

Séance 10

Concepts avancés et calcul numérique



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

Objectifs

- Constructions avancées en Python
 - Gestionnaire de contexte
 - Coroutine
- Interaction avec des librairies C
 - Utilisation de l'outil Cython
- Calcul numérique et symbolique avec la suite SciPy
 - Librairies numpy, matplotlib, scipy et sympy

Constructions avancées



Bloc else

- Un **bloc else** peut être attaché à plusieurs instructions

Associable à if, while, for et try

- **Comportements** différents du bloc else
 - Une fois que la boucle for s'est terminée (pas via break)
 - Lorsque la condition du while devient False (pas via break)
 - Lorsqu'aucune exception n'est apparue dans un try

Exemple : Trouver un élément (1)

- Le programme ne peut continuer que si un élément est trouvé

Utilisation d'une variable pour mémoriser résultat de recherche

```
1 fruits = ['apple', 'banana', 'kiwi']
2
3 found = False
4 for fruit in fruits:
5     if fruit == 'watermelon':
6         found = True
7
8 if not found:
9     raise ValueError('Watermelon no found')
10
11 print('Manage the watermelon')
12 # ... suite du programme
```

Exemple : Trouver un élément (2)

■ Simplification du code à l'aide d'un bloc else

Si la boucle arrive jusqu'au bout, c'est qu'on n'a pas trouvé

```
1  fruits = ['apple', 'banana', 'kiwi']
2
3  for fruit in fruits:
4      if fruit == 'watermelon':
5          break
6  else:
7      raise ValueError('Watermelon no found')
8
9  print('Manage the watermelon')
10 # ... suite du programme ...
```

Exemple : Traiter un fichier (1)

■ Ouverture d'un fichier et traitement de son contenu

Le fichier peut ne pas exister, gestion avec un try/except

```
1 try:
2     with open('data.txt') as file:
3         content = file.read()
4         print(content)
5         # ... suite du programme ...
6 except IOError:
7     print('Erreur de lecture')
```


Exemple : Traiter un fichier (2)

- Code à exécuter avec le contenu du **fichier une fois lu**

N'est pas susceptible de générer des `IOError`

```
1 try:
2     with open('data.txt') as file:
3         content = file.read()
4 except IOError:
5     print('Erreur de lecture')
6 else:
7     print(content)
8     # ... suite du programme ...
```

EAFP versus LBYL

- **Easier to ask for forgiveness than permission (EAFP)**
 - Supposition que les clés ou attributs corrects, sinon exception
 - Beaucoup de try/except
- **Look before you leap (LBYL)**
 - Tests explicites des préconditions avant tout appel
 - Beaucoup de if

Gestionnaire de contexte

- Contrôle d'un **bloc with** avec un gestionnaire de contexte

Tout comme l'itérateur contrôle la boucle for

- Introduit pour **simplifier les try/finally**

Libération de ressources et/ou restauration d'état avec finally

- Définition des méthodes **__enter__** et **__exit__**

Appelées à l'entrée du with et à la fin du bloc with

Exemple : Modification du print (1)

- Définition d'une classe représentant le **gestionnaire de contexte**

Remplacement de `sys.stdout.write` par une nouvelle fonction

```
1 class Mirror:
2     def __enter__(self):
3         import sys
4         self.__oldwrite = sys.stdout.write
5         sys.stdout.write = self._revwrite
6         return 'MIROIR'
7
8     def _revwrite(self, text):
9         self.__oldwrite(text[::-1])
10
11    def __exit__(self, type, value, traceback):
12        import sys
13        sys.stdout.write = self.__oldwrite
```

Exemple : Modification du print (2)

- Création d'un nouveau contexte avec l'instruction `with`
Comportement de `print` modifié à l'intérieur du contexte
- Contexte remis dans son état de départ à la fin du bloc `with`

```
1 with Mirror() as m:  
2     print(m)  
3     print('Hello World!')  
4 print('Normal')
```

```
RIORIM  
!dlroW olleH  
Normal
```

yield (1)

- Deux significations au verbe anglais “to yield”
 - **to produce** : produit un élément qui sera reçu par next
 - **to give way** : suspend l'exécution jusqu'au prochain next

```
1 def natural():
2     n = 1
3     while True:
4         yield n
5         n += 1
6
7 gen = natural()
8 print(next(gen))
9 print(next(gen))
```

```
1
2
```

yield (2)

