Python and Libraries

2022.8

(*) Reference

- Wes Mckinney, Python for Data Analysis: Data Wrangling with Pandas, NumPy, and Ipython, O'Reilly, 2012
- Many Internet sites

What is Anaconda?

- What is Anaconda?
  - Very popular Python development platform package for mathematics and science, and specially for data science and machine learning
  - Anaconda is a set of about a hundred packages including conda, numpy, scipy, ipython notebook, and so on.
- Why Anaconda?
  - > 400 packages available, 150 automatically installed
  - Free, open source
  - Support all major platforms
  - Very reliable and easy to use
  - Scale up to professional and commercial use (with fee)
- System requirements
  - Minimum 3 GB disk space

Anaconda Overview

- Installation
  - Download Anaconda from https://www.anaconda.com/download/
    - Select Python 3.7 version (for Windows)
- Where to start?
  - Command line
  - Launcher: Jupyter notebook, Spyder, Ipython console
- Relevant libraries
  - Pandas (http://paandas.pydata.org)
  - Numpy (http://www.numpy.org)
  - SciPy (http://www.scipy.org)
  - Matplotlib (http://matplotlib.org)

Anaconda Packages

- Over 150 packages are **automatically installed with Anaconda**
- Over 250 additional open source packages can be individually installed from the anaconda repository at the command line, by using the “%conda install” command.
- Thousands of other packages are available from Anaconda.org site
- Others can be downloaded using “%pip install” command that is included and installed with Anaconda.
- You can also make your own custom packages using the “%conda build” command, and upload them to Anaocnda.org or other repositories.

Managing conda and Anaconda

- **Conda** : package management system (included in anaconda)
- Managing conda and anaconda
  - conda info # verify conda is installed, check version number
  - conda update conda # update the conda command
  - conda update anaconda # update anaconda meta package
- Managing packages in Python
  - conda list # view list of packages and versions
  - conda search PKG # search for a package
  - conda install PKG # install packages
  - conda update PKG
- Many more . . . (see the document)

Essential Python Modules

package	Modules with description	
numpy		Foundational Package for scientific computing Multidimensional array objects and computational functions
pandas		Rich data structures and functions to facilitate data processing and analysis: DataFrame and Series
SciPy		Collection of packages for performing linear algebra, statistics, optimization, and more
matplotlib	Pyplot	Data visualization
sklearn (scikit-learn)	linear_model, cluster metrics model_selection	LinearRegression, SGDVassifier, LogisticRegression Kmeans accuracy_score, classification_report, confusion_matrix roc_curve, auc train_test_split

What is Python Language?

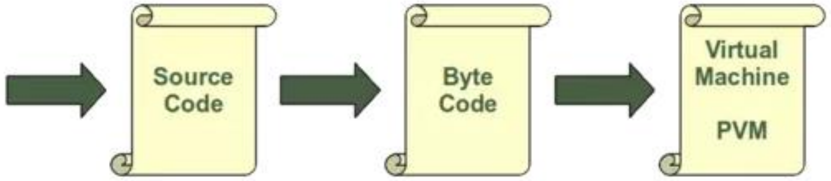
- Completely open source, started in early 1990
- **Script language (interpreter)** , i.e. no compiler
  - Directly translate source code (do not generate compiled code)
  - Converted to (platform-independent) bytecode (and Python Virtual Machine(PVM) interprets and executes it – slow)
- **Easy, Very portable**, mostly runnable on all supported platforms
- **Object-oriented** and Functional
- **Large standard libraries** with huge set of external modules
- **Dynamically typed:** variable type determined at run-time (no need of variable declaration), hence slow… but efficient memory usage

Python Scripts

- **Python script**: Collection of commands in a file designed to be executed like a program
- Use any editor to create a Python script, say, *myscript.py*
- No compilation needed
  - Python script is **interpreted**. More precisely, it is converted to byte code (.pyc), and then executed.
- Run script from command line
  - > python myscript.py
  - (ex) calculator, running scripts, test environment
- Run script in Notebook or IDE
  - **Jupyter** or Spyder, or other IDE
  - (ex) work processes (ideal for data processing and analysis), documentation, teaching and presentation

