Introduction to Scientific Python

Yang Liu

HPRC Short Course - Fall 2017
Contact the HPRC Helpdesk

Website: hprc.tamu.edu
Email: help@hprc.tamu.edu
Telephone: (979) 845-0219
Visit us in person: Henderson Hall, Room 114A

Help us, help you -- we need more info
- Which Cluster
- UserID/NetID
- Job id(s) if any
- Location of your jobfile, input/output files
- Application used if any
- Module(s) loaded if any
- Error messages
- Steps you have taken, so we can reproduce the problem
Slack for Python

https://pythoncs.slack.com/signup

I Create the slack workplace to discuss Python at TAMU. You need a .tamu email address to sign up.

We can discuss about python in slack.
About This Course

This course covers basic concept of

- object oriented programming
- numpy
- scipy
- matplotlib
Access Python Notebook on Titan/Crystal:
Step 1 and 2

We will mainly use notebook on titan/crystal for demonstrations and hands-on exercises. Python shells on Ada/Titan/Crystal will be used if there are problems with notebook.

Step 1: Login to https://titan.tamu.edu:8000 to access notebook for this short course.

You may connect to either titan.tamu.edu or crystal.tamu.edu to access the notebook.

You need a valid tamu Net Id and should be on campus network to connect to ada/titan/crystal. If you are off campus, you need first connect to tamu network via VPN.

Step 2: Click ‘Fall_2017_Scientific_Python’
Access Python Notebook on Titan/Crystal: Step 3 and 4

Step 3: Click ‘Scientific_Python.ipynb’

Step 4: This is the notebook for this course.
Close Python Notebook on Titan/Crystal:
Step 1 and 2

Step 3: Click ‘x’ to close the notebook

Step 1: Close the notebook

Step 4: Cling the ‘Running Tab’

Step 2: Clicking the ‘Running’ Tab
Close Python Notebook on Titan/Crystal:
Step 3 and 4

Step 3: Click ‘Shutdown’

Step 4: This is the interface after notebook shutdown

Step 3: Shutdown the notebook

Step 4: After the notebook shutdown
Object Oriented Programming

Class:
- Include both data and methods (functions)
- Used as a prototype to create instances

Instance:
- Has own data

Student
- Id
- Department
- set_department()
- display()

Student: student1
- Id = 101
- Department = 'Computer'
- set_department()
- display()

Student: student2
- Id = 125
- Department = 'Biology'
- set_department()
- display()
# Class Example: Student

```python
class Student:
    def __init__(self, name, department):
        self.name = name
        self.department = department

    def display(self):
        print('*****Student Information---begin*****
        print('Student Name: ', self.name)
        print('Department: ', self.department)
        print('*****Student Information---end*****

student_1 = Student('Mike Williams', 'Physics')
student_2 = Student('Eric Garcia', 'English')
student_1.display()
student_2.display()
```

# defines a class with name ‘Student’
# class constructor “__init__” is executed when an instance is created
# self refers to an instance itself

# instance student_1 has own data
# instance student_2 has own data
# instance student_1 can access/call class method display()
Class Variables

A class variable is shared among all instances

- When one instance changes a class variable, the other instances see the change

```
# when instance student_2 is created, the class variable count becomes 2. After that, student_1 also sees count as 2.
```

```python
class Student:
    count = 0
    def __init__(self, name, department):
        self.name = name
        self.department = department
        Student.count = Student.count + 1
    def display(self):
        print('*****Student Information ---begin*****
        print('Student Name: ', self.name)
        print('Department: ', self.department)
        print('Student Count: ', Student.count)
        print('*****Student Information ---end*****

student_1 = Student('Mike Williams', 'Physics')
student_1.display()
student_2 = Student('Eric Garcia', 'English')
student_1.display()
student_2.display()
```

# count is a class variable. To access it, use Student.count (class_name.class_variable)
Class Inheritance

A class (child) can inherit from another class (parent).

