Scientific Computation with Python

Yue-Xin Huang

School of Sciences, Great Bay University

August 1, 2025

Outline

Outline

Python

优点

- 简洁、简单，自动管理内存
- 解析型语言，不需要编译
- 存在很多函数包，调用各种包实现各式各样的功能
- 面向对象设计，易于打包和重复利用
- 用户文档丰富

缺点

- 慢
- 乱

Python 科学计算包

Numpy: numeric Python
Scipy: scientific Python
Matplotlib: graphics library

- 用 C、Fortran 语言实现，速度快
- 调用库：blas, lapack, arpack, Intel MKL
- 并行计算 (多进程、多线程)，GPU 运算

安装和管理

Linux

- **一般都已经安装好基本的 Python 环境**
  sudo apt-get install python-setuptools python-pip
- **用 pip 来管理 Python 环境**
- sudo pip install <package-name>
  pip search <name>
  pip list
  sudo pip remove <package-name>

Windows

- **官网 (www.python.org) 下载**
- Anaconda: http://anaconda.org
- conda install <package-name>
  e.g. conda install numpy
  conda install -c conda-forge qutip

运行 Python

```
hyxin@hyxin-huang:~$ python
Python 2.7.12+ (default, Sep 17 2016, 12:08:02)
[GCC 6.2.0 20160914] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello world!"
Hello world!
>>>
```

在 Python 环境中直接运行代码

```
hyxin@hyxin-huang:~$ cat hello_world.py
#! /usr/bin/python

print "Hello world!"
hyxin@hyxin-huang:~$ python hello_world.py
Hello world!
```

代码写入文件

- Jupyter notebook: 交互式 shell，提供补全、缩进等功能；网页端打开 notebook
- Spyder: 类似 Matlab 的 Python 集成环境

Outline

数据类型和基本运算

整型 *(int)*

- 1, 0, 100, -234
- 64 位系统：$-2^{63}-1 \sim 2^{63}-1$

浮点型 *(float)*

- 1.0, 1.345, 2.3e5, -5.4e-6
- 2.2250738585072014e-308 $\sim$ 1.7976931348623157e+308

复数 *(complex)*

- 2.3+4.5j

布尔值

- True, False

字符 *(str)*

- “a”, “hello world”, 'xyz', 'I\'m Jim'

Operator

- $+$, $-$, $*$, $/$, //(整除), **(幂), %(取模)
- **逻辑运算**：and, or, not
- **判断运算**：$>$, $<$, $>=$, $<=$, $==$, !=

数据结构

列表（list）和元组（tuple）

- 列表：[ ], [1,2,3], ['ab, 'cd', 1, 2.3, 1.2+3.4j]
- 元组：(), (1,2,3), ('ab, 'cd', 1, 2.3, 1.2+3.4j), 一旦赋值后就不可更改

```
In [1]: a= [ 1, 2, 4, -4]

In [3]: a
Out[3]: [1, 2, 4, -4]

In [2]: len(a)
Out[2]: 4

In [4]: a.append(5)

In [5]: a
Out[5]: [1, 2, 4, -4, 5]

In [7]: a.pop()
Out[7]: 5

In [9]: a.remove(1)

In [10]: a
Out[10]: [2, 4, -4]
```

```
In [19]: a=[ 2,4,5,-2,-5]

In [20]: a[0]
Out[20]: 2

In [21]: a[2]
Out[21]: 5

In [22]: a[-1]
Out[22]: -5

In [23]: a[0:3]
Out[23]: [2, 4, 5]

In [24]: a[-2:]
Out[24]: [-2, -5]

In [25]: a.reverse()

In [26]: a
Out[26]: [-5, -2, 5, 4, 2]

In [27]: a.sort()

In [28]: a
Out[28]: [-5, -2, 2, 4, 5]
```

数据结构

```
In [29]: b=( 1, 4, 5, 7 )

In [30]: b
Out[30]: (1, 4, 5, 7)

In [31]: type(b)
Out[31]: tuple

In [32]: b[2]=100

TypeErrorTraceback (most recent call last)
<ipython-input-32-4383960d9766> in <module>()
----> 1 b[2]=100

TypeError: 'tuple' object does not support item assignment
```

不能给元组赋值

```
[2]: name='Jim'
     age=18
     print('%s is %d years old' % (name, age))

     Jim is 18 years old

[3]: x=1234
     y=4563
     x,y=y,x

[4]: print(x,y)

     4563 1234
```

元组一般用法

数据结构

字典（dic）和集合（set）

- 空字典 { }
- 字典的定义模式是：{'keyword': value,...}, 如 d= { 'name': 'Jim', 'age': 18 , 'weight': 50 }
- 集合定义 s= set([1,2,1])

```
[5]: d = {'name': 'Jim', 'age': 18, 'weight': 50}
     type(d)

[5]: dict

[6]: d['name']

[6]: 'Jim'

[7]: s = set([1,2,3,1,2,4])
     print(s)

     {1, 2, 3, 4}

[8]: type(s)

[8]: set
```

