Data Science: NumPy, Matplotlib

Woong-Kee Loh

2024

Python Libraries for Data Science (1/7)

Many popular Python libraries

- NumPy
- SciPy
- Pandas
- SciKit-learn (or SKlearn)

Visualization libraries

- matplotlib
- Seaborn

and many more …

Python Libraries for Data Science (2/7)

NumPy (Numerical Python):

- introduces objects for multidimensional arrays and matrixes, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects
- many other Python libraries are built on NumPy

Python Libraries for Data Science (3/7)

SciPy:

- collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics, and more
- built on NumPy

Link: https://www.scipy.org/scipylib/

Python Libraries for Data Science (4/7)

Pandas: data frame (각 column끼리 같은 type)

- adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)
  1차원 data
- provides tools for data manipulation of table-like data: reshaping, merging, sorting, slicing, aggregation, etc.
- allows handling missing data

Link: http://pandas.pydata.org/

Python Libraries for Data Science (5/7)

SciKit-Learn:

- provides machine learning algorithms: classification, regression, clustering, model validation etc.
예측, 평가
- built on NumPy, SciPy and matplotlib

Link: http://scikit-learn.org/

Python Libraries for Data Science (6/7)

matplotlib:

- Python 2D & 3D plotting library which produces publication quality figures in a variety of hardcopy formats
- a set of functionalities similar to those of MATLAB
- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization

Link: https://matplotlib.org/

Python Libraries for Data Science (7/7)

Seaborn: 보다 간결하게

- based on matplotlib
- provides high level interface for drawing attractive statistical graphics
- similar (in style) to the popular ggplot2 library in R

Link: https://seaborn.pydata.org/

Install Numpy and MatPlotLib

- Launch *Command Prompt* as *Administrator*
- Run:
  - python -m pip install -U pip
  - python -m pip install -U numpy matplotlib

NumPy

NumPy Basics

element 똑같은 type

- Main object: homogeneous multidimensional array
  - A table of elements (usually numbers), all of the same type
  - Indexed by a tuple of positive integers
  - Dimensions are called *axes*
- Example
  - An array having 2 axes
    - First axis (row) has a length of 2; second (column) 3

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

  - A point in 3D space [1, 2, 1] has one axis, i.e., is one-dimensional
    - That axis has 3 elements, so it has a length of 3

NumPy Basics (cont'd)

- *ndarray (n-dimensional array)*: NumPy's array class
  - Also known by the alias *array*

Attributes	Description
ndarray.ndim	the number of axes (dimensions)
ndarray.shape	dimensions of the array (a tuple of integers)
ndarray.size	total number of elements of the array
ndarray.dtype	an object describing the type of the elements
ndarray.itemsize	the size in bytes of each element
ndarray.data (pointer)	the buffer containing the actual elements

Creating and Viewing an Array

- Creating an array using the arange function
- Reshaping an array using the reshape function

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
```

Creating an Array (cont'd)

- Creating an array from a list or a tuple
  - Using *array* function
  - (list) [2,3,4]
  - (tuple) (2,3,4)

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

  - Error raised when calling *array* with multiple numeric arguments

```
>>> a = np.array(1,2,3,4)     # WRONG
>>> a = np.array([1,2,3,4])   # RIGHT
```

Creating an Array (cont'd)

- Creating an array from a sequence of sequences

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

- Explicitly specifying the array data type

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Creating an Array (cont'd)

- Creating an array with placeholder content
  - When the array size is known, but not the elements.

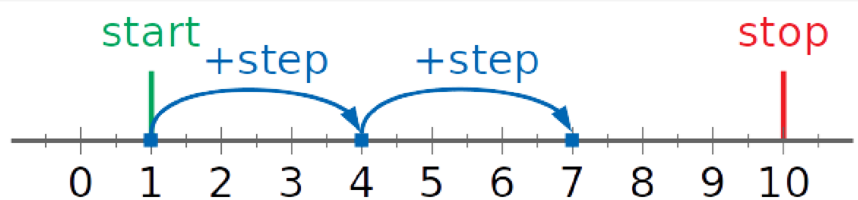
```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be specified
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]]], dtype=int16)
>>> np.empty( (2,3) )                                 # uninitialized, output may vary
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