- A child can access the attributes and methods of its parent.
- A child can override the methods of its parent.
- A child can add new attributes.

```python
# Graduate_student inherits from Student
# new method and attributes of Graduate_student
# Graduate_student overrides the display() method of its parent

class Student:
    def __init__(self, name, department):
        self.name = name
        self.department = department

def display(self):
    print('*****Student Information ---begin*****')
    print('Student Name: ', self.name)
    print('Department: ', self.department)
    print('*****Student Information ---end*****')

class Graduate_Student(Student):
    def set_advisor(self, advisor):
        self.advisor = advisor

def display(self):
    print('*****Graduate Student Information ---begin*****')
    print('Student Name: ', self.name)
    print('Department: ', self.department)
    print('*****Graduate Student Information ---end*****')

graduate_student_1 = Graduate_Student('Mike Williams', 'Physics')
graduate_student_1.display()
```

```bash
[yangliu@ada2 Python_Intermediate]$ ./graduate_student.py
*****Graduate Student Information ---begin*****
(Student Name: ' Mike Williams')
(Department: ' Physics')
*****Graduate Student Information ---end*****
```
Numpy and Scipy

Numpy is a fundamental package for scientific computing in python. It provides

- multidimensional array object and various derived objects (matrices, etc.)
- routines for fast operations on array: mathematical, shape manipulation, sorting, basic linear algebra, etc.

Scipy is a collection of mathematical algorithms and functions build on Numpy. It is organized into subpackages covering various computing domains

- Cluster: clustering algorithm
- Fftpack: fast Fourier Transform routines
- Linalg: linear algebra
- Optimize: optimization and root-finding routines
- Sparse: sparse matrices and associated routines
- ...
Numpy Array

A Numpy array (ndarray class) is a table of elements of the same type

- **Axis**: dimensions
- **Rank**: number of axis (dimensions)

An ndarray object has important attributes

- `ndim`: rank
- **Size**: total number of elements
- **Shape**: a tuple of integers (the size of each dimension)
- **Dtype**: element type

```python
>>> import numpy as np
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a.ndim
2
>>> a.size
6
>>> a.shape
(2, 3)
>>> a.dtype
dtype('int64')
```
Array of Zeros/Ones

- `zeros()`: creates an array full of 0
- `ones()`: creates an array full of 1
- Data type is `float64` by default, but can be set when creating arrays.

```python
>>> a = np.zeros((3,4))
>>> a
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> b = np.ones((2, 3, 4))
>>> b
array([[[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],
       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]]])
>>> a = np.zeros((3,4), dtype=np.int16)
>>> a
array([[ 0,  0,  0,  0],
       [ 0,  0,  0,  0],
       [ 0,  0,  0,  0]], dtype=int16)
```
Array of Numbers in a Sequence

- `arange(start, end, step):` the end number is NOT included. Due to finite floating point precision, it is difficult to predict the number of elements for `arange()`. So avoid to use this function when step is not an integer.

- `linspace(start, end, num_elements):` the end number maybe included

```python
>>> np.arange(1, 10, 0.3)
array([ 1. , 1.3, 1.6, 1.9, 2.2, 2.5, 2.8, 3.1, 3.4, 3.7, 4. ,
       4.3, 4.6, 4.9, 5.2, 5.5, 5.8, 6.1, 6.4, 6.7, 7. , 7.3,
       7.6, 7.9, 8.2, 8.5, 8.8, 9.1, 9.4, 9.7])

>>> np.linspace(1, 10, 30)
array([ 1.        ,  1.31034483,  1.62068966,  1.93103448,
       2.24137931,  2.55172414,  2.86206897,  3.17241379,
       3.48275862,  3.79310345,  4.10344828,  4.4137931 ,
       4.72413793,  5.03448276,  5.34482759,  5.65517241,
       5.96551724,  6.27586207,  6.5862069 ,  6.89655172,
       7.20689655,  7.51724138,  7.82758621,  8.13793103,
       8.44827586,  8.75862069,  9.06896552,  9.37931034,
       9.68965517, 10.        ]) # creates an array of 30 elements distributed evenly from 1 to 10
```
Precision for Floating Point Numbers

A number is stored as a binary number with finite bits.