- Arrêt de l'exécution de la fonction

Et la suite du code peut continuer à s'exécuter

- Spécification de la valeur renvoyée

Lorsque la fonction sera débloquée par l'appel next

- Deux constructions avec l'instruction yield

- Générateur
- Coroutine

Coroutine (1)

- L'exécution d'une **coroutine** peut être **interrompue et reprise**

Fonction qui contient au moins une instruction `yield`

- La coroutine peut recevoir des données grâce à **send**

L'instruction `yield` est utilisée à droite de l'affectation

- **Deux interactions** possibles avec une coroutine

- Fonction `next` avance jusqu'au prochain `yield`
- Méthode `send` débloque un `yield` en envoyant une valeur

Coroutine (2)

- next et send avancent jusqu'au **yield suivant**

Et renvoient l'éventuelle valeur associée

```
1 def compute():
2     a = yield -1
3     print('a =', a)
4     b = yield 99
5     print('b =', b)
6     yield a + b
7
8 gen = compute()
9 print(next(gen))
10
11 print(gen.send(12))
12 print(gen.send(8))
```

```
-1
a = 12
99
b = 8
20
```

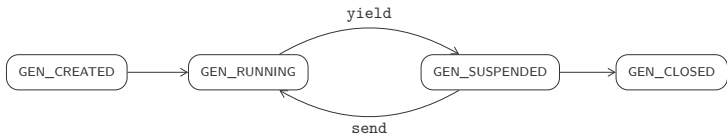
États d'une coroutine

- **Quatre états** différents pour une coroutine

Créé, en exécution, suspendue, terminée

- Méthode `send` ne fonctionne que dans l'état « **suspendue** »

Nécessité d'activer d'abord la coroutine avec `next`



Exemple : Additionneur

- Fonction `summer` calcule une **somme incrémentalement**

send renvoie la somme actuelle

```
1 def summer():
2     total = 0
3     while True:
4         value = yield total
5         total += value
6
7 s = summer()
8 next(s)
9
10 print(s.send(12))
11 print(s.send(3))
12 print(s.send(5))
```

```
12
15
20
```

Activation automatique

■ Décoration d'activation automatique d'une coroutine

La décoration appelle `next` sur la coroutine

```
1 def coroutine(f):
2     def primer(*args, **kwargs):
3         g = f(*args, **kwargs)
4         next(g)
5         return g
6     return primer
7
8 @coroutine
9 def summer():
10     total = 0
11     while True:
12         value = yield total
13         total += value
```