Python Execution

- Python is **interpreted language** and executed as:
  - Step 1 : (**compilation**) reads python code (.py file) and checks the syntax, and translate it into its equivalent form in intermediate language, “Byte code”. (.pyc file)
  - Step 2: (**interpretation**) Python Virtual Machine(PVM) converts the byte code into machine-executable code.




```
a = 10
b = 10
Sum = a+b
print(Sum)
```

```
(base) PS C:\Users\rtaje\Coding_Exercise> python -m dis test1.py
  1           0 LOAD_CONST               0 (10)
              2 STORE_NAME               0 (a)

  2           4 LOAD_CONST               0 (10)
              6 STORE_NAME               1 (b)

  3           8 LOAD_NAME                0 (a)
             10 LOAD_NAME                1 (b)
             12 BINARY_ADD
             14 STORE_NAME               2 (Sum)

  4          16 LOAD_NAME                3 (print)
             18 LOAD_NAME                2 (Sum)
             20 CALL_FUNCTION            1
             22 POP_TOP
             24 LOAD_CONST               1 (None)
             26 RETURN_VALUE
```

<u>Byte code example</u>

1. Line Number
2. offset position of byte code
3. name of byte code instruction
4. instruction’s argument
5. constants or names (in brackets)

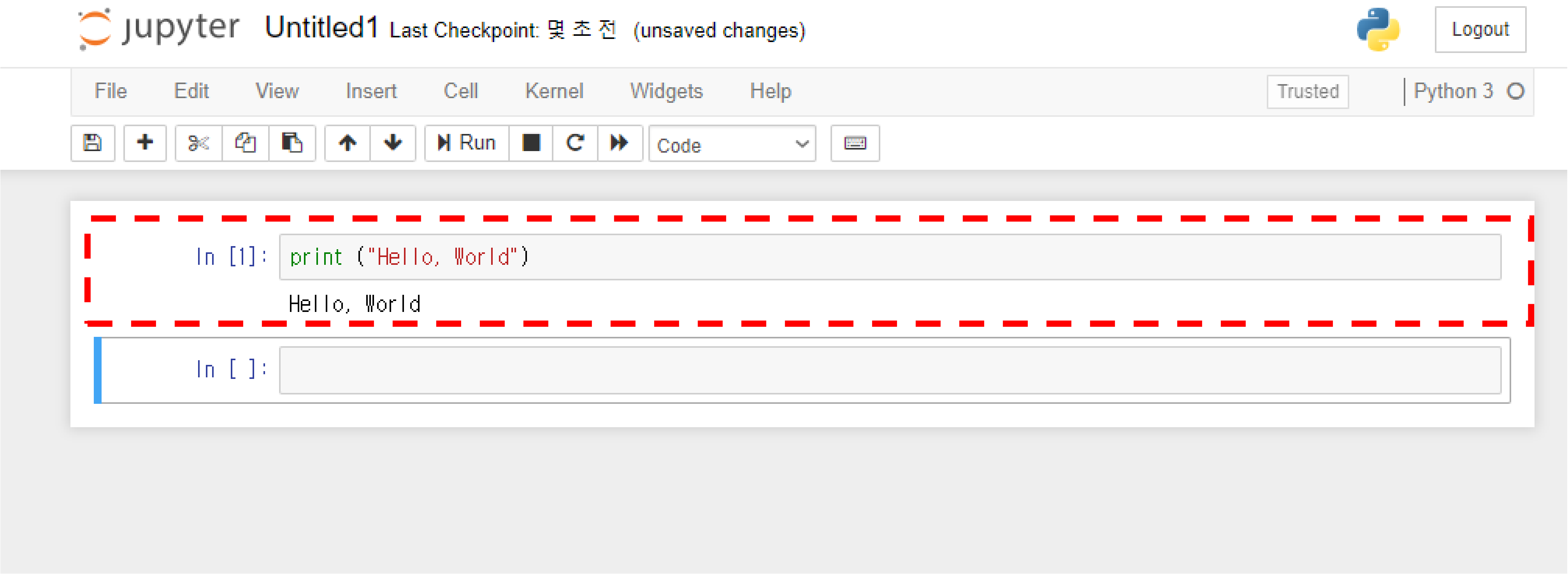
Jupyter Notebook

- Convenient web-based executable script files
  - Interactive code development
  - Cell-wise execution
  - No reloading of script (.py) files necessary
  - Easy to share
  - Excellent teaching tool
- Project Jupyter was born out of the IPython Project in 2014
  - Jupyter can support (or be interfaced with) other languages (Ruby, R, Julia, etc.)
- Requires Google Chrome or Mozilla Firefox
- On-line examples
  - https://nbviewer.jupyter.org