Creating a Sequence

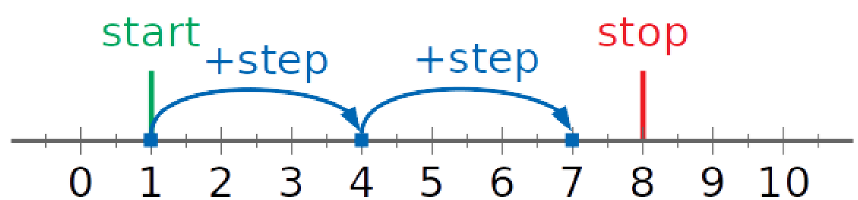
- The arange function returns an ndarray object containing evenly spaced values within a range

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

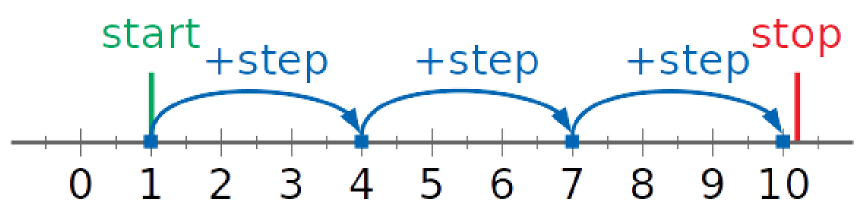
```
>>> np.arange(1, 10, 3)
array([1, 4, 7])
```




```
>>> np.arange(1, 8, 3)
array([1, 4, 7])
```




```
>>> np.arange(1, 10.1, 3)
array([1., 4., 7., 10.])
```



Creating a Sequence (cont'd)

- linspace (linear space): similar to arange

```
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lots of points
>>> f = np.sin(x)
```

Printing an Array

- Printing the layout
  - One-dimensional arrays are printed as rows, bi-dimensionals as matrixes, and tri-dimensionals as lists of matrixes

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

Printing an Array (cont'd)

- Printing a very large array
  - Automatically skips central part and prints only corners

```
>>> print(np.arange(10000))
[   0    1    2 ..., 9997 9998 9999]
>>> print(np.arange(10000).reshape(100,100))
[[   0    1    2 ...,   97   98   99]
 [ 100  101  102 ...,  197  198  199]
 [ 200  201  202 ...,  297  298  299]
 ...,
 [9700 9701 9702 ..., 9797 9798 9799]
 [9800 9801 9802 ..., 9897 9898 9899]
 [9900 9901 9902 ..., 9997 9998 9999]]
```

  - Disable this behavior and force the entire array to be printed

```
>>> np.set_printoptions(threshold=np.nan)
```

Basic Operations on Arrays

- Arithmetic operations
  - Applies element-wise on an array

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([ True, True, False, False])
```

Basic Operations on Arrays (cont'd)

- Arithmetic operations (*cont'd*)
  - Product operator * operates element-wise
  - Matrix product can be performed using the @ operator or the *dot* function or method (≥ python 3.5)

```
>>> A = np.array( [[1,1],
...               [0,1]] )
>>> B = np.array( [[2,0],
...               [3,4]] )
>>> A * B                       # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B                       # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)                    # another matrix product
array([[5, 4],
       [3, 4]])
```

Basic Operations on Arrays (cont'd)

- Operations of different types
  - Type of the resulting array corresponds to the more general or precise one

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([ 1.        ,  2.57079633,  4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

exp(x) is e^x, where *e* is Euler's number (2.7182818…)

Basic Operations on Arrays (cont'd)

- Unary operations
  - Implemented as methods of the ndarray class

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])
>>> a.sum()
2.5718191614547998
>>> a.min()
0.1862602113776709
>>> a.max()
0.6852195003967595
```

  - For *random* functions, see:
    - https://docs.scipy.org/doc/numpy/reference/routines.random.html

Basic Operations on Arrays (cont'd)

- Operation along a specified axis
  - By specifying the *axis* parameter

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                            # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                            # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                         # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Basic Operations on Arrays (cont'd)

- *Universal functions* (ufunc)
  - Mathematical functions such as sin, cos, and exp
  - Operate element-wise, and produce an array as output

```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([ 1.        ,  2.71828183,  7.3890561 ])
>>> np.sqrt(B)
array([ 0.        ,  1.        ,  1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([ 2.,  0.,  6.])
```

Indexing, Slicing and Iterating on Arrays

- To identify specific parts of an array to view or change.
- One-dimensional arrays can be indexed, sliced and iterated, much like Python lists