- A single-precision floating number is stored as 32 bits
- A double-precision floating number is stored as 64 bits.
- Quadruple precision is used for neutron start simulations
- A floating number maybe not the same as the stored binary number → this cause some errors

```python
>>> import numpy as np
>>> a = np.arange(7.8, 8.4, 0.05)
>>> a
array([ 7.8 ,  7.85,  7.9 ,  7.95,  8. ,  8.05,  8.1 ,  8.15,  8.2 ,
       8.25,  8.3 ,  8.35,  8.4 ])
>>> print('{0:.5f}'.format(a[12]))
8.39999999999985789145284797996282577514648437500000000
```

# Is it wrong that the end=8.4 is included in the array?
# No, it (the stored binary number) actually is not 8.4
Error for Comparing Floating Point Numbers

The result may surprise you when two floating point numbers are compared.

```python
>>> 0.1 + 0.1 + 0.1 == 0.3
False
>>> print('{0:.5f}'.format(0.1))
0.100000000000000055511151231257827021181583404541015625
>>> print('{0:.5f}'.format(0.3))
0.2999999999999999888977697537484344595763833190917968750
>>> print('{0:.5f}'.format(0.1 + 0.1 + 0.1))
0.30000000000000444089209850062616169452667236328125000
```

# Surprise! Error?

# the stored sum of 0.1 + 0.1 + 0.1 is indeed not equal to the stored number of 0.3

How to handle issues like this?

This cannot be covered in this course. You may need to study more on numeric computation. A simple approach is to claim that 0.1 + 0.1 + 0.1 == 0.3 is true if abs(0.1 + 0.1 + 0.1 – 0.3) < epsilon (a small number, e.g., 0.0000000001)
Random Array

Numpy random.random generates an array of random floating numbers between 0 and 1

```python
>>> a = np.random.random((2, 3))
>>> print(a)
[[ 0.02336401  0.19639539  0.33891337]
 [ 0.79757397  0.16154741  0.06970195]]
```

# generates a random array of shape (2,3)
Operations on Arrays

Arithmetic operations on arrays are element-wise

```python
>>> a = np.array([10, 20, 30])
>>> b = np.random.random((3))
>>> print(a)
[10 20 30]
>>> print(b)
[ 0.77840463 0.89514907 0.41390085]
>>> a + b
array([ 10.77840463, 20.89514907, 30.41390085])
>>> a - b
array([ 9.22159537, 19.10485093, 29.58609915])
>>> a < b
array([False, False, False], dtype=bool)
>>> np.sin(a)
array([-0.54402111, 0.91294525, -0.98803162])
>>> np.sqrt(a)
array([ 3.16227766, 4.47213595, 5.47722558])
```

# element-wise addition

# element-wise subtraction

# element-wise comparison

# function applied to each element
Matrix Product

Numpy ‘dot’ function generates a matrix product.

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.ones((2,2))
>>> print(a)
[[1 2]
 [3 4]]
>>> print(b)
[[ 1.  1.]
 [ 1.  1.]]
>>> a.dot(b)
array([[ 3.,  3.],
       [ 7.,  7.]])
>>> np.dot(a, b)
array([[ 3.,  3.],
       [ 7.,  7.]])
```

# 1 * 1 + 2 * 1 = 3
# 3 * 1 + 4 * 1 = 7
Array Indexing and Slicing

- An element of an array can be referred as a[i1, i2, …] (index starts from 0).

- A slicing start:end:step refers to one or more elements.

