Introduction à Python pour le calcul scientifique : bases du langage

très largement inspirée du cours de Pierre Navaro (IRMA, 2014)

Anne Cadiou

Laboratoire de Mécanique des Fluides et d'Acoustique

Séminaires et ateliers pour le calcul scientifique
PMCS2I - LMFA
Vendredi 3 février 2017

Table des matières

1 Historique

2 Calcul scientifique

3 Principes de base

4 Types de données

5 Opérateurs

6 Structures de contrôle

7 Fonctions

8 Classes

9 Fichiers

10 Modules

Table des matières

1 Historique
2 Calcul scientifique
3 Principes de base
4 Types de données
5 Opérateurs
6 Structures de contrôle
7 Fonctions
8 Classes
9 Fichiers
10 Modules

Historique

- Développé fin 1989 par Guido van Rossum (CWI, Amsterdam, Pays-Bas)
- Création de la Python Software Foundation en 2001

Caractéristiques

- Langage interprété écrit en C
- Langage open-source
- Langage utilisé dans de nombreux domaines
- S'interface avec un très grand nombre de langages
- Syntaxe aisée, facile à apprendre
- Disponible sous Unix, Windows, Mac OS X, Android, Blackberry...

Documentation abondante sur le web

https://docs.python.org

Principales versions actuelles

- Python 3.x sortie depuis fin 2008
  (first intentionally backwards incompatible release)

- Python 2.x encore largement uilisé pour le calcul scientifique
  (reste encore la version par défaut)

⇒ version décrite dans ce cours

Exemples de différences

Python 2

```
>>> print "Bonjour"
Bonjour
>>> 1/2
0
>>> 1./2
0.5
>>> from __future__ import division
>>> 1/2
0.5
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python 3

```
>>> print "Bonjour"
SyntaxError: invalid syntax
>>> print("Bonjour")
Bonjour
>>> 1/2
0.5
>>> range(10); list(range(10))
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Table des matières

1 Historique
2 Calcul scientifique
3 Principes de base
4 Types de données
5 Opérateurs
6 Structures de contrôle
7 Fonctions
8 Classes
9 Fichiers
10 Modules

Contraintes du calcul scientifique

- Développer un algorithme pour traiter un problème donné
- Implémenter l'algorithme pour le calcul
- Effectuer la simulation numérique en produisant des données
- Valider le comportement de la méthode et les résultats obtenus
- Analyser les performances du calcul
- Analyser et visualiser les résultats
- Présenter ses résultats

Pourquoi utiliser Python ?

- Syntaxe simple, apprentissage rapide
- Langage interprété : interactivité et scripts
- Langage impératif ou orienté objet
- Langage open-source, portable, très répandu et largement utilisé
- Permet le prototypage rapide d'un algorithme

  en s'appuyant sur de nombreuses bibliothèques pour le calcul scientifique
  - numpy
  - scipy
  - matplotlib
  - mpi4py
  - sage
  - ...

Table des matières

1 Historique

2 Calcul scientifique

3 Principes de base

4 Types de données

5 Opérateurs

6 Structures de contrôle

7 Fonctions

8 Classes

9 Fichiers

10 Modules

Interpréteurs sour Linux

$ python : interpréteur de base

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information
>>>
```

$ ipython : interpréteur interactif

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
Type "copyright", "credits" or "license" for more information.
IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details
In [1]:
```

IDE

Il existe un grand nombre d'IDE
(Integrated Development Environment)

- Spyder
- NetBeans
- PyDev (Eclipse)
- Pycharm
- KDevelop
- ...

avec généralement complétion de code et débogueur intégré

Notebook

Une manière de se familiariser avec le langage peut consister à commencer en utilisant un notebook (IPython ou Jupyter), lancé par exemple par

```
jupyter notebook
```

bien que le développement d'un code passe par l'écriture de programmes, avec un éditeur (et éventuellement l'aide d'un IDE)

Attention : les programmes python ont pour extension `.py` alors que les notebook sont des fichiers `.ipynb`

Écriture d'un script python

test.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
a=2
print type(a), a
```

Écriture d'un script python

test.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
a=2
print type(a), a
```

Exécution

```
$ python test.py
<type 'int'> 2
```

Passer en mode interactif après exécution du script

```
$ python -i test.py
<type 'int'> 2
>>>
```