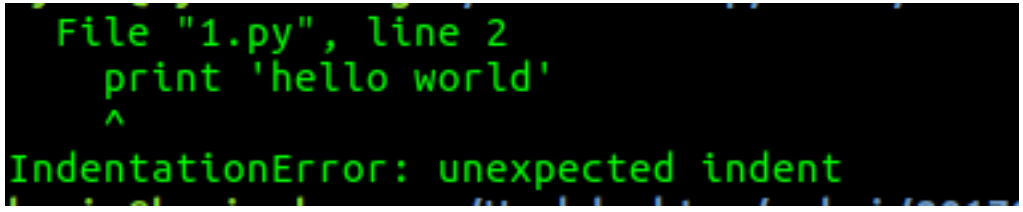
字典与集合

强制缩进

- 其它语言（C/C++, Fortran）的缩进，大多数情况下都是为了美观和可读性
- 对于 Python，行首的缩进和对齐是极其重要的。在逻辑行首的空白（空格和制表符）用来决定逻辑行的缩进层次，从而用来决定语句的分组
- 同一层次语句必须有相同的缩进

```
# file: 1.py

a= 100
  print('hello world')
```




```
# file: 2.py

a= 111
print('hello world')
for i in range(0,10):
    a= a+i
    print i
print a
```

控制流

if 语句

```
if 判断语句:
    语句 1
    语句 2
elif:
    语句 3
    语句 4
else:
    语句 5
    语句 4
```

```
[9]: if 3>4:
         print('Hello')
     elif 3==4:
         print('world')
     else:
         print('!!!')

     !!!

[10]: if True:
          print('It is True')

      It is True

[11]: if 3<4:
      print('hello')

        Cell In[11], line 2
          print('hello')
          ^
      IndentationError: expected an indented block after 'if' statement on line 1
```

if 语句用法

注意：if 语句在结尾处包含一个冒号——我们通过它告诉 Python 下面跟着一个语句块，接下来的语句要缩进并且对齐，这些语句隶属于 if，直至缩进结束（对 for，while 也一样）。

控制流

for 语句

for 变量 in 序列:
语句 1
语句 2
语句 3

- for 语句在一序列的对象上递归，即逐一使用队列中的每个项目
- 序列可以是 list，tuple，dic，set 等

```
[12]: lst = [1, 4, 9, 16, 25]
      for i in lst:
          print(i)

1
4
9
16
25

[13]: for i,value in enumerate(lst):
          print(i,value)

0 1
1 4
2 9
3 16
4 25
```

- 建立列表 x= [i**2 for i in range(0,5)]

控制流

while 语句

while 判断语句:
    语句 1
    语句 2
    语句 3
else:
    语句 4
    语句 5
    语句 6

[14]:
```
i = 1
while i<10:
    print(i)
    i = i+2
else:
    print('i>=10')
```

```
1
3
5
7
9
i>=10
```

函数

```
def function-name(参数表):
    ''' document of the function '''
    i=1
    ...
    return xxx
```

- 参数可以是 Python 内的任何元素：整型，浮点，列表，字典，元组，集合等待
- function-name.__doc__ 查看函数的文档，等同于 help(function-name)

函数

```
def power(x,n):
    tmp= 1.0
    for i in range(0,n):
        tmp= tmp*x
    return tmp
```

默认参数

```
def power1(x,n=2):
    tmp= 1.0
    for i in range(0,n):
        tmp= tmp*x
    return tmp
```

```
In [468]:
def power(x,n):
    tmp= 1.0
    for i in range(0,n):
        tmp= tmp*x
    return tmp

def power1(x,n=2):
    ''' return x**n, n= 2 by default '''
    tmp= 1.0
    for i in range(0,n):
        tmp= tmp*x
    return tmp

In [469]:
print power(4,3), power1(5,3)

64.0 125.0

In [470]:
power1(5)

Out[470]:
25.0

In [473]:
print power1.__doc__ # 与 help(power1) 等价

 return x**n, n= 2 by default
```

可变参数的函数

- 参数个数可变
- 如：$\sum_i x_i$，每次传入的 x_i 个数不一样

```
def summ(lst):
    ss= 0.0
    for i in lst:
        ss= ss+ i
    return ss
```

```
def summ1(*lst):
    ss= 0.0
    for i in lst:
        ss= ss+ i
    return ss
```

```
In [64]: lst= [1,2,3,4]
         summ(lst)
Out[64]: 10.0

In [65]: summ(1,2,3,4)

TypeErrorTraceback (most recent call last)
<ipython-input-65-3a8018c67e03> in <module>()
----> 1 summ(1,2,3,4)

TypeError: summ() takes exactly 1 argument (4 given)
```

```
In [66]: summ1(1,2,3,4)
Out[66]: 10.0

In [67]: lst= [1,2,3,4]
         summ1(*lst)
Out[67]: 10.0

In [68]: summ1()
Out[68]: 0.0
```

