NUS AI SUMMER EXPERIENCE 2019

NumPy & Matplotlib

PAN BINBIN

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF ISEM

Numeric and scientific applications

- As you might expect, there are a number of third-party packages available for numerical and scientific computing that extend Python's basic math module.
- These include:
- NumPy/SciPy – numerical and scientific function libraries.
- Numba – Python compiler that support JIT compilation.
- ALGLIB – numerical analysis library.
- Pandas – high-performance data structures and data analysis tools.
- PyGSL – Python interface for GNU Scientific Library.
- ScientificPython – collection of scientific computing modules.

Scipy and friends

- By far, the most commonly used packages are those in the SciPy stack. We will focus on these in this class. These packages include:
- NumPy
- SciPy
- Matplotlib – plotting library.
- IPython – interactive computing.
- Pandas – data analysis library.
- SymPy – symbolic computation library.

numpy

- Let's start with NumPy. Among other things, NumPy contains:
- A powerful N-dimensional array object.
- Sophisticated (broadcasting/universal) functions.
- Tools for integrating C/C++ and Fortran code.
- Useful linear algebra, Fourier transform, and random number capabilities.
- Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

NUMPY

- The key to NumPy is the ndarray object, an *n*-dimensional array of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:
- NumPy arrays have a fixed size. Modifying the size means creating a new array.
- NumPy arrays must be of the same data type, but this can include Python objects.
- More efficient mathematical operations than built-in sequence types.

Numpy datatypes

To begin, NumPy supports a wider variety of data types than are built-in to the Python language by default. They are defined by the numpy.dtype class and include:

- intc (same as a C integer) and intp (used for indexing)
- int8, int16, int32, int64
- uint8, uint16, uint32, uint64
- float16, float32, float64
- complex64, complex128
- bool_, int_, float_, complex_ are shorthand for defaults.

These can be used as functions to cast literals or sequence types, as well as arguments to numpy functions that accept the dtype keyword argument.

Numpy datatypes


```
import numpy as np
x = np.float32(1.0)
x
```

```
1.0
```

```
y = np.int_([1,2,4])
y
```

```
array([1, 2, 4])
```

```
z = np.arange(3, dtype=np.uint8)
z
```

```
array([0, 1, 2], dtype=uint8)
```

```
z.dtype
```

```
dtype('uint8')
```


NUMPY ARRAYS

- There are a couple of mechanisms for creating arrays in NumPy:
- Conversion from other Python structures (e.g., lists, tuples).
- Built-in NumPy array creation (e.g., arange, ones, zeros, etc.).
- Reading arrays from disk, either from standard or custom formats (e.g. reading in from a CSV file).
- and others …

NUMPY ARRAYS

- In general, any numerical data that is stored in an array-like container can be converted to an ndarray through use of the array() function. The most obvious examples are sequence types like lists and tuples.

```
x = np.array([2,3,1,0])
x = np.array([2, 3, 1, 0])
x = np.array([[1,2.0],[0,0],(1+1j,3.)])
x = np.array([[ 1.+0.j, 2.+0.j], [ 0.+0.j, 0.+0.j], [ 1.+1.j, 3.+0.j]])
```


NUMPY ARRAYS

- There are a couple of built-in NumPy functions which will create arrays from scratch.
- zeros(shape) -- creates an array filled with 0 values with the specified shape. The default dtype is float64.
- ones(shape) -- creates an array filled with 1 values.
- arange() -- creates arrays with regularly incrementing values.

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.arange(2, 10, dtype=np.float)
```

```
array([ 2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

```
np.arange(2, 3, 0.1)
```

```
array([ 2. ,  2.1,  2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9])
```

NUMPY ARRAYS

- arange() -- creates arrays with regularly incrementing values.

```
np.arange(2, 3, 0.1)
```

```
array([ 2. ,  2.1,  2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9])
```

- linspace() -- creates arrays with a specified number of elements, and spaced equally between the specified beginning and end values.

```
np.linspace(1., 4., 6)
```

```
array([ 1. ,  1.6,  2.2,  2.8,  3.4,  4. ])
```

- random.random(shape) – creates arrays with random floats over the interval [0,1).

```
np.random.random((2,3))
```

```
array([[ 0.05951219,  0.79863919,  0.7177081 ],
       [ 0.00493471,  0.76923206,  0.83393762]])
```

-

NUMPY ARRAYS ATTRIBUTES

- ndarray.ndim
  - the number of axes (dimensions) of the array i.e. the rank.
- ndarray.shape
  - the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the rank, or number of dimensions, ndim.
- ndarray.size
  - the total number of elements of the array, equal to the product of the elements of shape.