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6, exclusive, set
every 2nd element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,   729])
>>> a[ : :-1]                                 # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
```

Working with Multidimensional Arrays

- One index per axis; the indices are given in a tuple separated by commas

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]                       # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                        # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                      # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Working with Multidimensional Arrays (cont'd)

- Missing indexes are considered complete slices

```
>>> b[-1]                                   # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

- b[i] is treated as i followed by as many : as needed to represent the remaining axes
  - e.g., x[1] is equivalent to x[1,:,:,:,:] for x with 5 axes
- Also allows writing using *dots* (…)
  - e.g., x[1,2,...] is equivalent to x[1,2,:,:,:],
    x[...,3] to x[:,:,:,:,3] and
    x[4,...,5,:] to x[4,:,:,5,:]

Working with Multidimensional Arrays (cont'd)

- Iterating over multidimensional arrays
  - Done with respect to the first axis

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

- Iterating over all the elements
  - Use the *flat* attribute

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
...
```

Array Shape

- Given as the number of elements along each axis

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.shape
(3, 4)
```

Changing the Array Shape: ravel, reshape, transpose

- Returns a modified array, without changing the original array

```
>>> a.ravel()  # returns the array, flattened
array([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.,  8.,  9.,  3.,  6.])
>>> a.reshape(6,2)  # returns the array with a modified shape
array([[ 2.,  8.],
       [ 0.,  6.],
       [ 4.,  5.],
       [ 1.,  1.],
       [ 8.,  9.],
       [ 3.,  6.]])
>>> a.T  # returns the array, transposed
array([[ 2.,  4.,  8.],
       [ 8.,  5.,  9.],
       [ 0.,  1.,  3.],
       [ 6.,  1.,  6.]])
>>> a.T.shape
(4, 3)
>>> a.shape
(3, 4)
```

Note: ravel, flatten

- ravel, flatten
  - returns a flattened one-dimensional array
- flatten vs. ravel
  - flatten returns a copy
  - ravel returns a view of the original array whenever possible. This isn't visible in the printed output, but if you modify the array returned by ravel, it may modify the entries in the original array.
  - If you modify the entries in an array returned from flatten this will never happen.
  - ravel is faster since no memory is copied, but you have to be more careful about modifying the array it returns.

Changing the Array Shape: resize

- The *ndarray.resize* method modifies the original array

```
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.resize((2,6))
>>> a
array([[ 2.,  8.,  0.,  6.,  4.,  5.],
       [ 1.,  1.,  8.,  9.,  3.,  6.]])
```

- dimension = -1
  - The other dimensions are automatically calculated

```
>>> a.reshape(3,-1)
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
```

Stacking Arrays

- vstack (vertical stack) and hstack (horizontal stack)

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

Stacking Arrays (cont'd)

- Stacking 1D arrays
  - The function *column_stack* stacks 1D arrays as columns into a 2D array; equivalent to *hstack* only for 2D arrays
  - cf. The function *row_stack* is equivalent to *vstack* for any input arrays

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))     # with 2D arrays
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
>>> a = np.array([4.,2.])
>>> b = np.array([3.,8.])
>>> np.column_stack((a,b))     # returns a 2D array
array([[ 4., 3.],
       [ 2., 8.]])
>>> np.hstack((a,b))           # the result is different
array([ 4., 2., 3., 8.])
>>> a[:,newaxis]               # this allows to have a 2D columns vector
array([[ 4.],
       [ 2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[ 4.,  3.],
       [ 2.,  8.]])
>>> np.hstack((a[:,newaxis],b[:,newaxis]))   # the result is the same
array([[ 4.,  3.],
       [ 2.,  8.]])
```

Note: newaxis

- *newaxis* expression is used to *increase the dimension* of an existing array by *one more dimension*.
- *np.newaxis* also comes in handy when you want to convert a 1D array to either a *row vector* or a *column vector.*

```
# 1D array
arr = np.arange(4)
arr.shape
   (4,)

# make it a row vector by inserting an axis along first dimension
row_vec = arr[np.newaxis, :]
row_vec.shape
   (1, 4)

# make it a column vector by inserting an axis along second dimension
col_vec = arr[:, np.newaxis]
col_vec.shape
   (4, 1)
```

Splitting Arrays

- *Hsplit* splits an array along horizontal axis

