Introduction to scientific computing

Yunkai Zhou

Department of Mathematics
Southern Methodist University
Dallas, Texas 75075

yzhou@smu.edu

Spring, 2017

Main contents of this course

- Introduction to Python
- Solving linear equations
- Curve fitting
- Numerical differentiation
- Numerical integration
- Solving ODEs (initial value problem)
- (optional) Solving ODEs (boundary value problem)

Basics elements of Python (1)

- Data types
  1. Primitive Data Types:
     - 1.1 Numeric type: `int, float, complex` (Python2 also has `long` int type)
     - 2.2 Strings
     - 3.3 Boolean: `True` or `False`
  2. Container data type: (Structured data type)
     - 2.1 Tuples (immutable)
     - 2.2 Lists (mutable)
     - 2.3 Dictionaries (mutable)
     - 2.4 Arrays, including vectors and matrices (mutable)
- Variables and assignments
- Input and output (from screen)
- Input and outputs (from files, i.e., file operations); paths of files

Basics elements of Python (2)

- Branching inside a program
  (conditional branching: comparisons and logical operations)
- Loops (Iterations)
  (The `for` loop and the `while` loop)
- Handling exceptions
- Functions:
  1. Definition;
  2. Scoping – namespace;
  3. Arguments:
     a) positional arguments;
     b) keyword arguments; default of a keyword argument
- Modules (import)

Variables and assignments (1)

Variable names in Python:

- Can contain alpha-numerical characters (a-z, A-Z, 0-9); it is case sensitive.
- Can contain some special characters, such as _.
- Variable names must start with a letter. (i.e., cannot start with a number)
- Python reserved keywords cannot be used as variable names. The keywords are: `and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield.`

Assignments are done with the $=$ sign.

```
>>> color="green"
>>> color
'green'
>>> a1= 100;  a1              #Note: the ';' is used to put >1 commands in one line
100
>>> name='blah';  name
'blah'
>>> type(name)
<type 'str'>
>>> type(color)
<type 'str'>
```

Note: Python is dynamically typed. (No need to specify the type when creating a variable.)

Variables and assignments (2)

In Python, one can assign values to multiple variables with a single "="

```
>>> x, y, z = 'moon', 'sun', 'stars'
>>> x
'moon'
>>> y
'sun'
>>> z
'stars'

### swapping values of two variables is easy
>>> x, y = y, x
>>> x
'sun'
>>> y
'moon'
```

Variables and assignments (2)

In Python, one can assign values to multiple variables with a single "="

```
>>> x, y, z = 'moon', 'sun', 'stars'
>>> x
'moon'
>>> y
'sun'
>>> z
'stars'

### swapping values of two variables is easy
>>> x, y = y, x
>>> x
'sun'
>>> y
'moon'
```

Assignment can be done together with arithmetic:

```
>>> a = 10
>>> a += 10; a       #a += b  same as  a = a+b
20
>>> a -= 4;  a       #a -= b  same as  a = a-b
16
>>> a *=10;  a       #a *= b  same as  a = a*b
160
>>> a /= 2;  a       #a /= b  same as  a = a/b
80.0
```

Primitive data type: Numeric and boolean

- Integers: $\pm 1, 0, \pm 2$, etc
- Floats: $\pm 1.0, 0.0, \pm 2.0, \pm 3.0001$, etc
- Complex numbers: $2 \pm 5j$, etc
- Booleans: `True` and `False`

The data type can be checked by the `type()` command. Examples:

```
>>> type(0)
<type 'int'>
>>> a=-1
>>> type(a)
<type 'int'>

>>> a + 10
9

>>> b=1.0
>>> type(b)
<type 'float'>

>>> c=2+3j
>>> type(c)
<type 'complex'>

>>> type(False)
<type 'bool'>
>>> type(1==0)
<type 'bool'>
```

Basic operations for numeric data types

```
>>> a=9;  b=2;  c=-5
>>> a/2     #automatically upgrade to float type in Python 3
4.5         #(in Python2 the value is kept as int type, thus 9/2=4)

>>> int(a/2)
4
>>> float(9/2)
4.5
>>> d = c % 2     #remainder
>>> d
1

>>> sqrt(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined

>>> import math      #import the math module
>>> math.sqrt(9)
3.0

>>> math.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error

>>> import cmath     #import the complex math module
>>> cmath.sqrt(-1)
1j
```

Basic operations for numeric data types

All numeric types (except some of them for complex types) support the following operations, sorted by ascending priority.

x + y	sum
x - y	difference
x * y	product
x / y	quotient
x // y	floored quotient
x % y	remainder of x/y
- y	y negated
abs(y)	absolute value or magnitude of y
int(y)	convert y to integer
float(y)	convert y to float
complex(x, y)	complex number with real part x, imaginary part y. y can default to 0, i.e., complex(x) produce (x + 0j)
c.conjugate()	conjugate of the complex number c
divmod(x, y)	the pair of (x // y, x % y)
pow(x, y), or x**y	x to the power y (note y can be any number in Python3)

Some examples of the basic operations

```
>>> divmod(10,3)
(3, 1)

>>> [10/3, 10//3, 10%3]
[3.3333333333333335, 3, 1]

>>> [ 2**10,  1024**(1/10),  pow(2, 5), pow(32, 1/5) ]
[1024, 2.0, 32, 2.0]

>>> [ (-1)**(1/2),  pow(-1, 0.5) ]    ##not allowed in Python2
[(6.123233995736766e-17+1j), (6.123233995736766e-17+1j)]

>>> [(-2)**1j,  9**3j]
[(0.033241827008856 65+0.027612020368333007j),(0.9527936007966461+0.3036187647049527j

>>> [10**3, 1e3,  10**(-2), 1e-2]
[1000, 1000.0, 0.01, 0.01]

>>> 10**10 == 1e10
True
>>> 10**10 == 1e+10
True
>>> 10**(-10) == 1e-10
True

>>> [type(10**3), type(1e3)]        #be careful: for integer n, 10**n is of int type,
[<class 'int'>, <class 'float'>]    #but 1e+n is of float type (so range(1, 10**3)
                                    #is fine,  while range(1, 1e+3) is not)
```

Basic operations for numeric data types

All real numbers (int, float) support the following operations:

round(x, n)	round x to n decimal digits. (if n is omitted, it defaults to 0)
math.floor(x)	the greatest integer $<=$ x
math.ceil(x)	the smallest integer $>=$ x
math.trunc(x)	truncate x to an integer by removing its decimal parts

```
>>> import math

>>> (math.ceil(9/2),  math.floor(9/2))     #this returns a tuple (later)
(5, 4)

>>> (math.ceil(-5/2),  math.floor(-5/2))
(-2, -3)

>>> (math.trunc(-2.1), math.trunc(-2.99),  math.trunc(5.01), math.trunc(5.9))
(-2, -2, 5, 5)

>>> x=-math.pi
>>> (round(x),  round(x, 2),  round(x,4), round(x, 8), round(x, 15), round(x, 16))
(-3, -3.14, -3.1416, -3.14159265, -3.141592653589793, -3.141592653589793)

>>> round(x/10, 1000) #so it is not 'exactly' to n decimal digits
-0.3141592653589793   #roughly, a float (0.????) has around 16 decimal points
```

Basic operations for Boolean data type

The `or, and, not` are the Boolean operations, ordered by ascending priority:

x `or` y	True if either x or y is True
x `and` y	True only if both x and y are True
`not` x	True only if x is False

Comparison operations: They have same priority, but higher priority than Boolean operations `(or, and, not)`.

$<$, $>$	strictly less, greater than
$<=$, $>=$	less, greater than
$==$	equal
$!=$	not equal
`is`	A `is` B is True if A and B are the same object
`is not`	A `is not` B is True if A and B are not the same object
`in`	True if in a sequence type (list, tuple, range)
`not in`	True if not in a sequence type

Basic operations for Boolean data type

Examples

```
>>> 1>0 or 1<0
True
>>> 1>0 and -1<=0
True
>>> not (1>0)
False

>>> 'A' < 'a'  and 'Z' < 'a'
True
>>> 'A' < 'Z'  and 'Z' < 'a' < 'z'
True

>>> 1 < 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() < str()

>>> str(1) < str(2) < str(9) < 'A'
True                                #it returns True if all comparisons are True

>>> str(9) < str(10)
False
```

Primitive data type: Strings

String is a variable type used to store any type of texts. E.g.,

```
>>> s='hello math3315 and cse3365'; print(s)
hello math3315 and cse3365

>>> print(s[1:5]);  print(s[0:5])
ello
hello

>>> print(s[:10]);  #from the very beginning (default start=0)
hello math

>>> print(s[-7:])   #to the very end (i.e., last 7)
cse3365

>>> print(s[::2])   #step=2  (from very beginning to very end)
hlomt31 n s36

>>> print(s[::-1])  #from very end to very beginning
5633esc dna 5133htam olleh

>>> print(s[::-2])
53ecda53ha le
```

Note: Default index starts with 0.

The ":" is for index slicing. The general format is `[start:end:step]`.

Default step is 1. Negative step means going backwards.

Quotes of a string

A string contains immutable sequences of Unicode code points. String can be created in a variety of (mostly equivalent) ways:

- Single quotes: 'hello'
- Double quotes: "hello"
- Triple quotes: "'hello'", " " "hello" " "

Triple quoted strings can span multiple lines (All whitespace will be included in the string literal. Convenient for in-code document.)

Different type of quotes provide can be useful for quote embedding.

```
>>> 'embed "double quotes" in single quotes'
'embed "double quotes" in single quotes'
>>> "embed 'single quotes' in double quotes"
"embed 'single quotes' in double quotes"
>>> '''embed 'single quotes' in triple quotes'''
"embed 'single quotes' in triple quotes"
>>> """embed 'single quotes' in triple double quotes"""
"embed 'single quotes' in triple double quotes"
>>> """embed "double quotes" in triple double quotes"""
'embed "double quotes" in triple double quotes'

>>> "embed "double quotes" in double quotes"
  File "<stdin>", line 1
    "embed "double quotes" in double quotes"
                ^
SyntaxError: invalid syntax
```

Basic string operations

```
>>> a='cat';  b="dog"
>>> a+b
'catdog'
>>> a+'.'+b
'cat.dog'
>>> 3*a
'catcatcat'
>>> 5*(a+'-'+b)
'cat-dogcat-dogcat-dogcat-dogcat-dog'
>>> 5*a+'-'+b
'catcatcatcatcat-dog'

>>> c=100
>>> a+c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> a+str(c)    #the mighty str() transform numeric to str type
'cat100'

>>> print("value = %f" % 2.0)        #can use C-style string formatting
value = 2.000000
>>> print("value = %.3f" % 2.0)
value = 2.000

>>> s2 = "value1 = %.2f. value2 = %d" % (3.1415926536,  2.71828)
>>> s2
'value1 = 3.14. value2 = 2'
```

Useful methods for a string object: `str()`, `len()` and `.join()`, `.split()`, `.count()`, `.upper()`, `.lower()`, `.title()`, `.strip()`, `.lstrip()`, `.rstrip()`, `.find()`, `.rfind()`, `.replace()`, `.zfill()`, ...

```
>>> s0 = "Python has long scary fangs!!!!!"
>>> len(s0)
32
>>> s0.upper()
'PYTHON HAS LONG SCARY FANGS!!!!!'
>>> s0.lower()
'python has long scary fangs!!!!!'

>>> s0.count('!!')
2
>>> s0.count('!')
5
>>> s0.find(' ')  #first position of ' '
6
>>> s0.rfind(' ') #last position of ' ' (or, find 1st from right)
21
>>> s0.replace(' ', '+')
'Python+has+long+scary+fangs!!!!!'

>>> s0.split(' ')
['Python', 'has', 'long', 'scary', 'fangs!!!!!']

>>> '_'.join(s0.split(' '))
'Python_has_long_scary_fangs!!!!!'
>>> '/'.join(s0)
'P/y/t/h/o/n/ /h/a/s/ /l/o/n/g/ /s/c/a/r/y/ /f/a/n/g/s/!/!/!/!/!'
```

Data type: Container types

1. Tuples: (immutable)
   defined by ( ); e.g., (1, 2, 'a', 'b', 'c')

2. Lists: (mutable)
   defined by [ ]; e.g., [1, 2, 'a', 'b', 'c']

3. Dictionaries: (mutable)
   defined by { }; e.g., {'1st':1, '2nd':2, '3rd':'a', "4th":'b', "last":'c'} ;
   essentially a dictionary is a list where each item has a name-tag, the syntax is {`key: value, ...`}, where the `key` should be immutable, such as a string, number, or tuple; while the `value` can be any data type.

Access of elements in a container type data:

- Although tuples, list, and dictionaries are defined using ( ), [ ], { } respectively, their elements are accessed using [`id`],
  - for a tuple or a list, the `id` is the position index (integers);
  - for a dictionary, the `id` is the key name (immutable data)

Data types: Tuples

Tuples are created using the syntax (..., ..., ...), the () may be omitted:

```
>>> T = ('cat', 'dog', "honey", 3315, 3365)

>>> type(T)
<type 'tuple'>

>>> T[0]
'cat'

>>> T[-1]
3365

>>> T[2]
'honey'

>>> T[2]="tea"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment


>>> tup = 'a', 'b', 'c+d', 'e ... f'     #note the () can be omitted
>>> tup
('a', 'b', 'c+d', 'e ... f')
>>> type(tup)
<type 'tuple'>
```

Data types: Lists

Lists can be created using the syntax [..., ..., ...] and other ways.
Each element of a list can be of any valid data type.

```
>>> L= [ 2, 'a', 1.0,  1-1j ]
>>> type(L)
<type 'list'>
>>> type(L[-1])
<type 'complex'>

>>> L[-1]= str(L[-1])       #list elements can be changed
>>> L
[2, 'a', 1.0, '(1-1j)']
>>> type(L[-1])
<type 'str'>                #L[-1] changed from a numeric type to a string

>>> L.append('today')       #new elements can be added to a list
>>> L
[2, 'a', 1.0, '(1-1j)', 'today']
>>> L[0:-1]
[2, 'a', 1.0, '(1-1j)']


>>> tup = ('a', 'b', 'c')
>>> list(tup)               #list() can be used to create a list from iterables
['a', 'b', 'c']
```

Basic List Properties and Operations

Adding, inserting, removing, or sorting elements from a list:

- `.append()`, `.pop()`, `.insert()`, `.remove()`, `.sort()`, `.extend()`; `del`

```
>>> L=[];  L.append('z'); L.append('XYZ'); L.append(123)
>>> L                    #note append() adds to the end
['z', 'XYZ', 123]

>>> L.insert(1, 'A')     #insert() can add to a given position in the list
>>> L
['z', 'A', 'XYZ', 123]
>>> L.insert(0, 'B');    L
['B', 'z', 'A', 'XYZ', 123]

>>> L.insert(-1, 'B'); L      #insert() is done right before a given position
['B', 'z', 'A', 'XYZ', 'B', 123]

>>> L.remove('B')     #.remove() only removes the 1st element with specified value
>>> L
['z', 'A', 'XYZ', 'B', 123]

>>> del L[3]; L          #del can delete elements in a specified positions
['z', 'A', 'XYZ', 123]
>>> del L[1:2]; L
['z', 123]
```

Other methods include `.clear()`, `.copy()`, `.count()`, `.reverse()`.

Nested Lists

- A list can be inhomogeneous and arbitrarily nested
- A nested list leads to the difference between shallow copy (clone) and deep copy (discussed later)

Examples:

```
>>> nested_list = [1, [2, [3, [4, [5]]]]]

>>> nested_list[1]
[2, [3, [4, [5]]]]

>>> nested_list[1][0]
2

>>> nested_list[1][1]
[3, [4, [5]]]

>>> nested_list[1][1][1]
[4, [5]]

>>> nested_list[1][1][1][1]
[5]
```

Data type: Range

Lists play an important role in Python. They are frequently used in loops and other flow control structures (discussed later).

A number of functions can be used to generate lists of various types, including the `list()` command, and the `range()` command with syntax `range(start, stop, step)`.

- The `start` defaults to 0 if not present;
- The `step` defaults to 1 if not present.
- The `stop` value is not included in the range! (Same as in index slicing)

```
>>> range(10)                    #using the default start=0 and step=1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]     #using the default step=1
>>> type(range(1, 10))
<type 'list'>      #a range in Python2 is a list, in Python3 it is a range type

>>> b = range(-5, 11, 2); b
[-5, -3, -1, 1, 3, 5, 7, 9]
>>> type(b)
<type 'list'>      #again this is Python2, used to show the items in the range

>>> range(10, 0, -2)
[10, 8, 6, 4, 2]
```

`range()` creates a sequence data type that can be conveniently iterated over.

Advantage of a range type

In Python3 a range type is no longer a list type. It has memory advantage over a regular list or tuple, especially for large (or very long) lists.
This is because a range type only needs to store three values `start, stop, step`, it can calculate individual items or subranges on the fly (i.e., as needed); while a regular list or tuple type needs to store all of its elements. A range data type may be considered as an implicit or dynamic list (of integers only).