Écriture d'un script python

test.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
a=2
print type(a), a
```

Exécution

```
$ python test.py
<type 'int'> 2
```

Passer en mode interactif après exécution du script

```
$ python -i test.py
<type 'int'> 2
>>>
```

Dans une console ipython

```
$ ipython
In [1]: run test.py
<type 'int'> 2
```

Éléments de syntaxe

Indentation : permet de différentier les blocs d'instruction

```
for n in range(0,5):
        print "n= %s"%n
print "Done"
```

Commentaires :

```
#!/usr/bin/env python
"""
SYNOPSIS
 methode pour appeler le programme
DESCRIPTION
 ce que fait le programme

AUTHOR
 anne cadiou

VERSION
  $ Id version 1.0 $
"""
# modules
import numpy as np
import matplotlib.pyplot as plt
```

Table des matières

1 Historique

2 Calcul scientifique

3 Principes de base

4 Types de données

5 Opérateurs

6 Structures de contrôle

7 Fonctions

8 Classes

9 Fichiers

10 Modules

Les types de données

En Python tout est **objet**.
Python possède 5 types de donnée standard :

- des nombres
- des chaînes de caractères
- des listes
- des tuples
- des dictionnaires

Les objets sont créés quand une valeur leur est attribuée.

Allocation mémoire

L'assignation du type de la variable est implicite : l'allocation mémoire a lieu au moment de l'assignation.

```
>>> x = 10; y = x;
>>> print id(x); print id(y)
29712512
29712512
```

Le nom `x` est attaché à l'objet entier `10`. `x` et `y` pointent sur le même espace mémoire (qui contient la valeur `10`). Réassigner `x` n'affecte pas `y` car ils ne pointent plus sur le même espace mémoire.

```
>>> x = 3
>>> print id(x); print id(y);
29712680
29712512
>>> print y
10
```

Il peut y avoir plusieurs zones mémoire par valeur mais les zones mémoires contiennent la même valeur.

```
>>> id(10.0)
29783688
>>> id(10.0)
38155984
```

Les nombres

```
>>> x = 10; print type(x) #entier 32 bits
<type 'int'>
>>> x = 10L; print type(x) #entier long
<type 'long'>
>>> x = 10.; print type(x) #réel 64 bits (sur machine 64 bits)
<type 'float'>
>>> x = 1.+2.j; print type(x) #imaginaire
<type 'complex'>
>>> x=1.24e-6/2; type(x); x
<type 'float'>
6.2e-07
```

Les chaînes de caractères

```
>>> str = 'Bonjour'
>>> print str                    #affiche la variable
Bonjour
>>> str += " tout le monde " #concaténation de 2 chaînes de caractères
>>> str += 3*'!'                 #réplique 3 fois la chaîne ' !', à la fin
>>> print str
Bonjour tout le monde !!!
```

Les simples et doubles quotes définissent les chaînes de caractères. Un sous-ensemble de la chaîne de caractère peut être accédé avec l'opérateur `[]`, indexé de 0 à -1.

```
>>> str[0]                                #1er élément
'B'
>>> str[-1]                               #dernier élément
'!'
>>> str[3:7]                              #sélection du 4ieme au 7ieme caractère
'jour'
>>> str[:3]                               #jusqu'au 3ieme caractère
'Bon'
>>> str[-1::-1]                           #...inverse
'!!! ednom el tuot ruojnoB'
```

Manipulation de chaînes de caractères

Les chaînes de caractères sont des objets sur lesquels on applique des fonctions et des méthodes pour les manipuler

```
function(objet); objet.method();
```

Exemples :

```
>>> str = 'Bonjour tout le monde !!!'
>>> len(str)                          #renvoie la taille
25
>>> str.find('on')                    #recherche la sous-chaîne
1
>>> str.replace('jour','soir')        #remplace une chaîne par une autre
'Bonsoir tout le monde !!!'
>>> str.lower()                       #tout passer en minuscules
'bonjour tout le monde !!!'
>>> str.split(' ')
['Bonjour', 'tout', 'le', 'monde', '!!!']
>>> i=int('2'); x=float("5.3"); type(i); type(x); print i,x
<type 'int'>
<type 'float'>
2 5.3
```

Afficher toutes les méthodes de la classe :