Jupyter Notebook or Lab

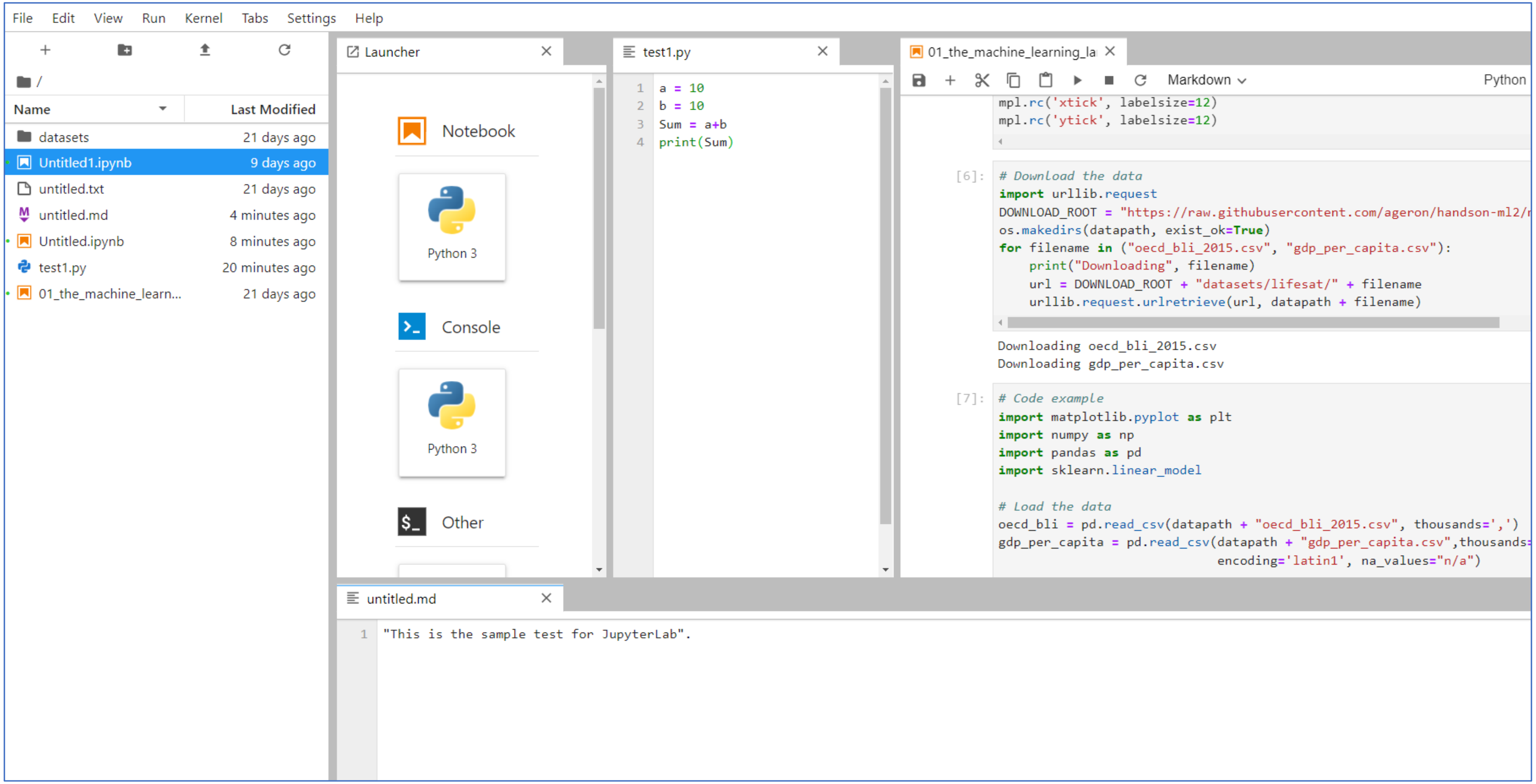
- To start, from command line, enter “jupyter notebook” or click the icon “Jupyter Notebook” from startup menu and set the type as “Python 3”.
- For JupyterLab, enter “jupyter lab”. (You can go to your working directory before entering JupyterLab, or you can navigate to your working directory in JupyterLab.)
- You will have either “Coding” cell or “Markdown” cell. (Jupyter Notebook)
- You can access “Notebook”, “Console”, “Text File”, “Markdown”, “Terminal”. (JupyterLab)
- Markdown (documentation) guides:
  - https://colab.research.google.com/notebooks/markdown_guide.ipynb