NUMPY ARRAYS ATTRIBUTES

- ndarray.dtype
  - an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. NumPy provides many, for example bool_, character, int_, int8, int16, int32, int64, float_, float8, float16, float32, float64, complex_, complex64, object_.
- ndarray.itemsize
  - the size in bytes of each element of the array. E.g. for elements of type float64, itemsize is 8 (=64/8), while complex32 has itemsize 4 (=32/8) (equivalent to ndarray.dtype.itemsize).
- ndarray.data
  - the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

NUMPY ARRAYS INDEXING

- Single-dimension indexing

```
x = np.arange(10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
x[2]
```

```
2
```

```
x[-2]
```

```
8
```

- Multi-dimensional indexing.

```
x.shape=(2,5)
x
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

```
x[1,3]
```

```
8
```

```
x[1,-1]
```

```
9
```

SUBARRAY


```
x
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

```
x[0]
```

```
array([0, 1, 2, 3, 4])
```

```
x.shape
```

```
(2, 5)
```

```
x[0].shape
```

```
(5,)
```


SLICING


```
y = np.arange(35).reshape(5,7)
y
```

```
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, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
```

```
y[1:5:2,::3]
```

```
array([[ 7, 10, 13],
       [21, 24, 27]])
```

ARRAY OPERATIONS


```
a = np.arange(5)
b = np.arange(5)
a+b
```

```
array([0, 2, 4, 6, 8])
```

```
a-b
```

```
array([0, 0, 0, 0, 0])
```

```
a**2
```

```
array([ 0,  1,  4,  9, 16], dtype=int32)
```

```
a>3
```

```
array([False, False, False, False,  True], dtype=bool)
```

```
a*b
```

```
array([ 0,  1,  4,  9, 16])
```

- Basic operations apply element-wise. The result is a new array with the resultant elements.

  Operations like *= and += will modify the existing array.

IN-BUILT METHODS


```
a = np.random.random((2,3))
a
```

```
array([[ 0.05470163,  0.71011834,  0.08593443],
       [ 0.82096601,  0.87886273,  0.50198452]])
```

```
a.sum()
```

```
3.0525676593169293
```

```
a.min()
```

```
0.054701632750315032
```

```
a.max(axis=0)
```

```
array([ 0.82096601,  0.87886273,  0.50198452])
```

```
a.max(axis=1)
```

```
array([ 0.71011834,  0.87886273])
```

- Max
- Min
- Sum,
- etc

```
a = np.arange(24)
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])
```

```
a.shape = (2,12)
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]])
```

```
a.reshape(4,6)
```

```
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]])
```

```
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]])
```

- An array shape can be manipulated by a number of methods.

  resize(size) will modify an array in place.

  reshape(size) will return a copy of the array with a new shape.


```
a.resize(4,6)
```

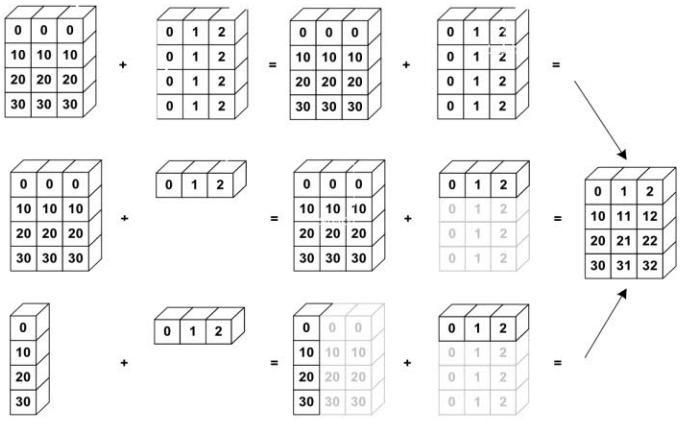
```
a
```

```
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]])
```

- resize(size) will modify an array in place.

  reshape(size) will return a copy of the array with a new shape.