```
>>> help('str')
```

Les listes, tuples (n-uplets) et séquences

- une liste est délimitée par des "[", les n-uplets par des "(".
- les listes sont "mutables" au contraire des n-uplets.

Exemple :

```
>>> L=[1,2,3]
>>> type(L)
<type 'list'>
>>> T=(1,2,3)
>>> type(T)
<type 'tuple'>
```

Caractère "mutable" des listes :

```
>>> a = b = c = [1,2,3]
>>> print id(a),id(b),id(c)
139661177879512 139661177879512 139661177879512
>>> print a[0]; a[0] = 4
1
>>> print a,b,c
[4, 2, 3] [4, 2, 3] [4, 2, 3]
>>> a = b = c = (1,2,3)
>>> print id(a),id(b),id(c)
139661177923856 139661177923856 139661177923856
>>> print a[0]; a[0] = 4     #opération invalide (objet non modifiable)
1
TypeError: 'tuple' object does not support item assignment
```

Manipulation des listes

Les listes peuvent être modifiées ou complétées en appliquant les méthodes de la classe `list`.

```
>>> L=[1,2,3,'un','deux','trois',False]
>>> type(L[-1])
<type 'bool'>
>>> type(L[0])
<type 'int'>
>>> type(L[3])
<type 'str'>
>>> len(L)
7

>>> M=[4,'quatre']
>>> L.insert(3,M[0])
>>> print L
[1, 2, 3, 4, 'un', 'deux', 'trois', False]
>>> L.pop(-1)
False
>>> print L
[1, 2, 3, 4, 'un', 'deux', 'trois']
>>> L.append(M[1])
>>> print L
[1, 2, 3, 4, 'un', 'deux', 'trois', 'quatre']
>>> L.index('un')
4
```

Les listes : copie

Ne pas confondre la copie d'une référence et la copie des éléments d'une liste.

```
>>> L=['Dans', 'python', 'tout', 'est',  'objet']
>>> M=L
>>> M[-1]='bon'
>>> print M
['Dans', 'python', 'tout', 'est', 'bon']
>>> print L
['Dans', 'python', 'tout', 'est', 'bon']

>>> L=['Dans', 'python', 'tout', 'est',  'objet']
>>> M=L[:]
>>> M[-1]='bon'
>>> print M
['Dans', 'python', 'tout', 'est', 'bon']
>>> print L
['Dans', 'python', 'tout', 'est', 'objet']
```

Accéder aux éléments d'une liste

Slicing [debut : fin : pas]

```
>>> a = 'Demonstrate slicing in Python'.split()
>>> print 'Taille de a = ' + str(len(a))
Taille de a = 4
>>> print range(len(a))
[0, 1, 2, 3]
>>> print a[:]
['Demonstrate', 'slicing', 'in', 'Python']
>>> print a[::2]
['Demonstrate', 'in']
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '
     __delslice__', '__doc__', '__eq__', '__format__', '__ge__', '
     __getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__', '
     __iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__'
     , '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
      '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__', '
     __sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend', '
     index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> for index in range(len(a)):
...     print index, a[index]
0 Demonstrate
1 slicing
2 in
3 Python
```

```
>>> for index,elt in enumerate(a):
...     print index, elt
0 Demonstrate
1 slicing
2 in
3 Python
```

Les dictionnaires

- On les appelle aussi "tableaux associatifs"
- Les indices sont appelés "clés".
- `d['Nom']` affiche l'élément dont la clé est 'Nom'
- `d.keys()` affiche les clés
- `d.items()` affiche les éléments du dictionnaire

```
animal = {}
animal['nom'] = "girafe"
animal['taille'] = 5.0
animal['poids'] = 1100
>>> animal['nom']='girafe'
>>> animal['taille'] = 5.
>>> animal['poids']=1100
>>> print animal
{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}
>>> print animal.keys()
['nom', 'poids', 'taille']
>>> print animal.items()
[('nom', 'girafe'), ('poids', 1100), ('taille', 5.0)]
```

Table des matières

1 Historique

2 Calcul scientifique

3 Principes de base

4 Types de données

5 Opérateurs

6 Structures de contrôle

7 Fonctions

8 Classes

9 Fichiers

10 Modules

Les opérateurs

Affectation

```
=
```

Addition, soustraction, ...