Jupyter notebook Python 3



- ❖ In place: Ctrl + Enter
- ❖ To execute cell and move to next cell: Shift + Enter
  - Create new cell if necessary
- ❖ To execute and insert new cell: Alt + Enter

JupyterLab



File Edit View Run Kernel Tabs Settings Help
Name
Last Modified
datasets 21 days ago
Untitled1.ipynb 9 days ago
untitled.txt 21 days ago
untitled.md 4 minutes ago
Untitled.ipynb 8 minutes ago
test1.py 20 minutes ago
01_the_machine_learn... 21 days ago
Launcher
Notebook
Python 3
Console
Python 3
Other
test1.py
a = 10
b = 10
Sum = a+b
print(Sum)
01_the_machine_learning_la
Markdown
Python
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
[6]:
# Download the data
import urllib.request
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/
os.makedirs(datapath, exist_ok=True)
for filename in ("oecd_bli_2015.csv", "gdp_per_capita.csv"):
print("Downloading", filename)
url = DOWNLOAD_ROOT + "datasets/lifesat/" + filename
urllib.request.urlretrieve(url, datapath + filename)
Downloading oecd_bli_2015.csv
Downloading gdp_per_capita.csv
[7]:
# Code example
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model
# Load the data
oecd_bli = pd.read_csv(datapath + "oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv(datapath + "gdp_per_capita.csv",thousands
encoding='latin1', na_values="n/a")
untitled.md
"This is the sample test for JupyterLab".

Keyboard Shortcuts - Jupyter

- Command mode (press ESC to enable)

In command mode	
Shift-Enter	run cell, select below
Ctrl-Enter	run selected cells
Alt-Enter	run cell and insert below
a/b	insert cell above/below
x/c	cut selected cells / copy selected cells
Shift-v / v	paste cells above/below
Shift-m	merge selected cells, or current cell with the cell below if only one cell is selected

In command mode	
l	toggle line numbers
o	toggle output of selected cells
h	show keyboard shortcuts
Shift-Space	scroll notebook up
Space	scroll notebook down
Window-/	toggle comment

In edit mode (press Enter)	
Ctrl-Shift-Minus	Split cell at cursor

Notebook Cell Types

- Code cells
  - Edit and execute cells inline, generates output as text, figures, HTML tables
  - Syntax highlighting, tab completion, introspection
  - Default for inserted cells
- Markdown cells
  - Rich text input, including HTML and LaTex
  - Cell replaced by text output when executed (**Documents**)
- Raw text cells
  - Executed as input (no formatting)
  - Cell remains in place
- Heading cells
  - Levels 1 through 6, similar to Microsoft Word
  - Can be used to generate Table of Contents

Colab from Google

- Free cloud service from Google
  - A Jupyter notebook environment that requires no setup to use
  - Supports free GPU/TPU, and Runs entirely in the cloud
  - provides a maximum GPU runtime of 8~12 hours ideally at a time
- Useful Shortcuts

actions	colab	jupyer
Convert to code cell	Ctl-M Y	Y
Convert to text cell	Ctl-M M	M
Split at cursor	Ctl-M – (minus sign)	Ctrl Shift -
Merge two cells	Ctl-M /	Shift M
Show keyboard shortcuts	Ctl-M H	H
Interrupt execution	Ctl-M I	II