注：在 summ1 函数中，参数个数可变，其实是传入的参数自动组装成一个元组，参与运算

关键字参数

关键子参数允许传入任意个含参数名的参数，以此可以方便扩展函数的用法

```
def print_information( **kw ):
    for i in kw:
        print i, 'is' , kw[i]
```

[15]:
```
def print_information(**kw):
    for i in kw:
        print(i, 'is', kw[i])
```

[16]:
```
print_information(name='Jim', age= 18, weight= 50)
```

```
name is Jim
age is 18
weight is 50
```

[19]:
```
Jim= {'name': 'Jim', 'age': 18, 'weight': 50}
print_information(**Jim)
```

```
name is Jim
age is 18
weight is 50
```

注：在 print_information 函数中，不仅参数个数可变，变量名称也可变，其实是传入的参数自动组装成一个字典 (dic)，参与运算

函数

Python 内的任何函数都可以表示成如下形式：

```
func(vars, *args, **kw)
```

参数定义的顺序必须是：

- 必选参数
- 默认参数
- 可变参数
- 关键字参数

高阶函数用法

- 函数的函数

```
def func_func( f, *lst ):
    for i in lst:
        tmp= tmp+ f(i)
    return tmp
```

- Python 内建的 map() 函数: map(func, list)
- Python 内建的 reduce() 函数:reduce(func, list): 类似于 Baker-Hausdorff 公式的展开

```
In [78]: func_func( np.sin, 1, 2, 3 )
Out[78]: 1.8918884196934453

In [79]: func_func( np.cos, 1, 2, 3 )
Out[79]: -0.865837027279448

In [80]: map( np.sin, [1,2,3] )
Out[80]: [0.8414709848078965, 0.90929742682568171, 0.14112000805986721]

In [86]: reduce( power, [ 2, 3, 4 ] )
Out[86]: 4096.0
```

power(power(x1,x2),x3)= $(2^3)^4 = 8^4$

- filter(func, list), sorted(list, func), ...

高阶函数用法

- 返回函数

```
def func_sum(lst):
    def tmp(x):
        final= 0.0
        for i in lst:
            final= final+ i**x
        return final
    return tmp
```

```
In [99]: f= func_sum( 1,2,3,4)

In [103]: type(f)
Out[103]: function

In [101]: f(2)
Out[101]: 30.0
```

- 匿名函数: 如定义 l= lambda x: x**2，即可使用 l(x) 函数，返回 x^2

类和对象

Python 下所有的元素都可以看成是一个对象：变量，函数，模块

- 用 class 语句创建一个类
- self: 指当前类，类似 C++ 的 self，fortran 和 java 的 this
- ___init___ 方法: 在类的一个对象被建立时，马上执行

```
[20]: class person:
          def __init__(self, name, age, weight):
              self.name= name
              self.age= age
              self.weight= weight

          def print_information(self):
              print('Name is %s' % self.name)
              print('age is %d' % self.age)
              print('weight is %d' % self.weight)

[21]: jim= person('Jim', 18, 50)

[22]: jim.age

[22]: 18

[23]: jim.print_information()

      Name is Jim
      age is 18
      weight is 50
```

定义了一个叫 person 的类，person 类中有两种方法 (method)，___init___ 在定义对象时马上执行，print_infor 打印信息，利用 person 定义一个对象 jim

Outline

模块 (module)

引入模块

- import module-name [as xx]
  e.g.: import sys, import numpy, ...
  import numpy as np
- from module-name import *: 这样引入可能会引入非常多的函数，有可能会拖慢 Python，也可能造成不同模块间的函数冲突
  e.g.: from qutip import *
- from module-name import xxx: 只引入给定模块的特定函数
  e.g.: from scipy.fftpack import fft, ifft

编写自己的模块

- 每个 Python 程序都是一个模块
- 保存为 filename.py
- 就可以在别的程序中引入：import filename

模块

[25]:
```
import sys
```

[28]:
```
sys.platform
```

[28]:
```
'linux'
```

[29]:
```
import numpy as np
```

[30]:
```
np.linspace(0,10,21)
```

[30]:
```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
        5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. ])
```

[33]:
```
from qutip import *
```

[34]:
```
destroy(4)
```

[34]:
```
Quantum object: dims=[[4], [4]], shape=(4, 4), type='oper', dtype=Dia, isherm=False
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1.414 & 0 \\ 0 & 0 & 0 & 1.732 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

模块的使用

Numpy

- Numpy 提供向量、矩阵、高维数据结构的处理
- Python 下的数值计算，底层是用 C 和 Fortran 实现的

[35]:
```
import numpy as np
```

[36]:
```
np.linspace(0,1,11)
```

[36]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

[38]:
```
a= np.array([1,2,3,4])
b= np.array([[1,2,3],[4,5,6],[7,8,9]])
type(a), type(b)
```

[38]:
```
(numpy.ndarray, numpy.ndarray)
```

[39]:
```
print( a.max(), a.mean(), b.size, b.shape )
```

