itérable : Itérations sur des couples

for numéro, nom in enumerate(itérable) :

for numéro, nom in enumerate(itérable, d) :

for e1, e2, ... in zip(itérable1, itérable2, ...) :

Parcours en parallèle de plusieurs itérables ; s'arrête dès qu'on arrive à la fin de l'un d'entre eux

Gestion basique d'exceptions

```

try :
    bloc à essayer
except :
    bloc exécuté en cas d'erreur

```

x,y = -1.2,0.3

print("Pt",2,"(",x,",",",",y+4,")")

→ Pt 2 = (-1.2, 4.3)

↓ Un espace est inséré à la place de chaque virgule séparant deux objets consécutifs. Pour mieux maîtriser l'affichage, utiliser la méthode de formatage `str.format`

s = input("Choix ? ")

↓ `input` renvoie toujours une chaîne de caractères ; la convertir si besoin vers le type désiré

Importation de modules

Module `mon_mod` ⇒ Fichier `mon_mod.py`

- Importation d'objets par leurs noms**
- `from mon_mod import nom1, nom2`
- Importation avec renommage**
- `from mon_mod import nom1 as n1`
- Importation du module complet**
- `import mon_mod`
- `...`
- `... mon_mod.nom1 ...`
- Importation du module complet avec renommage**
- `import mon_mod as mm`
- `...`
- `... mm.nom1 ...`

Programme utilisé comme module

- Bloc-Test** (non lu en cas d'utilisation du programme `mon_mod.py` en tant que module)

```

if __name__ == "__main__":
    Bloc d'instructions
    :

```

from time import time

time

`debut = time()`

`: (instructions)`

`durée = time() - début`

↓ Évaluation d'une durée d'exécution, en secondes

Mémento Python 3 pour le calcul scientifique

Aide numpy/scipy

`np.info(nom_de_la_fonction)`

`import numpy as np`

Fonctions mathématiques

En calcul scientifique, il est préférable d'utiliser les fonctions de `numpy`, au lieu de celles des modules basiques `math` et `cmath`, puisque les fonctions de numpy sont vectorisées : elle s'appliquent aussi bien à des scalaires (`float`, `complex`) qu'à des vecteurs, matrices, tableaux, avec des durées de calculs minimisées.

`np.pi, np.e`

→ Constantes π et e

`np.abs, np.sqrt, np.exp, np.log, np.log10, np.log2`

→ `abs`, racine carrée, exponentielle, logarithmes népérien, décimal, en base 2

`np.cos, np.sin, np.tan` → Fonctions trigonométriques (angles en radians)

`np.degrees, np.radians` → Conversion radian→degré, degré→radian

`np.acccos, np.arcsin` → Fonctions trigonométriques réciproques

`np.arctan2(y, x)` → Angle dans $]-\pi, \pi]$

`np.cosh, np.sinh, np.tanh` (trigonométrie hyperbolique)

`np.arcsinh, np.arccosh, np.arctanh`

Tableaux numpy.ndarray : généralités

Un tableau `T` de type `numpy.ndarray` (« n-dimensional array ») est un conteneur homogène dont les valeurs sont stockées en mémoire de façon séquentielle.

`T.ndim` → « dimension d » = nombre d'indices (1 pour un vecteur, 2 pour une matrice)

`T.shape` → « forme » = plages de variation des indices, regroupées en `tuple (n0, n1, ..., nd-1)` : le premier indice varie de 0 à `n0-1`, le deuxième de 0 à `n1-1`, etc.

`T.size` → nombre d'éléments, valant `n0 × n1 × ... × nd-1`

`T.dtype` → type des données contenues dans le tableau (`np.bool`, `np.int32`, `np.uint8`, `np.float`, `np.complex`, `np.unicode`, etc.)