JupyterLab

- What is JupyterLab?
  - a next-generation web-based user interface for Project Jupyter
  - Fully support Jupyter Notebook
  - Enables users to use text editor, terminals, data file viewers, and others
- Installation (in command shell)
  - conda install –c conda-forge jupyterlab # when using conda
  - pip install jupyterlab # when using pip
- Supported browsers
  - Firefox, Chrome, Safari
- Starting jupyterlab
  - Type “jupyter lab” in command shell

Python Language

- What is Python?
  - Widely used general purpose high-level language
  - There are two major versions : Python 2 and Python 3
  - Object-oriented
  - Interpreted language
  - Two modes: Interactive Mode and Script Mode
- Important Concepts
  - Objects, attributes, and methods
  - Functions vs. object methods
  - Object references
  - Mutable and immutable objects

Python Programs

- a program is a sequence of definitions and commands
  - definitions evaluated
  - commands executed by Python interpreter in a shell
- commands (statements) instruct interpreter to do something
- can be typed directly in a shell or stored in a file that is read into the shell and evaluated
- Programs manipulate data objects, and Objects are:
  - Scalar (can not be subdivided): basic type
  - non-Scalar (have internal structure that can be accessed): container type

Data types

- Basic Types (Scalar Objects)
  - Int, float, bool, NoneType (special and has one value, None)
- Container types
  - String: sequence of characters, “Hello”
  - List: can contain any types of variables, mutable, [1, 2.3, ”Welcome”]
  - Tuple: can read, but not overwrite (to make computation fast), immutable, (1, 3, [2,3])
  - Dictionary (or dict): only access by keys, mutable, {“name”:”Kim”, “age”:25}
- Array (or ndarray)
  - Defined in numpy : similar to list, but much more efficient
  - all the elements are of the same type (int, float, Boolean, string, or other object)
  - Element-wise operation (vector operation)
- DataFrame and Series
  - Defined in pandas: provides data processing and analysis capabilities
  - Built on top of Numpy functionality
  - Table-shaped: “columns” and “index”

Printing to Console

- When you use Jupyter notebook:

```
In [1]: 3+2
Out[1]: 5

In [2]: print(3+2)
        5
```

"Out" tells you it's an interaction within the shell only

No "Out" means it is actually shown to a user, apparent when you edit/run files

Binding variables and values

- Equal sign is an **assignment** of a value to a variable name

  pi = 3.14159

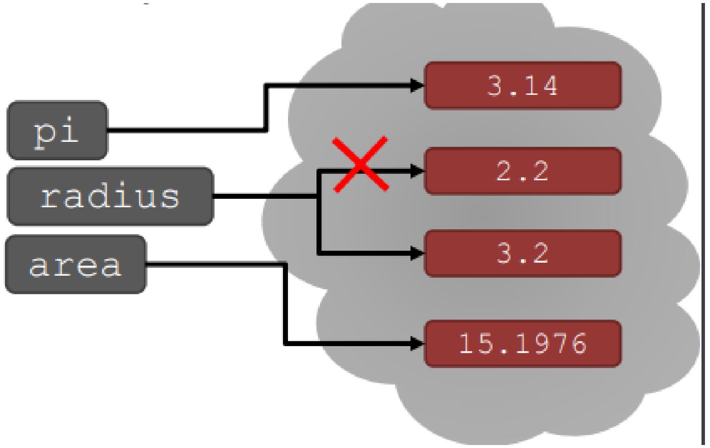
- **Assignment** of expression:

  radius = radius + 1

  - expression on the right (evaluated to a value)
  - variable name on the left
  - Equivalent to `radius += 1`