```
4 2.5 9 (3, 3)
```

[42]:
```
c= np.array([[1+1j,2+2j],[3+3j,4+4j]])
print( 'Conjugate of c=',c.conjugate() )
print( 'sum of c=', c.sum() )
```

```
Conjugate of c= [[1.-1.j 2.-2.j]
 [3.-3.j 4.-4.j]]
sum of c= (10+10j)
```

[47]:
```
d= b[0:2,0:2] # 截取矩阵b中的数据
```

[50]:
```
c * d  # 对应的元素相乘
```

[50]:
```
array([[ 1. +1.j,  4. +4.j],
       [12.+12.j, 20.+20.j]])
```

[51]:
```
c@d # 矩阵相乘
```

[51]:
```
array([[ 9. +9.j, 12.+12.j],
       [19.+19.j, 26.+26.j]])
```

Numpy

```
[52]: import numpy.linalg as lg
```

```
[53]: np.diag([1,2,3])
```

```
[53]: array([[1, 0, 0],
             [0, 2, 0],
             [0, 0, 3]])
```

```
[54]: m= np.matrix([[1,2+2j,3],[2,3+3j,4],[3,4,5]])
```

```
[56]: m.H # conjugate transpose
```

```
[56]: matrix([[1.-0.j, 2.-0.j, 3.-0.j],
              [2.-2.j, 3.-3.j, 4.-0.j],
              [3.-0.j, 4.-0.j, 5.-0.j]])
```

```
[63]: m.I # inverse of m
```

```
[63]: matrix([[-1.87500000e+00-0.125j,  1.25000000e+00+0.25j ,
                1.25000000e-01-0.125j],
              [-6.93889390e-18+0.25j ,  1.38777878e-17-0.5j  ,
               -6.93889390e-18+0.25j ],
              [ 1.12500000e+00-0.125j, -7.50000000e-01+0.25j ,
                1.25000000e-01-0.125j]])
```

```
[62]: iden= m@m.I
      np.where( np.abs(iden)< 1e-10, 0, iden )
```

```
[62]: array([[1.+1.38777878e-17j, 0.+0.00000000e+00j, 0.+0.00000000e+00j],
             [0.+0.00000000e+00j, 1.+4.16333634e-17j, 0.+0.00000000e+00j],
             [0.+0.00000000e+00j, 0.+0.00000000e+00j, 1.+2.77555756e-17j]])
```

Numpy

In [522]:
```
f= open('file.txt', 'w')

f.write('hello world')
a= np.array( [1,2,3,4])
f.write(a) # file类 存储的是字符串
f.close()
```

In [521]:
```
f= open('file.txt', 'r')
b= f.read()
b
```

Out[521]:
```
'hello world\x01\x00\x00\x00\x00\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x03\x00\x00\x00\x00\x00
x00\x00\x00\x00\x00'
```

In [516]:
```
np.savetxt('array.txt',a,fmt='%.4f')
```

In [517]:
```
!cat array.txt
```

```
1.0000
2.0000
3.0000
4.0000
```

In [519]:
```
b= np.genfromtxt('array.txt')
print b
```

```
[ 1.  2.  3.  4.]
```

数据存储

Scipy

- Clustering package (**`scipy.cluster`**)
- Constants (**`scipy.constants`**)
- Discrete Fourier transforms (**`scipy.fftpack`**)
- Integration and ODEs (**`scipy.integrate`**)
- Interpolation (**`scipy.interpolate`**)
- Input and output (**`scipy.io`**)
- Linear algebra (**`scipy.linalg`**)
- Miscellaneous routines (**`scipy.misc`**)
- Multi-dimensional image processing (**`scipy.ndimage`**)
- Orthogonal distance regression (**`scipy.odr`**)
- Optimization and root finding (**`scipy.optimize`**)
- Signal processing (**`scipy.signal`**)
- Sparse matrices (**`scipy.sparse`**)
- Sparse linear algebra (**`scipy.sparse.linalg`**)
- Compressed Sparse Graph Routines (**`scipy.sparse.csgraph`**)
- Spatial algorithms and data structures (**`scipy.spatial`**)
- Special functions (**`scipy.special`**)
- Statistical functions (**`scipy.stats`**)
- Statistical functions for masked arrays (**`scipy.stats.mstats`**)
- C/C++ integration (**`scipy.weave`**)

Scipy

In [196]:
```
import scipy.integrate as inte
```

积分

In [206]:
```
def f(x):
    return np.exp(-x)* np.sin(x)
result, error = inte.quad( f,0,np.Inf) # quad(函数，积分下限，积分上限)
print '结果=',result, '误差=', error
```