↳ `shp` est la forme du tableau créé, `data_type` le type de données contenues dans le tableau (`np.float64` si l'option `dtype` n'est pas utilisée)

générateurs

`T = np.empty(shp, dtype=data_type)` → pas d'initialisation

`T = np.zeros(shp, dtype=data_type)` → tout à 0/False

`T = np.ones(shp, dtype=data_type)` → tout à 1/True

▪ Tableaux de même forme que `T` (même type de données que `T` si ce n'est pas spécifié) :

`S = np.empty_like(T, dtype=data_type)`

`S = np.zeros_like(T, dtype=data_type)`

`S = np.ones_like(T, dtype=data_type)`

▪ Un vecteur `V` est un tableau à un seul indice

▪ Comme pour les listes, `V[i]` est le $(i+1)$ ème coefficient, et l'on peut extraire des sous-vecteurs par : `V[:2], V[-3:], V[:::-1]`, etc.

Si `c` est un nombre, les opérations `c*V, V/c, V+c, V-c, V//c, V%c, V**c` se font sur chaque coefficient

Si `U` est un vecteur de même dimension que `V`, les opérations `U+V, U-V, U*V, U/V, U//V, U%V, U**V` sont des opérations terme à terme

▪ Produit scalaire : `U.dot(V)` ou `np.dot(U, V)` ou `U@V`

Vecteurs

générateurs

`np.linspace(a, b, n)`

→ `n` valeurs régulièrement espacées de `a` à `b` (bornes incluses)

`np.arange(x_min, x_max, dx)`

→ de `x_min` inclus à `x_max` exclu par pas de `dx`

générateurs

`np.linspace(a, b, n)`

→ `n` valeurs régulièrement espacées de `a` à `b` (bornes incluses)

`np.arange(x_min, x_max, dx)`

→ de `x_min` inclus à `x_max` exclu par pas de `dx`

générateurs

↳ Sans l'option `axis`, un tableau est considéré comme une simple séquence de valeurs

`T.max(), T.min(), T.sum()`

Statistiques

`T.argmax(), T.argmin()` indices séquentiels des extrêmes

`T.sum(axis=d)` → sommes sur le $(d-1)$ -ème indice

`T.mean(), T.std(), T.std(ddof=1)` moyenne, écart-type

`V = np.unique(T)` valeurs distinctes, sans ou avec les effectifs

`V, N = np.unique(T, return_counts=True)`

`np.cov(T), np.corrcoef(T)` matrices de covariance et de corrélation ; `T` est un tableau $k \times n$ qui représente n répétitions d'un vecteur de dimension k ; ces matrices sont $k \times k$.

Modules random et numpy.random

Tirages pseudo-aléatoires

`import random`

`random.random()`

→ Valeur flottante dans l'intervalle $[0,1]$ (loi uniforme)

`random.randint(a, b)`

→ Valeur entière entre `a` inclus et `b` inclus (équiprobabilité)

`random.choice(L)`

→ Un élément de la liste `L` (équiprobabilité)

`random.shuffle(L)`

→ `None`, mélange la liste `L` « en place »

`import numpy.random as rd`

`rd.rand(n0, ..., nd-1)`

→ Tableau de forme (n_0, \dots, n_{d-1}) , de flottants dans l'intervalle $[0,1]$ (loi uniforme)

`rd.randint(a, b, shp)`

→ Tableau de forme `shp`, d'entiers entre `a` inclus et `b` exclu (équiprobabilité)

`rd.randint(n, size=d)`

→ Vecteur de dimension `d`, d'entiers entre `0` et `n-1` (équiprobabilité)

`rd.choice(Omega, n, p=probas)` → Tirage avec remise d'un échantillon de taille `n` dans `Omega`, avec les probabilités `probas`

`rd.choice(Omega, n, replace=False)` → Tirage sans remise d'un échantillon de taille `n` dans `Omega` (équiprobabilité)

↳ Le passage maîtrisé `list` ↔ `ndarray` permet de bénéficier des avantages des 2 types

Conversions

`T = np.array(L)` → Liste en tableau, type de données automatique

`T = np.array(L, dtype=data_type)` → Idem, type spécifié

`L = T.tolist()` → Tableau en liste

`new_T = T.astype(data_type)` → Conversion des données

`S = T.flatten()` → Conversion en vecteur (la séquence des données telles qu'elles sont stockées en mémoire)