```
>>> a = np.floor(10*np.random.random((2,12)))
>>> a
array([[ 9.,  5.,  6.,  3.,  6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 1.,  4.,  9.,  2.,  2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])
>>> np.hsplit(a,3)   # Split a into 3
[array([[ 9.,  5.,  6.,  3.],
       [ 1.,  4.,  9.,  2.]]), array([[ 6.,  8.,  0.,  7.],
       [ 2.,  1.,  0.,  6.]]), array([[ 9.,  7.,  2.,  7.],
       [ 2.,  2.,  4.,  0.]])]
>>> np.hsplit(a,(3,4))   # Split a after the third and the fourth column
[array([[ 9.,  5.,  6.],
       [ 1.,  4.,  9.]]), array([[ 3.],
       [ 2.]]), array([[ 6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])]
```

- *vsplit* splits along vertical axis, and *array_split* allows to specify along which axis to split

Object References

- No copy at all
  - Simple assignments make no copy of array objects

```
>>> a = np.arange(12)
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same ndarray object
True
>>> b.shape = 3,4    # changes the shape of a
>>> a.shape
(3, 4)
```

  - Python passes mutable objects as references, so function calls make no copy

```
>>> def f(x):
...     print(id(x))
...
>>> id(a)                           # id is a unique identifier of an object
148293216
>>> f(a)
148293216
```

View (Shallow Copy) of an Object

- Different array objects can share the same data
- The view method creates a new array object that looks at the same data

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a                        # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c.shape = 2,6                      # a's shape doesn't change
>>> a.shape
(3, 4)
>>> c[0,4] = 1234                      # a's data changes
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

View (Shallow Copy) of an Object (cont'd)

- Slicing an array returns a view of it

```
>>> s = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:,1:3]"
>>> s[:] = 10           # s[:] is a view of s. Note the difference between s=10 and s[:]=10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Deep Copy

- Makes an independent copy of an array and its data

```
>>> d = a.copy()                          # a new array object with new data is created
>>> d is a
False
>>> d.base is a                           # d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Linear Algebra

- Simple array operations

```
>>> import numpy as np
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> print(a)
[[ 1.  2.]
 [ 3.  4.]]

>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])

>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> u = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> j = np.array([[0.0, -1.0], [1.0, 0.0]])

>>> j @ j        # matrix product
array([[-1.,  0.],
       [ 0., -1.]])
```

Same as a.T for ndim≥2

Functions and Methods

- Array Creation
  - arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, r, zeros, zeros_like
- Conversions
  - ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat
- Manipulations
  - array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack

Functions and Methods (cont'd)

- Questions
  - all, any, nonzero, where
- Ordering
  - argmax, argmin, argsort, max, min, ptp, searchsorted, sort
- Operations
  - choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum
- Basic Statistics
  - cov, mean, std, var
- Basic Linear Algebra
  - cross, dot, outer, linalg.svd, vdot

Running .py Using IDLE

- In IDLE (1)
  - Open a .py program by choosing *File→ Open* menu or pressing Ctrl+O
  - Run the program by choosing *Run→ Run Module* menu or pressing F5
- In Windows file explorer
  - Right-click a .py program and choose *Edit with IDLE→ Edit with IDLE* menu
  - Run the program in IDLE
- In IDLE (2) or a Python program

```
>>> import os
>>> os.chdir("c:/work") # change to a python program folder
>>> # use slash (/) or double backslash (\\); case-insensitive
>>> exec(open("sampleData.py").read())
...
>>>
```

NumPy Exercises

NumPy: slicing rows of an array

```
import numpy as np
X = np.array ( [ [ 1, 2, 3] , [ 4, 5, 6] , [ 7, 8, 9] , ] ,
                        [ 11, 12, 13] , [ 14, 15, 16] , [ 17, 18, 19] ,
                        [ 21, 22, 23] , [ 24, 25, 26] , [ 27, 28, 29] ] )
X[1]
```

NumPy: slicing columns of an array

```
import numpy as np
X = np.array ( [ [ 1, 2, 3] , [ 4, 5, 6] , [ 7, 8, 9] , ] ,
                    [ 11, 12, 13] , [ 14, 15, 16] , [ 17, 18, 19] ,
                    [ 21, 22, 23] , [ 24, 25, 26] , [ 27, 28, 29] ] )
X[ :, 1]
```

NumPy: dicing (slicing rows and columns of an array)

```
import numpy as np
X = np.array ( [ [ 1, 2, 3] , [ 4, 5, 6] , [ 7, 8, 9] , ] ,
                  [ 11, 12, 13] , [ 14, 15, 16] , [ 17, 18, 19] ,
                  [ 21, 22, 23] , [ 24, 25, 26] , [ 27, 28, 29] ] )
print X[ 1, 1]
print X[:, 1, 1]
print X[1, :, 1]
print
print X[1:2, 1:2]
```

NumPy: Array max, min