The memory efficiency of a range type is gained by giving up the type of items it can hold: A range type can only hold integers.

```
>>> range(0, 5, 0.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range() integer step argument expected, got float.

>>> range(0.1, 5.1, 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range() integer end argument expected, got float.

>>> ran = range(20); ran
range(0, 20)            ##in Python3, range items are not stored, thus not shown
>>> type(ran)
<class 'range'>         ##in Python3, a range has its own type
>>> ran[1]
1
>>> ran[1]=10           ##in Python3, a range item cannot be modified
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object does not support item assignment
```

Shallow copy and deep copy of lists (1)

The concepts of a shallow copy and a deep copy, if not understood, may lead to tricky bugs.

```
>>> a = range(10);  a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> b = a     #this is not a real copy (neither shallow nor deep)
>>> b is a
True          #b and a are the same object (list)

>>> b[0]='AA';  b[-1]='BB';  b
['AA', 1, 2, 3, 4, 5, 6, 7, 8, 'BB']

>>> a         #changes in b change corresponding elements in a (vice versa)
['AA', 1, 2, 3, 4, 5, 6, 7, 8, 'BB']


>>> c = a[:]  #this makes a shallow copy of a and store in c
>>> c
['AA', 1, 2, 3, 4, 5, 6, 7, 8, 'BB']
>>> c is a
False          #now c and a are different objects (lists)

>>> c[2]='shallow'
>>> c
['AA', 1, 'shallow', 3, 4, 5, 6, 7, 8, 'BB']
>>> a          #changing c does not appear to affect a (and vice versa)
['AA', 1, 2, 3, 4, 5, 6, 7, 8, 'BB']
```

Shallow copy and deep copy of lists (2)

Shallow copy appears to work just fine from previous example, but it can be unsafe when applied to a nested list (i.e., a list with more than one level of list nesting).

The term *deep* means recursively going deeper into all lower levels.

```
>>> lista = [1, 2,  [4, 5, 6]]
>>> listb=lista[:]
>>> listb[0]=200; listb
[200, 2, [4, 5, 6]]
>>> lista
[1, 2, [4, 5, 6]]     #lista appears unaffected by 1-level change in listb

>>> listb[-1][0]=400; listb
[200, 2, [400, 5, 6]]
>>> lista     #lista is affected by 2-level change in listb (due to shallow copy)
[1, 2, [400, 5, 6]]


>>> import copy
>>> listb=copy.deepcopy(lista);  listb
[1, 2, [400, 5, 6]]
>>> listb[-1][0]='$';  listb
[1, 2, ['$', 5, 6]]
>>> lista          #with deepcopy(), lista is isolated from any change in listb
[1, 2, [400, 5, 6]]
```

Various ways to copy a list

There are different ways to make a real copy (shallow or deep) of a list. They may have different efficiency.

(Check back to see this slide only after you become fairly familiar with the Python basics)

```
>>> mylist=[range(1000),  [range(100), 500*'A-to-Z']]  #create a test list

>>> slice_cp = mylist[:]                    #get a shallow copy via index slicing

>>> list_cp  = list(mylist)                 #get a shallow copy via list()

>>> pcp_cp   = mylist.copy()                #.copy() for a list obj is a shallow copy

>>> import copy
>>> ccp_cp   = copy.copy(mylist)
>>> dcp_cp   = copy.deepcopy(mylist)        #deepcopy can be slow for huge/deep list

>>> ext_cp = []; ext_cp.extend(mylist)      #shallow copy by .extend()

>>> app_cp = [];                            #shallow copy by .append()
>>> for x in mylist: app_cp.append(x)       #direct app_cp.append(mylist) is incorrect

>>> dl_cp = [ x for x in mylist]            #shallow copy via [ ] and loop

>>> slice_cp==list_cp==pcp_cp==ccp_cp==dcp_cp==ext_cp==app_cp==dl_cp==mylist
True                                        #they are all the 'same'
```

Data type: Dictionaries

- Dictionaries are like lists, except that each element is a pair of {`key : value`}, where `key` should be immutable, while `value` is mutable.
- The syntax for dictionaries is {key1:value1, ...,}.
- Main operations on a dictionary are (1) storing a value with some key; and (2) extracting the value given the key.
- To delete a key:value pair, use `del`.
- When a key is already in use and a new value associated with the key is added, then the old value associated with that key is forgotten.

Methods defined for a dictionary object (denote as D here)

1. D.`copy()`: a shallow copy of D
2. D.`keys()`: a set-like object containing D's keys
3. D.`values()`: an object containing D's values
4. D.`has_key(k)`: True if D has a key named k, False otherwise.
5. D.`pop(k)`: Remove specified key and return the corresponding value.
6. many others ...

Data type: Dictionaries

```
>>> ds={}    #this constructs an empty dict
>>> ds['A']=1;  ds['B']=2; ds['C']=3; ds  #create a dict by elementwise assignment
{'B': 2, 'C': 3, 'A': 1}

>>> ds2=dict(A=1, B=2, C=3)   #use keyword arguments of dict() to create a dict
>>> ds2
{'A': 1, 'C': 3, 'B': 2}      #dictionaries are unordered

>>> ds3=dict([('A', 1), ('B',2), ('C',3)]) #use dict() to make a list into a dict

>>> ds4=dict( zip(['A','B','C'], [1,2,3]) )  #use the zip() function

>>> ds2 == ds3 == ds4 == ds
True

>>> ds['A']='apple';  ds
{'B': 2, 'C': 3, 'A': 'apple'}  #same key 'A', now it holds a different value

>>> del ds['C'];  ds            #delete using key name as identifier
{'B': 2, 'A': 'apple'}

>>> ds4.keys()
dict_keys(['A', 'C', 'B'])
>>> ds4.values()
dict_values([1, 3, 2])
>>> ds4.pop('C')
3
>>> ds4
{'A': 1, 'B': 2}
```

Data type: Sets

Sets: (mutable, elements without duplication, and unordered)
defined by { }; e.g., {1, 2, 'a', 'b', 'c'} ;
A set contains unordered collections **without duplicates**.

```
>>> s={1, 1, 1, 2, 2, 3, 3, 3, 4, 5, 5, 5};  s
{1, 2, 3, 4, 5}

>>> s2=set((1, 1, 1, 2, 2, 3, 3, 3, 4, 5, 5, 5)); s2
{1, 2, 3, 4, 5}

>>> s==s2
True
>>> type(s)
<class 'set'>

>>> s is s2
False

>>> s[1]    #set is unordered, elements cannot be accessed via position index
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing

>>> s -= {1}; s
{2, 3, 4, 5}
>>> s |= {('a', 'b')}; s
{2, 3, 4, 5, ('a', 'b')}
```

Data type: Sets and frozensets

```
>>> s1 = set('python eats pigs')
>>> s1
{'o', 'e', 'n', ' ', 'h', 'y', 'a', 's', 'i', 'p', 't', 'g'}

>>> s1.remove('o')     #sets are mutable
>>> s1
{'e', 'n', ' ', 'h', 'y', 'a', 's', 'i', 'p', 't', 'g'}
>>> s1.add('run')
>>> s1
{'e', 'n', 'run', ' ', 'h', 'y', 'a', 's', 'i', 'p', 't', 'g'}

>>> s1.add(['aa',1])   #set's element cannot be mutable (such as a list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

>>> s1.add(('aa',1))   #an immutable can be a set element
>>> s1
{('aa', 1), 'n', 'run', ' ', 'h', 'y', 'a', 's', 'i', 'p', 't', 'g'}

>>> cities = frozenset(['dallas', 'houston', 'austin']); cities
frozenset({'austin', 'dallas', 'houston'})
>>> type(cities)
<class 'frozenset'>

>>> cities.add('LA')    #frozensets are immutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

The `len()` function

The `len()` is a built-in function that returns the length (number of items) of a sequence or collection. The argument is an object, which may be a sequence (such as a string, bytes, tuple, list, or range), or a collection (such as a dictionary, set, or frozen set).

```
>>> str1 = "python eats a lot"
>>> len(str1)
17

>>> set1 = set(str1); set1
{'o', 'e', 'n', 'l', ' ', 'h', 'y', 'a', 's', 'p', 't'}
>>> len(set1)
11

>>> rg1 = range(20)
>>> len(rg1)
20

>>> cities = frozenset(['dallas', 'houston', 'austin', 'waco']);
>>> len(cities)
4

>>> L = list(cities)
>>> L[0 : len(L)]
['austin', 'dallas', 'waco', 'houston']
```

Conditional branching

Main structure:

```
if condition:
   do something
else:
   do something_else
```

```
if cond1:
   do thing1
elif cond2:
   do thing2
else:
   do something_else
```

Conditional blocks can be nested, for example

```
if cond1:
   if cond2:      #cond1 and cond2
       do something
   else:          #cond1 and not cond2
       do something_else
else:
   if cond3:      #not cond1 and cond3
       do something
   else:          #not cond1 and not cond3
       do something_else
```

```
if condA:
   do thingA
elif condB:
   do thingB
elif condC:
   do thingC
elif condD:
   do thingD
else:
   do something_else
```

Python code block has no explicit begin or end or no curly braces to mark where the block starts and ends.

The only delimiter is a colon (:), and the indentation of the code itself!

```
#assign the larger value of two floats x and y to z
if x < y:
   z = y
else:
   z = x

#assign the smallest value of x, y, z to m
if x <= y:
  if x<= z:
     m = x
  else:
     m = z
else:
  if z<= y:
     m = z
  else:
     m = y

#assign the smallest value of x, y, z to m (another ways)
if x <= y:                          |    if x<=y and x<=z:
  if x<= z:                         |       m = x
     m = x                          |    elif y<=z and y<=x:
  else:                             |       m = y
     m = z                          |    else:
elif z<= y:                         |       m = z
     m = z                          |
else:                               |
     m = y                          |
```

The `for` loop

One of the most important functionalities in any programming language is iteration, i.e., repeated application of some operation to (likely updated) data.

In Python an iteration is done by a `for` loop or a `while` loop.

In a `for` loop, an iteration variable (automatically) steps through a given iterable (a sequence type data, such as a list, range, tuple, string), each iteration performs some task. The syntax of a `for` loop is

```
for var in iterable:
    do something
```

```
# compute 1 + 2 + 3 +... + 10000  and 1^2 + 2^2 + 3^2 + ... + 10000^2
s1= 0;  s2= 0
for i in range(1,10001):
    s1 = s1 + i
    s2 = s2 + i**2

# compute the inner product of two same length vectors v1, v2
dotprod = 0
for i in range(len(v1)):
    dotprod += v1[i]*v2[i]

# compute the series sin(x)+ sin(x**2)+ sin(x**3)+...+ sin(x**n), n is an integer
import math
ssum = math.sin(x)
for i in range(2,n+1):
    ssum += math.sin(x**i)
```

The `for` loop without indexes

In Python, a `for` loop need not iterate over integer indexes. It can also iterate over items of an iterable (e.g., sequential type data like lists, tuples, strings; or mapping type data like dictionaries.)

```
# iterate over a string:
# e.g., assume a string s is given, construct a list with len(s) elements,
# each element should be a char in s augmented by '__'
slst = []
for c in s:       #see how easily it iterates over each char in a string
    slst.append(c+'__')

# iterate over a list:
# e.g., create a list that contains only numeric items of another list
L =[ 'city', 10, 'map', 3.14, '%%', -2.71 ]
numL = []
for item in L:
   if type(item)==int or type(item)==float:
      numL.append(item)

# iterating over a dictionary: it defaults to iterating over the keys
dic=dict([('A', 10), ('B', 20),  ('C', 30), ('D', 40)])
for k in dic:      #this is the same as for k in dic.keys():
   print(k)        #what do you expect this print will produce?
   print(dic[k])

for v in dic.values(): #can be made to iterate over the values
   print(v)        #what do you expect this print will produce?
```

The `while` loop

A `while` loop needs an iteration condition to be met in order to perform the tasks inside a loop. The syntax is

```
while loop_condition:
    do something
    update the looping variable (or break)
```

```
# compute 1 + 2 + 3 +... + 10000  and 1^2 + 2^2 + 3^2 + ... + 10000^2
s1= 0;  s2= 0;  i= 1
while i <= 10000:
    s1 = s1 + i
    s2 = s2 + i**2
    i  = i+1

# compute the inner product of two same length vectors v1, v2
dotprod = 0;  i= 0
while i < len(v1):
    dotprod += v1[i]*v2[i];
    i = i+1

# construct a list with len(s) elements, each element is a char in a
# given string s, each char should be augmented by '__'
slst = [];  i= 0
while i < len(s):  #assume s is already given somewhere
    slst.append(s[i]+'__');
    i = i+1
```

Break out a Loop; skip part of a loop

The keyword `break` can be used to break out of a loop (if loops are nested, if breaks out of the most interior loop that encloses the `break` command).

```
#find the 1st three numbers in range(1,90000) that can be divided by 11, 15, 19
num = 0;  find=[]
for i in range(1,90000):
    if i%11==0 and i%15==0 and i%19==0:
       find.append(i);
       num += 1
    if num == 3:
       break

#request user input until user does what is requested
while True:
    number = input('Input a number within [1, 100]: ')
    if 1<= number <= 100:  break
    print('input out of bounds [1, 100], try again')
```

The keyword `continue` can be used to skip the remaining of the current iteration (and continue from the start of the next iteration).

```
#print all leap years in years 2000 to 3000
for i in range(2000, 3001):
   if i%4 != 0 or (i%100 == 0 and i%400 != 0):
      continue  #this skips the rest
   print('Year ', i, ' is a leap year')
```

'Equivalence' of `while` and `for` loop

- A `for` loop is more convenient when you know how many steps to perform the iteration (i.e., iteration bounds known).
- A `for` loop can always be written as a `while` loop, e.g.,

```
sumsq = 0
for i in range(1,1000):
    sumsq += i**2
```

```
sumsq = 0;  i=1
while i < 1000:
    sumsq += i**2
    i += 1
```

- When the iteration bounds are not known a-priori, a `while` loop may be more convenient. (You could use an 'infinite' `for` loop and break out when the condition is met.)
- A programming language with `while` loops and conditionals is Turing-complete. (A Turing-complete programming language is theoretically capable of expressing all tasks that can be accomplished by computers.) In other words, so far we have learned the most essential Python functionalities that can be used to solve fairly complicated problems.

Avoid a (potentially serious) pitfall in a loop

Rule of thumb: Do not change inside the loop the variables that are used to decide the loop-stopping bounds!
(May lead to very unexpected results/bugs that can be hard to catch.)

```
>>> m = 3
>>> for i in range(m):
...     print('outer i=', i, ' m=', m)
...     for j in range(m):
...         m = m-1
...         print('inner i=',i, ' j=', j, ' m=', m)
...

outer i= 0  m= 3
inner i= 0  j= 0  m= 2
inner i= 0  j= 1  m= 1
inner i= 0  j= 2  m= 0
outer i= 1  m= 0        #note: all inner loops were skipped due to modified m
outer i= 2  m= 0
```

```
>>> def removeDups(L1, L2):
...     '''remove from L1 any items that are also in L2  (buggy)'''
...     for e in L1:
...         if e in L2:
...           L1.remove(e)

>>> L1=['a','b','c','d'];  L2=['a','b', 1, 'd']
>>> removeDups(L1,L2); print('L1=', L1)
L1= ['b', 'c']          #note that 'b' is not removed as desired (a tricky bug!)
```

Functions (why use functions?)

Many sound reasons for using functions!

- A function can implement a set of often used operations so they can be utilized more than once. This makes reusing coding much easier.
- Using functions makes debugging easier, a programmer can focus on fully debugging one single function, then apply it elsewhere (with confidence that it is fully debugged).
- Using functions usually enhances the comprehensibility of your code (which usually results in better code quality).
- Using functions makes code development relatively independent (each function may be considered as a black box: given an interface that is agreed on, a developer can focus on how to implement the function; while the other developers can move on calling the function assuming that the function has been developed with its expected outputs).
- ...

In other languages, functions are also called *subroutines (Fortran), procedures (Pascal), subprograms, methods, ...*

Functions (definition)

A Python function is defined by the keyword `def`, followed by a function name and a parentheses `( )`, and a colon `:`. The syntax is

```
def function_name(list_of_arguments/variables):
    function body
    return some values (default to return None if nothing is explicitly returned)
```

Function arguments should be placed inside the parentheses `( )`.
The function body should be indented accordingly. For examples,

```
def maxVal(a, b):
    '''return the max of a and b'''
    if a >= b:
        return a
    else;
        return b
```

```
def sumpow(n, pow):
    '''return the sum of each integer
       to its power—pow from 1 to n'''
    s = 0
    for i in range(1,n+1):
        s += i**pow
    return s
```

```
>>> maxVal(10, 1)
10
>>> maxVal(-500, -100)
-100
>>> sumpow(100,1)
5050
>>> sumpow(100,2)
338350
```

Functions (positional arguments)

A common usage of functions is to compute one or more results by applying some 'functions' to given variables. The values of the variables (parameters) are passed to the function via arguments.
Every argument can be made a keyword argument, if no keywords are specified, they become positional arguments and are identified by their position in the list of arguments.
Arguments can become optional (in this case they are keyword arguments and default values are used). The optional arguments must follow the mandatory ones if they exist.

```
def test_position_vars(p1, p2, p3):
    print('p1=', p1)
    print('p2=', p2)
    print('p3=', p3)

### driver code ###
>>> A=1; B=2; C=3
>>> test_position_vars(B,C,A)
p1= 2
p2= 3
p3= 1
>>> test_position_vars(B*C, C+A, C**B)
p1= 6
p2= 4
p3= 9
```

Functions (keyword arguments)

Keyword arguments are very useful:

- They make it less vital to remember the positions of arguments – especially when there are a long list of arguments.
- They can be assigned default values at implementation, such that a function can be called with (much) fewer arguments than its definition.
- They also can make code easier to read (and maintain) with well-chosen keywords.