```python
>>> a = np.arange(10)
>>> a[0]
0
>>> a[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 10 is out of bounds for axis 0 with size 10
>>> b = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
>>> b[1, 3]
9
>>> b[1, ::2]
array([6, 8, 10])
>>> b[1, :4:2]
array([6, 8])
>>> b[1, 1:4:2]
array([7, 9])
```

# first element by indexing 0
# index starting from 0
# element by indexing [1,3]
# slicing: every other element in second row
# slicing: every other element until the 5th element in second row
# slicing: every other element staring from 2nd element until the 5th element in second row
Reshape Array

The shape of an array `a` can be changed by

- `a.shape = ()` – the shape of `a` is actually changed
- `a.resize()` – the shape of `a` is actually changed
- `a.reshape()` – the shape of `a` does not change

```python
>>> a = np.arange(24)
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
>>> b = a.reshape(3,8)
>>> print(b)
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]]
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
>>> a.shape = (2, 12)
>>> print(a)
[[ 0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]]
>>> a.resize(4, 6)
>>> print(a)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```
Stacking Arrays

Two arrays a and b can be stacked together

- numpy.vstack(a, b) – a and b are stacked vertically
- numpy.hstack(a, b) – a and b are stacked horizontally

```python
>>> a = np.arange(6).reshape(2,3)
>>> print(a)
[[0 1 2]
 [3 4 5]]
>>> b = np.ones((2,3))
>>> print(b)
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
>>> np.hstack((a, b))
array([[ 0.,  1.,  2.,  1.,  1.,  1.],
       [ 3.,  4.,  5.,  1.,  1.,  1.]])
>>> np.vstack((a, b))
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```
Splitting an Array

An array $a$ can be split into several smaller arrays

- `numpy.hsplit(a, 3)` – split array $a$ into 3 smaller arrays horizontally
- `numpy.vsplit(a, 3)` – split array $a$ into 3 smaller arrays vertically

```python
>>> a = np.arange(24).reshape(2,12)
>>> print(a)
[[  0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]]
>>> x, y, z = np.hsplit(a, 3)
>>> print(x)
[[ 0  1  2  3]
 [12 13 14 15]]
>>> print(y)
[[ 4  5  6  7]
 [16 17 18 19]]
>>> print(z)
[[ 8  9 10 11]
 [20 21 22 23]]
>>> x, y = np.vsplit(a, 2)
>>> print(x)
[[ 0  1  2  3  4  5  6  7  8  9 10 11]]
>>> print(y)
[[12 13 14 15 16 17 18 19 20 21 22 23]]
```

# split a into 3 smaller arrays horizontally

# split a into 3 smaller arrays vertically
Array Assignment: No Copy

Recall that assignment ‘a = b’ makes a refers to the same as b refers to: both a and b refer/point to the same object. So after the array assignment ‘a = b’, both a and b refers to the same array.

```python
>>> a = np.arange(12)
>>> b = a
>>> b is a
True
>>> id(a)
139978214113328
>>> id(b)
139978214113328
>>> b.shape = (3,4)
>>> print(a)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```
# both a and b refer to the same object

# changing the shape of a also changes the shape of b since both refers to the same object
Array View: Shallow Copy

For an array `a`, `a.view()` creates a new array object which refers to the same data of array `a`.

- Note that slicing of an array returns a view

```python
>>> a = np.arange(12)
>>> c = a.view()
>>> print(c)
[ 0  1  2  3  4  5  6  7  8  9 10 11]
>>> c.resize((3,4))
>>> print(c)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11]
>>> c[1, 2] = -3
>>> print(a)
[ 0  1  2  3  4  5  -3  7  8  9 10 11]
>>> b = c[:, 1:2]
>>> print(b)
[[1]
 [5]
 [9]]
>>> b[:] = -10
>>> print(c)
[[ 0 -10  2  3]
 [ 4 -10 -3  7]
 [ 8 -10 10 11]]
```

# `c` is a new array object, but refers to the same data of array `a`
# changing the shape of `c` does not change the shape of `a`
# changing an element of `c` also changes the same element of `a`
# slicing returns a view
Array Copy: Deep Copy

For an array a, `a.copy()` creates a new array object which has its own data, i.e., not refer to the same data of array a.

```python
>>> a = np.arange(24)
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
>>> d = a.copy()
>>> d is a
False
>>> d[5] = -10
>>> print(d)
[ 0  1  2  3  4  -10  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

# d is a new array and do not share the data of a
# changing an element of d does NOT change data of a
Array of Indexes for One Dimensional Array

For a one dimensional array a, an array of indexes from a can be used to select elements from array a to form a new array b

- Each element in the array of indexes is an index of array a
- The new array b has the same shape as the array of indexes