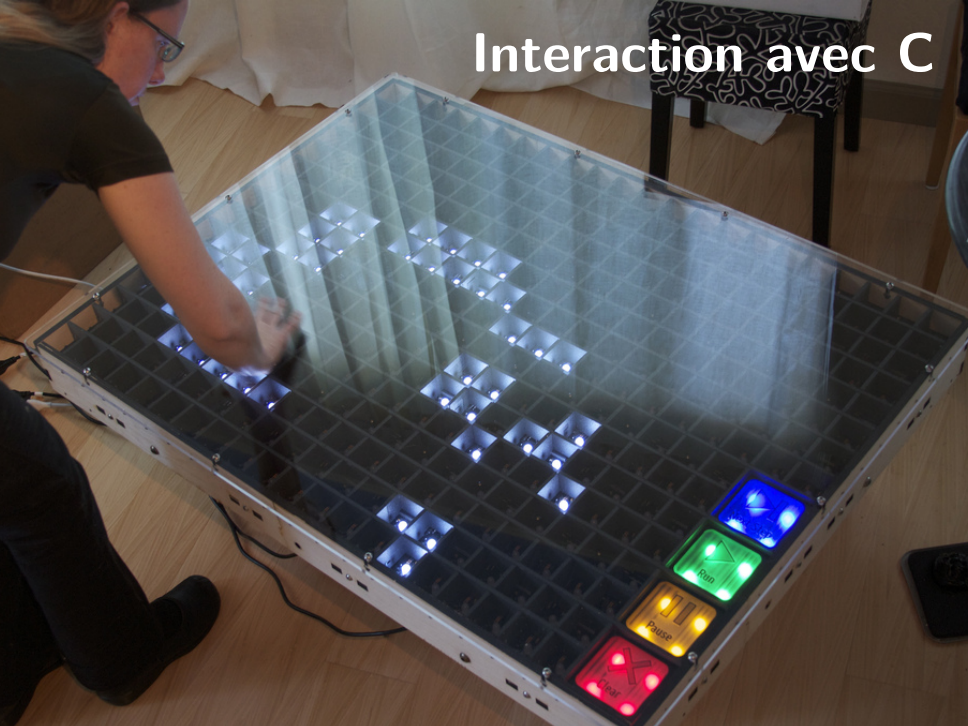
Valeur de retour

- Valeur de retour d'une coroutine obtenue avec une exception

Attribut value de l'exception de type StopIteration

```
1  @coroutine
2  def compute():
3      a = yield
4      b = yield
5      return a + b
6
7  try:
8      gen = compute()
9      gen.send(12)
10     gen.send(8)
11 except StopIteration as e:
12     print(e.value)
```

Interaction avec C



Optimisation de l'exécution

- Plusieurs techniques d'optimisation de la vitesse d'exécution
 - Utilisation de bibliothèques optimisées
 - Appel d'une bibliothèque externe en C
 - Compilation du code Python en natif
- Utilisation de l'outil Cython

Facilite l'écriture d'extensions en C pour Python

Optimisation de son Python

- Utiliser les **fonctions prédéfinies** comme `map`, `zip`...

- Par compréhension (9 secondes pour $2 \cdot 10^7$)

```
1 def sqrt(data):  
2     return [math.sqrt(i) for i in data]
```

- Utilisation de `map` (3 secondes pour $2 \cdot 10^7$)

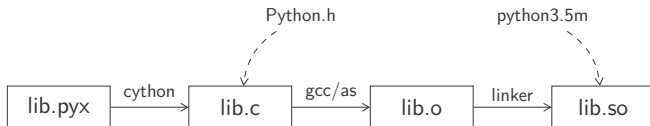
```
1 def sqrt(data):  
2     return list(map(math.sqrt, data))
```

- Faire appel depuis Python à du **code écrit en C**

Par exemple, librairie `ujson` pour traiter les fichiers JSON

Outil Cython

- **Plusieurs étapes** à suivre pour appeler du C depuis Python
 - Écrire sa librairie en C
 - Écrire un fichier pyx à compiler avec Cython
 - Compiler la librairie native
 - Import de la librairie depuis Python



Librairie C (1)

- Fichier `utils.c` qui contient les **fonctions de la librairie**
 - Rien de particulier à faire, juste son code en C

```
1  int cfact (int n)
2  {
3      int result = 1;
4      int i;
5      for (i = 1; i <= n; i++)
6      {
7          result *= i;
8      }
9      return result;
10 }
11
12 int csum (int a, int b)
13 {
14     return a + b;
15 }
```

Librairie C (2)

- Fichier `utils.h` qui contient les **entêtes des fonctions**

Reprendre les prototypes des fonctions de la librairie à exporter

```
1  #ifndef __SUM_H__
2  #define __SUM_H__
3
4  extern int cfact (int);
5  extern int csum (int, int);
6
7  #endif // __SUM_H__
```

Fichier pyx

- Création de **fonctions Python** qui font appel au code C

Typiquement juste faire l'appel à la fonction C

- Mot réservé `cdef` pour définir une **fonction externe**

Lien vers le fichier d'entête avec `extern from`

```
1 cdef extern from "utils.h":  
2     cdef int cfact (int)  
3     cdef int csum (int, int)  
4  
5 def fact(n):  
6     return cfact(n)  
7  
8 def sum(a, b):  
9     return csum(a, b)
```