```
+ - * / % // ** pow abs
```

Opérateurs logiques

```
and or not
```

Opérateurs de comparaison

```
==, is, !=, is not, >, >=, <, <=
```

Table des matières

1 Historique

2 Calcul scientifique

3 Principes de base

4 Types de données

5 Opérateurs

6 Structures de contrôle

7 Fonctions

8 Classes

9 Fichiers

10 Modules

if, elif, else

Syntaxe générale

```
if <test1>:
        <blocs instructions>
elif <test2>:
        <blocs instructions>
else:
        <blocs instructions>
```

Exemple

```
L = [1,3,6,8]
if 9 in L:
        print '9 est dans la liste L'
else:
        L.append(9)
```

while

Syntaxe générale

```
while <test1>:
        <blocs instructions>
        if <test2>: break       #sort de la boucle
        if <test3>: continue    #remonte au début de la boucle
else:                           #exécuté si la boucle se termine normalement
        <blocs instructions>
```

Exemple

```
x = y / 2
while x > 1:
        if y % x == 0:
                print y, ' est facteur de ',x
                break
        x -= 1
else:
        print y, ' est un nombre premier'
```

for

Syntaxe générale

```
for <cible> in <objet>:
        <blocs instructions>
        if <test1>: break       #sort de la boucle
        if <test2>: continue    #remonte au début de la boucle
else:                           #exécuté si la boucle se termine normalement
        <blocs instructions>
```

Exemples

```
sum = 0
for i in range(1,5):
        sum += i

L = [10.0]*20
print [ id(l) for l in L ]
```

Pour les tableaux de grande taille, préférer `xrange` à `range`

```
$ python -m timeit 'for i in range(1000000):' ' pass'
10 loops, best of 3: 46.6 msec per loop
$ python -m timeit 'for i in xrange(1000000):' ' pass'
10 loops, best of 3: 32.2 msec per loop
```

Les fonctions `map` et `zip`

- **zip** : permet de parcourir plusieurs séquences en parallèle

```
>>> L1 = [1, 2, 3]
>>> L2 = [4, 5, 6]
>>> for (x, y) in zip(L1, L2):
...     print x, y, '--', x + y
...
1 4 -- 5
2 5 -- 7
3 6 -- 9
```

Les fonctions `map` et `zip`

- **zip** : permet de parcourir plusieurs séquences en parallèle

```
>>> L1 = [1, 2, 3]
>>> L2 = [4, 5, 6]
>>> for (x, y) in zip(L1, L2):
...     print x, y, '--', x + y
...
1 4 -- 5
2 5 -- 7
3 6 -- 9
```

- **map** : applique une méthode sur une ou plusieurs séquences

```
>>> from math import factorial
>>> map(factorial,range(4))
[1, 1, 2, 6]
>>> def add(x,y):
...     return x+y
...
>>> map(add,L1,L2)
[5, 7, 9]
```

Les fonctions `map` et `zip`

- **zip** : permet de parcourir plusieurs séquences en parallèle

```
>>> L1 = [1, 2, 3]
>>> L2 = [4, 5, 6]
>>> for (x, y) in zip(L1, L2):
...     print x, y, '--', x + y
...
1 4 -- 5
2 5 -- 7
3 6 -- 9
```

- **map** : applique une méthode sur une ou plusieurs séquences

```
>>> from math import factorial
>>> map(factorial,range(4))
[1, 1, 2, 6]
>>> def add(x,y):
...     return x+y
...
>>> map(add,L1,L2)
[5, 7, 9]
```

$\Rightarrow$ map peut être beaucoup plus rapide qu'une boucle for.

Table des matières

1 Historique

2 Calcul scientifique

3 Principes de base

4 Types de données

5 Opérateurs

6 Structures de contrôle

7 Fonctions

8 Classes

9 Fichiers

10 Modules

Fonctions

L'indentation délimite le corps de la fonction.

Syntaxe générale

```
def <nom_fonction>(arg1, arg2, ... argN):
        blocs instructions
        return <valeur(s)>
```

Exemple

```
from math import sort
def norme(x, y):
   return sqrt(x*x+y*y)
>>> norme(3,4)
5
```

En l'absence de return une fonction renvoie la valeur None