BROADCASTING



- Operations on arrays of different sizes is possible if Numpy can transform these arrays so that they all have the same size: this conversion is called broadcasting.

BROADCASTING


```
a = np.tile(np.arange(0, 40, 10), (3, 1)).T
a
```

```
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
```

```
b = np.array([0, 1, 2])
b
```

```
array([0, 1, 2])
```

```
a + b
```

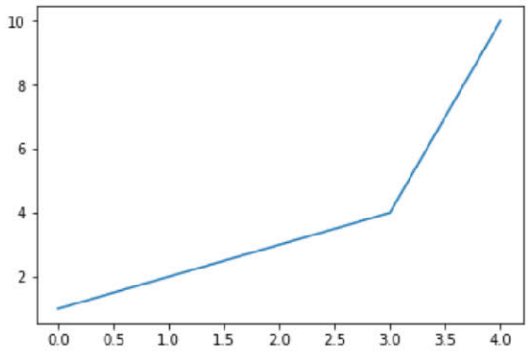
```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Matplotlib

- Important library of Python
- Useful for different types of charts

```
import matplotlib.pyplot as plt
%matplotlib inline
# Plot a line
plt.plot([1,2,3,4,10])
```

```
[<matplotlib.lines.Line2D at 0x15878a62358>]
```



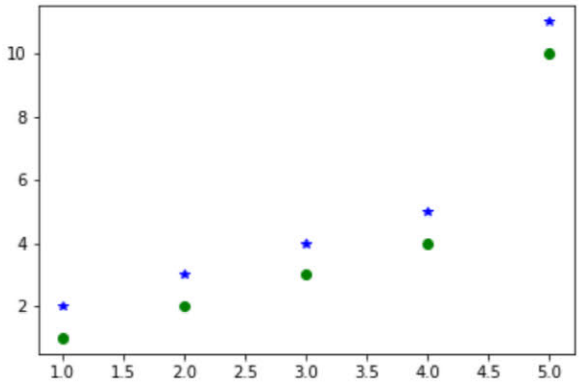
- A basis Scatter Plot with pyplot
- .plot() assumed the values of the X-items in the data.
- If you only want to see the plot, add lines in one shot.

plt.plot()

- The plt.plot accepts 3 basic arguments in the following order: (x, y, format).
- This format is a short hand combination of {color}{marker}{line}.
- the format 'go-' has 3 characters standing for: 'green colored dots with solid line'. By omitting the line part ('-') in the end, you will be left with only green dots ('go'), which makes it draw a scatterplot.
- Examples
  - 'r*--' : 'red stars with dashed lines'
  - 'ks.' : 'black squares with dotted line' ('k' stands for black)
  - 'bD-.' : 'blue diamonds with dash-dot line'.

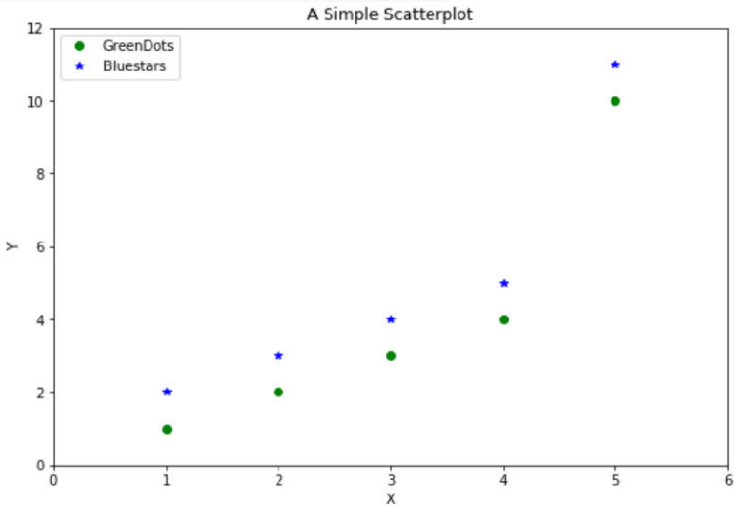
Two scatter plots in one figure