Compilation

- Compilation pour obtenir un **fichier binaire** de la librairie

Génération du fichier .so à importer ensuite dans Python

- Variable \$PYTHONPATH indiquant le chemin d'installation

Pour les fichiers Python.h et python3.5m

```
1 CC=cc
2 CFLAGS=-g -O2 -fpic -Wall -I$(PYTHONPATH)/Headers
3 LDFLAGS=-shared -L$(PYTHONPATH)/lib
4
5 all: utils.c lib.pyx
6     cython lib.pyx
7     $(CC) -c utils.c -o utils.o $(CFLAGS)
8     $(CC) -c lib.c -o lib.o $(CFLAGS) 'python3-config --cflags'
9     $(CC) -shared -o lib.so utils.o lib.o $(LDFLAGS) 'python3-
    config --libs'
```

Utiliser une librairie

- Placer le **fichier .so** à un endroit trouvable par Python

Le plus facile étant dans le même dossier que le programme

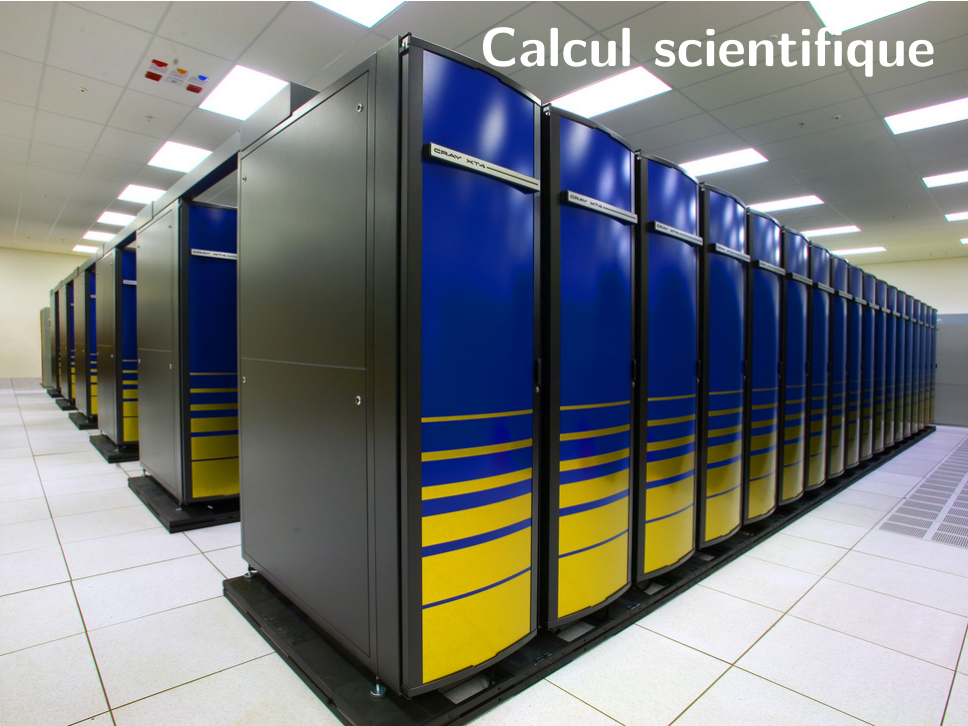
- **Importation** de la librairie et de ses fonctions avec `import`

`import xxx` OU `from xxx import yyy`

```
1 import lib
2
3 print(lib.fact(3))
4 print(lib.sum(12, 9))
```

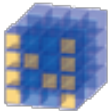
```
6
21
```

Calcul scientifique



- **Écosystème de logiciels** pour les maths, sciences et ingénierie

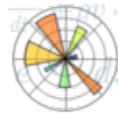
Collection de plusieurs bibliothèques coordonnées



NumPy



SciPy



Matplotlib

IP[y]:
IPython

IPython



SymPy



pandas

Librairie numpy

- Package de base pour le **calcul scientifique**

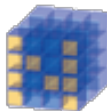
Intégration facilitée de code C/C++ et Fortran

- De nombreuses **fonctionnalités**

- Représentation de tableaux à N dimensions
- Fonctions sophistiquées
- Algèbre linéaire, transformée de Fourier
- ...