```
结果= 0.5 误差= 1.48759119734e-08
```

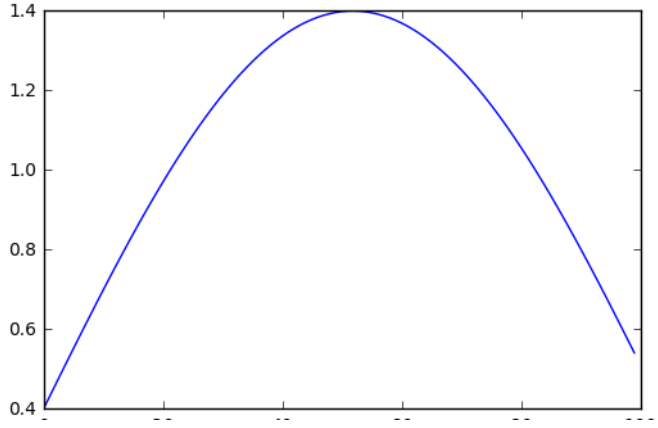
In [213]:
```
def ff(x,y):
    return np.exp(-x**2-y**2)
##  dblquad(函数，x下限，x上限，y下限，y上限)
### y下限和上限：要输入一个关于x的函数
result, error= inte.dblquad( ff, 0, np.Inf, lambda x:0, lambda x:np.Inf )
print '结果=',result, '误差=', error
```

```
结果= 0.785398163397 误差= 1.46476403803e-08
```

微分方程

In [252]:
```
def dy(y,x):
    dy1= np.cos(x)
    return dy1
y0= 0.4
xlist= np.linspace(0,3,100)
## 解微分方程 y'(x)= cos(x)
re=inte.odeint( dy, y0, xlist )
pylab.plot( re )
```

Out[252]:
```
[<matplotlib.lines.Line2D at 0x7fa506f12350>]
```

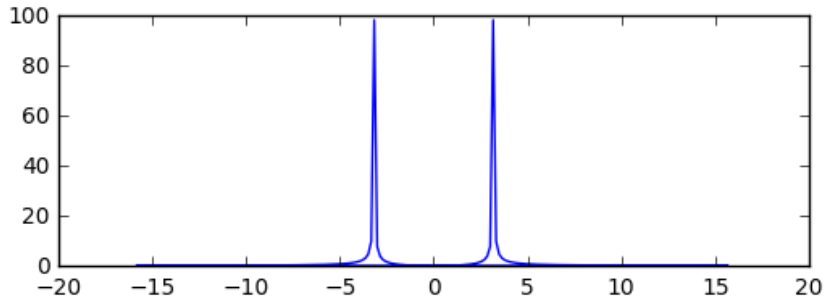


Scipy

In [253]:
```
import scipy.fftpack as fftpack
```

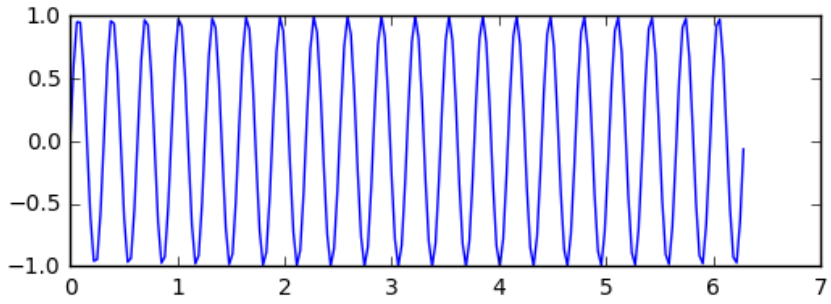
In [323]:
```
N=200
xlist= np.linspace(0,6.28,N)
dt= xlist[1]-xlist[0]
w= fftpack.fftfreq(N,dt)
ylist= np.sin(20*xlist) # 待变换函数
F= fftpack.fft(ylist)  # 快速离散傅立叶变换
pylab.subplots(figsize=(6,2))
pylab.plot( w,abs(F) )
```

Out[323]:
```
[<matplotlib.lines.Line2D at 0x7fa5057c1050>]
```



In [324]:
```
F_inv= fftpack.ifft(F)  # 反傅立叶变换
pylab.subplots(figsize=(6,2) )
pylab.plot(xlist,F_inv)
```

Out[324]:
```
[<matplotlib.lines.Line2D at 0x7fa505725610>]
```



快速傅立叶变换

线性代数

scipy.linalg是处理线性代数运算的

Solve equation $A\vec{x} = \vec{b}$

In [336]:
```
import scipy.linalg as lg
A= np.random.random([3,3])
b= np.random.random(3)
x=lg.solve(A,b) # 用函数solve求出x
print np.dot(A,x)-b # 验证 Ax-b 是否为0
```