```
# Draw two sets of points
plt.plot([1,2,3,4,5], [1,2,3,4,10], 'go')  # green dots
plt.plot([1,2,3,4,5], [2,3,4,5,11], 'b*')  # blue stars
plt.show()
```



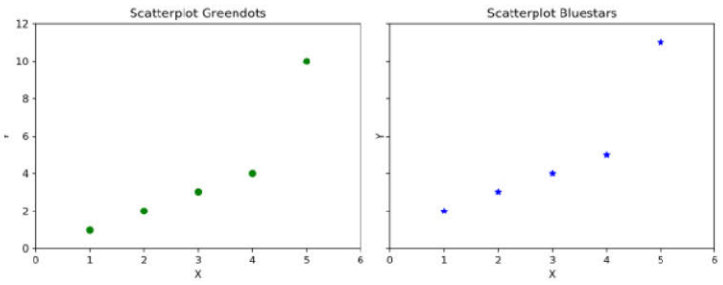
Two scatter plots in one figure

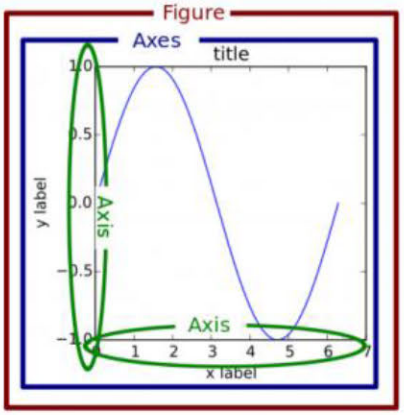

```
plt.figure(figsize=(9,6)) # 10 is width, 7 is height
plt.plot([1,2,3,4,5], [1,2,3,4,10], 'go', label='GreenDots')  # green dots
plt.plot([1,2,3,4,5], [2,3,4,5,11], 'b*', label='Bluestars')  # blue stars
plt.title('A Simple Scatterplot')
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim(0, 6)
plt.ylim(0, 12)
plt.legend(loc='best')
plt.show()
```



2 scatterplots in different panels

- Subplot (axes)





Two scatter plots in one figure


```
# Create Figure and Subplots
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(10,4), sharey=True, dpi=120)

# Plot
ax1.plot([1,2,3,4,5], [1,2,3,4,10], 'go')  # greendots
ax2.plot([1,2,3,4,5], [2,3,4,5,11], 'b*')  # bluestart

# Title, X and Y labels, X and Y Lim
ax1.set_title('Scatterplot Greendots'); ax2.set_title('Scatterplot Bluestars')
ax1.set_xlabel('X');  ax2.set_xlabel('X')  # x label
ax1.set_ylabel('Y');  ax2.set_ylabel('Y')  # y label
ax1.set_xlim(0, 6) ;  ax2.set_xlim(0, 6)   # x axis limits
ax1.set_ylim(0, 12);  ax2.set_ylim(0, 12)  # y axis limits

# ax2.yaxis.set_ticks_position('none')
plt.tight_layout()
plt.show()
```

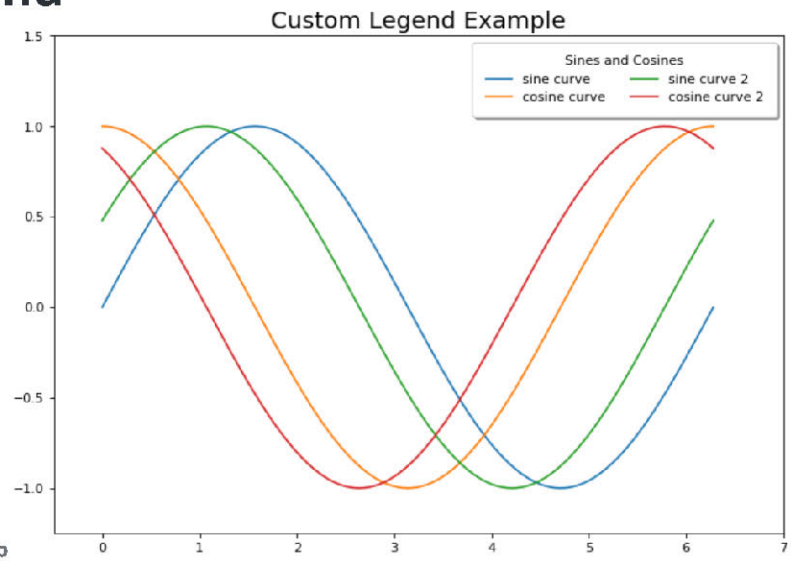
Plt vs subplot

- plt.xlabel() → ax.set_xlabel()
- plt.ylabel() → ax.set_ylabel()
- plt.xlim() → ax.set_xlim()
- plt.ylim() → ax.set_ylim()
- plt.title() → ax.set_title()

Set Legend