`np.unravel_index(ns, T.shape)` donne la position dans le tableau `T` à partir de l'index séquentiel `ns` (indice dans `S`)

Matrices

▪ Une matrice `M` est un tableau à deux indices

▪ `M[i, j]` est le coefficient de la $(i+1)$ -ième ligne et $(j+1)$ -ième colonne

▪ `M[i, :]` est la $(i+1)$ -ième ligne, `M[:, j]` la $(j+1)$ -ième colonne,

`M[i:i+h, j:j+l]` une sous-matrice $h \times l$

▪ Opérations : voir Vecteurs

▪ Produit matriciel : `M.dot(V)` ou `np.dot(M, V)` ou `M@V`

`M.transpose(), M.trace()` → transposée, trace

↳ Matrices carrées uniquement (algèbre linéaire) :

`import numpy.linalg as la` ("Linear algebra")

`la.det(M), la.inv(M)` → déterminant, inverse

`vp = la.eigvals(M)` → `vp` vecteur des valeurs propres

`vp, P = la.eig(M)` → `P` matrice de passage

`la.matrix_rank(M), la.matrix_power(M, p)`

`X = la.solve(M, V)` → Vecteur solution de `M X = V`

`B = (T==1.0)`

Tableaux booléens, comparaison, tri

`B = (abs(T)<=1.0)` → `B` est un tableau de booléens, de même forme que `T`

`B = (T>0) * (T<1)` Par exemple `B*np.sin(np.pi*T)` renverra un tableau de $\sin(\pi x)$ pour tous les coefficients `x` dans $[0,1]$ et de 0 pour les autres

`B.any(), B.all()` → booléen « Au moins un True », « Que des True »

`indices = np.where(B)` → tuple de vecteurs d'indices donnant les positions des True

`T[indices]` → extraction séquentielle des valeurs

`T.clip(v_min, v_max)` → tableau dans lequel les valeurs ont été ramenées entre `v_min` et `v_max`

`np.allclose(T1, T2)` → booléen indiquant si les tableaux sont numériquement égaux

Intégration numérique

`import scipy.integrate as spi`

`spi.odeint(F, Y0, vt)` → renvoie une solution numérique du problème de Cauchy $Y'(t) = F(Y(t), t)$, où $Y(t)$ est un vecteur d'ordre n , avec la condition initiale $Y(t_0) = Y0$, pour les valeurs de t dans le vecteur `vt` commençant par t_0 , sous forme d'une matrice $n \times k$

`spi.quad(f, a, b) [0]` → renvoie une évaluation numérique de l'intégrale : $\int_a^b f(t) dt$

Mémento Python 3 pour le calcul scientifique

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

`plt.figure(mon_titre, figsize=(W,H))` crée ou sélectionne une figure dont la barre de titre contient `mon_titre` et dont la taille est $W \times H$ (en inches, uniquement lors de la création de la figure)

`plt.plot(X,Y,dir_abrg)` trace le nuage de points d'abscisses dans `X` et d'ordonnées dans `Y` ; `dir_abrg` est une chaîne de caractères qui contient une couleur ("r"-ed, "g"-reen, "b"-lue, "c"-yan, "y"-ellow, "m"-agenta, "k" black), une marque (voir ci-dessous) et un type de ligne (" " pas de ligne, "-- plain, "- - dashed, ":" dotted, ...)