```python
>>> a = np.arange(12) + 10
>>> print(a)
[10 11 12 13 14 15 16 17 18 19 20 21]
>>> array_indices = np.array([0, 0, 3, 4, 3])
>>> a[array_indices]
array([10, 10, 13, 14, 13])
>>> array_indices = np.array([[0, 0, 2], [1, 3, 4]])
>>> a[array_indices]
array([[10, 10, 12],
       [11, 13, 14]])
```

# index can be reused in array of indexes.
# array of indexes is used to select elements from array a to form a new array which has the same shape as the array of indexes
Array of Indexes for Two Dimensional Array

For a two dimensional array \( a \), an array of indexes from \( a \) can be used to select elements from array \( a \) to form a new array \( b \):

- Each element in the array of indexes is the first dimension index of array \( a \)
- The new array \( b \) one more dimension than the shape of the array of indexes

```python
>>> a = np.arange(24).reshape(4,6)
>>> print(a)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
[12 13 14 15 16 17]
[18 19 20 21 22 23]]

>>> array_indices = np.array([[[0, 0, 2], [1, 3, 3]]])

>>> b = a[array_indices]

>>> print(b)
[[[ 0  1  2  3  4  5]
  [ 0  1  2  3  4  5]
  [12 13 14 15 16 17]]

[[ 6  7  8  9 10 11]
 [18 19 20 21 22 23]]]

>>> b.shape
(2, 3, 6)
```
Array of Indexes: Multidimensional Indexes

Indexes for more than one dimension in array `a` can be used to create a new array (below is an example for two-dimensional array `a`)

- Each element in `row_indexes` is the first dimension index of `a`
- Each element in `column_indexes` is the second dimension index of `a`
- `row_indexes` and `column_indexes` can be put together to form an array of index

```python
>>> import numpy as np
>>> a = np.arange(24).reshape(4,6)
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
>>> row_indexes = np.array([[0, 0], [1, 3]])
>>> column_indexes = np.array([[2, 2], [0, 4]])
>>> a[row_indexes, column_indexes]
array([[ 2,  2],
       [ 6, 22]])
```
Array Indexes: Boolean Indexes

Booleans can be used in an array of indexes to select elements from array `a`

- The array of indexes `b` should have the same shape of `a`
- An element of `a` is selected by `a[b]` if its corresponding boolean value in `b` is true

```python
>>> a = np.random.random(12).reshape(3, 4)
>>> print(a)
[[ 0.7574143  0.5185174  0.10785612  0.952208   ]
 [ 0.00364275  0.33689668  0.44678831  0.56410663   ]
 [ 0.27069099  0.91649263  0.99866993  0.1147783   ]
>>> b = a > 0.5
>>> b
array([[ True,  True, False,  True],
       [False, False, False,  True],
       [False, True, True, False]], dtype=bool)
>>> a[b]
array([ 0.7574143,  0.5185174,  0.952208 ,  0.56410663,  0.91649263,
       0.99866993])
>>> a[b] = 1
>>> print(a)
[[ 1.   1.0  0.10785612  1.   ]
 [ 0.00364275  0.33689668  0.44678831  1.   ]
 [ 0.27069099  1.   0.1147783  ]]

# creates an array of boolean indexes by comparison

# selects elements from a using the array of boolean index

# updates elements from a using the array of boolean index
Broadcasting

- Broadcasting in Numpy allows operations on arrays of different shapes

- Numpy compares the shapes of two arrays by starting with the trailing dimensions and working its way backward to ensure two arrays compatible. Two dimensions are compatible if
  - Their sizes are equal, or
  - One has a size of 1

- When one dimension has a size of 1 in an array, the array is copied along that dimension.