- Disponible en **open-source** sur Sourceforge

<https://sourceforge.net/projects/numpy>



Types de données

- Déclaration générale ou précision du **nombre de bits**

Booléens et nombres entiers, flottants et complexes

```
1 import numpy as np
2
3 v = np.bool_(True)
4 w = np.int_(99)
5 x = np.int32(123)
6 y = np.float64(12.99)
7 z = np.complex_(3j+2)
8
9 print(x, y, z)
```

```
True 99 123 12.99 (2+3j)
```

Tableau

- Plusieurs manières de créer des **tableaux** de valeurs

Depuis une structure Python, ou par des fonctions de `numpy`

```
1 print(np.array([1, 2, 3]))
2 print(np.array([[1, 2], [3, 4]]))
3 print(np.zeros(5))
4 print(np.ones((2, 3), dtype='int32'))
5 print(np.linspace(0, 2, 5))
```

```
[1 2 3]
[[1 2]
 [3 4]]
[ 0.  0.  0.  0.  0.]
[[1 1 1]
 [1 1 1]]
[ 0.   0.5  1.   1.5  2. ]
```

Opérations

- **Plusieurs attributs** disponibles sur les tableaux

Généralisation de l'opérateur d'indexation et de slicing

```
1 x = np.array([[1, 2], [3, 4], [5, 6]])
2
3 print(x.ndim)      # Dimension
4 print(x.shape)     # Forme
5 print(x.size)      # Nombre total d'éléments
6 print(x.dtype)     # Type de données stockées
7 print(x[0,1])
8 print(x[:,1])
```

```
2
(3, 2)
6
int64
2
[2 4 6]
```

Méthodes (1)

- **Plusieurs méthodes** disponibles sur les tableaux

Essentiellement opérations matricielles et d'accès aux éléments

```
1 x = np.array([[1, 2], [3, 4], [5, 6]])
2
3 print(x.ndim)      # Dimension
4 print(x.shape)     # Forme
5 print(x.size)      # Nombre total d'éléments
6 print(x.dtype)     # Type de données stockées
7 print(x[0,1])
8 print(x[:,1])
```

```
2
(3, 2)
6
int64
2
[2 4 6]
```

Méthodes (2)

```
1 x = np.array([[1, 2], [3, 4], [5, 6]])
2
3 print(x.reshape((2, 3)))
4 print(x.transpose())
5 print(x.flatten())
6
7 y = x[[0,2],:]
8
9 print(y)
10 print(y.diagonal())
11 print(y.trace())
12 print(y.sum(axis=1))
13 print(2 * y)
```

```
[[1 2 3]
 [4 5 6]]
[[1 3 5]
 [2 4 6]]
[1 2 3 4 5 6]
[[1 2]
 [5 6]]
[1 6]
7
[ 3 11]
[[ 2  4]
 [10 12]]
```

Librairie matplotlib

- Dessin de **graphes 2D** interactifs ou exportables

Génération de figures de qualité prêtes pour publication

- De nombreuses **fonctionnalités**

- Dessin de courbes
- Histogrammes
- Dessin 3D également possible
- ...

- Disponible en **open-source** sur GitHub

<https://github.com/matplotlib/matplotlib>



Dessin d'une fonction (1)

- Utilisation de la fonction `plot` pour dessiner des points

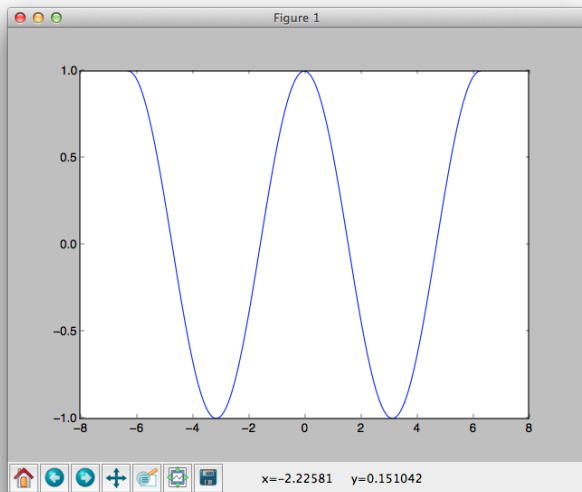
Création des vecteurs de données avec `numpy`