- Options courantes :

<code>label=...</code>	étiquette pour la légende
<code>linewidth=...</code>	épaisseur du trait (0 pour aucun trait)
<code>color=...</code>	couleur du trait : (<code>r</code> , <code>g</code> , <code>b</code>) ou (<code>m</code> , <code>c</code> , ...)
<code>dashes=...</code>	style de pointillé (liste de longueurs)
<code>marker=...</code>	forme de la marque :
<code>"o" ". "*" ">" "<" "^" "v" "s" "d" "D" "p" "h" "H" "8" "1" "2" "3" "4" "+" "x"</code>	
<code>markersize=...</code>	taille de la marque
<code>markeredgecolor=...</code>	couleur du contour
<code>markerfacecolor=...</code>	couleur de l'intérieur

`plt.axis("equal"), plt.grid()` repère orthonormé, quadrillage

`plt.xlim(a,b), plt.ylim(a,b)` plages d'affichage ; si $a > b$, inversion de l'axe

`plt.xlabel(axe_x, size=s, color=(r,g,b)), plt.ylabel(axe_y, ...)` étiquettes sur les axes, en réglant la taille `s` et la couleur de la police de caractères (`r`, `g` et `b` dans [0,1])

`plt.legend(loc="best", fontsize=s)` affichage des labels des "plot" en légende

`plt.show()` affichage des différentes figures et remise à zéro

`plt.twinx()` bascule sur une deuxième échelle des ordonnées apparaissant à droite du graphique

`plt.xticks(Xt), plt.yticks(Yt)` réglage des graduations des axes

`plt.subplot(nbL, nbC, numero)` début de tracé dans un graphique situé dans un *tableau de graphiques* à `nbL` lignes, `nbC` colonnes ; `numero` est le numéro séquentiel du graphique dans le tableau (voir ci-contre).

`plt.subplots_adjust(left=L, right=R, bottom=B, top=T, wspace=W, hspace=H)` ajustement des marges (voir ci-contre)

`plt.title(Titre_du_graphique)` rajout d'un titre au graphique en cours de tracé

`plt.suptitle(Titre_général)` rajout d'un titre à la fenêtre de graphiques

`plt.text(x,y, texte, fontdict=dico, horizontalalignment=HA, verticalalignment=HV)` tracé du texte `texte` à la position `(x,y)`, avec réglage des alignements (`HA` ∈ {"center", "left", "right"}, `HV` ∈ {"center", "top", "bottom"}) ; `dico` est un dictionnaire (voir ci-contre)

`plt.axis("off")` suppression des axes et du cadre

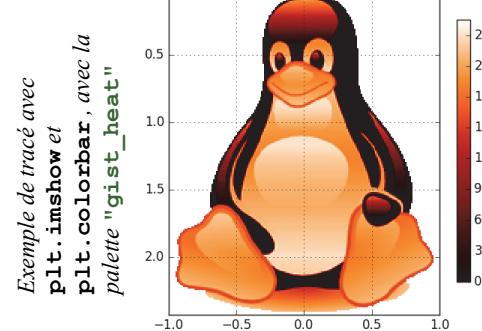
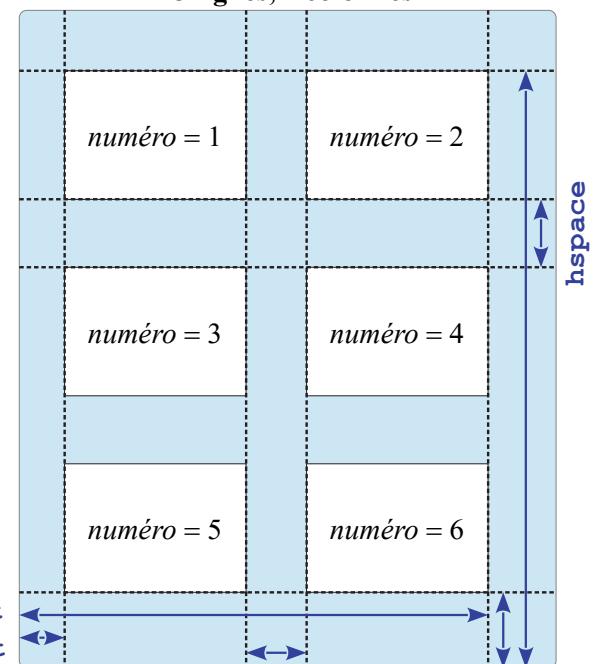
`plt.imshow(T, interpolation="none", extent=(gauche, droite, bas, haut))` tracé d'une image pixélisée non lissée à partir d'un tableau `T` ($nl \times nc \times 4$ format *RGBA*, $nl \times nc \times 3$ format *RGB*) ; l'option `extent` permet de régler la plage correspondant à l'image ($-0.5, nc-0.5, nl-0.5, -0.5$) par défaut