- Can **re-bind** variable names using new assignments

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = radius+1
```



Objects, attributes, and methods

- **Everything in Python is an object**.
  - Scalars, sequences, dictionaries, functions, DataFrames, modules, and more
  - Object is simply a <u>collection of data (variables) and methods (functions)</u> that act on those data.
- Each type of object has a set of
  - **Attributes**: Characteristics of the object
  - **Methods**: Functions that operate on the object (and possibly other objects)
- Attributes and methods are accessible by:
  - obj.attr_name
  - obj.method_name()

Functions vs. Object Methods

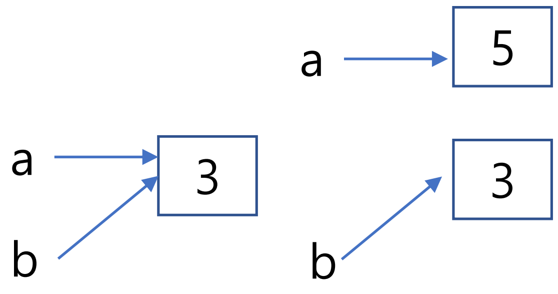
- Functions and object methods are essentially the same…
  - One or more bundled steps performed on some input
  - In some cases, there will be a function and an object method that do the same thing (e.g., sum)
- …BUT, they differ in how they are used
  - Functions are called on zero or more objects and return result(s) that can be assigned to a variable
  - Object methods are called by an object and can either update the calling object or return results

Mutable and Immutable Objects

- **Mutable** Objects
  - Can be modified via assignment or a function/method
  - Sets, Lists, dictionaries, arrays, dataframes, class instances
- **Immutable** Objects
  - Can not be modified
  - All other types including int, float, Boolean, strings, tuples

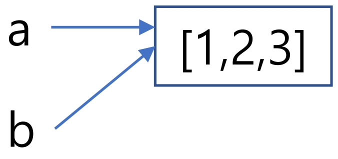
Mutable and Immutable (examples)

```
In [384]: a = 3                                # immutable variable
          b = a
          id(a), id(b)

Out[384]: (1681633568, 1681633568)
```

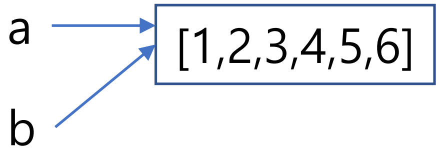



```
In [385]: a += 2                               # since it is immutable, a is newly created
          a,b, id(a), id(b)

Out[385]: (5, 3, 1681633632, 1681633568)
```

```
In [389]: # more examples
          a =[1,2,3]                           # when assigning a variable, you are assigning the reference.
          b = a                                # id(x) returns memory address of the object
          id(a),id(b)

Out[389]: (1559399868552, 1559399868552)
```




```
In [390]: a +=[4,5,6]                          # same id (interpreted as a.append([4,5,6]))
          a,b, id(a), id(b)                    # note that a = a + [4,5,6] will create a new object

Out[390]: ([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], 1559399868552, 1559399868552)
```



Object References

- Call-by-value? or Call-by-reference?

```
>>> def test(a):
        a = 2

>>> a = 1
>>> test(a); a
1
```

```
>>> def test2(a):
        a.append('world.')

>>> b = 'Hello'
>>> test2(b); b
['Hello', 'World.']
```

```
>>> a = 10
>>> b = a
>>> a += 100
>>> a, b
(110, 10)
>>> id(a), id(b)
(14073…7824, 1407…624)
```

```
>>> a = [1,2,3]
>>> b = a
>>> a += [4,5,6]
>>> a,b
([1,2,3,4,5,6],[1,2,3,4,5,6])
>>> id(a), id(b)
(225009…832, 225009…832)
```

Object References (2)

- **Call-by-Object** (or **call-by-Object Reference** or call-by-sharing)
  - If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like *call-by-value*. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable.
  - If mutable arguments are passed, they are also passed by object reference, but they can be changed in place in the function. If we pass a list to a function, we have to consider two cases:
    - Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope. (act like *call-by-reference*)
    - If a new list is assigned to the name, the old list will not be affected. (a new object is created)

Classes and Objects

- **Implementing a new object type with classes**
  - **define** the class
  - define **data attributes** and **methods**
  - data and methods are **common across all instances**

- **Using the new object type**
  - create **instance** of the object type (instance is **one specific** object)
  - do **operations**
  - Instance has the **structure of the class**

- Objects have attributes
  - Data attributes : how can you represent your object with data? (**what it is**)
  - Procedural attributes (behavior/operations/methods): how can someone interact with the object (**what it does**)
  - *'self'* is a special parameter referring to an instance of the class

Classes define example

class definition

name

class parent

variable to refer to an instance of the class

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None

myanimal = Animal(3)
```

what data initializes an `Animal` type

special method to create an instance

`name` is a data attribute even though an instance is not initialized with it as a param

one instance

mapped to `self.age` in class def

https://ocw.mit.edu/courses/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016