```
norme = lambda x,y: sqrt(x*x+y*y)
>>> norme(3,4)
5
```

Gestion des paramètres et récursivité

Valeur par défaut et utilisation des paramètres par leurs noms :

```
>>> def somme(a,b=5):
...     return a+b
...
>>> somme(1)
6
>>> somme(b="a",a="bc")
'bca'
```

Fonction `def avec un appel récursif`

```
def fibo( n ):
    """ Retourne le nombre
         de Fibonacci n """
    if n == 0 or n == 1:
       return n
   else:
       return fibo( n - 1 ) + fibo( n - 2 )
print fibo(9)
help(fibo)
```

Déclaration d'une fonction sans connaître ses paramètres

Fonction à nombre variable de paramètres

```
>>> def somme(*args):
...     print args
...     sum(args)
...     return sum(args)
>>> somme(4,5)
(4, 5)
9
```

Fonction à nombre variable de paramètres nommés

```
>>> def myfunc(**kwargs):
...     print kwargs
...
>>> myfunc(a=4, b='python', pi=3.14)
{'a': 4, 'pi': 3.14, 'b': 'python'}
```

Portée des variables

Chaque fonction possède son propre environnement.

```
>>> pi = 1.
>>> def deg2rad(theta):
...     pi = 3.14
...     return theta * pi / 180.
...
>>> deg2rad(45)
0.785
>>> pi
1.0
```

```
>>> def deg2rad(theta):
...     global pi
...     pi = 3.14
...     return theta * pi / 180
...
>>> pi = 1
>>> deg2rad(45)
0.785
>>> pi
3.14
```

```
>>> def rad2deg(theta):
...     return theta*180./pi
...
>>> rad2deg(0.785)
141.3
>>> pi = 3.14
>>> rad2deg(0.785)
45.0
```

Dans les fonctions, les variables ont une portée "locale".

Table des matières

1 Historique

2 Calcul scientifique

3 Principes de base

4 Types de données

5 Opérateurs

6 Structures de contrôle

7 Fonctions

8 Classes

9 Fichiers

10 Modules

Programmation Objet avec Python

En programmation procédurale il y a une séparation stricte entre code et données. En programmation objet les opérations sont des propriétés des données.

- Une classe python est composée d'attributs et de méthodes.

Programmation Objet avec Python

En programmation procédurale il y a une séparation stricte entre code et données. En programmation objet les opérations sont des propriétés des données.

- Une classe python est composée d'attributs et de méthodes.
- Instance : création d'un objet, appel de la méthode nommée __init__.

Programmation Objet avec Python

En programmation procédurale il y a une séparation stricte entre code et données. En programmation objet les opérations sont des propriétés des données.

- Une classe python est composée d'attributs et de méthodes.
- Instance : création d'un objet, appel de la méthode nommée __init__.
- Héritage : un objet qui dérive d'un autre objet possède les mêmes attributs et les mêmes méthodes.

Programmation Objet avec Python

En programmation procédurale il y a une séparation stricte entre code et données. En programmation objet les opérations sont des propriétés des données.

- Une classe python est composée d'attributs et de méthodes.
- Instance : création d'un objet, appel de la méthode nommée __init__.
- Héritage : un objet qui dérive d'un autre objet possède les mêmes attributs et les mêmes méthodes.
- Quand une méthode est appelée, une référence vers l'objet principal est passée en tant que premier argument. Par convention ce premier argument est toujours : `self`.

Une classe Python

Définition

```
class MaClasse:
   'Ma premiere classe'
   def __init__(self, i, j):
      self.i = i; self.j = j
   def write(self):
      print MaClasse: i=,self.i,j=,self.j
```

Une classe Python

Définition

```
class MaClasse:
   'Ma premiere classe'
   def __init__(self, i, j):
      self.i = i; self.j = j
   def write(self):
      print MaClasse: i=,self.i,j=,self.j
```

L'argument `self` est obligatoire pour la définition.

Utilisation

```
>>> obj1 = MaClasse(6,9)
>>> print obj1.i, obj1.j
6 9
>>> obj1.write()
MaClasse: i= 6 j= 9
```

Surcharge de l'opérateur print

Définition

```
class MaClasse:
   """Ma premiere classe
    avec surcharge de l'opérateur print """
   def __init__(self, i, j):
      self.i = i; self.j = j
   def __repr__(self):
      return MaClasse: i=+str(self.i)+j=+str(self.j)
```

Surcharge de l'opérateur print

Définition