- Ouverture de la fenêtre de dessin avec `show`

Possibilité d'avoir plusieurs dessins dans la même fenêtre

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(-2*np.pi, 2*np.pi, 256, endpoint=True)
5 y = np.cos(x)
6
7 plt.plot(x, y)
8 plt.show()
```


Dessin d'une fonction (2)



Paramétrage des dessins (1)

■ Configuration des courbes

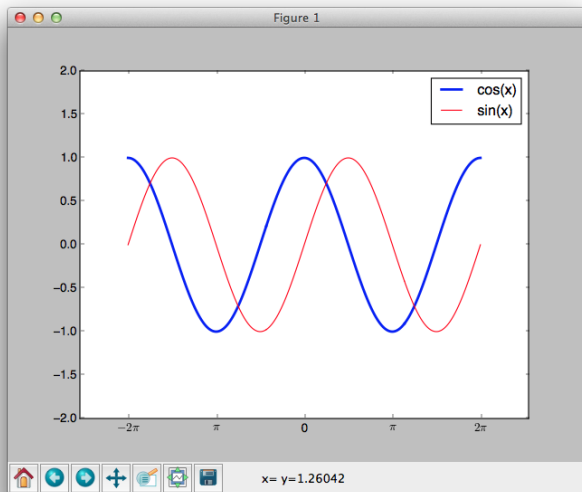
Épaisseur et couleur du trait, texte légende

■ Ajustement de la zone de dessin et autres éléments

Limites des axes, ticks et texte (support \LaTeX), légende

```
1 plt.plot(x, c, linewidth=2.5, label='cos(x)')
2 plt.plot(x, s, color='red', label='sin(x)')
3
4 plt.ylim(-2, 2)
5 plt.xticks(
6     [-2*np.pi, -np.pi, 0, np.pi, 2*np.pi],
7     [r'$-2\pi$', r'$-\pi$', '0', r'$\pi$', r'$2\pi$'],
8 )
9 plt.legend(loc='upper right')
```

Paramétrage des dessins (2)



Librairie scipy

- Package de base de **la stack scipy**

Algorithmes et fonctions utilitaires construits sur numpy

- De nombreuses **fonctionnalités**

- Intégration numérique
- Optimisation
- Distributions statistiques
- ...

- Disponible en **open-source** sur GitHub

<https://github.com/scipy/scipy>



Polynôme

- Polynôme représenté à l'aide de la fonction `poly1d`

Opérations et méthodes de manipulation des polynômes

```
1 from numpy import poly1d
2
3 p = poly1d([1, 2, -1])
4 print(p)
5 print(2 * p)
6 print(p ** 2)
7 print(p.deriv())
```

```
      2
1 x + 2 x - 1
      2
2 x + 4 x - 2
      4      3      2
1 x + 4 x + 2 x - 4 x + 1

2 x + 2
```

Vectorisation de fonction

- Transformer une fonction scalaire en **fonction vectorielle**

Transformation complètement transparente

```
1 import numpy as np
2
3 def add(a, b):
4     return a + b
5
6 vec_add = np.vectorize(add)
7
8 x = [1, 2, 3]
9 y = [7, 8, 9]
10 print(vec_add(x, y))
```

```
[ 8 10 12]
```

Intégration numérique

- **Intégration numérique** avec `scipy.integrate`

Spécification de la fonction à intégrer et des bornes

- Plusieurs **méthodes d'intégration** disponibles

- `quad` calcule une intégrale définie
- `romberg`, `trapez`, `simps`...

```
1 import scipy.integrate as integrate
2
3 r = integrate.quad(lambda x: -x + 1, 0, 1)
4 print(r)
```

```
(0.5, 5.551115123125783e-15)
```

Optimisation

■ Optimisation de fonction avec `scipy.optimize`

Spécification de la fonction à optimiser et de la méthode

```
1 from scipy.optimize import minimize
2
3 def obj(x):
4     return x ** 2 - x + 1
5
6 r = minimize(obj, 0, method='nelder-mead', options={'disp': True})
7 print(r.x)
```