`plt.imshow(T, interpolation="none", vmin=vmin, vmax=vmax, cmap=palette, extent=(gauche, droite, bas, haut))` tracé d'une image pixélisée non lissée à partir d'un tableau `T` rectangulaire $nl \times nc$ correspondant à des niveaux de gris sur la plage $[v_{\min}, v_{\max}]$, avec la palette de couleurs `palette` : voir https://matplotlib.org/examples/color/colormaps_reference.html

`plt.colorbar(schrink=c)` Affichage de l'échelle des couleurs du tracé précédent sur la plage $[v_{\min}, v_{\max}]$

Graphiques Matplotlib

3 lignes, 2 colonnes



Dictionnaires
Un dictionnaire `D`, de type `dict` (type itérable), se présente sous la forme :
`{clef_0:valeur_0, clef_1:valeur_1, ...}`
On peut accéder aux clefs par `D.keys()`, aux valeurs par `D.values()`, et obtenir un itérateur sur les couples par `D.items()` :
Parcours des clefs et valeurs du dictionnaire :
`for key, val in D.items():`
→ bloc d'instructions
Extraire une valeur par sa clef : `D[key]`
Compléter le dictionnaire :
`D[new_key] = new_val`
`dict1.update(dict2)`
Supprimer un entrée : `del D[key]`
Dictionnaire vide : `D = dict()` ou `D = {}`
Exemple pour la fonction `plt.text` :
`{ "family" : "Courier New", "weight" : "bold", "style" : "normal", "size" : 18, "color" : (0.0, 0.5, 0.8) }`

Calcul formel avec sympy

Nombres exacts

Rationnels : `sb.Rational(2,7)`
ou `sb.S(2)/7`

Irrationnels :
`sb.sqrt(2)` $\sqrt{2}$
`sb.pi, sb.E` π et e
`sb.I` i tel que $i^2=-1$

Il est conseillé d'utiliser un *notebook jupyter* (voir <https://jupyter.readthedocs.io/en/latest/>) avec en en-tête pour avoir de belles formules mathématiques à l'écran les instructions ci-contre :

Fonction indéfinie

`f = sb.Function("f")`

`sb.oo` → ∞ L'infini

Fonctions mathématiques

<code>sb.re, sb.im</code>	→ Parties réelle et imaginaire
<code>sb.sqrt, sb.exp</code>	→ Racine carrée, exponentielle
<code>sb.log, sb.factorial</code>	→ Logarithme népérien, factorielle
<code>sb.cos, sb.sin, sb.tan</code>	→ Fonctions trigonométriques
<code>sb.acos, sb.asin, sb.atan</code>	→ Fonctions trig. r.éiproques
<code>sb.atan2(y,x)</code>	→ Angle dans $]-\pi, \pi]$
<code>sb.cosh, sb.sinh, sb.tanh</code>	(trigonométrie hyperbolique)
<code>sb.acosh, sb.asinh, np.atanh</code>	

Expressions symboliques A et B

Égalités et équations

<code>A == B</code>	→ Booléen : identité parfaite des expressions
<code>A.equals(B)</code>	→ Booléen : égalité après simplifications
<code>sb.Eq(A,B)</code>	→ Équation : booléen seulement si l'équation peut être identifiée comme vraie ou fausse
<code>sb.Ne(A,B) ou Lt, Le, Gt, Ge</code>	→ Inéquations $\neq, <, \leq, >, \geq$

```
import sympy as sb
sb.init_printing()
```