Examples of compatible arrays
(1) A 2 x 3 x 3
    B 2 x 3 x 1
(2) A 2 x 3 x 3
    B 3

```python
>>> a = np.arange(18).reshape(2, 3, 3)
>>> print(a)
[[[0 1 2]
  [3 4 5]
  [6 7 8]]
[[9 10 11]
 [12 13 14]
 [15 16 17]]]
>>> b = np.arange(6).reshape(2,3,1)
>>> print(b)
[[[0]
  [1]]
 [[2]]
 [[[3]
   [4]]
  [[5]]]
>>> a + b
array([[ [0, 1, 2],
         [4, 5, 6],
         [8, 9, 10]],
        [[12, 13, 14],
         [16, 17, 18],
         [20, 21, 22]]])
```
Broadcasting: Array Copy

How was the array \( b \) copied to make it compatible with \( a \)?

```python
>>> a = np.arange(18).reshape(2, 3, 3)
>>> print(a)
[[[0 1 2]
  [3 4 5]
  [6 7 8]]
[[9 10 11]
 [12 13 14]
 [15 16 17]]]
>>> b = np.arange(6).reshape(2, 3, 1)
>>> print(b)
[[[0]
  [1]
  [2]]
[[3]
 [4]
 [5]]]
>>> a + b
array([[ [0, 1, 2],
  [4, 5, 6],
  [8, 9, 10]],
[[[12, 13, 14],
 [16, 17, 18],
 [20, 21, 22]]])
```

# before the copy, array \( b \) is

\[
\begin{bmatrix}
0 \\
1 \\
2 \\
3 \\
4 \\
5
\end{bmatrix}
\]

# after the copy, array \( b \) becomes

\[
\begin{bmatrix}
[0, 0, 0],
[1, 1, 1],
[2, 2, 2]
\end{bmatrix}
\]

broadcasting
Numpy Linear Algebra

Numpy provides basic matrix operations and simple linear algebra functions and methods.

```python
>>> a = np.array([[1, 2], [3, 4]])
>>> print(a)
[[1 2]
 [3 4]]
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> np.trace(a)
5
>>> b = np.array([[10, 20], [30, 40]])
>>> print(b)
[[10 20]
 [30 40]]
>>> np.dot(a, b)
array([[ 70, 100],
       [150, 220]])
```

# generates the transpose matrix of array a

# the inv() function in linear algebra module of Numpy calculates the inverse matrix of a

# calculates the trace (sum of the elements on the main diagonal) of array a

# calculate the matrix product of array a and b
Scipy Linear Algebra

Scipy.linalg provides all functions in numpy.linalg, plus some other more advanced ones.

- Scipy.linalg is preferred unless you do not want the dependency on scipy
- Scipy.linalg is always compiled with BLAS/LAPACK support (faster), while this is optional for numpy
- All of the lapack and blas libraries are available for use

Example

\[
\begin{align*}
x + 3y &= 10 \\
2x + 5y &= 20
\end{align*}
\]

Note: Linalg.solve(A, b) is faster than linalg.inv(A).dot(b)

```python
>>> A = np.array([[1, 3], [2, 5]])
>>> print(A)
[[1 3]
 [2 5]]
>>> b = np.array([[10], [20]])
>>> print(b)
[[10]
 [20]]
>>> linalg.solve(A,b)
array([[ 10.],
       [  0.]])
```

# solve the equation Ax = b
Matplotlib

Matplotlib is a python 2D plotting library

- Produces publication quality figures
- Generates plot, histograms, power spectra, bar charts, etc.
  - Pyplot interface of matplotlib provides a Matlab-like interface
  - Full control of line styles, font properties, etc are provided via an object oriented interface or a set of functions
  - Toolkits available: basemap, cartopy, mplot3d, seaborn, ggplot, etc.

https://matplotlib.org/1.5.3/users/screenshots.html
Figure

A figure contains all plot elements.