```
Optimization terminated successfully.
      Current function value: 0.750000
      Iterations: 23
      Function evaluations: 46
[ 0.5]
```

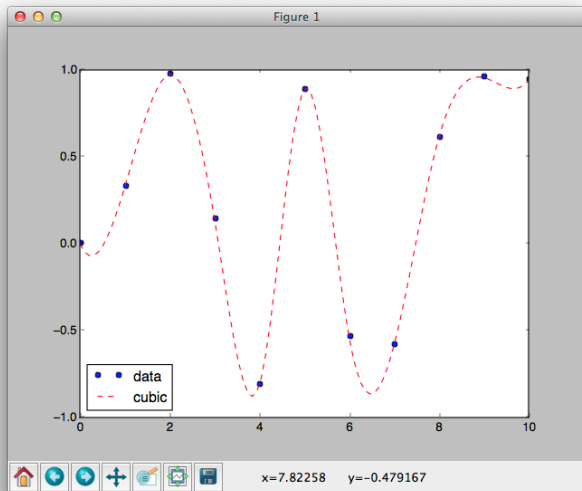

Interpolation

■ Interpolation de points avec `scipy.interpolate`

Plusieurs types de fonctions possibles

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 from scipy.interpolate import interp1d
4
5 x1 = np.linspace(0, 10, 11, endpoint=True)
6 y1 = np.sin(x1 ** 2 / 3)
7
8 f = interp1d(x1, y1, kind='cubic')
9 x2 = np.linspace(0, 10, 100, endpoint=True)
10
11 plt.plot(x1, y1, 'o', color='blue', label='data')
12 plt.plot(x2, f(x2), '--', color='red', label='cubic')
13 plt.legend(loc='best')
14 plt.show()
```

Interpolation (2)



Algèbre linéaire (1)

■ Différences entre `numpy.array` et `numpy.ndarray`

```
1 import numpy as np
2 from scipy import linalg
3
4 A = np.array([[1, 2], [3, 4]])
5 b = np.array([[5], [6]])
6
7 print(A.T)
8 print(linalg.inv(A))
9 print(A * b)
10 print(A.dot(b))
11 print(linalg.det(A))
```

```
[[1 3]
 [2 4]]
[[-2.  1. ]
 [ 1.5 -0.5]]
[[ 5 10]
 [18 24]]
[[17]
 [39]]
-2.0
```

Algèbre linéaire (2)

■ Résolution de systèmes d'équations linéaires

Résolution explicite lente ou spécifique rapide

```
1 import numpy as np
2 from scipy import linalg
3
4 A = np.array([[1, 2], [3, 4]])
5 b = np.array([[5], [6]])
6
7 print(linalg.inv(A).dot(b))
8 print(linalg.solve(A, b))
```

```
[[ -4. ]
 [  4.5]]
[[ -4. ]
 [  4.5]]
```

Librairie sympy

- Système de type **Computer Algebra System** (CAS)

Complètement écrite en Python

- De nombreuses **fonctionnalités**

- Polynôme, dérivée, intégrale, résolution équations
- Combinatoire, math discrètes, algèbre matricielle
- Géométrie, dessin de graphe
- ...

- Disponible en **open-source** sur GitHub

<https://github.com/sympy/sympy>



Déclaration des symboles

- **Déclaration de symboles** avec la fonction `symbols`

Symboles séparés par un espace sous forme d'un str

- Importance de l'**ordre de déclaration** des symboles

Objet de type `sympy.core.symbol.Symbol`

```
1 from sympy import *
2
3 x, y = symbols('x y')
4 print(x)
5 print(type(y))
```

```
x
<class 'sympy.core.symbol.Symbol'>
```

Substitution (1)

- Remplacement d'un symbole par une expression avec subs

Appel de la méthode sur une expression

- La méthode subs renvoie des expressions symboliques

```
1 expr1 = 2 * x**2 - y + 1
2 expr2 = expr1.subs(x, 2)
3
4 print(expr1)
5 print(expr1.subs(x, y**2))
6 print(expr1.subs([(x, 2), (y, 9)]))
```

```
-y + 9
2*y**4 - y + 1
0
```

Substitution (2)

- Récupération d'un **nombre flottant** avec evalf

Possibilité de substituer des valeurs et donner la précision

```
1 expr = 2 * x**2 - y + 1
2
3 expr.evalf(subs={x: 1, y: 1/3})
4 expr.evalf(4, subs={x: 1, y: 1/3})
```

```
2.6666666666666667
2.667
```