Symboles

<code>sb.symbols("a,a_0,a^*)</code>	→ (a, a_0, a^*) (tuple)
<code>sb.symbols("x", real=True)</code>	→ réel x
<code>sb.symbols("y", nonzero=True)</code>	→ réel y non nul
<code>sb.symbols("j,k", integer=True)</code>	→ entiers relatifs j et k
<code>sb.symbols("m", integer=True, positive=True)</code>	→ $m \in \mathbb{N}^*$
<code>sb.symbols("n", integer=True, nonzero=True)</code>	→ $m \in \mathbb{Z}^*$
<code>sb.symbols("s", zero=False)</code>	→ symbole s non nul
<code>sb.symbols(..., nonnegative=True)</code>	→ positif ou nul
<code>sb.symbols(..., negative=True)</code>	→ strictement négatif
<code>sb.symbols(..., nonpositive=True)</code>	→ négatif ou nul
<code>sb.symbols(..., complex=True)</code>	→ complexe
<code>sb.symbols(..., imaginary=True)</code>	→ imaginaire pur

Expressions symboliques A et B

Manipulation d'expressions

<code>A+B, A-B, A*B, A/B, etc.</code>	→ Opérations mathématiques
<code>A.diff(x), A.diff(x,n), A.diff(x,3,y,2,z)</code>	→ $\frac{\partial A}{\partial x}, \frac{\partial^n A}{\partial x^n}, \frac{\partial^6 A}{\partial x^3 \partial y^2 \partial z}$
<code>A.expand(), A.simplify()</code>	→ Développer, simplifier
<code>A.factor(), A.together()</code>	→ Factoriser, réduire une fraction
<code>A.collect(x)</code>	→ Regrouper les termes par rapport à x
<code>A.apart(x)</code>	→ Décomposition en élément simples par rapport à x

Mémento Python 3 pour le calcul scientifique

Calcul formel avec sympy (suite)

Il est conseillé d'utiliser un **notebook jupyter** (voir <https://jupyter.readthedocs.io/en/latest/>) avec en en-tête pour avoir de belles formules mathématiques à l'écran les instructions ci-dessous :

```
import sympy as sb
sb.init_printing()
```

Expr. symboliques **A** (de **t**) et **B** (de **x** et **y**)

```
x,y,t = sb.symbols("x,y,t", real=True)
```

A.integrate(t) ou **sb.integrate(A,t)**
→ primitive de **A** par rapport à **t**

B.integrate(x,y) ou **sb.integrate(B,x,y)**
→ primitive de **B** par rapport à **x** et à **y**

A.integrate((t,tinf,tsup)) ou **sb.integrate(A,(t,tinf,tsup))**
→ intégrale de **tinf** à **tsup** de **A**

sb.integrate(B,(x,a,b),(y,c,d))
→ intégrale double de **B** sur $[a,b] \times [c,d]$

sb.integrate(tx,(t,1,sb.oo))** → $\begin{cases} -\frac{1}{x+1} & \text{for } x < -1 \\ \infty \int t^x dt & \text{otherwise} \end{cases}$

sb.integrate(tx,(t,1,sb.oo),conds="none")** → $\frac{-1}{x+1}$
(on se place dans le cas où l'intégrale est définie)

Intégrales et primitives

Si **L** est une séquence d'expressions symboliques, **sum(L)** renvoie leur somme

Ak est une expression symbolique de **k**

```
k,n = sb.symbols("k,n", integer=True)
```

sb.summation(Ak,(k,kmin,kmax)) → $\sum_{k=k_{\min}}^{k_{\max}} A_k$
Exemple : **sb.summation(k**2,(k,0,n)).factor()** → $\frac{n(n+1)(2n+1)}{6}$

Sommes, finies ou infinies

Si **L** est une séquence d'expressions symboliques, **sum(L)** renvoie leur somme

Ak est une expression symbolique de **k**

```
k,n = sb.symbols("k,n", integer=True)
```

sb.summation(Ak,(k,kmin,kmax)) → $\sum_{k=k_{\min}}^{k_{\max}} A_k$
Exemple : **sb.summation(k**2,(k,0,n)).factor()** → $\frac{n(n+1)(2n+1)}{6}$

Résolution algébrique d'équations

sb.solve(équations,inconnues) où **équations** est une séquence d'équations, ou d'expressions qui doivent s'annuler, et **inconnues** l'inconnue ou la liste des inconnues. Renvoie la liste des solutions, si elles sont calculables par **sympy**, chaque solution étant soit une expression, soit un tuple d'expressions, soit un dictionnaire (option « **dict=True** »).