```
## note: keyword arguments need to follow positional arguments (why?)
def  pos_keywrod_vars(p1, p2, pow1=1, pow2=2):
   print(['p1=', p1, 'p2=', p2, 'pow1=', pow1, 'pow2=', pow2])
   return p1**pow1 + p2**pow2

#### driver code ####
>>> pos_keywrod_vars(10, 20)
['p1=', 10, 'p2=', 20, 'pow1=', 1, 'pow2=', 2]
410
>>> pos_keywrod_vars(10, 20, pow2=0)
['p1=', 10, 'p2=', 20, 'pow1=', 1, 'pow2=', 0]
11
>>> pos_keywrod_vars(10, 20, pow1=0)
['p1=', 10, 'p2=', 20, 'pow1=', 0, 'pow2=', 2]
401
>>> pos_keywrod_vars(10, 20, pow2=1, pow1=2)
['p1=', 10, 'p2=', 20, 'pow1=', 2, 'pow2=', 1]
120
>>> pos_keywrod_vars(2, 1, pow2=0, pow1=10)
['p1=', 2, 'p2=', 1, 'pow1=', 10, 'pow2=', 0]
1025
```

Another keyword argument example

```
## the following can be called without arguments
def yes_or_no(retries=5, reminder='Try again!'):
    """return True if user input 'y', 'ye', 'yes';
       return False if user input 'n', 'no', 'nop', 'nope';
    """
    while True:
        ok = input("Input yes/no:  ").lower()
        if ok in ('y', 'ye', 'yes'):  return True
        if ok in ('n', 'no', 'nop', 'nope'):  return False
        retries = retries - 1
        if retries <= 0:
            raise ValueError('invalid user response')
        print(str(retries)+ ' retries left. '+ reminder)
```

```
>>> yes_or_no()
Input yes/no:  'q'
4 retries left. Try again!
Input yes/no:  'ye'
True
```

```
>>> yes_or_no(reminder="hurry up")
Input yes/no:  'o'
4 retries left. hurry up
Input yes/no:  'no'
False
```

```
>>> yes_or_no(retries=10, reminder="don't give up")
Input yes/no:  'p'
9 retries left. don't give up
Input yes/no:  'Yes'
True
```

Default value of a keyword argument

Warning on a potentially tricky issue: The default value of a keyword argument is evaluated only once during repeated call to the function. Thus, if the default is a mutable object such as a list or dictionary, then unintended values may be accumulated in the default (which leads to a different default from what you initially intended).

```
#this example shows the accidental change of the 'default'
def dbug(a, L=[]):
    L.append(a)
    return L

###driver code
>>> dbug(10)
[10]
>>> dbug(100)       #if L=[] is the default, the return should be just [100]
[10, 100]
>>> dbug(1000)
[10, 100, 1000]
>>> L2=dbug(1)      #assigning the returned value to a new variable
>>> L2              #does not help this buggy issue
[10, 100, 1000, 1]
>>> L3=dbug('a')
>>> L3
[10, 100, 1000, 1, 'a']
```

Approach to avoid such issue: Do not write such weird code, practice *defensive coding*, including not using mutable if an immutable will do, not changing loop bounds inside a loop, etc.

Functions (scope of variables)

There are two type of variables:

1. Local: Variable created inside a function is local to that function and inaccessible outside of the function.
2. Global: Variable defined in the driver (the one used to call functions etc) of a code are global variables. They are accessible by functions they call (thus they can have unintended consequences.)

One can define global variables inside a function using the `global` keyword, but this practice is not recommended. (It makes code lose locality, thus harder to debug and maintain.)

```
#these are in the script file named scope.py
def scope_test():
    A=100;  B=200
    print('local A=', A)
    print('local B=', B)
    global C
    print('global C=', C)
    C=300
    print('global D=', D)

####driver code #####
A=1;  B=2;  C=3;  D=4
print('A=%d, B=%d, C=%d, D=%d' % (A,B,C,D))
scope_test()
print('A=%d, B=%d, C=%d, D=%d' % (A,B,C,D))
```

Here is the output executing the scope.py:

```
>>> exec(open("scope.py").read())
A=1, B=2, C=3, D=4
local A= 100
local B= 200
global C= 3
global D= 4
A=1, B=2, C=300, D=4
```

Functions (data communication)

- A function can return multiple values; in that case, the order matters at the receiving end.
- Mutable Data (such as lists) passed as function arguments may be modified (*call-by-reference*), but be very careful about this!
  - *Call-by-reference*: The function modifies the value in the same *reference* of the argument, but does not create a local variable using the same name.
  - *Call-by-value*: If a local variable is created using the same name as the mutable argument, then the associated change with this local variable will not be returned to the caller. (Any earlier changes made via *call-by-reference* to the argument will still be returned to the caller.)

```
def data_communicate(list1, list2):
   list1 =[list1, list1] #list1 is call-by-value since a local list1 is created
   list2.pop()    #list2 is call-by-reference since no local list2 is created
   return  len(list1),  (list1, list2),  ['a','b','c','d']

##driver/caller part
L1=['a', 1, 2, 3];  L2=[10,  50, 'BB', 'EE'];

a, tup1, L3 = data_communicate(L1, L2)  #observe what values are stored
print(a)                                #in the receiving a, tup1, L3
print(tup1)
print(L3)

print(L1)  #note L1 is not modified
print(L2)  #note L2 is modified
```

Function call: Input and Output

A critical concept on function definition and its usage:

- A function may perform some tasks without returning a value;
- The far more common cases are that a function performs some tasks and return some related values that are useful. In this case, in order to reuse the returned values, one needs a "receiver" to store the returned values from a function;
- Depending on how many items a function returns, the "receiver" would faithfully receives what are returned. One needs to be careful about matches of the returned values with the receiver.

```
def  poly(x, c0, c1, c2):
    return c0, c0+c1*x, c0+c1*x+c2*x**2


>>> p = poly(2, 1, 1, 1)
>>> p
(1, 3, 7)
>>> p[0], p[1], p[2]
(1, 3, 7)

>>> a0, a1, a2 = poly(2, 1, 1, 1)
>>> ( a0, a1, a2 )
(1, 3, 7)
```

List comprehension 1

List comprehension refers to a concise way to create a list by enclosing a `for` loop inside a list constructor sign `[ ]`. The simplest syntax is

`[(an expression of var)  for var in iterable]`,

where the `iterable` is often a sequence type data (e.g., list, range, tuple, string) or a dictionary. It returns a list whose items are the result of sequentially applying the `expression` (e.g., a function) to the items in the `iterable`.

```
>>> [ t*2 for t in "Python3" ]
['PP', 'yy', 'tt', 'hh', 'oo', 'nn', '33']

>>> [s*0.5 for s in range(1, 11)]  #in Python, 0.5*range(1, 11) is not allowed
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]


>>> [ key*5  for key in {'A':1, 'B':2,  'C':3} ]
['AAAAA', 'CCCCC', 'BBBBB']

>>> [(x,y) for x in range(-1,2) for y in range(8,11)]
[(-1, 8), (-1, 9), (-1, 10), (0, 8), (0, 9), (0, 10), (1, 8), (1, 9), (1, 10)]

>>> [(x,y) for x in range(-1,2) for y in range(8,11) if x**2+y**2 < 100]
[(-1, 8), (-1, 9), (0, 8), (0, 9), (1, 8), (1, 9)]
```

List comprehension 2

The list comprehension

```
[(an expression of var)   for var in iterable if conditionals]
```

essentially is a concise way to express a loop:

```
newlist=[]
for var in iterable:
    if conditionals:
        newlist.append(expression applied to var)
```

```
# here is a loop for the example:  [s**2  for s in range(-5,5) ]
nl=[];
for s in range(-5,5):
    nl.append(s**2)

# here is a loop for the example:
# [(x,y) for x in range(-1,2) for y in range(8,11) if x**2+y**2 < 100]
nlist=[];
for x in range(-1,2):
    for y in range(8,11):
        if x**2 + y**2 < 100:
            nlist.append((x,y))
```

There can be more than one loop/condition in the list comprehension, for e.g.,

```
>>> X=range(1,20);
>>> [(x,y,z) for x in X for y in X for z in X if x**2+y**2==z**2]
[(3, 4, 5), (4, 3, 5), (5, 12, 13), (6, 8, 10), (8, 6, 10), (8, 15, 17),
 (9, 12, 15), (12, 5, 13), (12, 9, 15), (15, 8, 17)]

>>> [(x,y,z) for x in X for y in X for z in X if x**2+y**2==z**2 and x<=y<=z]
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15)]

>>> nlst=[];
>>> for x in X:
...     for y in X:
...         for z in X:
...             if x**2+y**2==z**2 and x<=y<=z:
...                 nlst.append((x,y,z))
...
>>> nlst
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15)]

#can reduce # of iterations if one can adopt conditional into loop bounds
>>>[(x,y,z) for x in range(1,20) for y in range(x,20) for z in range(y,20)
    if x**2+y**2==z**2]
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15)]

>>> nlst=[]; n=20
>>> for x in range(1,n):
...     for y in range(x,n):
...         for z in range(y,n):
...             if x**2+y**2==z**2:
...                 nlst.append((x,y,z))
...
>>> nlst
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15)]
```

The lambda function

The lambda keyword can be used to create anonymous functions (these functions need not be named, they are usually simple and can be written in one line).

```
#define a function f(x)= square of x
f = lambda x : x**2

#this f defined above is equivalent to
def f(x):
    return x**2
```

```
>>> f = lambda t : t**2

>>> g = lambda y : y**3

>>> for i in range(-3, 3): print([i, f(i), g(i)])
...
[-3, 9, -27]
[-2, 4, -8]
[-1, 1, -1]
[0, 0, 0]
[1, 1, 1]
[2, 4, 8]
```

lambda function can conveniently combine with the map(), filter() functions.

The `map()` and the `filter()` functions

The syntax of `map()` is `map(function, iterable, ...)`, it applies the `function` to every item of the `iterable` and returns an object consisting the results. (Can also be done via list comprehension.)

```
>>>map( (lambda x: x **2),  (-1, 2, 3, 9))
<map object at 0x2b374964fda0>

>>>list( map( lambda x: x**2, (-1,2,3,9)) )  #i.e. [x**2 for x in (-1,2,3,9)]
[1, 4, 9, 81]

>>>list( map( lambda x: x**3, [-1,2,3,9]) )  #i.e. [x**3 for x in [-1,2,3,9]]
[-1, 8, 27, 729]

>>>list(map( lambda x: x*3,  "python") ) #same as [ x*3 for x in "python" ]
['ppp', 'yyy', 'ttt', 'hhh', 'ooo', 'nnn']
```

The `filter()` returns an object containing only elements in the sequence that passes the filter (i.e., if the function returns True).

```
>>> alist = [-8, -6, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list(filter( lambda x: x >0,  alist) )
[1, 2, 3, 4]
>>> list(filter( lambda x: x % 2 ==0,  alist) )
[-8, -6, -2, 0, 2, 4]
>>> list(map( lambda x: x % 2 ==0,  alist) )  #'map' compared with 'filter'
[True, True, False, True, False, True, False, True, False, True]
```

Dictionary and set comprehensions

```
#dictionary comprehension
>>> {i*2:j*5 for i, j in {1:'a', 2:'b', 3:'c', 4:'d', 5:'e'}.items()}
{8: 'ddddd', 2: 'aaaaa', 4: 'bbbbb', 10: 'eeeee', 6: 'ccccc'}

>>> {i*2:i*5 for i in {1: 'a', 2: 'b', 3:'c', 4:'d', 5:'e'}.keys()}
{8: 20, 2: 5, 4: 10, 10: 25, 6: 15}

>>> {i*2:i*5 for i in {1: 'a', 2: 'b', 3:'c', 4:'d', 5:'e'}.values()}
{'aa': 'aaaaa', 'cc': 'ccccc', 'dd': 'ddddd', 'ee': 'eeeee', 'bb': 'bbbbb'}

>>> {i*2:i**2 for i in range(10)}
{0: 0, 2: 1, 4: 4, 6: 9, 8: 16, 10: 25, 12: 36, 14: 49, 16: 64, 18: 81}

>>> {i:(i**2, i**3) for i in range(5)}
{0: (0, 0), 1: (1, 1), 2: (4, 8), 3: (9, 27), 4: (16, 64)}


#set comprehension
>>> {x**2 for x in range(11)}
set([0, 1, 4, 100, 81, 64, 9, 16, 49, 25, 36])

>>> {x**2 for x in range(-10,11)}
set([64, 1, 100, 0, 49, 9, 16, 81, 25, 4, 36]) #duplications are removed

#output the prime numbers in a given range
>>> {x for x in range(2, 70) if all(x%k!=0 for k in range(2, x))}
set([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67])
```

Tuple comprehension

Requires the `tuple()` generator keyword:

```
#tuple comprehension
>>> tuple(i**3 for i in range(-10,5))
(-1000, -729, -512, -343, -216, -125, -64, -27, -8, -1, 0, 1, 8, 27, 64)

>>> tuple(s*3 for s in {1: 'a', 2: 'b', 3:'c', 4:'d', 5:'e'}.keys() )
(3, 6, 9, 12, 15)

>>> tuple(s*3 for s in {1: 'a', 2: 'b', 3:'c', 4:'d', 5:'e'}.values() )
('aaa', 'bbb', 'ccc', 'ddd', 'eee')

>>> tuple((i, i**3) for i in range(-10,10) if i%2==0)
((-10, -1000), (-8, -512), (-6, -216), (-4, -64), (-2, -8), (0, 0), (2, 8),
(4, 64), (6, 216), (8, 512))

>>> tuple({i:i**2} for i in range(-10,10) if i%2!=0)
({-9: 81}, {-7: 49}, {-5: 25}, {-3: 9}, {-1: 1}, {1: 1}, {3: 9}, {5: 25},
{7: 49}, {9: 81})

>>> tuple([i,i**2] for i in range(-10,10) if i%3==0)
([-9, 81], [-6, 36], [-3, 9], [0, 0], [3, 9], [6, 36], [9, 81])

>>> f = lambda x : x*(x+1)
>>> tuple([i,f(i)] for i in range(-10,10) if i%3==0)
([-9, 72], [-6, 30], [-3, 6], [0, 0], [3, 12], [6, 42], [9, 90])
```

Modules

A Python module is really just a text-file containing normal Python code. The code (generally) contains definitions of variables, functions, and/or classes that perform specific tasks which are expected to be called-for often.
You can reuse these functions and classes conveniently in other places of your code, simply by importing the related modules.
There are four common ways to import a module (or modules):

- `import module1, module2, ...` : Each function or variable imported from its associated module can only be accessed by putting name of the module plus a '.' in front of its name.
- `import module_name as alias` : Each function or variable imported can only be accessed by putting the alias name plus a '.' in front of its name.
- `from module_name import *` : All variables, functions, classes from the imported module will be accessible with names as defined in the module.
  (convenient, but may be wasteful and prone to name clashes)
- `from module_name import var1, var2, func1, ...` : Only the imported stuffs are accessible (names need not be preceeded by their `module_name`.
  Not imported stuffs are not accessible.)

```
>>> import sys, math
>>> sys.version
'3.4.3 (default, Oct 14 2015, 20:28:29) \n[GCC 4.8.4]'
>>> math.cos(math.pi)
-1.0
```

```
>>> import numpy as np
>>> np.sqrt([16, 9, 4, np.pi**2])
array([ 4.        ,  3.        ,  2.        ,  3.14159265])
>>> np.random.rand(1,4)
array([[ 0.82662719,  0.75383984,  0.13668734,  0.92577824]])
```

```
>>> from numpy import *
>>> sqrt([16, 9, 4, pi**2])
array([ 4.        ,  3.        ,  2.        ,  3.14159265])
>>> random.rand(1,5)
array([[ 0.59448252,  0.1241577 ,  0.14977436,  0.12175439,  0.81305565]])
```

```
>>> from numpy import random          #assume no previous import of numpy
>>> random.rand(1,5)
array([[ 0.80022709,  0.29715391,  0.06049658,  0.67376975,  0.31110592]])
>>> numpy.pi                          #only the imported ones are 'defined'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'numpy' is not defined
>>> sqrt(4)                           #only the imported ones are 'defined'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

Input and Output (via screen)

- The syntax of `input()` is `some_var = input(some_string)`.
  In Python3, inputs are always stored as string type objects.

```
>>> A=input('Input A = ')      #input instruction sting better be informative
Input A = 90
>>> [A, type(A)]
['90', <class 'str'>]

>>> B=input('Input a list here: ')
Input a list here: [1, 2, 3]
>>> [B, type(B)]               #Python saves raw input as a string type
['[1, 2, 3]', <class 'str'>]

>>> [eval(B), type(eval(B))]   #use eval() to evaluate input to proper type
[[1, 2, 3], <class 'list'>]    #B=eval(input('Input a list:')) get it directly
```

- Output is by `print()`, with syntax `print(some_string)`. The string to be printed can contain formatting instructions.
- A few formats for `print()`. (essentially it is *string concatenation)*
  1. *Python-style: (mighty, automatic type-adjustment, ...)*
  2. *C-style: (neat, versatile, ...)*
  3. *String concatenation via '+':*
  4. *String concatenation via '.join()':*

Input and Output (via screen)

- The syntax of `input()` is `some_var = input(some_string)`.
  In Python3, inputs are always stored as string type objects.

```
>>> A=input('Input A = ')      #input instruction sting better be informative
Input A = 90
>>> [A, type(A)]
['90', <class 'str'>]

>>> B=input('Input a list here: ')
Input a list here: [1, 2, 3]
>>> [B, type(B)]               #Python saves raw input as a string type
['[1, 2, 3]', <class 'str'>]

>>> [eval(B), type(eval(B))]   #use eval() to evaluate input to proper type
[[1, 2, 3], <class 'list'>]    #B=eval(input('Input a list:')) get it directly
```

- Output is by `print()`, with syntax `print(some_string)`. The string to be printed can contain formatting instructions.
- A few formats for `print()`. (essentially it is *string concatenation)*
  1. *Python-style: (mighty, automatic type-adjustment, ...)*
     Syntax: `'{} {}...{}'.format(v1,v2, ...,vn)`
  2. *C-style: (neat, versatile, ...)*
  3. *String concatenation via '+':*
  4. *String concatenation via '.join()':*

Input and Output (via screen)