```
plt.figure(figsize=(10,7), dpi=80)
X = np.linspace(0, 2*np.pi, 1000)
sine = plt.plot(X,np.sin(X)); cosine = plt.plot(X,np.cos(X))
sine_2 = plt.plot(X,np.sin(X+.5)); cosine_2 = plt.plot(X,np.cos(X+.5))
plt.gca().set(ylim=(-1.25, 1.5), xlim=(-.5, 7))
plt.title('Custom Legend Example', fontsize=18)

# Modify legend
plt.legend([sine[0], cosine[0], sine_2[0], cosine_2[0]],   # plot items
           ['sine curve', 'cosine curve', 'sine curve 2', 'cosine curve 2'],
           frameon=True,                                    # legend border
           framealpha=1,                                    # transparency of border
           ncol=2,                                          # num columns
           shadow=True,                                     # shadow on
           borderpad=1,                                     # thickness of border
           title='Sines and Cosines')                       # title
plt.show()
```

Set Legend



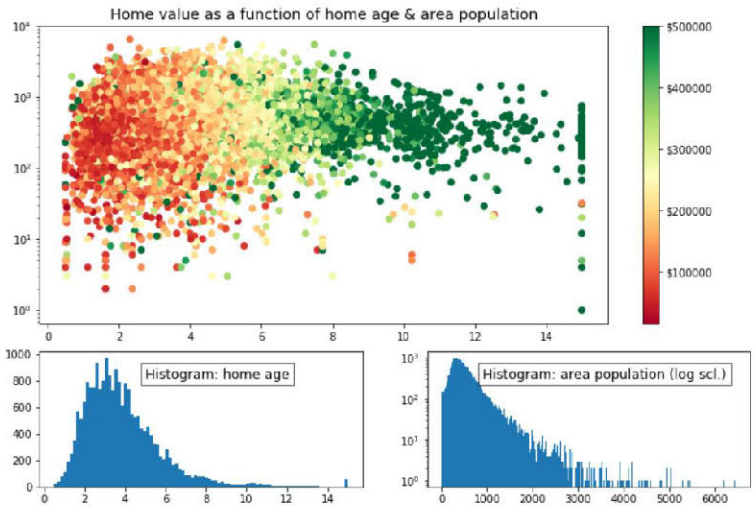
Multi Plots


```
from io import BytesIO
import tarfile
from urllib.request import urlopen

url = 'http://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.tgz'
b = BytesIO(urlopen(url).read())
fpath = 'CaliforniaHousing/cal_housing.data'

with tarfile.open(mode='r', fileobj=b) as archive:
    housing = np.loadtxt(archive.extractfile(fpath), delimiter=',')

y = housing[:, -1]
pop, age = housing[:, [4, 7]].T

def add_titlebox(ax, text):
    ax.text(.55, .8, text,
        horizontalalignment='center',
        transform=ax.transAxes,
        bbox=dict(facecolor='white', alpha=0.6),
        fontsize=12.5)
    return ax