Exemples : **sb.solve(sb.Eq(x**4,1),x)** → [-1,1,-i,i]
sb.solve(x2-3,x,dict=True)** → [(x:- $\sqrt{3}$],[x: $\sqrt{3}$]]
sb.solve([x2+y**2-5,x-y-1],[x,y])** → [(-1,-2),(2,1)]

• Calcul de constantes en fonction des conditions initiales sur une expression :

```
a,b,x,u0,v0 = sb.symbols("a,b,x,u_0,v_0")
U = a*sb.exp(x) + b*sb.exp(-2*x)
CI = [ sb.Eq(U.replace(x,0), u0), \
        sb.Eq(U.diff(x).replace(x,0),v0) ]
sb.solve(CI,[x,y],dict=True) → [a:  $\frac{2u_0}{3} + \frac{v_0}{3}$ , b:  $\frac{u_0}{3} - \frac{v_0}{3}]$ 
```

☞ L'ensemble des équations différentielles

Équations différentielles

que **sympy** sait résoudre est pour l'instant assez limité.

Il faut procéder en 2 temps : 1/ Résolution des équations différentielles ;
2/ Détermination des constantes en fonction des conditions initiales et/ou aux bords.

sb.dsolve(équations,inconnues) renvoie une équation ou une liste d'équations. De chaque équation **eq**, de la forme **sb.Eq(f(x),solu)**, on peut extraire la solution **solu** par **eq.rhs** (right-hand side).

Exemple d'équation différentielle :

```
r = sb.symbols("r"); f = sb.Function("f")
EDO = sb.Eq(f(r).diff(r,2)+f(r).diff(r)/r+f(r)/r**2,0)
solu = sb.dsolve(EDO,f(r)).rhs → C1 sin(log(r))+C2 cos(log(r))
```

Exemple de système différentiel (linéaire à coefficients constants) :

```
x,y,z = [sb.Function(c) for c in "xyz"]
t = sb.symbols("t")
SDO = [sb.Eq(x(t).diff(t),y(t)-z(t)), \
        sb.Eq(y(t).diff(t),x(t)+z(t)), \
        sb.Eq(z(t).diff(t),x(t)+y(t)+z(t))]
Leq = sb.dsolve(SDO,[x(t),y(t),z(t)])
[e.rhs for e in Leq] → [-C1 e^{-t} - C2 e^t - C3(t-1) e^t,
                           C1 e^{-t} + C2 e^t + C3(t+1) e^t,
                           2 C2 e^t + C3(2t+1) e^t]
```

Les constantes à trouver ensuite sont définies par :

```
sb.symbols("C1,C2,C3[etc]")
```

Exemple : **sb.solve([solu.replace(r,1)-a, \ solu.diff(r).replace(r,1)-b], \ sb.symbols("C1,C2"))** → {C1:b, C2:a}

Limites et développements limités

sb.limit(A,x,x0,dir="+")

Limite, développement limité

→ Limite de l'expression **A** lorsque **x** tend vers **x0** dans la direction **dir** ("+-", "+" ou "-")

A.series(x,x0,n)

→ Développement limité de **A** en **x0** à l'ordre **n**

Exemple : **sb.cos(2*x).series(x,0,6)** → $1 - 2x^2 + 2x^4/3 + O(x^6)$

A.series(x,x0,n).replace(sb.O,lambda *args : 0)

→ Développement limité sans le **O** ($(x-x0)^n$)

Approximation par différences finies

On cherche à approcher la dérivée **d**-ième d'une fonction indéfinie **f** au point **x** à l'ordre **n**. Le nombre de points discrétils à considérer est **d+n**.