```
import numpy as np
dataset = np.array ( [ [ 2, 4, 6, 8, 3, 2, 5] ,
                       [ 7, 5, 3, 1, 6, 8, 0] ,
                       [ 1, 3, 2, 1, 0, 0, 8] ] )
print np.max (dataset, axis=1) – np.min(dataset, axis=1)
```

NumPy: Array value normalization

```
import numpy as np
a = np.array ( [ [ 15.0, 20.0, 22.0, 75.0, 40.0, 35.0 ] )
a = a * .01
print a
```

NumPy: matrix and vector multiplication

```
import numpy as np
a = np.array ( [ [ 2, 4, 6, 8] )
b = np.array ( [ [ 1, 2, 3, 4],
                 [ 2, 3, 4, 5],
                 [ 3, 4, 5, 6],
                 [ 4, 5, 6, 7] ] )
c = np.dot (a, b)
print c
```

NumPy: matrix multiplication

```
import numpy as np
a = np.array ( [ [ 2, 4, 6, 8],
                   [ 1, 3, 5, 7] ] )
b = np.array ( [ [ 1, 2],
                   [ 2, 3],
                   [ 3, 4],
                   [ 4, 5] ] )
c = np.dot (a, b)
print c
```

Arrays and Constructors

- >>> a = ones((3,3),float)
- >>> print a
- [[1., 1., 1.],
- [1., 1., 1.],
- [1., 1., 1.]]
- >>> b = zeros((3,3),float)
- >>> b = b + 2.*identity(3) #"+" is overloaded
- >>> c = a + b
- >>> print c

Overloaded Operators

- >>> b = 2.*ones((2,2),float) #overloaded
- >>> print b
- [[2.,2.],
- [2.,2.]]
- >>> b = b+1   # Addition of a scalar is
- >>> print b   #   element-by-element
- [[3.,3.],
- [3.,3.]]
- >>> c = 2.*b # Multiplication by a scalar is
- >>> print c   #   element-by-element

Array Functions

- >>> from LinearAlgebra import *
- >>> a = zeros((3,3),float) + 2.*identity(3)
- >>> print inverse(a)
- >>> print determinant(inverse(a))
- >>> print diagonal(a)
- >>> print diagonal(a,1)

Operations on Arrays

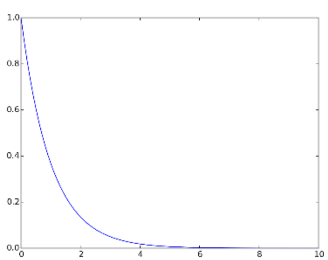
- Calculate a+b, a-b, a*b, a/b
- Save the result in c
- print c

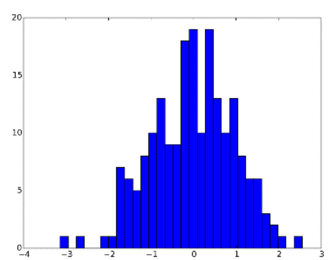
```
import numpy as np

a = np.array([1,2,3])
b = np.array([4,5,6])
```

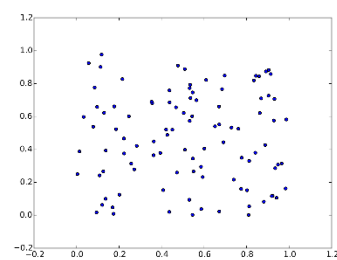
matplotlib

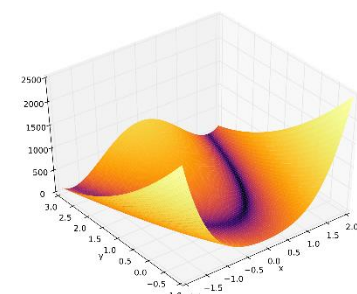
Matplotlib

- Plotting library for Python and NumPy; designed to be as usable as MATLAB
- *pyplot* is a Matplotlib module; a collection of command style functions making matplotlib work like MATLAB
- *pylab* is a module that bulk imports matplotlib.pyplot and NumPy; deprecated and its use is strongly discouraged





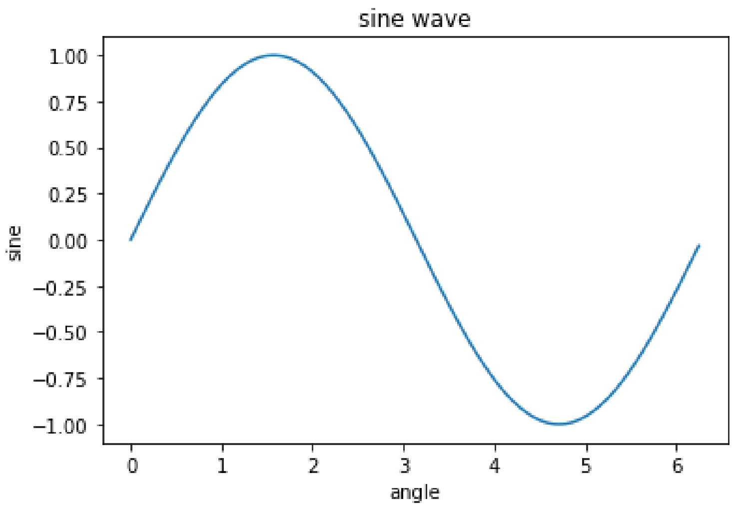




Simple Line Plot

- Angle in radians vs its sine value