- The syntax of `input()` is `some_var = input(some_string)`.
  In Python3, inputs are always stored as string type objects.

```
>>> A=input('Input A = ')      #input instruction sting better be informative
Input A = 90
>>> [A, type(A)]
['90', <class 'str'>]

>>> B=input('Input a list here: ')
Input a list here: [1, 2, 3]
>>> [B, type(B)]               #Python saves raw input as a string type
['[1, 2, 3]', <class 'str'>]

>>> [eval(B), type(eval(B))]   #use eval() to evaluate input to proper type
[[1, 2, 3], <class 'list'>]    #B=eval(input('Input a list:')) get it directly
```

- Output is by `print()`, with syntax `print(some_string)`. The string to be printed can contain formatting instructions.
- A few formats for `print()`. (essentially it is *string concatenation)*
  1. *Python-style: (mighty, automatic type-adjustment, ...)*
     Syntax: `'{} {}...{}'.format(v1,v2, ...,vn)`
  2. *C-style: (neat, versatile, ...) Syntax* `'%s %f %e %d' % (var_tuple)`
  3. *String concatenation via '+':*
  4. *String concatenation via* `'.join()'`*:*

Input and Output (via screen)

- The syntax of `input()` is `some_var = input(some_string)`.
  In Python3, inputs are always stored as string type objects.

```
>>> A=input('Input A = ')      #input instruction sting better be informative
Input A = 90
>>> [A, type(A)]
['90', <class 'str'>]

>>> B=input('Input a list here: ')
Input a list here: [1, 2, 3]
>>> [B, type(B)]               #Python saves raw input as a string type
['[1, 2, 3]', <class 'str'>]

>>> [eval(B), type(eval(B))]   #use eval() to evaluate input to proper type
[[1, 2, 3], <class 'list'>]    #B=eval(input('Input a list:')) get it directly
```

- Output is by `print()`, with syntax `print(some_string)`. The string to be printed can contain formatting instructions.
- A few formats for `print()`. (essentially it is *string concatenation)*
  1. *Python-style: (mighty, automatic type-adjustment, ...)*
     Syntax: `'{} {}...{}'.format(v1,v2, ...,vn)`
  2. *C-style: (neat, versatile, ...) Syntax* `'%s %f %e %d' % (var_tuple)`
  3. *String concatenation via '+': (simple to use, need* `str()` *for non-string types.)*
  4. *String concatenation via* `'.join()'`*:*

Input and Output (via screen)

- The syntax of `input()` is `some_var = input(some_string)`.
  In Python3, inputs are always stored as string type objects.

```
>>> A=input('Input A = ')      #input instruction sting better be informative
Input A = 90
>>> [A, type(A)]
['90', <class 'str'>]

>>> B=input('Input a list here: ')
Input a list here: [1, 2, 3]
>>> [B, type(B)]               #Python saves raw input as a string type
['[1, 2, 3]', <class 'str'>]

>>> [eval(B), type(eval(B))]   #use eval() to evaluate input to proper type
[[1, 2, 3], <class 'list'>]    #B=eval(input('Input a list:')) get it directly
```

- Output is by `print()`, with syntax `print(some_string)`. The string to be printed can contain formatting instructions.
- A few formats for `print()`. (essentially it is *string concatenation)*
  1. *Python-style: (mighty, automatic type-adjustment, ...)*
     Syntax: `'{} {}...{}'.format(v1,v2, ...,vn)`
  2. *C-style: (neat, versatile, ...) Syntax* `'%s %f %e %d' % (var_tuple)`
  3. *String concatenation via '+': (simple to use, need* `str()` *for non-string types.)*
  4. *String concatenation via* `'.join()'`*: (similar to using '+')*

Python-style format

Python-style output formats are mighty, handles automatic type-transformation, very convenient to use:

Syntax: `" {} {}...".format(var1,var2, ...),`

where each pair of empty curly braces serve as a place-holder for a corresponding (by position) variable listed in the `format()` function. In the resulting string, all listed variables will take the corresponding places reserved for them.

```
>>> strA='Mr. {} is {} years old and {} feet tall.'.format('Brown', 40, 7.51)
>>> print(strA)
Mr. Brown is 40 years old and 7.51 feet tall.

##can put position index in {}, but make sure the indexes match with vars.
>>> strB="Mr. {1} is {2} years old and {0} feet tall".format('Brown', 40, 7.51)
>>> print(strB)
Mr. 40 is 7.51 years old and Brown feet tall

##recognizes all python types, can print without explicit instruction on types
>>> L1=['a', 'b', 'c', 'z']
>>> T1=tuple((1.0, 2, 3, 4, -5.1))
>>> D1={"A":(1,2), "B":['e', 'f']}
>>> print(' list={} \n tuple={} \n dict={}'.format(L1, T1, D1) )
 list=['a', 'b', 'c', 'z']
 tuple=(1.0, 2, 3, 4, -5.1)
 dict={'B': ['e', 'f'], 'A': (1, 2)}
```

C-style format

C-style output formats are neat, versatile, more 'scientifically tunable', ...

Approximate syntax: `" %s %f %e %d ..." % (var1, var2, ...)`

where '%s' is for string type, '%f' for float type, '%e' for exponential format of float type, and '%d' for integer type. For each type, you can specify number of digits to show, such as '%10.3e'.

```
>>> from math import pi
>>> print('The %s value in float is %f, in exp is %e, it rounds to %d.'
...           % ('Pi', pi, pi, pi))
The Pi value in float is 3.141593, in exp is 3.141593e+00, it rounds to 3.

>>> print('The %s value in float is %.2f, in exp is %.2e, it rounds to %5d.'
...           % ('Pi', pi, pi, pi))
The Pi value in float is 3.14, in exp is 3.14e+00, it rounds to     3.

>>> print('%s = %.3f,  1000*%s = %.3e;  round(10*%s)=%6d.'
...           % ('Pi', pi, 'Pi', 1000*pi, 'Pi', round(10*pi)))
Pi = 3.142,  1000*Pi = 3.142e+03;  round(10*Pi)=    31.

>>> A5 = "Ms. %s is %d years old and %10.5f feet tall" % ('Green', 20,  2*pi)
>>> print(A5)
Ms. Green is 20 years old and    6.28319 feet tall
```

Review of string concatenation

An output format is essentially constructed via string concatenation. We have learned four ways doing the concatenation:

- By "+"
- By a C-style formatting
- By a Python-style formatting
- By the `.join()` method provided for the string class

```
## A review example for string concatenation
>>> s1="hello";  s2="python";  s3=3000

>>> A1 = s1 + ' ' + s2  + ' ' + str(s3);     #need str() for s3
>>> print(A1)
hello python 3000

>>> A2 ="%s %s %s" % (s1,s2,s3)    #C-style format, %s auto transform s3 to str type
>>> print(A2)
hello python 3000

>>> A3 = "{} {} {}".format(s1,s2,s3)         #Python-style format
>>> print(A3)
hello python 3000

>>> A4 = ' '.join([s1, s2, str(s3)])         #need str() for s3
>>> print(A4)
hello python 3000

>>> A1 == A2 == A3 == A4
True
```

Review of string concatenation

```
## Another review example for string concatenation
>>> s1="hello";   s2="python";   s3=3000

>>> A1 = s1 + '\\' + s2  + '\\' + str(s3) +'!'   #note '\' won't work here (why?)
>>> B1 = s1 + '+' + s2  + '+' + str(s3) +'!!!'

>>> A2 ="%s\%s\%s!" % (s1,s2,s3)                 #C-style format
>>> B2 ="%s+%s+%s!!!" % (s1,s2,s3)

>>> A3 = "{}\{}\{}!".format(s1,s2,s3)            #Python-style format
>>> B3 = "{}+{}+{}!!!".format(s1,s2,s3)

>>> A4=''.join(['\\'.join([s1, s2, str(s3)]),'!']) #need two .join() for this task
>>> B4=''.join(['+'.join([s1, s2, str(s3)]),'!!!'])

>>> [A1 == A2 == A3 == A4,  B1 == B2 == B3 == B4]
[True, True]

>>> [A1, A2, A3]
['hello\\python\\3000!', 'hello\\python\\3000!', 'hello\\python\\3000!']

>>> print("\t A1={},  A2={} \n\t A3={},  A4={}".format(A1, A2, A3, A4))
 A1=hello\python\3000!,  A2=hello\python\3000!
 A3=hello\python\3000!,  A4=hello\python\3000!

>>> print("\t B1={},  B2={} \n\t B3={},  B4={}".format(B1, B2, B3, B4))
 B1=hello+python+3000!!!,  B2=hello+python+3000!!!
 B3=hello+python+3000!!!,  B4=hello+python+3000!!!
```

Fine tuning the Python-style format

Is there a way to make the Python-style as precise as the C-style format?
The answer is a cool Yes!

```
>>> import math
>>> print("e={:.5f},  pi={:.15f}".format(math.e, math.pi))
e=2.71828,  pi=3.141592653589793
>>> print("e={:.7f},  pi={:.15e}".format(math.e*100, math.pi/1000))
e=271.8281828,  pi=3.141592653589793e-03

>>> print("{:.30s}={:.7e},  {:.10s}={:.15f}".format('Approximate e value',
          math.e*100, 'Approximate pi value', math.pi/1000))
Approximate e value=2.7182818e+02,  Approximat=0.003141592653590

>>> '{:{prec}} = {:{prec}}'.format('Gibberberish', math.e, prec='.3')
'Gib = 2.72'
>>> '{:{prec}} = {:{prec}}'.format('Gibberberish', math.e, prec='.7')
'Gibberb = 2.718282'
>>> '{:{prec}} = {:{prec}}'.format('Gibberberish', math.e, prec='.9')
'Gibberber = 2.71828183'
>>> '{:{p1}} = {:{p2}}'.format('Gibberberish', math.e, p1='.12', p2='.16')
'Gibberberish = 2.718281828459045'

>>> from datetime import datetime
>>> exam_date=datetime(2016, 11, 1, 9, 30)
>>> '{:%Y-%m-%d %H:%M}'.format(exam_date)
'2016-11-01 09:30'
>>> '{:{dfmt} {tfmt}}'.format(exam_date, dfmt='%m/%d/%Y', tfmt='%H:%M')
'11/01/2016 09:30'
```

Input and Output (via files)

The term "file" may be traced down to 1952, when early computers used punch cards and files were really documents stored in real cabinets or folders.

Most basic file manipulations include reading and writing. (Here 'writing' includes all types of editing, such as appending, deleting, or modifying.)

- Read from a text file: Use the `open()` function with syntax

  `fh = open(filename, 'r')` or `fh = open(filename)` ,

  where `filename` is a string – the name of the file to read; the `'r'` instructs the computer to open the file for `r`eading, this argument is optional and it defaults to `'r'`.

  The function returned a file handle (object), attributes and methods defined for file objects can be applied to process the opened file via the given handle.

- Write to a text file: First, need to `open()` with `'w'` assigned to the 2nd argument, as in `fh = open(filename, 'w')`; then, need to call the `.write()` method for real writing. E.g.,

```
my1stf = open('my_first_edit.txt', 'w')
my1stf.write('add a test line')
```

- When finished processing a file, it needs to be closed it via the `.close()` method to keep the data final. For the file with handle `fh`, closing it is simply `fh.close()`.

Example of a file read and write operation

Goal: Read in a text file, remove all leading whitespace, add line number (in the form of '#: ') to the beginning of each line, and save all the modified lines in a new file.

```
# assume the file to open is named 'if_poem.txt', which exists in current dir;
# and the new file to write to is named 'if_poem2.txt'

fin  = open("if_poem.txt")
fout = open("if_poem2.txt","w")
linum = 1           #used to count line numbers

for line in fin:   #important trick: iterate over all lines in fin (one by one)
    newline = line.lstrip()       #this removes all leading whitespace in line
    fout.write("{}{}{}\n".format(linum, ": ", newline))
    linum += 1

fin.close()
fout.close()
```

Common file IO modes

The file IO modes are controlled by the value of the 2nd parameter passed to the `open()` function, as in `open=(filename, mode)`.

1. Read only or Write only mode:
   - `'r'` : For reading only. When first opened, file pointer points to the beginning of file.
   - `'w'` : For writing only. Overwrites the file if it exists, or creates anew if it doesn't exist.
   - `'a'` : For appending only. When first opened, file pointer points to the end of file if it exists, otherwise `'a'` is the same as `'w'`.
2. Both Read and Write mode:
   - `'r+'` : For reading and writing. When first opened, file pointer points to the beginning of file.
   - `'w+'` : For reading and writing. When first opened, if the file exists, it overwrites the file; if not exits, it creates a new file with the given name for reading and writing.
   - `'a+'` : For reading and appending. When first opened, file pointer points to the end of the file if it exists, otherwise `'a+'` is the same as `'w+'`.

These modes are for text files. For a binary file, add a `b` before or after `r,w,a`, such as `'rb'`, `'br+'`, `'wb+'`, etc.

Other file IO commands

- The `open()` together with a `for` loop operate on files line by line. This has the advantage of being able to work on very large files (since they are processed one line at a time).
- For small to medium sized files, sometimes it is more convenient to load them in as a whole (instead of line by line). Two ways to do this: (Assuming a file handle `fh` is made ready)
  - `fh.readlines()` : This returns a list, its i-th item is the (i+1)-th line in the file associated with `fh`.
  - `fh.read()` : This returns a string containing all lines in the file associated with `fh`, concatenated from beginning to end, including carriage returns and line feeds.

```
# example of .readlines() and .read()
>>> fh=open('if_poem.txt')  #read in a text file with 39 lines
>>> lsta= fh.readlines()    #assign the whole content to lsta

>>> [len(lsta), type(lsta)]
[39, <class 'list'>]        #got a list from .readlines()

>>> lstb=fh.readlines();  [len(lstb), type(lstb)]
[0, <class 'list'>]         #got a list with no items (no rewind of fh)

>>> fh.seek(0); lstb=fh.readlines(); [len(lstb), type(lstb)]
0
[39, <class 'list'>]        #rewind via .seek() solves the 0 items problem

>>> fh=open('if_poem.txt')  #got the file handle again
>>> stra = fh.read();  [len(stra), type(stra)]
[2480, <class 'str'>]       #got a string from .read()
```

A few tricks in file IO

1. Automatic closing of a file: The `with` statement combined with `open()` has the advantage of automatically closing the opened file after the indented block of `with` is finished execution. (Thus it avoids the not uncommon mistake of forgetting to `.close()` an opened file.)

```
fhr = open("test.txt", 'r')

with open("testwrite.txt", "w") as fhw:
    for line in fhr:
        fhw.write("{}\n".format(line))
```

2. To avoid on-screen output of extra blank lines, use `.rstrip(os.linesep)`.

```
#demonstrate how to set and remove linesep explicitly
def  echofile(filename):
    #file=open(filename,'rU')          #'U'="Universal", os.linesep='\n' (python2)
    file=open(filename, newline='\n') #python3 way to set os.linesep='\n'
    for line in file:                 #iterates over all lines in the file one by one
        print(line.rstrip('\n')) #.rstrip('\n') avoids printing extra blank lines
    file.close()
```

3. The `print()` function can also be used to write to a file. That is, after a file handle `fh` is made ready to write, the `fh.write(some_string)` can also be achieved by `print(some_string, file=fh)`.

Read from and write to a same file (?)

- To open a file for reading and writing at the same time, the `open()` modes can be `'r+'`, `'w+'` or `'a+'`.
- Assuming the file exists and we do not want to overwrite (or delete) it, and the edit is not just appending at the end, then we should use the `'r+'` mode.
- However, read from and write to a same file at the same time usually may not be a good idea (may need to re-position the file pointer using `.seek()` and `.tell()` etc if you need in-line replacements).
- It is often more convenient to write to a new file. A combined `'r'` and `'w'` modes for two different files would do the same task, as in

```
with open('inputfile', 'r') as infile:
    with open('outputfile', 'w') as outfile:
        for line in infile:
            modline=(some modification to line)
            outfile.write(modline)
```

(This way may be much easier than working strictly with only one file, since you don't need to 'manually' reset file pointers.)

Handling exceptions

Two types of most common errors:

1. Syntax Errors: A statement or expression is syntactically correct. (likely the most common errors you got when you learning Python.) These are errors caught before execution.
2. Exceptions: These are errors caught during execution. The More common exceptions are `TypeError`, `ValueError`, `IndexError`, `NameError`, others include `ZeroDivisionError`, `IOError`, `OSError`...

Exceptions can be handled via the `Try` and `except` block, the syntax is

```
try:
   do something
except:        #this part is reached only when the try part caught error)
   raise exception (or print exception message)
```

```
while True:
   val = input('Enter an integer: ')
   try:
      val = int(val)
      print('The square of the number you entered is', val**2)
      break  #exit the while loop
   except ValueError:
      print('Your input ', val, ' is not an integer')
```

Exceptions with informative messages

- System error message may be collected and print out.
- Try-except block can have more than 1 Exceptions, e.g.,

```
try:
    f = open('myfile.txt')
    num = float(f.readline().strip())
except OSError as err1:
    print("OS error: {}".format(err1))
except ValueError as err2:
    print("Could not convert data to float: {}".format(err2))
except:
    print("Unexpected error caught")
    raise
```

- More than 1 exceptions can be put together into a tuple. E.g.,

```
try:
    do_some_tasks
except (IOError, ValueError, TypeError) as err:
    print('exception caught,  got error message: {}'.format(err))
```

Or better, simply use the `Exception` keyword, e.g.,

```
try:
    do_some_tasks
except Exception as err:
    print('exception caught,  got error message: {}'.format(err))
```

Line continuation in Python

Python adopts implied-line-continuation via parentheses, brackets and braces.
A (long) statement can be broken over multiple lines by wrapping expressions in a pair of parentheses, brackets, or braces. This is the preferred Python way to do line continuation.
If you prefer not to use the implied-line-continuation, a trailing backslash can be used for line continuation. In either scenario, better indent code properly for easier human reading.

```
#use implied line continuation
f = (lambda x: x+
               x**2 +
               x**3 )

[(x, y)   ##list comprehension written as nested loops
    for x in range(20)
       for y in range(20)
           if x**2 + y**2 <=100
]

#use trailing backslash (no penalty if you use '\' inside  () [] or {})
#but you cannot add anything after trailing '\', including whitespace or comment
g = lambda x: x + \
              x**2 + \
              x**3

[(x, y) \
    for x in range(20) \
       for y in range(30) \
           if x**2 + y**2 <=100 \
]
```

Line continuation

There are a few cases in which line continuation has to be done with a backslash.

```
#a quoted string cannot be broken into multiple lines without '\'
>>> print('this is a quoted string \
... separated in two lines')
this is a quoted string separated in two lines
>>> print('this is a quoted string
  File "<stdin>", line 1
    print('this is a quoted string
                                  ^
SyntaxError: EOL while scanning string literal

>>> ('A= \
... B')
'A= B'
>>> ('A=
  File "<stdin>", line 1
    ('A=
       ^
SyntaxError: EOL while scanning string literal

#can use implied-line-continued whenever the quotes are paired in one line
>>> ('A= B'
... )
'A= B'
>>> print('this is a quoted string '
... + 'separated in two lines')
this is a quoted string separated in two lines
```

Some useful Python tricks

- `dir()`: Return a quick list of the defined symbols or "attributes" of an object (such as an imported module, e.g., `import numpy; dir(numpy)`).
- `help()`: Return documentation strings for various modules, functions, and methods. (Once you find an attribute of an obj via `dir()`, you can use `help()` to see doc for that attribute, e.g., `help(numpy.random)`.)
- `import os`: after the import, you can use most of system commands allowed by your OS, e.g., `os.getcwd()`, `os.chdir()`, `os.mkdir()`, `os.execl()`.