Ces points sont donnés dans une séquence **S**, par exemple $(x-h, x, x+h, x+2h)$

```
f = sb.Function("f")
```

```
x,h = sb.symbols("x,h", real=True)
```

f(x).diff(x,d).as_finite_difference(S) → approx. de $f^{(d)}(x)$

Exemple : **S = [x,x+h,x+2h]**

f(x).diff(x).as_finite_difference(S).together()
→ $\frac{-3f(x) + 4f(h+x) - f(2h+x)}{2h}$

Expression symbolique

A

B = A.replace(x,y) → **B** s'obtient en remplaçant **x** par **y** dans **A**

x peut être un symbole, une fonction, ou autre chose*

Exemples : **f(x).replace(x,y)** → **f(y)**

(x2).replace(x,x+y)** → $(x+y)^{**2}$

(x2).replace(2,y+1)** → $x^{**}(y+1)$

f(x).replace(f,g) → **g(x)**

sb.cos(x).replace(sb.cos,lambda t : t2)** → x^{**2}

dico est un dictionnaire

B = A.xreplace(dico) → **B** s'obtient en remplaçant simultanément dans **A** toutes les clefs de **dico** par les expressions correspondantes ; ces clefs sont des symboles, ou des « sous-expressions complètes »*

Exemples : **(x+2*y).xreplace({x:y,y:x})** → $2x+y$

(x+2*y).xreplace({x:y,y:y+1}) → $3y+2$

(x+x2).xreplace({x**2:y})** → $x+y$

Calcul littéral **V = sb.pi*r**2*h** suivi d'une application numérique : **float(V.xreplace({r:0.1,h:0.2}))** → $\approx 6.283e-03$

(* Voir ci-dessous : « Manipulation avancée d'expressions »).

Résultat symbolique → Fonction numérique

A est une expression symbolique contenant les symboles **x,y,z**

Fnum = sb.lambdify((x,y,z), A, "numpy") définit une fonction numérique des variables **x,y** et **z**

Fnum = sb.lambdify((x,y,z), A, (dico,"numpy")) indique, à l'aide du dictionnaire **dico**, la correspondance entre les fonctions de **sympy** (en chaîne de caractères) et les fonctions numériques à utiliser. Des exemples sont donnés dans le tableau ci-dessous.

Tableau de correspondance de quelques fonctions

import scipy.special as sf	(certaines sont automatiques avec numpy)
"factorial" : sf.factorial	"atan2" : np.arctan2
"binomial" : sf.binom	"besselj" : sf.jn
"erf" : np.erf ou sf.erf	"bessely" : sf.yN
"erfinv" : sf.erfinv	"zeta" : sf.zeta
"sinc" : lambda x : np.sinc(x/np.pi)	
"lowergamma" : lambda s,x : sf.gamma(s)*sf.gammainc(s,x)	

M = sb.Matrix(liste de listes)

M.det(), **M.trace()**, **M.inv()** → déterminant, trace, inverse

M.eigenvals() → valeurs propres, avec ordres de multiplicité

sb.diag(a1,...,an) → matrice diagonale de coef. diagonaux **a1, ..., an**

M+N, **a*M**, **M@N** → somme, produit par un scalaire, produit matriciel

☞ La notion de vecteur n'existe pas en **sympy**; il est assimilé indûment et confusément à une matrice-colonne et/ou à une matrice-ligne.

A = 2*a*x+y3**

Manipulation avancée d'expressions

sb.srepr(A) → $\text{Add}(\text{Mul}(\text{Integer}(2), \text{Symbol('a')}, \text{Symbol('x'))}, \text{Pow}(\text{Symbol('y')), Integer(3)))$
sous-expression complète sous-expression complète

A.func, A.args → **sympy.core.add.Add, (2*a*x, y**3)**

X,Y = sb.Wild("X"), sb.Wild("Y") symboles indéfinis

A.match(a*X+Y) → **{X:2*x, Y:y**3}** (dictionnaire)

A.replace(y3,4*sb.sin(y))** → $2ax + 4\sin(y)$