```
class MaClasse:
   """Ma premiere classe
    avec surcharge de l'opérateur print """
   def __init__(self, i, j):
      self.i = i; self.j = j
   def __repr__(self):
      return MaClasse: i=+str(self.i)+j=+str(self.j)
```

Utilisation

```
>>> obj1 = MaClasse(6,9)
>>> print obj1.i, obj1.j
6 9
>>> print obj1
MaClasse: i= 6 j= 9
```

Surcharge des opérateurs

Une classe vecteur

```
class Vecteur:
   dimension = 2
   def __init__(self, x=0, y=0):
      self.x=x
      self.y=y
   def __eq__(self, vB):
      return (self.x==vB.x) and (self.y==vB.y)
   def __add__(self, vB):
      return Vecteur(self.x+vB.x,self.y+vB.y)
   def __sub__(self, vB):
      return Vecteur(self.x-vB.x,self.y-vB.y)
   def __mul__(self, c):
      if isinstance(c,Vecteur):
         return self.x*c.x+self.y*c.y
      else:
         return Vecteur(c*self.x,c*self.y)
```

Surcharge des opérateurs (suite)

Manipulation

```
>>> U = Vecteur(); U.x,U.y = 0,1
>>> V = Vecteur(); V.x,V.y = 1,0
>>> print U, V
<__main__.Vecteur instance at 0x2b8ee0> <__main__.Vecteur instance at 0x2b8f30>
>>> W = U+V
>>> print W
<__main__.Vecteur instance at 0x2b8f80>
>>> Vecteur.dimension=3

>>> print 'Dimension de U=' + str(U.dimension)
Dimension de U=3
>>> print 'Dimension de V=' + str(V.dimension)
Dimension de V=3
>>> U.dimension=4
>>> print 'Dimension de V=' + str(V.dimension)
Dimension de V=3
>>> Vecteur.dimension=5
>>> print 'Dimension de U=' + str(U.dimension)
Dimension de U=4
>>> print 'Dimension de V=' + str(V.dimension)
Dimension de V=5
```

Héritage

Définition de la sous classe

```
class MaSousClasse(MaClasse):
   def __init__(self, i, j, k):
      MaClasse.__init__(self,i,j)
      self.k = k
   def write(self):
      print MaSousClasse: i=,self.i,j=,self.j, \
          k=, self.k
```

Héritage

Définition de la sous classe

```
class MaSousClasse(MaClasse):
   def __init__(self, i, j, k):
      MaClasse.__init__(self,i,j)
      self.k = k
   def write(self):
      print MaSousClasse: i=,self.i,j=,self.j, \
          k=, self.k
```

Utilisation

```
>>> obj1 = MaClasse(6,9)
>>> obj1.write()
MaClasse: i= 6 j= 9
>>> obj2 = MaSousClasse(6,9,12)
>>> obj2.write()
MaSousClasse: i = 6  j= 9  k =  12
```

Exemple de structure de programme

```
#! /usr/bin/env python
""" DESCRIPTION classe """
# classes
class Point(object):
  """coordonnees d'un point dans l'espace cartesien 2D"""
  def __init__(self,x=0.0,y=0.0):
    self.x = x; self.y = y
  def __str__(self):
    return "Point (%s,%s)"%(self.x,self.y)
  def trace(self):
    fig = plt.figure();
    X=[self.x]; Y=[self.y]; plt.plot(X, Y, "ko-",mfc="r",mec="k");
    plt.show()
    return
# fonctions
def distance(P0,P1):
  d=np.sqrt((P0.x-P1.x)**2+(P0.y-P1.y)**2)
  return d
# programme
def test():
  A = Point(0.0,0.0); B = Point(1.0,0.0);
  print A; A.trace(); print "distance AB : ",distance(A,B)
  return
# entete general du module
if __name__ == "__main__":
    import numpy as np
    import matplotlib.pyplot as plt
    test()
```

Une classe qui s'utilise comme une fonction

Définition

```
class F:
   def __init__(self, a=1, b=1, c=1):
      self.a=a; self.b=b; self.c=c
   def __call__(self, x, y):
      return self.a+self.b*x+self.c*y*y
```

Une classe qui s'utilise comme une fonction

Définition