There are a few ways to run a Python script file. You can open it in spyder and run it there; you can also directly run via command line; or, in an iterative Python console, you can run your file using `exec(open("").read())`.

E.g., assume that you name your python script as `mytest.py`, then

- to use the command line to run it, open a terminal and cd into the dir that contains your script file, then type in `python mytest.py`. (This assumes the python executable is in your PATH, if not, you need to specify the full path of your python command before your script name.)
- to run it in a python console, you can simply use `exec(open("mytest.py").read())`. (This assumes the `mytest.py` file is in your current working dir, if not, you should specify the full path of `mytest.py` in the `open()` command.)

Measuring time of Python code

There are a few ways to measure the wall-time or cpu time of your code. Some relatively simple to use ones: (may need to run several times then take average)

- Use the `timeit` module inside your code. This way you can time any specific code-block you want to time, simply by enclosing it inside two `default_timer()`. E.g.,

```
from timeit import default_timer
tstart = default_timer()
# put the code-block to be timed here
elapsed_time = default_timer() - tstart
```

- Use the `time` module inside your code, e.g.,

```
import time  #may need to replace .clock() below by .process_time() in Python3
tstart = time.clock()
# put the code-block to be timed here
elapsed_time = time.clock() - tstart
```

- Use the `-m cProfile` followed by `-m pstats` on command line. This will provide far more details (than the two approaches above) of the cpu time as well as number of calls to specific functions etc of your code. E.g.,

```
##save profile data in a file with the -o option (e.g. -o stats.prof)
%python -m cProfile -o stats.prof  your_script_file
##after the run, look at the profiling data saved in stats.prof using
%python -m pstats stats.prof
##you'll see a prompt (here it is stats.prof%), then type 'stat funct_name',
##where 'funct_name' is a function in your_script_file, to see the details.
```

Introduction to numpy

Overview:

- Numpy allows users to call optimized C and/or Fortran code while writing in Python.
- The main data structure is `array`, with type `numpy.ndarray`. The array type include common structures such as vectors, matrices, and tensors.
- Fast for vectorizable array operations (in the spirit of Matlab).

Many scientific calculations are naturally vecterizable, including

- Matrix multiplications and many other linear algebra routines;
- Applying a fixed function (e.g., sine or cosine) to an entire array;
- Generating random arrays.

Why use ndarray as in numpy?

numpy `ndarray` appears to be like lists in Python. Why not just stick with list type? Why bother introducing the new array type?
Main reason: Efficiency !

- Python lists are very general and dynamically typed. A list can contain any type of objects in it (i.e., inhomogeneous). Applying mathematically functions to all items in the list in a vectorized fashion may not be doable or at best inefficient.
- Numpy ndarrays are homogeneous and statically typed, all items in the array are of same type (mostly numerical). The static typing allows efficient application of mathematical functions in vectorized form, and these can be done using compiled languages C and/or Fortran.

numpy arrays

Basic numpy functions: `.empty()`, `.zeros()`, `.ones()`, `.identity()` or `.eye()`, `.diag()`.
The shape of the generated array can be controlled by a tuple specifying the dimension.

```
>>> np.empty(3)          #.empty() generate an 'empty' array of desired shape
array([  6.91074585e-310,   3.07494838e-316,   1.58101007e-322])
>>> np.empty((2,3))
array([[  6.91074585e-310,   3.04679375e-316,   3.95252517e-323],
       [  3.95252517e-323,   0.00000000e+000,   0.00000000e+000]])

>>> [ np.zeros(3), np.ones(4) ]
[array([ 0.,  0.,  0.]), array([ 1.,  1.,  1.,  1.])]
>>> np.zeros((3,4))      #generate 3 x 4 zero matrix
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones((2,5))       #generate 2 x 5 matrix of all ones
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])

>>> np.eye(3)            #3x3 identity matrix, same as np.identity(4)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

>>> np.diag([3,5,1])   #.diag() generates a diagonal matrix from input list/array.
array([[3, 0, 0],  #if input is a matrix, .diag() returns an array of its diagonal
       [0, 5, 0],
       [0, 0, 1]])
```

numpy useful functions .array(), .arange()

- .array() can be used to create numpy array (in particular, '.ndarray')

```
>>> import numpy as np
>>> lst = range(10)
>>> L = np.array(lst)
>>> [type(lst), type(L)]
[<class 'range'>, <class 'numpy.ndarray'>]

>>> 0.5*L    #this is allowed (vectorized multiplication)
array([ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5])

>>> 0.5*lst      #this is not allowed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'float' and 'range'
```

- .arange() is similar to .range(), but it returns an '.ndarray' type.

Syntax: `arange(start, stop, step=1, dtype=None)`

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(5, 20, 3)
array([ 5,  8, 11, 14, 17])

>>> np.arange(1, 4, 0.5)    #better use .linspace() for non-integer ranges
array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5])
>>> np.arange(2.6, 4, 0.2)
array([ 2.6,  2.8,  3. ,  3.2,  3.4,  3.6,  3.8])
```

numpy useful function .linspace()

```
linspace(start, stop, num=50, endpoint=True, retstep=False)
    Returns `num` evenly spaced samples, calculated over the
    interval [`start`, `stop` ].
    The endpoint of the interval can optionally be excluded.
```

```
>>> np.linspace(0, 1, num=9)   #note: default endpoint is True
array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ,  0.625,  0.75 ,  0.875,  1.   ])

>>> np.linspace(0, 10, num=10, endpoint=False)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

>>> np.linspace(0, 10, endpoint=False)
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8,  2. ,
        2.2,  2.4,  2.6,  2.8,  3. ,  3.2,  3.4,  3.6,  3.8,  4. ,  4.2,
        4.4,  4.6,  4.8,  5. ,  5.2,  5.4,  5.6,  5.8,  6. ,  6.2,  6.4,
        6.6,  6.8,  7. ,  7.2,  7.4,  7.6,  7.8,  8. ,  8.2,  8.4,  8.6,
        8.8,  9. ,  9.2,  9.4,  9.6,  9.8])

#how to generate [0.2, 0.25, 0.3, ...., 1.05] ? (at least two ways)
>>> np.linspace(0.2, 1.05, (1.05-0.2)/0.05 +1)
array([ 0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,  0.45,  0.5 ,  0.55,  0.6 ,
        0.65,  0.7 ,  0.75,  0.8 ,  0.85,  0.9 ,  0.95,  1.  ,  1.05])
>>> np.linspace(0.2, 1.1, (1.1-0.2)/0.05, endpoint=False)
array([ 0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,  0.45,  0.5 ,  0.55,  0.6 ,
        0.65,  0.7 ,  0.75,  0.8 ,  0.85,  0.9 ,  0.95,  1.  ,  1.05])
```

numpy array: shape, reshape

```
#for numpy.ndarray M,  the  M.shape returns a tuple of dimension
>>> M=np.linspace(1,20,20); M
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,  11.,
        12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.])
>>> M.shape            #.shape is a tuple, thus it cannot be M.shape()
(20,)

>>> N= M.reshape(4,5); N
array([[  1.,   2.,   3.,   4.,   5.],   #note the row-wise ordering!
       [  6.,   7.,   8.,   9.,  10.],
       [ 11.,  12.,  13.,  14.,  15.],
       [ 16.,  17.,  18.,  19.,  20.]])
>>> N.shape
(4, 5)

>>> P=M.reshape(2,10); P
array([[  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.],
       [ 11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.]])
>>> P.shape
(2, 10)
```

Other Numpy data descriptors include: `.dtype` — data type, `.itemsize` — bytes per array item, `.ndim` — number of dimensions, `.nbytes` — total bytes of array, `.size` — number of items in array.

```
>>> [M.dtype,  M.ndim,  M.itemsize,  M.size,  M.nbytes]
[dtype('float64'), 1, 8, 20, 160]
>>> [P.dtype,  P.ndim,  P.itemsize,  P.size,  P.nbytes]
[dtype('float64'), 2, 8, 20, 160]
```

numpy array: copy

Same as in Python list, numpy array may have the *call-by-reference* issue. Need a true copy to avoid unwanted side-effect.

```
>>> x=np.linspace(1,5,11); x
array([ 1. ,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,  3.8,  4.2,  4.6,  5. ])
>>> y=x       #make y to point to the same address of x
>>> y[0]=0; y[-1]=0
>>> x         #x is changed by the change in y
array([ 0. ,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,  3.8,  4.2,  4.6,  0. ])

>>> xx=np.linspace(0,5,6); xx
array([ 0.,  1.,  2.,  3.,  4.,  5.])
>>> yy=xx[:]
>>> yy[0]=8; yy[1]=7
>>> xx        #copy via index slicing does not work here !!
array([ 8.,  7.,  2.,  3.,  4.,  5.])

>>> X=np.linspace(10,20, 11); X
array([ 10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.])
>>> Y=list(X);  Z=X.copy()    #both are valid 'shallow-copy' (only 1 level)
>>> Y[0]=0; Y[-1]=0;  Z[-1]=30; Z[-2]=0
>>> Y
[0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 0]
>>> Z
array([ 10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,   0.,  30.])
>>> X
array([ 10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.])
```

The `np.copyto(b, a)` can also be used to make a true copy of numpy array a and store in b.

numpy array: indexing

The syntax is the same as index slicing `[start:stop:step]` for Python sequence types (strings, lists, etc). For a multi-dimension numpy array, same indexing rule holds for each dimension.

```
>>> import numpy as np
>>> M=np.arange(1., 25).reshape(4,6);  M
array([[  1.,   2.,   3.,   4.,   5.,   6.],
       [  7.,   8.,   9.,  10.,  11.,  12.],
       [ 13.,  14.,  15.,  16.,  17.,  18.],
       [ 19.,  20.,  21.,  22.,  23.,  24.]])

>>> (M[1,1],  M[2,4],  M[-1, 0],  M[-1,-2])
(8.0, 17.0, 19.0, 23.0)

>>> M[:,1]      #2nd column
array([  2.,   8.,  14.,  20.])

>>> M[0,:]      #1st row
array([ 1.,  2.,  3.,  4.,  5.,  6.])

>>> M[0:2, 2:]
array([[  3.,   4.,   5.,   6.],
       [  9.,  10.,  11.,  12.]])

>>> M[0:4,0:4]
array([[  1.,   2.,   3.,   4.],
       [  7.,   8.,   9.,  10.],
       [ 13.,  14.,  15.,  16.],
       [ 19.,  20.,  21.,  22.]])
```

```
>>> M[-3:,-3:]
array([[ 10.,  11.,  12.],
       [ 16.,  17.,  18.],
       [ 22.,  23.,  24.]])

>>> M[2:4, 2:5]
array([[ 15.,  16.,  17.],
       [ 21.,  22.,  23.]])

>>> M[-2:, 0:3]
array([[ 13.,  14.,  15.],
       [ 19.,  20.,  21.]])

>>> M[0:3, 0 :6 :2]
array([[  1.,   3.,   5.],
       [  7.,   9.,  11.],
       [ 13.,  15.,  17.]])
```

Numpy random arrays: often used functions

- `np.random.rand(d1, d2, ...)`: Create an array of shape (d1, d2, ...) whose items are random samples from a uniform distribution over [0, 1).
- `np.random.randn(d1, d2, ...)`: Create an array of shape (d1, d2, ...) whose items are random samples from standard normal distribution.
- `np.random.normal(`σ`, `μ`, size=(d1, d2, ...))`: Create an array of shape (d1, d2, ...) whose items are random samples from normal distribution with mean σ and standard deviation μ.
- `np.random.randint(low, high, size=(d1, d2, ...))`: Create an array of shape (d1, d2, ...) whose items are random integers drawn from discrete uniform distribution in [low, high).
- `np.random.laplace(`σ`, `λ`, size=(d1, d2, ...))`: Create an array of shape (d1, d2, ...) whose items are random samples from Laplace distribution with mean σ and exponential decay λ.
- `np.random.shuffle(x)`: Randomly shuffle the items in the input array x. (When x has more than 1 dim, to shuffle all items instead of the ones in the 1st dim, reshape x into 1 dim first, shuffle, then reshape back.)
- `np.random.choice(x, size=(d1, d2,...), replace=True, p=None)`, random choice from given array `x`, by default the choice is with replacement and with uniform distribution over all entries in `x`. Choice probability can be given via `p`. If `x` is an integer `n`, then `x=np.arange(n)`.

Numpy random arrays

The tuple used to control the shape of a randomly generated array is `size=(d1,d2,...)`, it defaults to 1 (i.e., generate only one random number) if not given.

```
>>> import numpy as np
>>> [np.random.rand(), np.random.randn(), np.random.normal()]
[0.8121580446674673, -0.20167117488599684, -0.067162940267359 24]

>>> [np.random.rand(2), np.random.randn(3)]
[array([ 0.9690785 ,  0.02537005]), array([ 0.69905858, -0.25999029, -0.27275752])]

>>> np.random.rand(2,5)
array([[ 0.08438024,  0.53420951,  0.60934605,  0.76362634,  0.2152678 ],
       [ 0.10716023,  0.94743159,  0.06901283,  0.12752239,  0.15627961]])

>>> np.random.randn(3,4)
array([[-0.46381411, -1.22834981, -1.11043212,  0.88301188],
       [ 0.30787985, -0.83778999, -0.0339636 , -0.104661  ],
       [-0.40986742, -1.14611435, -1.23068656,  1.90628689]])

>>> A= np.random.normal(-10, 2, (3,5));  A
array([[-10.53647098,  -9.23432843,  -7.60737642, -12.65568273,  -9.62449061],
       [ -8.04144009, -11.22679572,  -8.52044469, -14.20031506, -10.07048319],
       [ -9.37341539, -10.67328335,  -8.35509204,  -9.09881004, -11.87010034]])

>>> B=np.random.randint(-10, 10, (3,8)); B
array([[-10,  -7,  -8, -10,  -4,  -2,  -5,  -3],
       [ -2,  -8,   5,  -6,   9,  -7, -10,  -9],
       [  1,  -2,  -4,  -8,   7,   3,  -8,  -1]])
```

Numpy random arrays

```
>>> import numpy as np

>>> C=np.array(range(12));
>>> np.random.shuffle(C); C
array([ 3,  4,  5,  9, 11,  0,  8, 10,  6,  7,  2,  1])

>>> np.random.choice(C, 12)
array([ 2,  2,  4, 11, 11,  1,  4,  7,  3, 11,  5, 11])

>>> np.random.choice(C, 12, replace=False)
array([ 0,  3, 11,  5,  9,  1, 10,  8,  2,  6,  7,  4])

>>> np.random.choice(12, 12)
array([10,  8,  9,  1,  5,  7,  7,  1, 11,  6,  9,  4])

>>> np.random.choice(12, 12, p=[0, 0.2, 0.6, 0.2, 0,0,0,0,0,0,0,0])
array([2, 2, 3, 2, 1, 2, 1, 2, 2, 2, 2, 2])
>>> np.random.choice(12, 12, p=[0, 0.2, 0.6, 0.2, 0,0,0,0,0,0,0,0])
array([2, 3, 2, 2, 2, 1, 3, 2, 2, 2, 2, 2])
>>> np.random.choice(12, 12, p=[0, 0.2, 0.6, 0.2, 0,0,0,0,0,0,0,0])
array([2, 2, 2, 3, 2, 1, 2, 2, 1, 2, 2, 1])
>>> np.random.choice(12, 12, p=[0, 0.2, 0.6, 0.2, 0,0,0,0,0,0,0,0]).reshape(3,4)
array([[1, 2, 2, 2],
       [2, 1, 3, 2],
       [3, 3, 1, 2]])
```

numpy array: sorting

The functions are `.sort()`, which sorts the input array in non-decreasing order and overwrites the array;
and `.argsort()`, which returns an array containing the position indexes that would sort the array.

```
>>> import numpy as np
>>> a=np.random.rand(5);  a
array([ 0.72102822,  0.87673033,  0.49757161,  0.49294696,  0.36479718])

>>> ia=a.argsort()     #.argsort() returns the position indexes of sorted order
>>> ia
array([4, 3, 2, 0, 1])

>>> b = a.copy()       #make a copy of a (to show the 'fancy-indexing' later)

>>> a.sort()           #.sort() sort in non-decreasing order

>>> a                  #note a is changed (overwritten)
array([ 0.36479718,  0.49294696,  0.49757161,  0.72102822,  0.87673033])

>>> b[ia]              #this is 'fancy indexing', should be the same as sorted a
array([ 0.36479718,  0.49294696,  0.49757161,  0.72102822,  0.87673033])
```

numpy array: sorting

For multi-dimension array, one can specify which dimension/axis to sort along by specifying the `axis`. If the dimension is n, then `axis` can be $0, 1, ..., n-1$.