```
[  0.00000000e+00   0.00000000e+00   1.11022302e-16]
```

本征值本征态

eig(A)返回矩阵A的本征值和本征向量，eigh(H)中矩阵H是厄米的矩阵

In [337]:
```
eva,eve= lg.eig(A) # 求A的本征值和本征向量
print eva
```

```
[ 1.62646085+0.j          0.17503666+0.19074463j  0.17503666-0.19074463j]
```

In [339]:
```
print A.dot(eve[:,0])- eva[0]*eve[:,0]
# 验证eva，eve是否为A的本征系统 A* x1 - v1* x1
```

```
[  6.66133815e-16+0.j   6.66133815e-16+0.j   4.44089210e-16+0.j]
```

SVD分解

svd(A) 返回 w, x, v, 满足A= w *diag(x)* v

In [340]:
```
w,x,v= lg.svd(A)
print w.dot( np.diag(x) ).dot(v)-A
```

```
[[  3.33066907e-16   2.63677968e-16   7.21644966e-16]
 [  5.55111512e-16   1.11022302e-16   3.88578059e-16]
 [  2.22044605e-16  -1.66533454e-16   2.22044605e-16]]
```

线性代数运算

Scipy

若矩阵 A 大多数元素为零，可以写成稀疏矩阵的形式以节省内存

- a.data: 一个列表，按顺序存 A 的所有元素
- a.indptr: 一个列表，a.indptr[i]= 前 i-1 行不为 0 的元素个数
- a.indices: 一个列表，存各个元素所在的列
- a.shape: 矩阵的维度，存为 (m,n)

[64]:
```
from scipy.sparse import csr_matrix
```

[66]:
```
a= csr_matrix([[1,2,0],[0,0,3],[4,0,5]])
print( a.data )
print( a.indptr )
print( a.indices )
```

```
[1 2 3 4 5]
[0 2 3 5]
[0 1 2 0 2]
```

[68]:
```
row= np.array([0,0,1,2,2])
col= np.array([0,1,2,0,2])
data= np.array([1,2,3,4,5])
mtx= csr_matrix((data,(row,col)), shape=(3,3))
print( mtx.todense() )
```

```
[[1 2 0]
 [0 0 3]
 [4 0 5]]
```

Scipy

[69]:
```
import scipy.sparse as spa
import scipy.sparse.linalg as splg
import time
```

[ ]:
```
a= spa.random(1000,1000,0.1) #定义一个1000*1000的稀疏矩阵，密度为0.1
start= time.time()
#求最小的本征值，可以定制arpack的迭代次数和精度
#很多时候，求最小本征值时，由于精度不够，得不到结果
splg.eigs(a,2, which='SM', maxiter=2000,tol=1E-2)
end= time.time()
print(f'The program use {end-start} seconds')
```

[90]:
```
a= spa.random(10000,10000,0.000001)
start= time.time()
x1=splg.eigs(a,10, which='LM')
end= time.time()
print(f'The program use {end-start} seconds')
```

```
The program use 0.007411003112792969 seconds
```

[91]:
```
start= time.time()
x2=splg.eigs(a,10, sigma=1, which='LM')
end= time.time()
print(f'The program use {end-start} seconds')
```