gridsize = (3, 2)
fig = plt.figure(figsize=(12, 8))
ax1 = plt.subplot2grid(gridsize, (0, 0), colspan=2, rowspan=2)
ax2 = plt.subplot2grid(gridsize, (2, 0))
ax3 = plt.subplot2grid(gridsize, (2, 1))
ax1.set_title('Home value as a function of home age & area population',
              fontsize=14)
sctr = ax1.scatter(x=age, y=pop, c=y, cmap='RdYlGn')
plt.colorbar(sctr, ax=ax1, format='$%d')
ax1.set_yscale('log')
ax2.hist(age, bins='auto')
ax3.hist(pop, bins='auto', log=True)

add_titlebox(ax2, 'Histogram: home age')
add_titlebox(ax3, 'Histogram: area population (log scl.)')
```

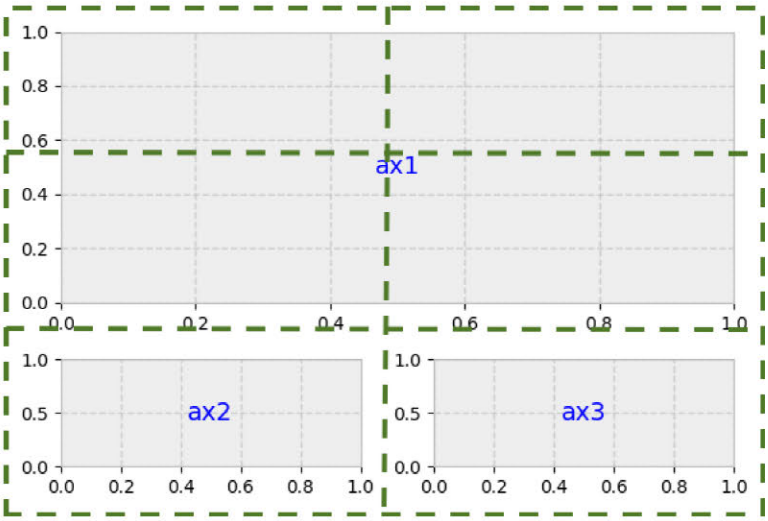
Multi Plots


```
from io import BytesIO
import tarfile
from urllib.request import urlopen

url = 'http://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.tgz'
b = BytesIO(urlopen(url).read())
fpath = 'CaliforniaHousing/cal_housing.data'

with tarfile.open(mode='r', fileobj=b) as archive:
    housing = np.loadtxt(archive.extractfile(fpath), delimiter=',')

y = housing[:, -1]
pop, age = housing[:, [4, 7]].T

def add_titlebox(ax, text):
    ax.text(.55, .8, text,
        horizontalalignment='center',
        transform=ax.transAxes,
        bbox=dict(facecolor='white', alpha=0.6),
        fontsize=12.5)
    return ax

gridsize = (3, 2)
fig = plt.figure(figsize=(12, 8))
ax1 = plt.subplot2grid(gridsize, (0, 0), colspan=2, rowspan=2)
ax2 = plt.subplot2grid(gridsize, (2, 0))
ax3 = plt.subplot2grid(gridsize, (2, 1))
ax1.set_title('Home value as a function of home age & area population',
              fontsize=14)
sctr = ax1.scatter(x=age, y=pop, c=y, cmap='RdYlGn')
plt.colorbar(sctr, ax=ax1, format='$%d')
ax1.set_yscale('log')
ax2.hist(age, bins='auto')
ax3.hist(pop, bins='auto', log=True)

add_titlebox(ax2, 'Histogram: home age')
add_titlebox(ax3, 'Histogram: area population (log scl.)')
```


Multi Plots

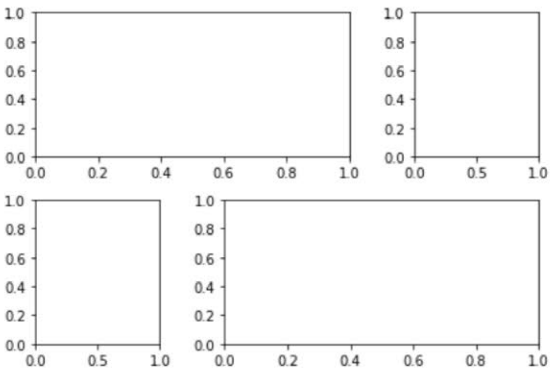


Multi Plots



Multi Plots with GridSpec