```
class F:
   def __init__(self, a=1, b=1, c=1):
      self.a=a; self.b=b; self.c=c
   def __call__(self, x, y):
      return self.a+self.b*x+self.c*y*y
```

Appel

```
>>> f = F(a=2,b=4)
>>> v = f(2,1) + f(1.2,0)
>>> print v
17.8
```

Pratique lorsque l'on souhaite séparer les paramètres d'une fonction (a,b,c) des variables indépendantes (x,y).

Table des matières

1 Historique

2 Calcul scientifique

3 Principes de base

4 Types de données

5 Opérateurs

6 Structures de contrôle

7 Fonctions

8 Classes

9 Fichiers

10 Modules

Entrées sorties

input et raw_input

```
>>> a=input("a? ")
a? [1,2,3]
>>> a
[1, 2, 3]
>>> type(a)
<type 'list'>
>>> a = raw_input("a? ")
a? [1,2,3]
>>> a
'[1,2,3]'
>>> type(a)
<type 'str'>
```

Création d'un objet fichier

```
f=open("fichier.txt")
for i in f:
        print (i)
f.close()
```

```
$ date > fichier.txt
$ python
>>> f=open("fichier.txt",mode='r')
>>> for i in f:
...     print i
...
Ven 10 jui 2011 15:07:34 CEST

>>> f.close()
```

Mode 'r' (lecture) 'w' écriture 'a' ajout 'b' binaire

Les méthodes de l'objet fichier

- f.close()
- f.read() : lit l'ensemble du fichier et le renvoie sous forme de chaîne.
- f.readline() : lit et renvoie une ligne du fichier de f, retour inclus.
- f.readlines() : lit et renvoie une liste de toutes les lignes, où chaque ligne est représentée par une chaîne
- f.write(s) : écrit la chaîne s
- f.writelines(lst) : écrit la liste de chaîne lst

Autres méthodes : modules pickle, h3py, numpy.loadtxt ...

Table des matières

1 Historique
2 Calcul scientifique
3 Principes de base
4 Types de données
5 Opérateurs
6 Structures de contrôle
7 Fonctions
8 Classes
9 Fichiers
10 Modules

Module python

dans un fichier fibo.py

```
def fib(n):
   """ write Fibonacci series up to n """
   a, b = 0, 1
   while b < n:
      print b
      a, b = b, a+b
```

Module python

dans un fichier fibo.py

```
def fib(n):
   """ write Fibonacci series up to n """
   a, b = 0, 1
   while b < n:
      print b
      a, b = b, a+b
```

Appel de la fonction fib

Méthode 1

```
import fibo
fibo.fib(1000)
```

Méthode 2

```
from fibo import fib
fib(1000)
```

Exécution d'un module python

```
def print_fib(n):
   "écrit la série de Fibonacci jusqu'à n"
   a, b = 0, 1
   while b < n:
      print b,
      a, b = b, a+b
def list_fib(n):
   "retourne la série de Fibonacci jusqu'à n "
   result,a,b =[],0,1
   while b < n:
       result.append(b)
       a, b = b, a+b
   return result
```

Exécution d'un module python

```
def print_fib(n):
   "écrit la série de Fibonacci jusqu'à n"
   a, b = 0, 1
   while b < n:
      print b,
      a, b = b, a+b
def list_fib(n):
   "retourne la série de Fibonacci jusqu'à n "
   result,a,b =[],0,1
   while b < n:
       result.append(b)
       a, b = b, a+b
   return result

if __name__ == '__main__':
   print_fib(1000)
   print list_fib(100)
```

```
$ python fibo.py
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Importation d'un module python

- `import fibo`
- `import fibo as f`
- `from fibo import print_fib, list_fib`
- `from fibo import *` (importe tous les noms sauf variables et fonctions privées)

Remarque : En Python, les variables ou les fonctions des objets du langage commencent par __.