```
>>> A=np.random.randn(3,4);  A
array([[ 0.17192638, -0.01110778,  0.09352861, -0.80519429],
       [-0.16181836, -1.89493851, -0.48703992,  1.08379986],
       [ 0.14086322, -2.07361723, -0.71599613,  1.33880704]])

>>> A.sort(axis=0);  #sort along each column
>>> A
array([[-0.16181836, -2.07361723, -0.71599613, -0.80519429],
       [ 0.14086322, -1.89493851, -0.48703992,  1.08379986],
       [ 0.17192638, -0.01110778,  0.09352861,  1.33880704]])

>>> A.sort(1)        #sort along each row (keyword axis can be omitted)
>>> A
array([[-2.07361723, -0.80519429, -0.71599613, -0.16181836],
       [-1.89493851, -0.48703992,  0.14086322,  1.08379986],
       [-0.01110778,  0.09352861,  0.17192638,  1.33880704]])
```

After sorting along each axis, the array will be non-decreasing along all axes/dimensions.

Numpy array: basic vectorized operations

Scaling of vectors or matrices, vector dot product, matrix-vector product, matrix-matrix product, applying a fixed function to a vector elementwisely, etc, can all be done in vectorized manner.

```
>>> import numpy as np

>>> v=np.array(range(10))

>>> v**2                    #same as v*v
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])

>>> v+1
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

>>> np.log(v + 1)
array([ 0.        ,  0.69314718,  1.09861229,  1.38629436,  1.60943791,
        1.79175947,  1.94591015,  2.07944154,  2.19722458,  2.30258509])
>>> np.cos(v)               #math.cos() is only for scalars
array([ 1.        ,  0.54030231, -0.41614684, -0.9899925 , -0.65364362,
        0.28366219,  0.96017029,  0.75390225, -0.14550003, -0.91113026])

>>> v2=np.ones(10)
>>> np.dot(v, v2)           #vectorized dot product, same as v.dot(v2) here
45.0

>>> np.greater(v, 3*v2)     #compare vectors component-wisely
array([False, False, False, False,  True,  True,  True,  True,  True,  True],
dtype=bool)
```

Numpy array: basic vectorized operations

```
>>> import numpy as np
>>> v=np.array(range(1,10))
>>> M=np.ones((9,9))
>>> M*v        #not the mat-vect product expected (since M is array, not matrix)
array([[ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
       [ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.]])

>>> M.dot(v)
array([ 45.,  45.,  45.,  45.,  45.,  45.,  45.,  45.,  45.])

>>> np.dot(M,v)
array([ 45.,  45.,  45.,  45.,  45.,  45.,  45.,  45.,  45.])

>>> M @ v      #the '@' is the 'product' one would expect for M*v
array([ 45.,  45.,  45.,  45.,  45.,  45.,  45.,  45.,  45.])

>>> np.matmul(M,v)  #.matmaul() is matrix-matrix multiplication
array([ 45.,  45.,  45.,  45.,  45.,  45.,  45.,  45.,  45.])
```

Numpy array: basic vectorized operations

```
#continue from example in previous slide: we can transform numpy array
#into numpy matrix, such that '*' would behave as standard product for
#a 'matrix times matrix' operation
>>> M=np.ones((9,9))
>>> v=np.array(range(1,10))

>>> A=np.matrix(M)          #transform array to matrix
>>> vt=np.matrix(v).T       #.T means transpose
>>> A*vt                    #for matrix data type, '*' behaves as expected
matrix([[ 45.],
        [ 45.],
        [ 45.],
        [ 45.],
        [ 45.],
        [ 45.],
        [ 45.],
        [ 45.],
        [ 45.]])

>>> v3=np.random.randn(1,5); v3
array([[-0.21126959,  0.47500691,  0.8362255 ,  0.75218421, -0.17085784]])

>>> v4=np.sign(v3); v4
array([[-1.,  1.,  1.,  1., -1.]])

>>> v3_nrm1, v3_nrm2 = [np.dot(v4, v3.T),  np.dot(v3, v3.T)]
>>> print('norm1={}, norm2={}'.format( v3_nrm1, v3_nrm2 ))
norm1=[[ 2.44554405]], norm2=[[ 1.56451297]]
```

Numpy: matrix-matrix product

Assume M and B are size $n \times n$ numpy arrays, there are at least five ways to do standard matrix-matrix product $M * B$ in Python3:

- By `np.dot(M, B)` or `M.dot(B)`
- By `np.matmul(M, B)`
- By `M @ B`
- By transforming arrays to matrixes via `np.matrix()` then use '$*$'

```
>>> import numpy as np
>>> n = 200
# generate two matrices of size n x n
>>> M = np.random.randint(1, 5, size=(n,n))
>>> B = np.random.rand(n,n)

>>> MB1 = np.dot(M, B)
>>> MB2 = M.dot(B)
>>> MB3 = np.matmul(M, B)
>>> MB4 = M @ B
>>> MB5 = np.matrix(M) * np.matrix(B)

# now check the difference of the products (by checking their norms)
>>> [np.linalg.norm(MB1-MB2), np.linalg.norm(MB2-MB3), np.linalg.norm(MB3-MB4)]
[0.0, 0.0, 0.0]

>>> from numpy import linalg as LA              #make shorter name via alias
>>> [LA.norm(MB1-MB2), LA.norm(MB2-MB3), LA.norm(MB3-MB4), LA.norm(MB4-MB5)]
[0.0, 0.0, 0.0, 0.0]
```

Vectorizing a function: numpy.vectorize()

```
>>> import math, numpy
>>> v = numpy.arange(10)
>>> math.sin(v)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars

>>> math.sin = numpy.vectorize(math.sin)
>>> math.sin(v)
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])

>>> math.exp(v)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars

>>> math.exp=numpy.vectorize(math.exp)
>>> math.exp(v)
array([  1.00000000e+00,   2.71828183e+00,   7.38905610e+00,
         2.00855369e+01,   5.45981500e+01,   1.48413159e+02,
         4.03428793e+02,   1.09663316e+03,   2.98095799e+03,   8.10308393e+03])
>>> numpy.sin(v)
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
>>> numpy.exp(v)
array([  1.00000000e+00,   2.71828183e+00,   7.38905610e+00,
         2.00855369e+01,   5.45981500e+01,   1.48413159e+02,
         4.03428793e+02,   1.09663316e+03,   2.98095799e+03,   8.10308393e+03])
```

Numpy .linalg submodule

```
>>> dir(np.linalg)
['LinAlgError', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__path__', '__spec__', '_numpy_tester',
'_umath_linalg', 'absolute_import', 'bench', 'cholesky', 'cond', 'det',
'division', 'eig', 'eigh', 'eigvals', 'eigvalsh', 'info', 'inv',
'lapack_lite', 'linalg', 'lstsq', 'matrix_power', 'matrix_rank', 'multi_dot',
'norm', 'pinv', 'print_function', 'qr', 'slogdet', 'solve', 'svd', 'tensorinv',
'tensorsolve', 'test']

>>> n = 200
>>> M = np.random.normal(1,5, size=(n,n))    #generate a size n x n matrix
>>> b = np.random.rand(n)                    #generate a random vector
>>> x = np.linalg.solve(M, b)                #solve M*x = b
>>>
>>> error = np.linalg.norm(M.dot(x) - b);    #check error ||M*x-b||
>>> print(error)
1.67807972175e-13

>>> np.linalg.cond(M)                        #compute condition number of M
856.78694700310712

>>> from numpy import linalg as LA           #same as 'import numpy.linalg as LA'
>>> eval, evect = LA.eig(M)                  #compute eigendecomposition of M
>>>
>>> max([ LA.norm(M@evect[:,j] - evect[:,j]*eval[j]) for j in range(n) ])
1.0658675198866689e-12
```

Numpy: Saving and loading data via files

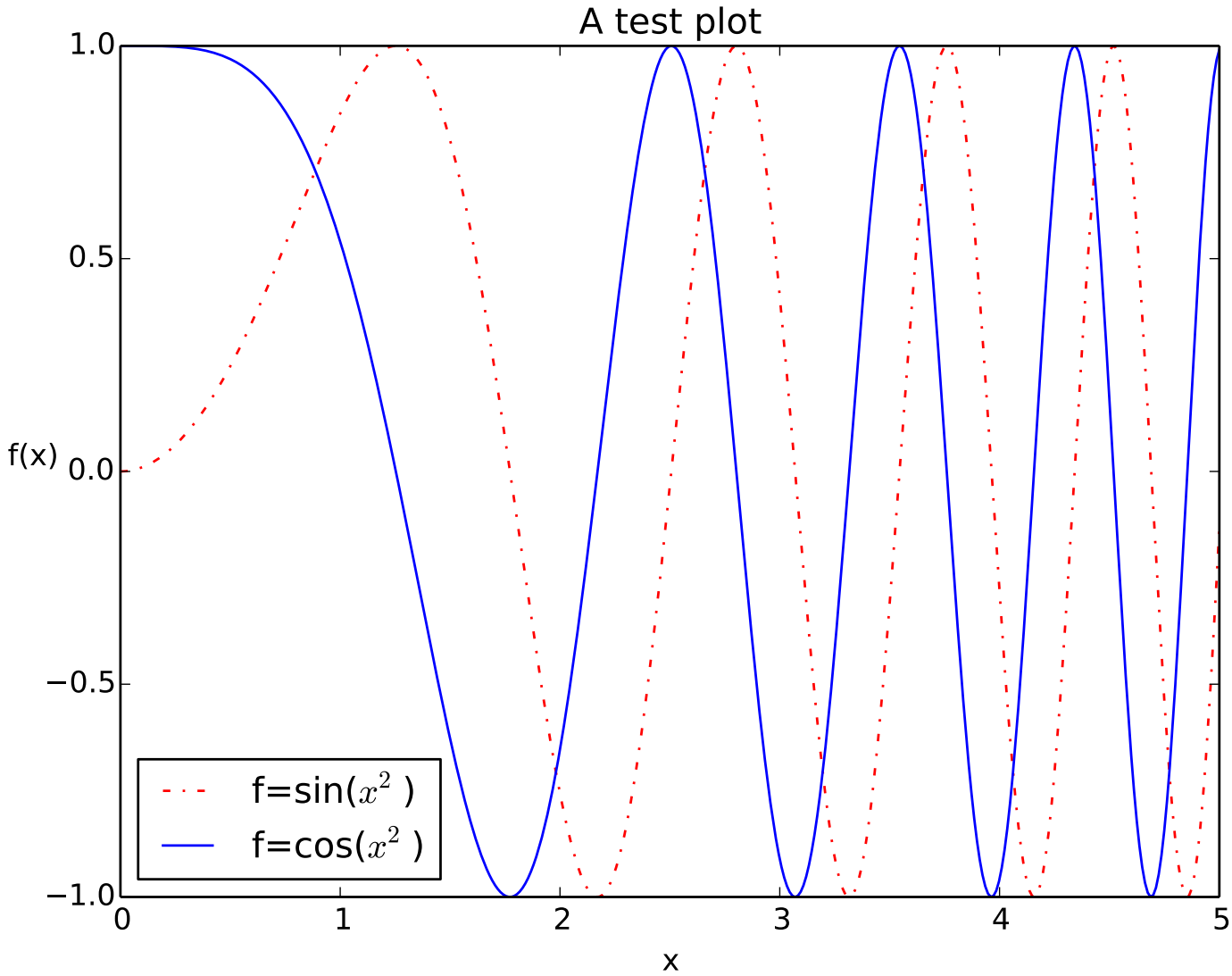
Numpy provides the `.save()` and `.savez()` functions to save data to a file using the the native `npy` and compressed `npz` format.
The same data can be loaded from the saved file via the `.load()` function.
The `.savetxt()` function can also be used to save data in a file in plain text format, which can then be read via Python built-in `open()`, or read by Numpy `.genfromtxt()` function.

```
#e.g. on saving and loading data (assume numpy is imported as np)
>>> M = np.array([[1., 2., 3.], [4., 5., 6.], [7., 8., 9.]])

>>> np.save('matrix_M.npy', M)    #save data in M in a file named matrix_M.npy
>>> N = np.load('matrix_M.npy')   #load M from file, pass the data to array N
>>> N
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])

>>> np.savetxt('Mmatrix.csv', M,  delimiter=',')  #saves in csv format
>>> np.savetxt('Mmatrix.txt', M)      #save in text, default delimiter=' '
>>> N1=np.genfromtxt('Mmatrix.txt')
>>> N1
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
>>> N2=np.genfromtxt('Mmatrix.csv',delimiter=',') #the delimiter cannot be omitted
>>> N2
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
```

Introduction to matplotlib

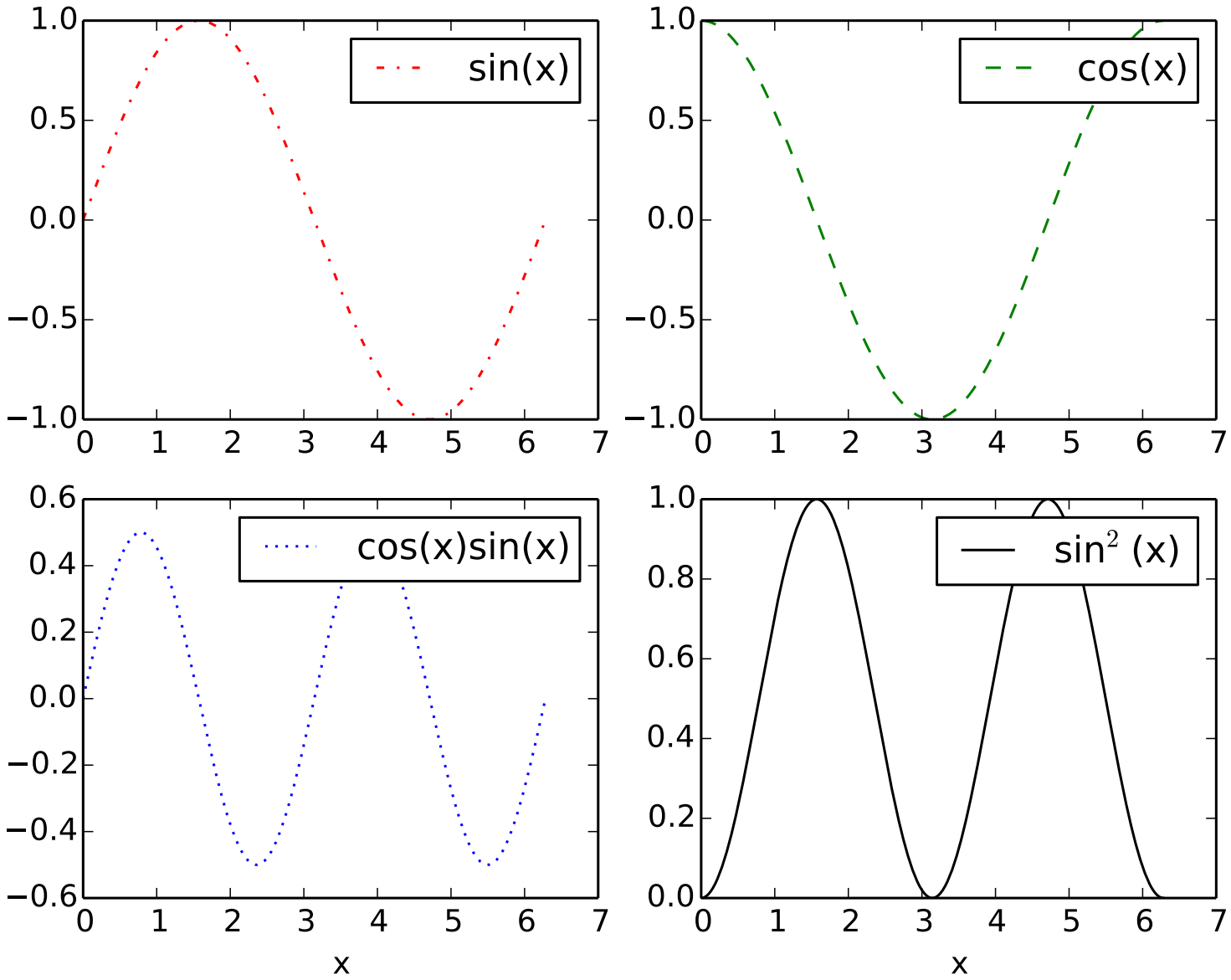
Matplotlib is an outstanding graphics library for high quality 2D plots, with toolkits for 3D plots. The output can easily be saved in usual formats (png, pdf, jpg, jpeg, eps, svg, ...), or output as LaTeX text (pgf).
Main commands: `plt.subplots()` together with `.plot()` .

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 5, 400)
y = np.sin(x**2)
z = np.cos(x**2)

#plotting just one figure
fig, aix = plt.subplots()
aix.plot(x, y, 'r-.', label='f=sin($x^2$)')
aix.plot(x, z, 'b-', label='f=cos($x^2$)')
aix.set_title('A test plot')
aix.legend(loc='best')
aix.set_xlabel('x')
aix.set_ylabel('f(x)', rotation=0)
fig.savefig('sin_cos.pdf')            #can use plt.savefig() here
fig.show()              #can use plt.show() or plot.show(fig) here
```


A test plot
1.0
0.5
0.0
−0.5
−1.0
f(x)
0
1
2
3
4
5
x
f=sin(x^2)
f=cos(x^2)