```
The program use 0.009476900100708008 seconds
```

Matplotlib

- 简单的画图函数库
- 能对图内所有元素进行定制
- 支持 Latex 公式
- 可以导出为多种格式: jpg, png, pdf, eps, ...

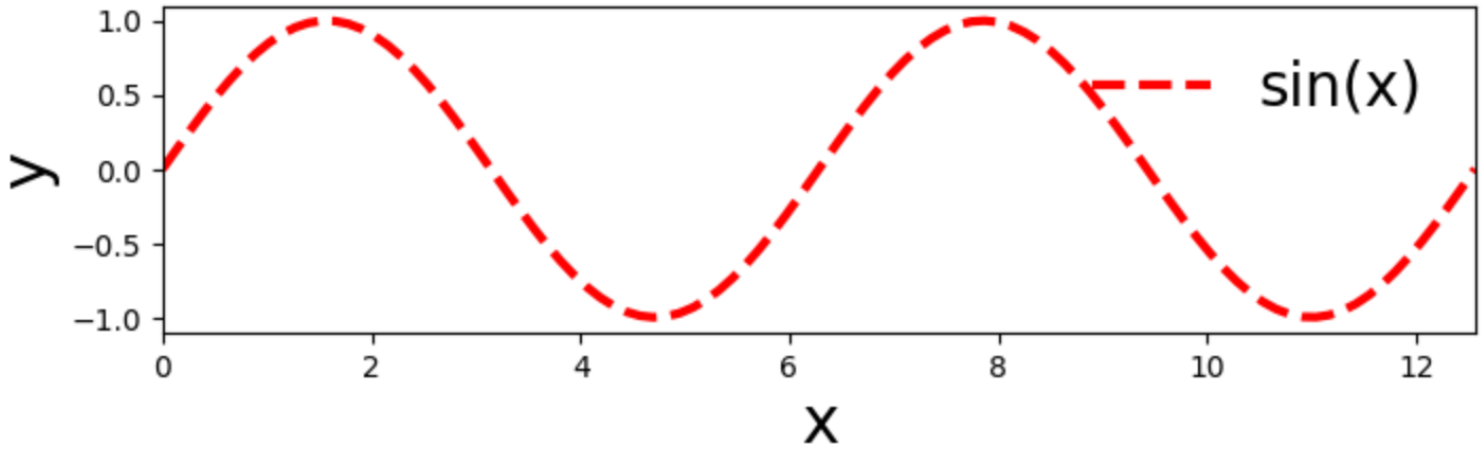
Matplotlib

[105]:
```
import matplotlib.pyplot as plt
```

[116]:
```
xlist= np.linspace(0,4*np.pi,100)
ylist= np.sin(xlist)
```

[125]:
```
fig,axe= plt.subplots(1,figsize=(8,2))
axe.plot(xlist,ylist,ls='--',c='r',lw=3,label=r'sin(x)')
axe.set_xlabel(r'x',fontsize=22)
axe.set_ylabel(r'y',fontsize=22)
axe.set_xbound(0,4*np.pi)
axe.legend(loc=0, fontsize=20, frameon=False)
```

[125]:
```
<matplotlib.legend.Legend at 0x7846f1a327e0>
```

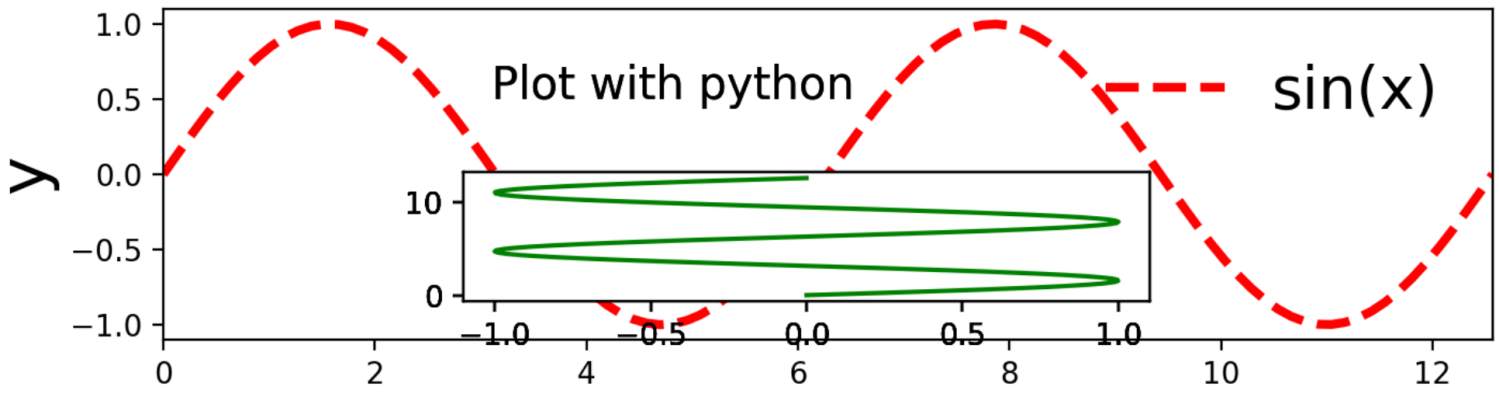


Matplotlib

[136]:
```
ax2=·fig.add_axes([0.3,0.2,0.4,0.3])
ax2.plot(ylist,xlist,'g')
axe.text(3.1,0.5,'Plot·with·python',·fontsize=15)
fig.savefig('1.png',dpi=200)
```

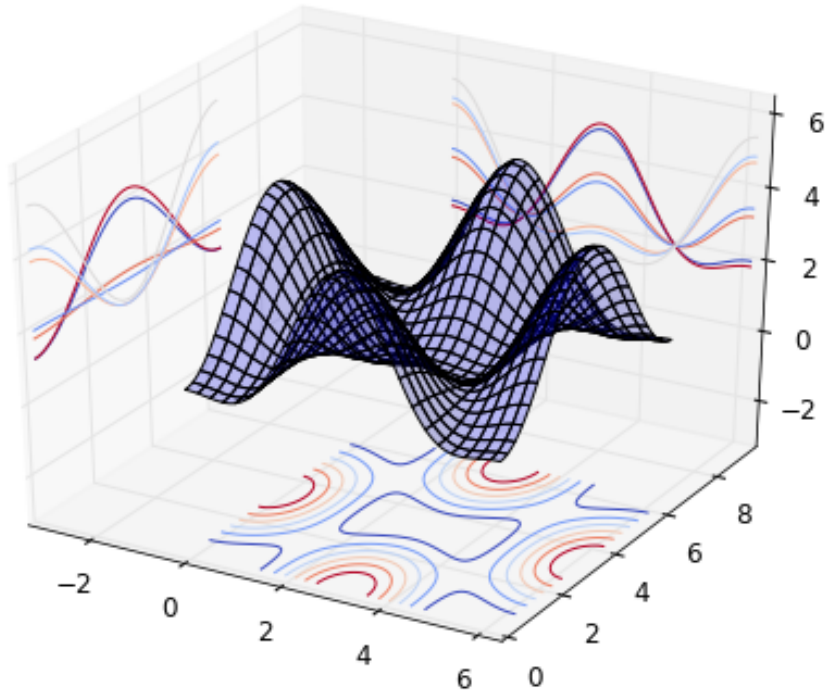
[137]:
```
from·IPython.display·import·Image
Image('1.png')
```

[137]:



Matplotlib

- pcolor(x,y,z), imshow, contour(x,y,z): 2D 图，登高线图 ...
- plot_surface(x,y,z), plot_wireframe(x,y,z): 3D 图



其他

- Sympy: 符号运算
- Qutip: Quantum toolbox in Python
- Kwant: numerical calculations on tight-binding models with a strong focus on quantum transport
- re：正则表达式
- scikit-learn：机器学习
- ...

Outline

多进程