- matplotlib.figure module provides full control of figures
- matplotlib.pyplot provides figure() to create a figure

```python
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> plt.show()
```

# import pyplot module from matplotlib
# create a new empty figure
# show the new empty figure
Plot

Matplotlib.plot() plot lines and/or markers to the current figure.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,2,3,4], [1,4, 9,16])
[<matplotlib.lines.Line2D object at 0x7faa514e64e0>]
>>> plt.plot([1,2,3,4], [1,4, 9,16], 'ro')
[<matplotlib.lines.Line2D object at 0x7faa514e6ef0>]
>>> plt.show()
```

# create and plot on the default figure, as shown in the left figure below
# plot on the default figure (override the previous one), as shown in the right figure below

# by default plot() plots a line for data
# 'ro' in the plot command creates markers (red 'o') for data
Multiple Figures

A class variable is shared among all instances.

```python
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,2,3,4], [1,4,9,16])
[<matplotlib.lines.Line2D object at 0x7faa531fb630>]
>>> plt.figure()
<matplotlib.figure.Figure object at 0x7faa531fb7f0>
>>> plt.plot([1,2,3,4], [1,4,9,16], 'ro')
[<matplotlib.lines.Line2D object at 0x7faa5314ef28>]
>>> plt.show()

# create a new figure
# make the second plot on the new/current figure
# show all figures
```
Axes

A figure can have more than one Axes. Each axes can be plotted separately.

- `pyplot.axes(left, bottom, hight, width)` creates an axes where all values are in fraction (0 to 1) coordinate.

```python
>>> import matplotlib.pyplot as plt
>>> plt.axes([0.1, 0.1, 0.3, 0.3])
<matplotlib.axes.Axes at 0x7fa5145cd68>
>>> plt.plot([1,2,3,4], [1,4, 9, 16])
[<matplotlib.lines.Line2D object at 0x7fa5142e438>]
>>> plt.axes([0.5, 0.5, 0.3, 0.3])
<matplotlib.axes.Axes at 0x7fa5142ee10>
>>> plt.plot([1,2,3,4], [1,4, 9, 16], 'ro')
[<matplotlib.lines.Line2D object at 0x7fa51466f60>]
>>> plt.show()

# create an axes for plotting

# create second axes for plotting
```
Axis

An axes has two or three axis.

- pyplot.xlim(min_x, min_y) sets the data limits on x-axis (ylim for y-axis)
- Pyplot.xlabel() sets the label for x-axis (ylabel for y-axis)
- Pyplot.xticks() sets the ticks/marks on x-axis (yticks for y-axis)

```python
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,2,3,4], [1,4,9,16])
[<matplotlib.lines.Line2D object at 0x7faa514d40b8>]
>>> plt.xlim(0, 9)
(0, 9)
>>> plt.xticks([2, 3, 4, 5])
[<matplotlib.axis.XTick object at 0x7faa513a7e48>, <matplotlib.axis.XTick object at 0x7faa513f36a0>, <matplotlib.axis.XTick object at 0x7faa5655f8d0>, <matplotlib.axis.XTick object at 0x7faa514d6438>], [a list of 4 Text xticklabel objects])
>>> plt.ylim('apples')
<matplotlib.text.Text object at 0x7faa531529e8>
>>> plt.show()
```

# sets the x range to be [0, 9]
# sets the x ticks/marks to be 2, 3, 4, and 5
# sets the y label to be ‘apples’
Legend and Title

- Pyplot.legend() adds a legend to a figure
- Pyplot.title() adds a title to a figure

```python
>>> from matplotlib import pyplot as plt
>>> from numpy.random import randn
>>> z = randn(10)
>>> red_dot, = plt.plot(z, "ro", markersize=15)
>>> blue_cross, = plt.plot(z[:5], 'b+', markeredgewidth=3, markersize=15)
>>> plt.legend([red_dot, (red_dot, blue_cross)], ["Attr A", "Attr A+B"])
<matplotlib.legend.Legend object at 0x7faa50fbc5c0>
>>> plt.title('Experiment A')
<matplotlib.text.Text object at 0x7faa51006588>
>>> plt.show()
```

# add legend by default to upper right corner
# add figure title 'Experiment A'
More on Matplotlib and Python

- https://www.labri.fr/perso/nrougier/teaching/matplotlib/
- http://nbviewer.jupyter.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-4-Matplotlib.ipynb
- http://nb.bianp.net/sort/views/