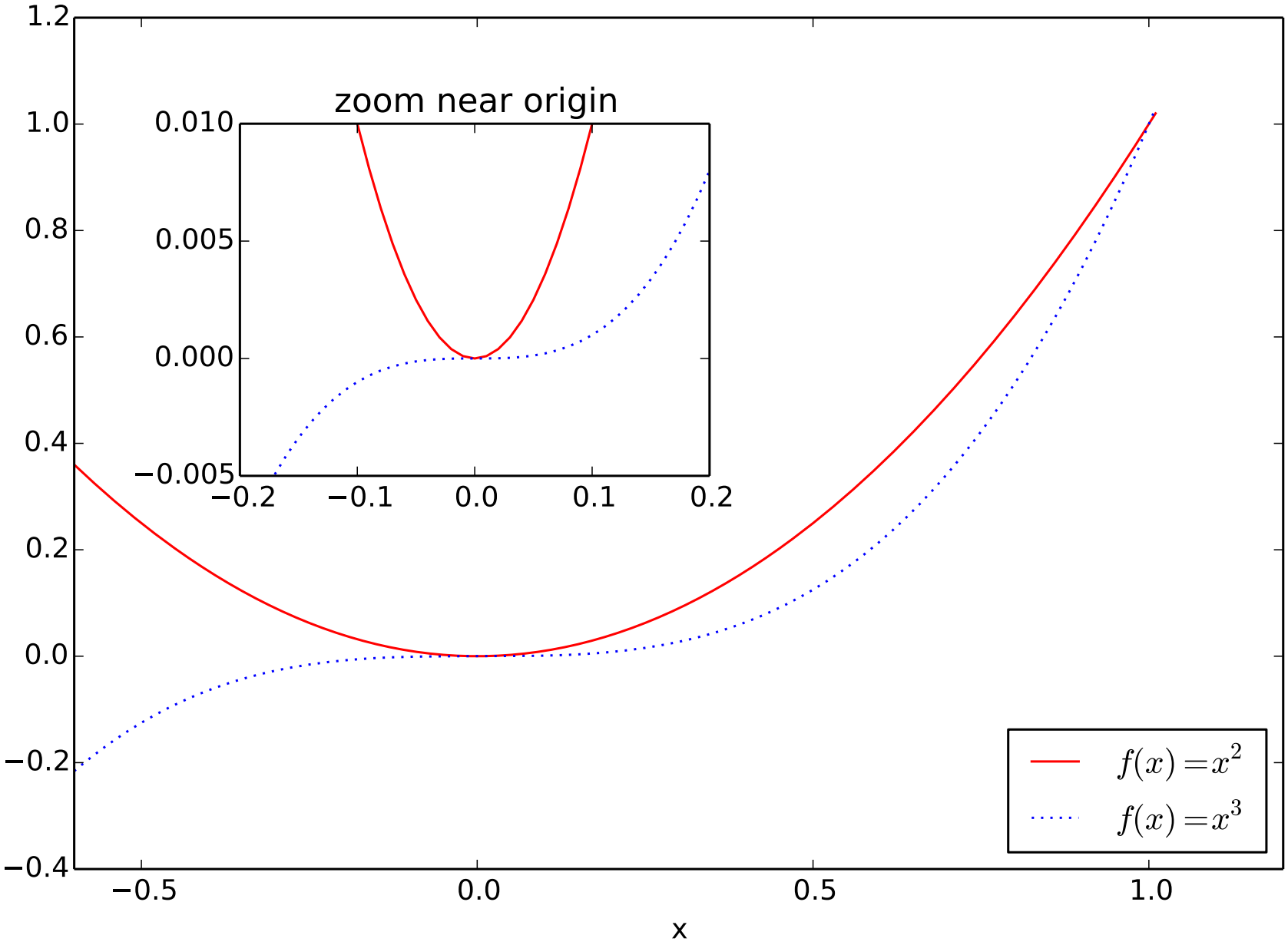
plt.subplots() with more than one subplot

```
from pylab import plt
import numpy as np

x = np.linspace(0, 2*np.pi, 200)
sinx = np.sin(x)
cosx = np.cos(x)

fig2, axmu = plt.subplots(2,2)
#create a 2 rows by 2 columns blocks for subplot.
#note axmu is 2x2 here, it can not be called as axmu[0] to axmu[3]

axmu[0,0].plot(x, sinx, 'r-.', label='sin(x)')
axmu[0,1].plot(x, cosx, 'g--', label='cos(x)')
axmu[1,0].plot(x, sinx*cosx, 'b:',  label='cos(x)sin(x)')
axmu[1,1].plot(x, sinx*sinx, 'k-',  label='sin$^2$(x)')

axmu[0][0].legend(); axmu[0][1].legend()
axmu[1][0].legend(); axmu[1][1].legend()

axmu[1,0].set_xlabel('x')
axmu[1,1].set_xlabel('x')

fig2.savefig('subplot4.pdf')

plt.show(fig2)
```


sin(x)
cos(x)
cos(x)sin(x)
sin^2 (x)
x
x

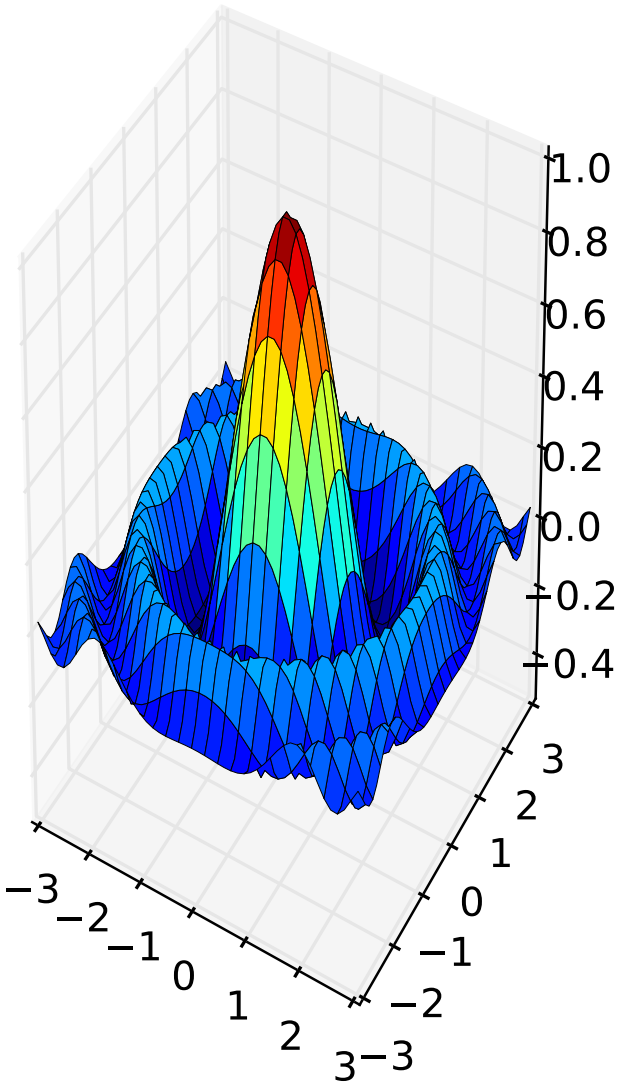
Zooming in

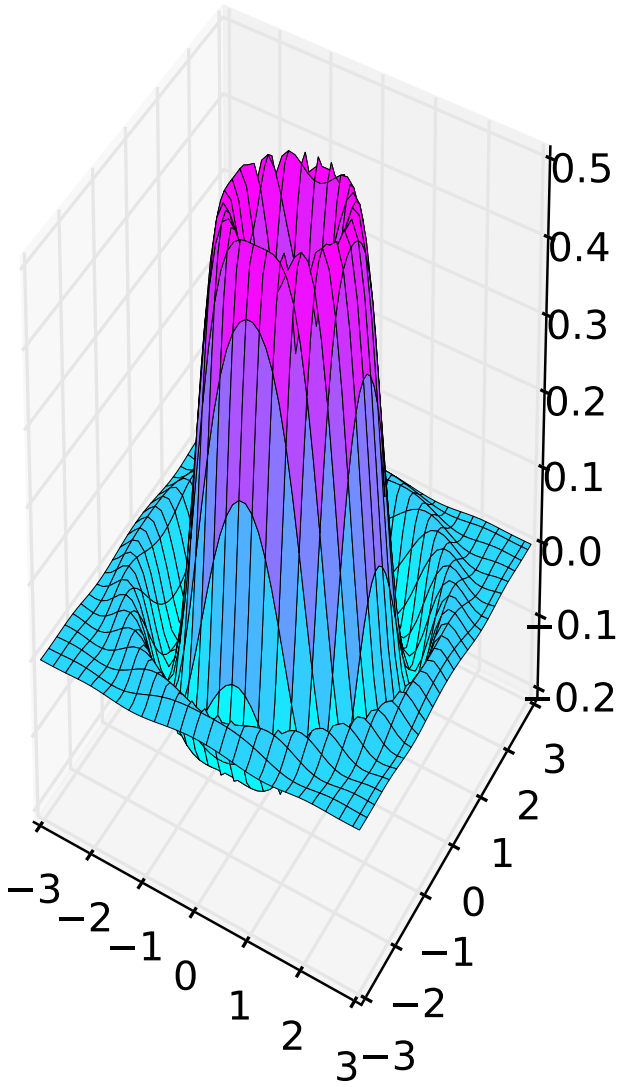
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
x = np.arange(-0.6, 1.02, 0.01)
ax.plot(x, x**2, 'r-', label='$f(x)=x^2$')
ax.plot(x, x**3, 'b:', label='$f(x)=x^3$')
ax.legend(loc='lower right')
ax.set_xlabel('x')
fig.tight_layout()

#create an inset, zoom in plots operate on this inset axis
inset_ax = fig.add_axes([0.2, 0.5, 0.35, 0.35]) #left, bottom, width, height
inset_ax.plot(x, x**2, 'r-', x, x**3, 'b:')
inset_ax.set_title('zoom near origin')

# set inset axis range
inset_ax.set_xlim(-.2, .2)
inset_ax.set_ylim(-.005, .01)

# set inset axis tick locations
inset_ax.set_yticks([-.005, 0, 0.005, 0.01])
inset_ax.set_xticks([-0.2, -0.1, 0, 0.1, .2]);

fig.savefig('zoomin.pdf')
fig.show()
```


zoom near origin
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.5
0.0
0.5
1.0
x
0.010
0.005
0.000
−0.005
−0.2
−0.1
0.0
0.1
0.2
$f(x) = x^2$
$f(x) = x^3$

3D plot

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import cm

from mpl_toolkits.mplot3d.axes3d import Axes3D    #3D toolkit

def f1(x, y):  return np.cos(x**2 + y**2) / (1. + x**2 + y**2)

def f2(x, y):  return np.exp((-x**2 - y**2)/2.) * np.sin(x**2 + y**2)

xgrid = np.linspace(-3, 3, 50)
ygrid = xgrid
x, y = np.meshgrid(xgrid, ygrid)

#this uses the Matlab plot style instead of the previous Python OOP style
fig = plt.figure()  #create a figure

ax = fig.add_subplot(121, projection='3d')
ax.plot_surface(x, y, f1(x, y), cmap=cm.jet, rstride=2, cstride=2,  lw=0.25)
ax.set_zlim(-0.5, 1.0)

ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(x, y, f2(x, y), cmap=cm.cool,  rstride=2, cstride=2, lw=0.25)
ax2.set_zlim(-.2, .5)

fig.savefig('surfaces.pdf')
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−3
−2
−1
0
1
2
3
−3
−2
−1
0
1
2
3


0.5
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−3
−2
−1
0
1
2
3
−3
−2
−1
0
1
2
3

Introduction to scipy

- Scipy is a module built on top of Numpy for array related operations. SciPy functions are often wrappers around industry-standard Fortran/C libraries (LAPACK, QUADPACK, FFTPACK, etc).
- Scipy provides modules that make many scientifically significant algorithms available at the higher-level through Scipy function calls. Useful modules include
  - Fourier transform (`scipy.fftpack`)
  - Integration (`scipy.integrate`): `quad, dblquad, tplquad, odeint, ode`
  - Interpolation (`scipy.interpolate`)
  - Linear algebra (`scipy.linalg`) (More functionalities than `numpy.linalg`)
  - Optimization (`scipy.optimize`)
  - Signal processing (`scipy.signal`)
  - Sparse matrices (`scipy.sparse`)
  - Sparse linear algebra (`scipy.sparse.linalg`)
  - Special functions (`scipy.special`)
  - Statistics (`scipy.stats`)

Note: These modules are not made automatically available after `import scipy` (thus they may not be considered as submodules of Scipy). Using any of them requires explicit import, such as `import scipy.fftpack as fft`, or `import scipy.linalg as linalg` .

A few examples of Scipy calls

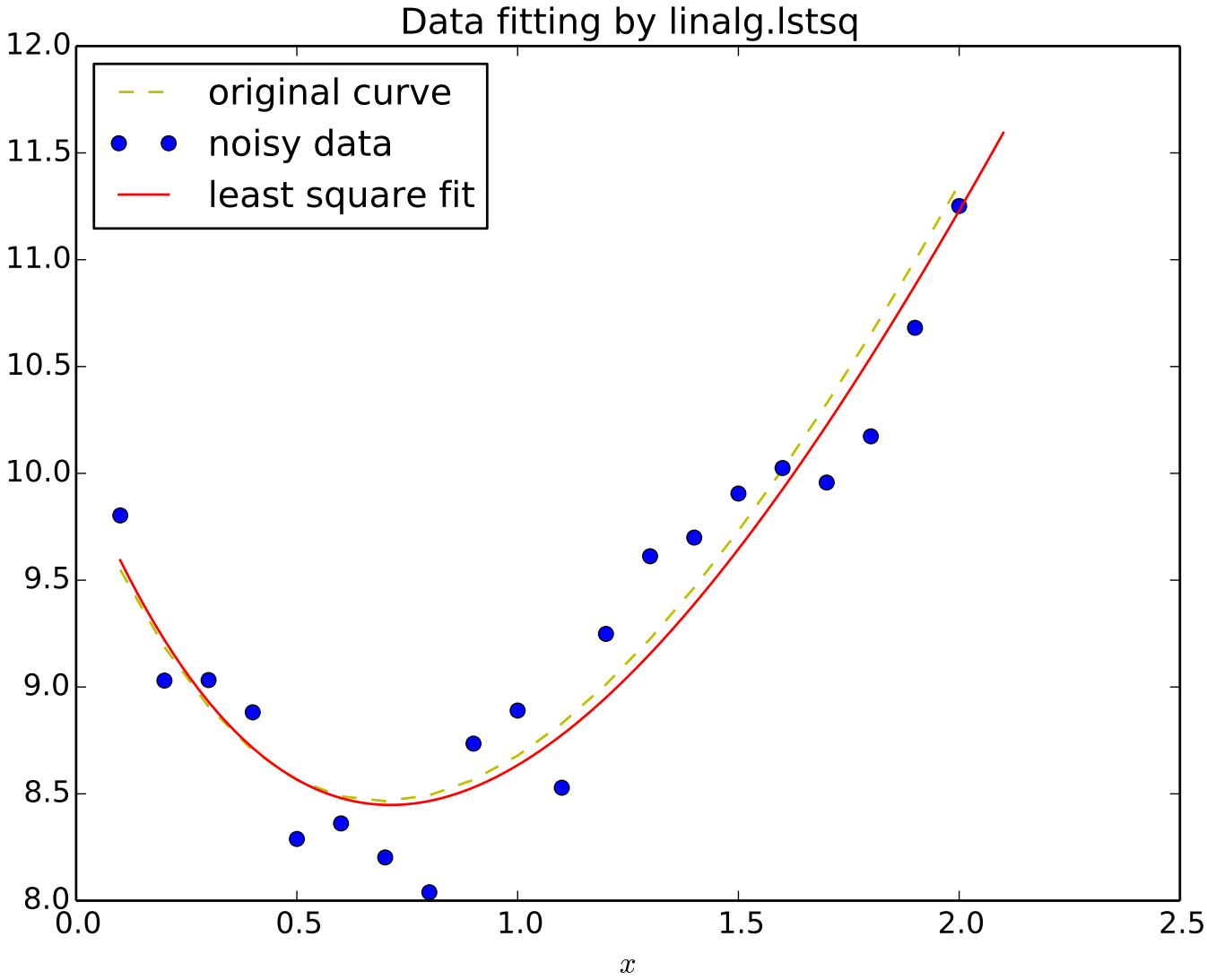
Linear algebra: `.solve()`, `.lstsq()`, `.eig()` in `scipy.linalg`.