Importation d'un module python

- `import fibo`
- `import fibo as f`
- `from fibo import print_fib, list_fib`
- `from fibo import *` (importe tous les noms sauf variables et fonctions privées)

Remarque : En Python, les variables ou les fonctions des objets du langage commencent par __. On peut accéder aux attributs du module avec:

- __dict__ : dictionnaire utilisé par le module pour l'espace de noms des attributs
- __name__ : nom du module
- __file__ : fichier du module
- __doc__ : documentation du module

Chemin de recherche d'un module python

Recherche dans sys.path

- dans le répertoire courant
- dans PYTHONPATH si défini (même syntaxe que PATH)
- dans un répertoire par défaut (sous Linux : /usr/lib/python)
- `from fibo import *` (importe tous les noms sauf variables et fonctions privées)

Chemin de recherche d'un module python

Recherche dans sys.path

- dans le répertoire courant
- dans PYTHONPATH si défini (même syntaxe que PATH)
- dans un répertoire par défaut (sous Linux : /usr/lib/python)
- `from fibo import *` (importe tous les noms sauf variables et fonctions privées)

Ajout de mon dans sys.path

```
import sys
sys.path.append('le/chemin/de/mon/module')
import mon_module
```

Exemple d'un module avec différents répertoires

```
monModule/ Paquetage de niveau supérieur
        __init__.py Initialisation du paquetage monModule
        sous_module1/ Sous-paquetage
                __init__.py
                fichier1_1.py
                fichier1_2.py ...
        sous_module2/ Sous-paquetage
                __init__.py
                fichier2_1.py
                fichier2_2.py ...
```

Le fichier __init__.py est obligatoire pour que Python considère les répertoires comme contenant des paquetages. Il peut être vide ou contenir du code d'initialisation.

Les modules standards

- sys : variables système et accès aux options passées en ligne de commande.

Les modules standards

- sys : variables système et accès aux options passées en ligne de commande.
- os : informations sur l'OS, manipulations de fichiers et gestions des processus.

Les modules standards

- sys : variables système et accès aux options passées en ligne de commande.
- os : informations sur l'OS, manipulations de fichiers et gestions des processus.
  - getcwd() : renvoie le chemin menant au répertoire courant

Les modules standards

- sys : variables système et accès aux options passées en ligne de commande.
- os : informations sur l'OS, manipulations de fichiers et gestions des processus.
  - getcwd() : renvoie le chemin menant au répertoire courant
  - abspath(path) : renvoie le chemin absolu de path

Les modules standards

- sys : variables système et accès aux options passées en ligne de commande.
- os : informations sur l'OS, manipulations de fichiers et gestions des processus.
  - getcwd() : renvoie le chemin menant au répertoire courant
  - abspath(path) : renvoie le chemin absolu de path
  - exists(path) : renvoie True si path désigne un fichier ou un répertoire existant, False sinon

Les modules standards

- `sys` : variables système et accès aux options passées en ligne de commande.
- `os` : informations sur l'OS, manipulations de fichiers et gestions des processus.
  - getcwd() : renvoie le chemin menant au répertoire courant
  - abspath(path) : renvoie le chemin absolu de path
  - exists(path) : renvoie True si path désigne un fichier ou un répertoire existant, False sinon
- `math` : pi, sqrt, cos, sin, tan,...

Les modules standards

- `sys` : variables système et accès aux options passées en ligne de commande.
- `os` : informations sur l'OS, manipulations de fichiers et gestions des processus.
  - getcwd() : renvoie le chemin menant au répertoire courant
  - abspath(path) : renvoie le chemin absolu de path
  - exists(path) : renvoie True si path désigne un fichier ou un répertoire existant, False sinon
- `math` : pi, sqrt, cos, sin, tan,...
- `string`

Les modules standards

- `sys` : variables système et accès aux options passées en ligne de commande.
- `os` : informations sur l'OS, manipulations de fichiers et gestions des processus.
  - getcwd() : renvoie le chemin menant au répertoire courant
  - abspath(path) : renvoie le chemin absolu de path
  - exists(path) : renvoie True si path désigne un fichier ou un répertoire existant, False sinon
- `math` : pi, sqrt, cos, sin, tan,...
- `string`
- `time`

Références

- La documentation officielle : `http://docs.python.org/tutorial`
- "Learning Python" de Mark Lutz et David Ascher chez O'Reilly
- "Python Scripting for Computational Science" de Hans Petter Langtangen chez Springer
- "Cours de Python" de Patrick Fuchs et Pierre Poulain : `http://www.dsimb.inserm.fr/~fuchs/python/`
- Formation Python en calcul scientifique `http://calcul.math.cnrs.fr/spip.php?article164`