```
# GridSpec
import matplotlib.gridspec as gridspec
fig = plt.figure()
grid = plt.GridSpec(2, 3)  # 2 rows 3 cols
plt.subplot(grid[0, :2])  # top left
plt.subplot(grid[0, 2])   # top right
plt.subplot(grid[1, :1])  # bottom left
plt.subplot(grid[1, 1:])  # bottom right
fig.tight_layout()
```

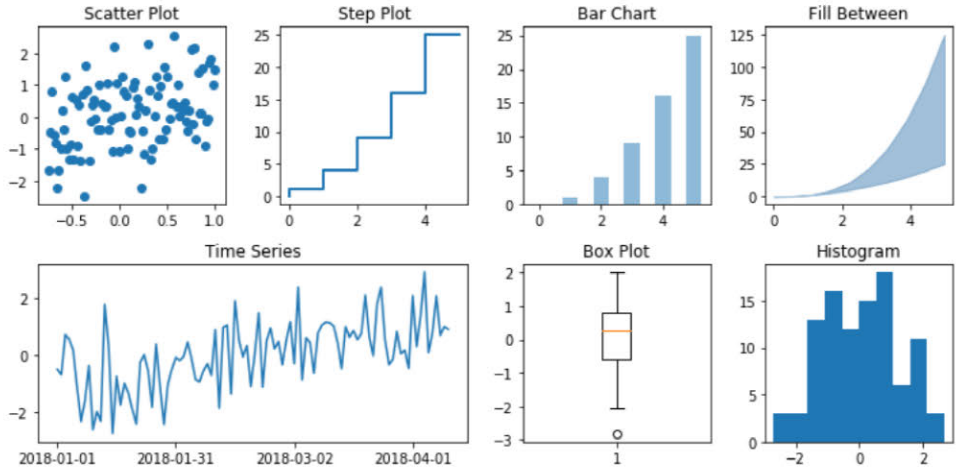


Histograms, Boxplots and Time Series, etc


```
import pandas as pd

# Setup the subplot2grid Layout
fig = plt.figure(figsize=(10, 5))
ax1 = plt.subplot2grid((2,4), (0,0))
ax2 = plt.subplot2grid((2,4), (0,1))
ax3 = plt.subplot2grid((2,4), (0,2))
ax4 = plt.subplot2grid((2,4), (0,3))
ax5 = plt.subplot2grid((2,4), (1,0), colspan=2)
ax6 = plt.subplot2grid((2,4), (1,2))
ax7 = plt.subplot2grid((2,4), (1,3))

# Input Arrays
n = np.array([0,1,2,3,4,5])
x = np.linspace(0,5,10)
xx = np.linspace(-0.75, 1., 100)

# Scatterplot
ax1.scatter(xx, xx + np.random.randn(len(xx)))
ax1.set_title("Scatter Plot")

# Step Chart
ax2.step(n, n**2, lw=2)
ax2.set_title("Step Plot")
```

```
# Bar Chart
ax3.bar(n, n**2, align="center", width=0.5, alpha=0.5)
ax3.set_title("Bar Chart")

# Fill Between
ax4.fill_between(x, x**2, x**3, color="steelblue", alpha=0.5);
ax4.set_title("Fill Between");

# Time Series
dates = pd.date_range('2018-01-01', periods = len(xx))
ax5.plot(dates, xx + np.random.randn(len(xx)))
ax5.set_xticks(dates[::30])
ax5.set_xticklabels(dates.strftime('%Y-%m-%d')[::30])
ax5.set_title("Time Series")

# Box Plot
ax6.boxplot(np.random.randn(len(xx)))
ax6.set_title("Box Plot")

# Histogram
ax7.hist(xx + np.random.randn(len(xx)))
ax7.set_title("Histogram")

fig.tight_layout()
```


Histograms, Boxplots and Time Series, etc



PAN BINBIN

BINBIN.PAN@NUS.EDU.SG

NUS AI SUMMER EXPERIENCE 2019

Object Oriented Programming in Python

NATIONAL UNIVERSITY OF SINGAPORE
DEPARTMENT OF ISEM