```
from matplotlib import pyplot as plt
import numpy as np
import math #needed for definition of pi
x = np.arange(0, math.pi*2, 0.05)
y = np.sin(x)
plt.plot(x,y)
plt.xlabel("angle")
plt.ylabel("sine")
plt.title('sine wave')
plt.show()
```

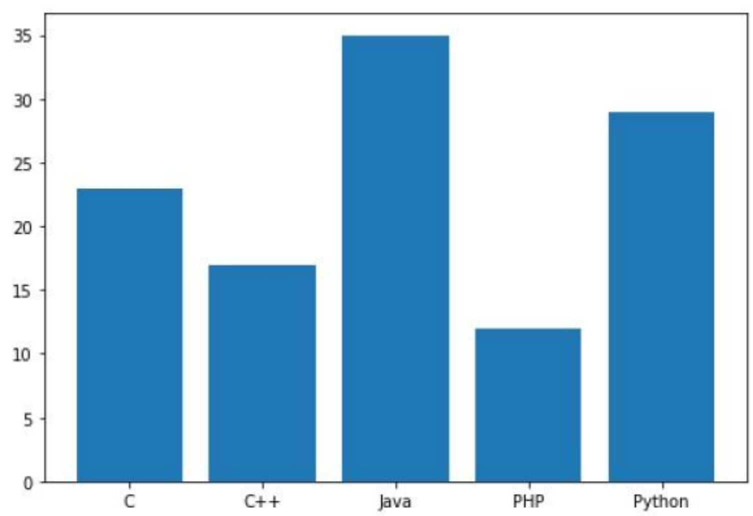


  - NOTE: See the reference for complete descriptions on pyplot functions and their parameters
  - https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html

Bar Chart

- Comparisons among discrete categories

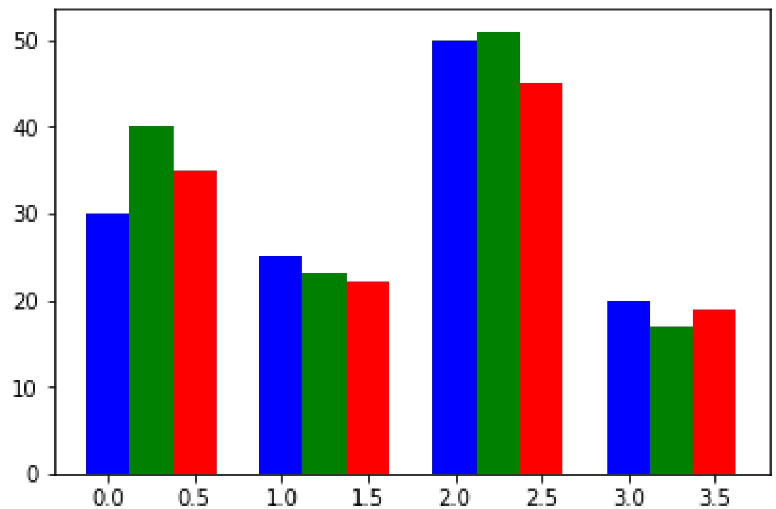
```
import matplotlib.pyplot as plt
langs = ['C', 'C++', 'Java', 'Python', 'PHP']
students = [23,17,35,29,12]
plt.bar(langs,students)
plt.show()
```



Bar Chart (cont'd)

- Compare several quantities

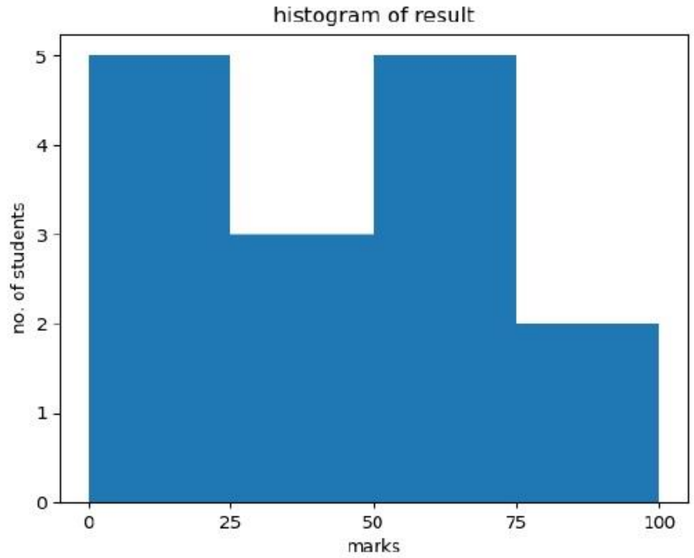
```
import numpy as np
import matplotlib.pyplot as plt
data = [[30, 25, 50, 20],
[40, 23, 51, 17],
[35, 22, 45, 19]]
X = np.arange(4)
plt.bar(X + 0.00, data[0], color = 'b', width = 0.25)
plt.bar(X + 0.25, data[1], color = 'g', width = 0.25)
plt.bar(X + 0.50, data[2], color = 'r', width = 0.25)
plt.show()
```



Histogram

- Represent the distribution of numerical data

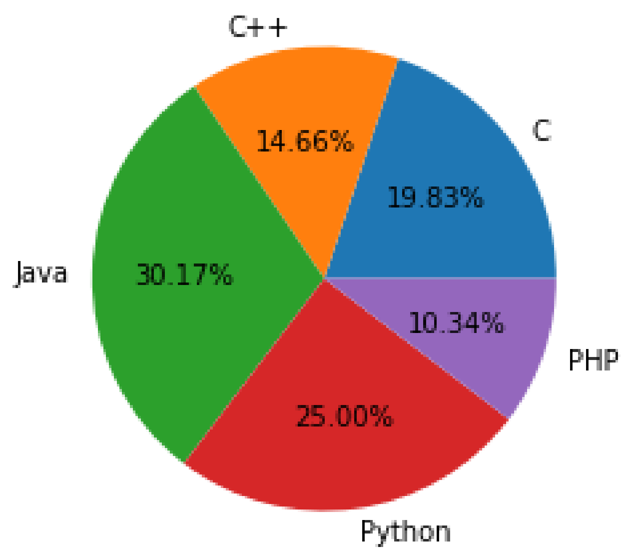
```
from matplotlib import pyplot as plt
import numpy as np
a = np.array([22,87,5,43,56,73,55,54,11,20,51,5,79,31,27])
plt.hist(a, bins = [0,25,50,75,100])
plt.title("histogram of result")
plt.xticks([0,25,50,75,100])
plt.xlabel('marks')
plt.ylabel('number of students')
plt.show()
```



Pie Chart

- Display one series (percentage) of data

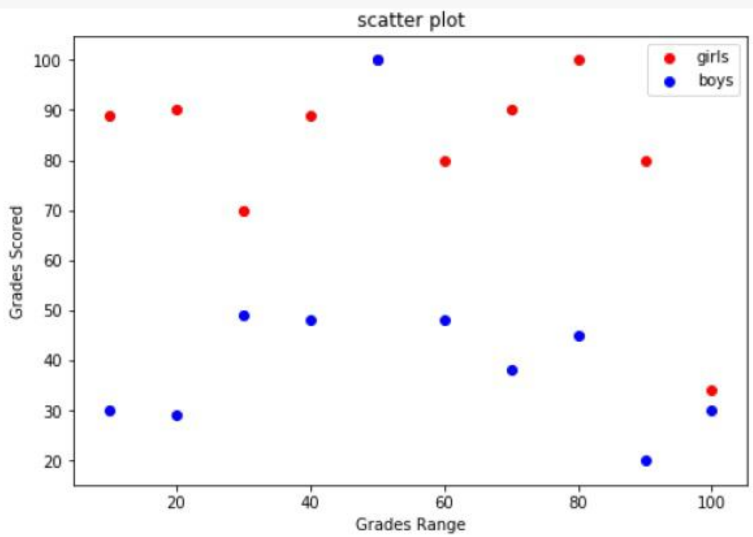
```
from matplotlib import pyplot as plt
import numpy as np
langs = ['C', 'C++', 'Java', 'Python', 'PHP']
students = [23,17,35,29,12]
plt.pie(students, labels = langs, autopct='%1.2f%%')
plt.show()
```



Scatter Plot

- Show how much one variable affects another