Simplification

- **Simplification d'une expression** avec `simplify`

Fonction générique qui choisit la meilleure technique

- **Plusieurs types** de simplifications spécialisés

- Pour les polynômes : `factor` et `expand`
- Pour les formules trigonométriques : `trigsimp`
- ...

```
1 expr = 3 * x**2 + y * x**2 - 4 + sin(x)**2 + cos(x)**2
2 print(simplify(expr))
3 print(factor(simplify(expr), x))
```

```
x**2*y + 3*x**2 - 3
x**2*(y + 3) - 3
```

Analyse (1)

- Calcul de **dérivée, intégrale et limite** d'une expression

Avec les fonctions `diff`, `integrate` et `limit`

- Spécification de la variable de dérivation, intégration...

```
1 expr = sin(x + 2*y)
2 print(diff(expr, x))
3 print(integrate(expr, y))
4 print(integrate(expr, (y, 0, pi/2)))
5 print(limit(expr, x, 0))
```

```
cos(x + 2*y)
-cos(x + 2*y)/2
cos(x)
sin(2*y)
```

Analyse (2)

- **Fonction non évaluée** avec Derivative, Integral et Limit

Calcul avec la méthode doit

```
1 expr1 = Derivative(sin(x + 2*y), x)
2 print(type(expr1))
3 print(expr1.doit())
4
5 expr2 = diff(expr1, x)
6 print(expr2)
7 print(expr2.doit())
```

```
<class 'sympy.core.function.Derivative'>
cos(x + 2*y)
Derivative(sin(x + 2*y), x, x)
-sin(x + 2*y)
```

Résolution d'équations (1)

- Résolution d'équations et systèmes d'équations avec `solve`

Spécification des équations et des variables

- Utilisation d'un objet `Eq` pour représenter une équation

Mais pas nécessaire pour la résolution d'équations

```
1 expr = Eq(x ** 2, -4)
2 print(solve(expr))
3
4 print(solve([x + y - 2, 2*x - 1], (x, y)))
```

```
[-2*I, 2*I]
{x: 1/2, y: 3/2}
```

Résolution d'équations (2)

- Définition du **domaine de recherche** avec solveset

Par exemple $S.Reals$, $S.Complexes$...

```
1 expr = Eq(x ** 2, -4)
2
3 print(solveset(expr, x, domain=S.Complexes))
4 print(solveset(expr, x, domain=S.Reals))
```

```
{-2*I, 2*I}
EmptySet()
```

Matrice

- **Matrices** représentées avec l'objet `Matrix`

Éléments déclarés comme un liste de listes

- **Plusieurs méthodes** d'interrogation et manipulation

```
1 m = Matrix([[1, 2], [3, 4], [5, 6]])  
2  
3 print(M.shape)  
4 M.row(0)  
5 M.col(-1)
```

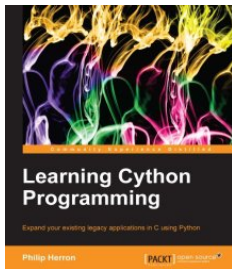
```
(3, 2)  
Matrix([[1, 2]])  
Matrix([  
[2],  
[4],  
[6]])
```

Opérations matricielles

- **Plusieurs opérations** utilisables avec des opérateurs
 - Addition, soustraction, multiplication par un scalaire
 - Multiplication matricielle, exponentiation
- **Transposition** avec l'opérateur `.T`

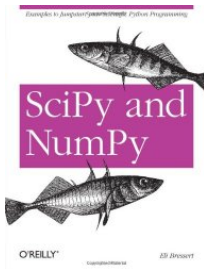
```
1 A = Matrix([[1, 2], [3, 4]])
2 B = eye(2)
3
4 print(A + B)
5 print(2 * A)
6 print(A * B)
7 print(A ** 3)
8
9 print(A.T)
```

Livres de référence



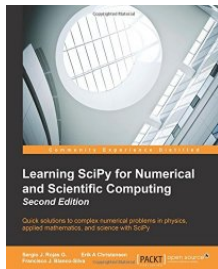
ISBN

978-1-783-28079-7



ISBN

978-1-449-30546-8



ISBN

978-1-783-98770-2

Crédits

- Photos des livres depuis Amazon
- <https://www.flickr.com/photos/danielmennerich/8626764931>
- <https://www.flickr.com/photos/oskay/4904174150>
- <https://www.flickr.com/photos/berkeleylab/3592326251>