```
import scipy.linalg as linalg
import numpy as np

b = np.random.rand(500,5)                    #generate multiple RHS vectors

A = np.random.randn(500,500)                 #generate a square array
X = linalg.solve(A, b)                       #this solves A*X = b
print('err=',linalg.norm(A.dot(X)-b, 2))    #verify that error is small

M = np.random.randn(500,200)                 #generate a non-square array
X = linalg.lstsq(M, b)           #X is a tuple, X[0] solves  min||MY-b||
Y = random.random(X[0].shape);   #X[0] contains the solution of type ndarray
#for any Y of same shape as X[0],  one should have ||MY -b|| >= ||MX[0] -b||,
#although ||MX[0] -b|| sometimes may be quite large
errY=linalg.norm(M.dot(Y)-b);  errX=linalg.norm(M.dot(X[0])-b);
print('||MY-b||={}, ||MX-b||={}'.format(errY, errX))

#solve a standard eigen-problem:   Ax= \lambda x
eva, evt = linalg.eig(A)
print('err=',  linalg.norm(A.dot(evt) - evt.dot(np.diag(eva))) )

#solve a generalized eigen-problem:   Ax= \lambda B x
B = np.random.random(A.shape)
evag, evtg = linalg.eig(A, B)
print('err=',  linalg.norm(A.dot(evtg) - B.dot(evtg).dot(np.diag(evag))) )
```

A few examples of Scipy calls

Data fitting using least square: `linalg.lstsq`

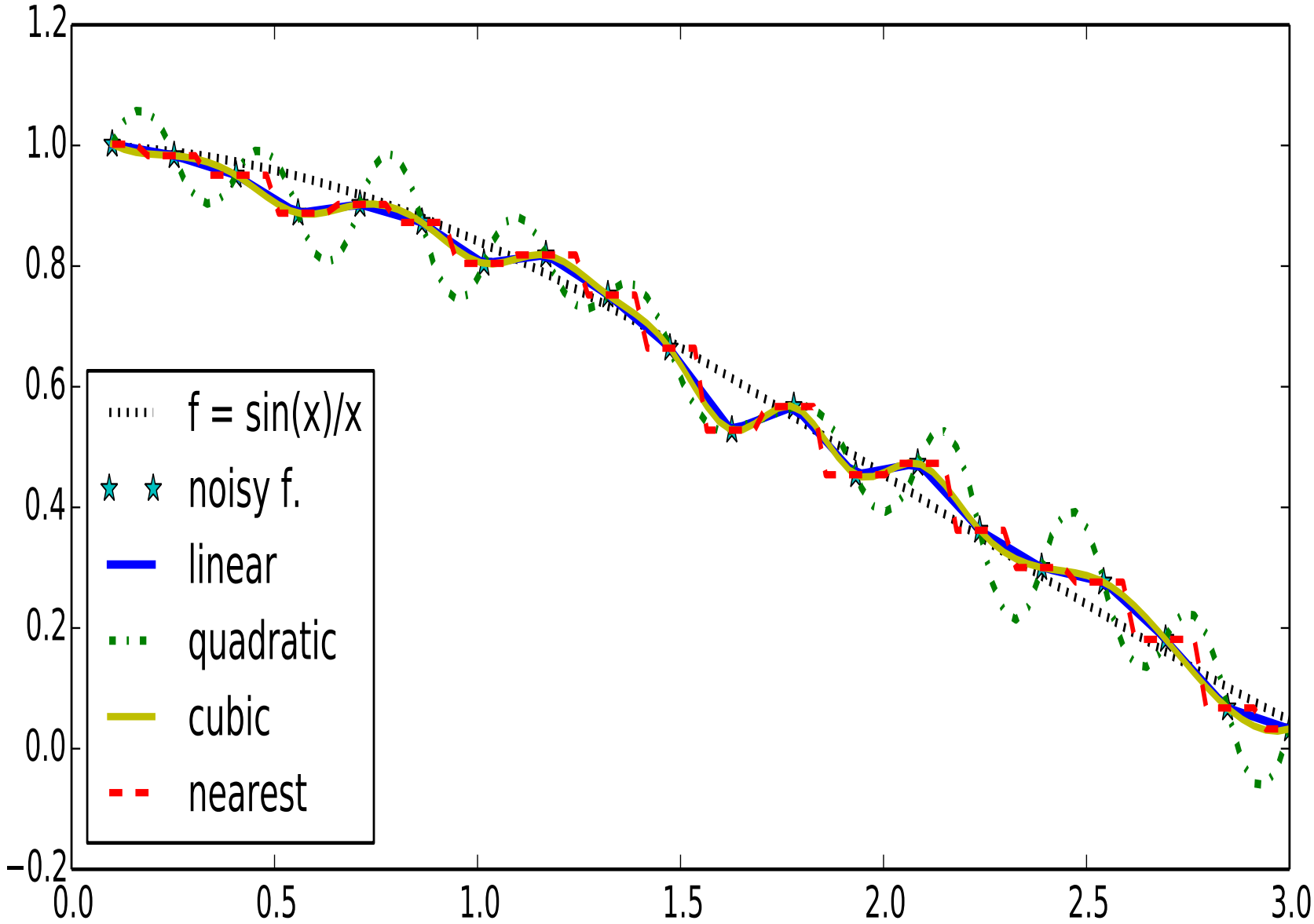
```
import scipy.linalg as linalg
import numpy as np
import matplotlib.pyplot as plt

x = np.r_[0.1:2.1:0.1]          #assign x coordinates of data points
c1, c2 = 10.0, 5.0              #assign coefficients
y = c1*np.exp(-x) + c2*x        #the curve to get the data from

#perturb y to get the noisy data points
z = y + 0.03 * np.max(y) * np.random.randn(len(y))

#construct the coefficient matrix
A = np.c_[np.exp(-x)[:, np.newaxis], x[:, np.newaxis]]

c, resid, rank, sigma = linalg.lstsq(A, z) #get the least square solution
x2 = np.r_[0.1:2.1:100j]
#use least square solution in c to generate fitting curve
y2 = c[0]*np.exp(-x2) + c[1]*x2

plt.plot(x, y, 'y--', x, z,'bo', x2, y2, 'r-')
plt.legend(('original curve', 'noisy data', 'least square fit'), loc='best')
plt.xlabel('$x$')
plt.title('Data fitting by linalg.lstsq')
plt.savefig('lsqfit.pdf')
plt.show()
```


Data fitting by linalg.lstsq
original curve
noisy data
least square fit
12.0
11.5
11.0
10.5
10.0
9.5
9.0
8.5
8.0
0.0
0.5
1.0
1.5
2.0
2.5
x

A few examples of Scipy calls

Interpolation: `interp1d` in `scipy.interpolate`

```
from scipy.interpolate  import  interp1d
import numpy as np;  import matplotlib.pyplot as plt

f = lambda x : np.sin(x)/x
x = np.linspace(0.1, 3, 20)     #choose evenly spaced points
px= np.linspace(0.1, 3, 100)    #later interpolate to more points
ymean= f(x) + 0.05*np.random.randn(len(x))  #small perturbation to f(x)
ytrue= f(px)

#get linear, quadratic,  cubic, and nearest interpolations
interp_lin = interp1d(x,  ymean)   #default: kind='linear'
interp_sq  = interp1d(x,  ymean, kind='quadratic')
interp_cub = interp1d(x,  ymean, kind='cubic')
interp_near= interp1d(x,  ymean, kind='nearest')

y_lin=interp_lin(px);  y_sq=interp_sq (px);
y_cub=interp_cub(px);  y_near= interp_near(px)

fig, ax = plt.subplots(figsize=(10,4))
ax.plot(px, ytrue, 'k:',  lw=2.5, label='f = sin(x)/x')  #lw=linewidth
ax.plot(x,  ymean, 'c*',  markersize=8, label='noisy f.')
ax.plot(px, y_lin, 'b-',  lw=2.3, label='linear')
ax.plot(px, y_sq,  'g-.', lw=2.3, label='quadratic')
ax.plot(px, y_cub, 'y-',  lw=2, label='cubic')
ax.plot(px, y_near,'r--', lw=2, label='nearest')
ax.legend(loc='lower left');
plt.savefig('interp1d.pdf');  plt.show()
```


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.5
1.0
1.5
2.0
2.5
3.0
f = sin(x)/x
noisy f.
linear
quadratic
cubic
nearest

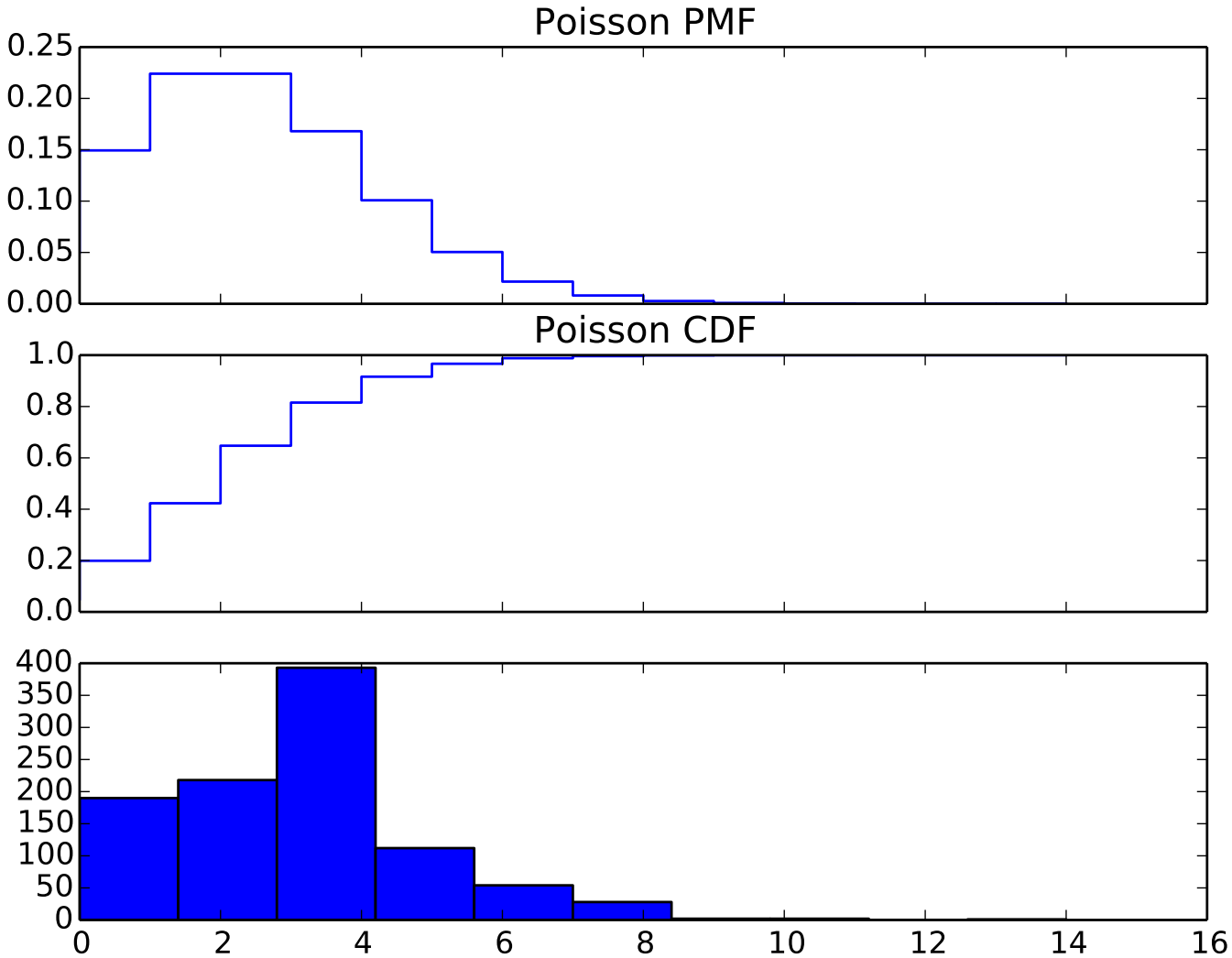
A few examples of Scipy calls

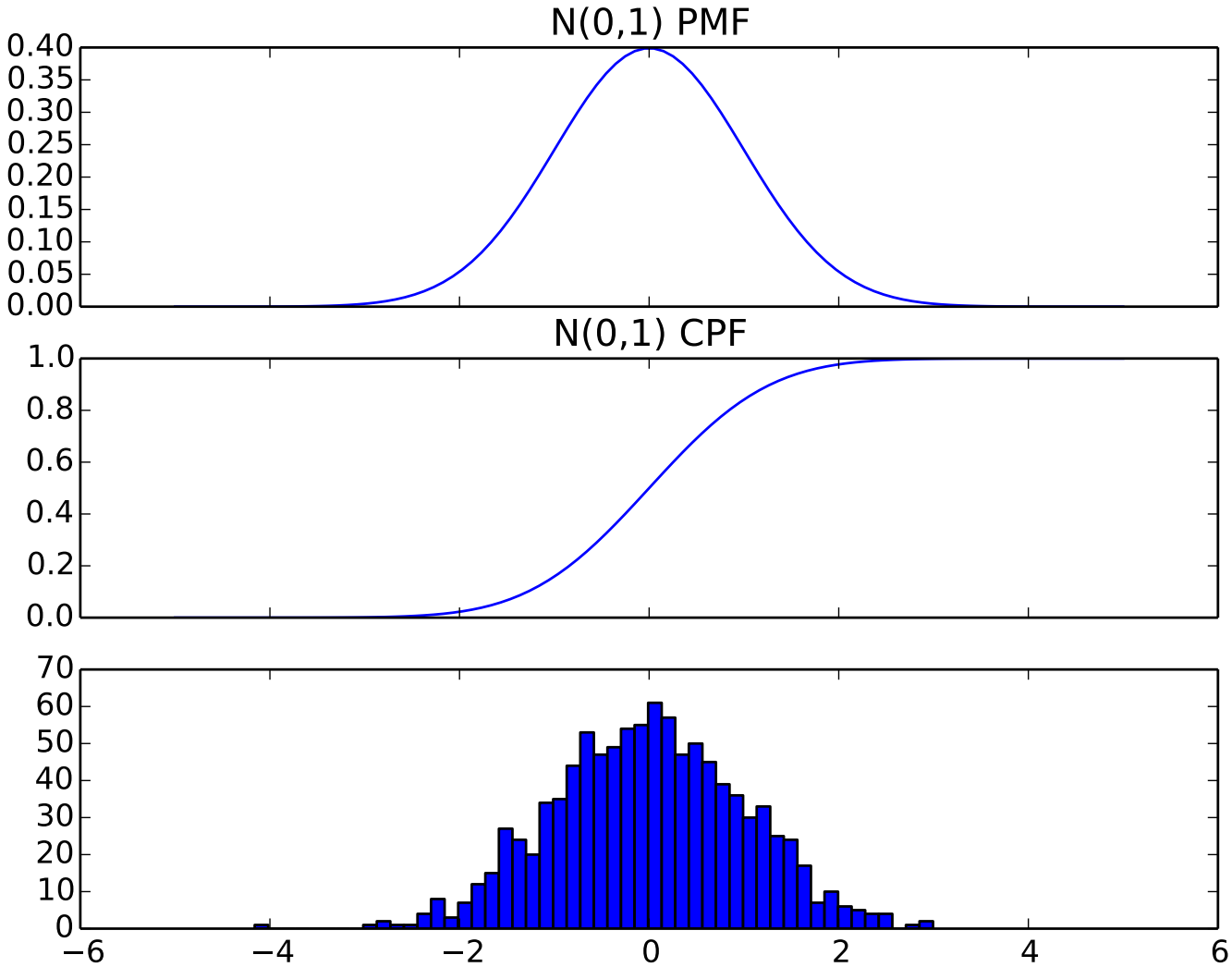
Integration: `quad, dblquad, tplquad` in `scipy.integrate`.

```
>>> from scipy.integrate import  quad,  dblquad, tplquad
>>> import numpy as np

#integrate cos(x) from 0 to pi/2
>>> val, err = quad( (lambda x: np.cos(x)),  0,  np.pi/2)
>>> print("Integral value= {:.5f},  absolute error= {:.9e}".format(val, err))
Integral value= 1.00000,  absolute error= 1.110223025e-14

>>> def integrand(x): return np.sin(x)/x
>>> val, err = quad( integrand,  0,  100)
>>> print("Integral value= {:.5f},  absolute error= {:.9e}".format(val, err))
Integral value= 1.56223,  absolute error= 3.421480815e-10

>>> invexp = lambda x: np.exp(-x)
>>> quad(invexp, 0,  np.inf)          #integrate e^(-x) from 0 to infinity
(1.0000000000000002, 5.842606742906004e-11)

>>> sqinvexp = lambda x, y: np.exp(-x**2 -y**2)
>>> dblquad(sqinvexp, 0,  np.inf,  lambda x: 0,  lambda x: np.inf)
(0.7853981633973343, 6.294671496421311e-09)   #note pi/4=0.7853981633974483

>>> tpinvexp = lambda x, y, z: np.exp(-x**2 -y**2 -z**2)
>>> low1=lambda x: 0; high1=lambda x: np.inf   #inner bounds need to be functions
>>> low2=lambda x,y:0;  high2=lambda x,y: np.inf
>>> tplquad(tpinvexp, 0,  np.inf, low1, high1, low2, high2)
(0.6960409996034802, 1.3469747760390168e-08)
```

A few examples of Scipy calls (scipy.stats)

```
from scipy import stats   #import scipy.stats as stats
import numpy as np
from pylab import plt

X = stats.poisson(3.0)    #create a r.v. with Poisson distribution
X.mean(), X.std(), X.var()

n = np.arange(0,15)
fig1, axes = plt.subplots(3,1, sharex=True)

# plot the probability mass function (PMF)
axes[0].step(n, X.pmf(n));  axes[0].set_title('Poisson PMF')
# plot the commulative distribution function (CDF)
axes[1].step(n, X.cdf(n));  axes[1].set_title('Poisson CDF')
# plot histogram of 1000 random realizations of the stochastic variable X
axes[2].hist(X.rvs(size=1000));
fig1.savefig('poisson.pdf');  plt.show()

Y = stats.norm()  #create a random variable with standard normal distribution
Y.mean(), Y.std(), Y.var()

y = np.linspace(-5,5,100)
fig2, axes = plt.subplots(3,1, sharex=True)
axes[0].plot(y, Y.pdf(y)); axes[0].set_title('N(0,1) PMF')
axes[1].plot(y, Y.cdf(y)); axes[1].set_title('N(0,1) CPF')
axes[2].hist(Y.rvs(size=1000), bins=50);
fig2.savefig('normal.pdf');  plt.show()
```


Poisson PMF
0.25
0.20
0.15
0.10
0.05
0.00
Poisson CDF
1.0
0.8
0.6
0.4
0.2
0.0
400
350
300
250
200
150
100
50
0
0
2
4
6
8
10
12
14
16


N(0,1) PMF
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
N(0,1) CPF
1.0
0.8
0.6
0.4
0.2
0.0
70
60
50
40
30
20
10
0
−6
−4
−2
0
2
4
6