```
import matplotlib.pyplot as plt
girls_grades = [89, 90, 70, 89, 100, 80, 90, 100, 80, 34]
boys_grades = [30, 29, 49, 48, 100, 48, 38, 45, 20, 30]
grades_range = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
plt.scatter(grades_range, girls_grades, color='r')
plt.scatter(grades_range, boys_grades, color='b')
plt.xlabel('Grades Range')
plt.ylabel('Grades Scored')
plt.title('scatter plot')
plt.show()
```

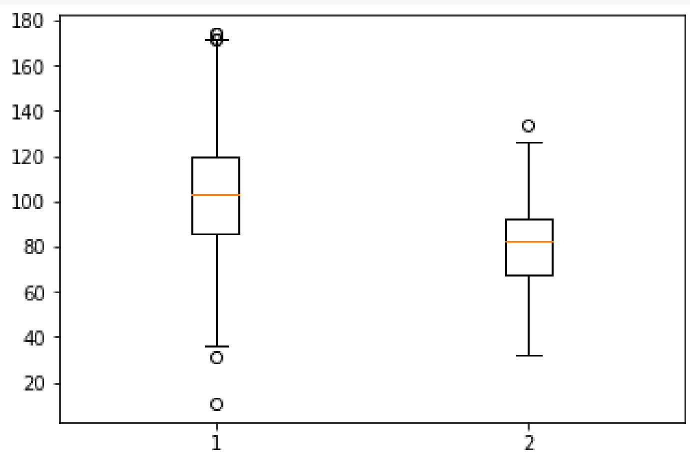


Box Plot

- Display a summary of a set of data
  - Minimum, first quartile, median, third quartile, and maximum

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(10)
dataSet1 = np.random.normal(100, 30, 200)
dataSet2 = np.random.normal( 80, 20, 200)
plotData = [dataSet1, dataSet2]
plt.boxplot(plotData)
plt.show()
```