```
from multiprocessing import Process
import os

# 子进程要执行的代码
def run_proc(name):
    print 'Run child process %s (%s)...' % (name, os.getpid())

if __name__=='__main__':
    print 'Parent process %s.' % os.getpid()
# 用Process启动一个进程
    p = Process(target=run_proc, args=('test',))
    print 'Process will start.'
    p.start()
# p.join()这里等待所有进程结束，再继续下面的命令
    p.join()
    print 'Process end.'
```

多进程

如果要启动大量的子进程，可用 Pool 实现

```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print 'Run task %s (%s)...' % (name, os.getpid())
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print 'Task %s runs %0.2f seconds.' % (name, (end - start))

if __name__=='__main__':
    print 'Parent process %s.' % os.getpid()
    p = Pool()
# 启动5个进程
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print 'Waiting for all subprocesses done...'
    p.close() # jion()之前必须调用close()
# 等待子进程结束
    p.join()
    print 'All subprocesses done.'
```

多进程

In [478]:
```
from multiprocessing import Pool
import os, time, random

def long_time_task(name):
    print 'Run task %s (%s)...' % (name, os.getpid())
    start = time.time()
    time.sleep(random.random() * 3)
    end = time.time()
    print 'Task %s runs %0.2f seconds.' % (name, (end - start))

if __name__=='__main__':
    print 'Parent process %s.' % os.getpid()
    p = Pool()
# 启动5个进程
    for i in range(5):
        p.apply_async(long_time_task, args=(i,))
    print 'Waiting for all subprocesses done...'
    p.close() # jion()之前必须调用close()
# 等待子进程结束
    p.join()
    print 'All subprocesses done.'
```

```
Parent process 6440.
Run task 0 (12028)...
Run task 1 (12029)...
Run task 2 (12030)...
Run task 3 (12031)...
Waiting for all subprocesses done...
Task 2 runs 0.73 seconds.
Run task 4 (12030)...
Task 0 runs 0.97 seconds.
Task 1 runs 2.31 seconds.
Task 3 runs 2.62 seconds.
Task 4 runs 2.49 seconds.
All subprocesses done.
```

多线程

- 多任务可由多进程完成，也可由多线程完成
- thread 和 threading 都可以处理多线程任务
- 任何进程都至少一个线程，称为主线程

```
import time, threading

# 新线程执行的代码:
def loop():
    print 'thread %s is running...' % threading.current_thread
    n = 0
    while n < 5:
        n = n + 1
        print 'thread %s >>> %s' % (threading.current_thread()
        time.sleep(1)
    print 'thread %s ended.' % threading.current_thread().name

print 'thread %s is running...' % threading.current_thread().n
t = threading.Thread(target=loop, name='LoopThread')
t.start()
t.join()
print 'thread %s ended.' % threading.current_thread().name
```

并行

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

Outline

其它

- 尽量不使用 from xxx import *，使用 import XXX as yyy 效率会高很多
- for、while 循环不要超过两层
- 数值计算尽量用 numpy, scipy 里面的函数

In [490]:
```
n= 300
A= np.random.random([n,n])
B= np.random.random([n,n])
start= time.clock()
C= np.dot(A,B)
end= time.clock()
print end - start
```

```
0.00934000000007
```

In [489]:
```
start= time.clock()
C= np.zeros([n,n])
for i in range(0,n):
    for j in range(0,n):
        for k in range(0,n):
            C[i,j]= C[i,j]+ A[i,k]* B[k,j]
end= time.clock()
print end - start
```

```
30.875038
```

参考

- doc.python.org
- numpy.org
- scipy.org
- qutip.org