μ = 100, σ = 30, size = 200



NumPy Exercise-1

- We want to compute the BMI (body mass index) of 100 students.
  - BMI = weight / (height * height)
    (* weight in kilograms, height in meters *)
- Create a wt array and an ht array, each of size 100.
  - Fill the wt array with 100 random float numbers between 40.0 and 90.0.
  - Fill the ht array with 100 random integers between 140 and 200 (centimeters).
- Compute the BMI for the 100 students, store them in a bmi array, and print the array.
- Post the screen of the Python/NumPy code and the first 10 elements of the bmi array to CyberCampus.

MatPlotLib Exercise-2

- Draw the bar chart, histogram, pie chart, and scatter plot of the (height, weight) data in the NumPy exercise. (Use 4 categories for the BMI index)

BMI	Weight status
Below 18.5	Underweight
18.5–24.9	Healthy
25.0–29.9	Overweight
30.0 and above	Obese

- Post the screen of the Python/MatPlotLib code and the plots to CyberCampus.

MatPlotLib Exercise-2 (cont'd)

- Bar chart
  - Plot the student distribution for each bmi level (#bars = 4)
- Histogram
  - Plot the student distribution for each bmi level (#bins = 4)
- Pie chart
  - Plot the ratio of students for each bmi level
- Scatter plot
  - Plot (height, weight) points

End of Lesson

- Acknowledgements
  - Original sources of this presentation
  - https://docs.scipy.org/doc/numpy/user/quickstart.html
  - https://www.tutorialspoint.com/matplotlib/index.htm
- See also
  - https://docs.scipy.org/doc/numpy/reference/index.html
  - https://www.tutorialspoint.com/numpy/index.htm
  - https://matplotlib.org/api/_as_gen/matplotlib.pyplot.htm
  - https://matplotlib.org/tutorials/index.html