Importing Modules and Scripts

- **Modules**
  - Simply a python file with a .py extension (module name is the file name)
  - Can define functions, classes and variables
- **Packages**
  - Directory which contains multiple packages or modules
  - **Must** contain a special file called __init__.py (indicates it is a Python package)
- Modules and Python scripts are loaded in the same manner. For a module or Python script P (.py):
  - import P [as p]
  - from module_name import *   // import all functionality
  - from module_name import f, g, h   // import specific functions
  - Import foo.bar (or from foo import bar) // import module bar from package foo
- **Built-in modules (standard library)**
  - https://docs.python.org/3/library/

Indexing and Slicing

- for container variables: lists, arrays, tuples, and strings
  - e.g., A = [1,2,3]
- **Indexing**: access item in a sequence
  - Python is zero-based: A[0] = 1, A[1] = 2, A[2] = 3
  - Negative indices: A[-1] = 3, A[-2] = 2
- **Slicing**: access subset of a sequence [*start: stop: step*]
  - A[start=0: stop=len(A): step=1]
  - Slicing ends before the *stop* (excluding *stop*): A[0:1] = [1]; A[0:2] = [1,2]
- Examples
  - A[:] = A[::] = A = [1,2.3]
  - A[1:] = [2,3], A[:2] =[1,2]
  - A[::2] = [1,3]
  - A[::-1] = [3,2,1]
  - A[-1:0:-1] = [3,2]
  - More…

NumPy (Numerical Python)

- Foundation for scientific computing
  - Linear algebra, math functions, and random number generation
- Provides useful data structures (array)
  - Ndarray (or array) : similar to lists, but much more powerful
  - Vectorization: fast operations on arrays of data without the need for loops
- Primary Use:
  - Fast vectorized array operations for data munging, cleaning, filtering, transforming
  - Built-in common array algorithms
  - Efficient descriptive statistics
  - Data alignment and relational data manipulations for merging and joining multiple data sets
  - Expressing conditional logic ad array expressions instead of loops

NumPy - array

- Numerical Python N-dimensional arrays
  - Similar to list, but much more powerful
- Fast, flexible container for data
  - Numerical, boolean, or string data
- Perform mathematical operations on entire data sets without loops (**vectorization**)
- Some common attributes (i.e., arr.*attr*)
  - ndim: Number of array dimensions (e.g., 1, 2, 3)
  - shape: Size of each dimension (e.g., (2, 4))
  - dtype: Data type for the array (e.g., int8, float64)
- Use tab completion to explore attributes and methods

Slicing: list and array

- 1-D array slicing (quite often used)

```
a = np.arrange(10)    # a = array([0,1,2,3,4,5,6,7,8,9])
a[start:end]          # items start through end-1
a[start:]             # items start through the rest of the array
a[:end]               # items from the beginning through end-1
a[:]                  # a copy of the whole array
a[start:end:step]     # start through not past end, by step

a[-1]                # last item in the array
a[-2:]               # last two items in the array
a[:-2]               # everything except the last two item
a[::-1]              # all items in the array, reversed
a[1::-1]             # the first two items, reversed
a[:-3:-1]            # the last two items, reversed
a[-3::-1]            # everything except the last two items, reversed
```

Slicing: list and array

- 2-D array slicing (to split loaded data into input(X) and the output(y))

```
X = [:, :-1]   # select al the rows and all columns except the last one
y = [:, -1]    # select all rows again, and index just the last column
```

Array operations

- Between Arrays and Scalars -- Broadcasting
  - All basic operations are applied **element-wise**
  - +, -, /, *, **, %, etc.
- Universal Functions (ufunc)
  - Unary (on a single array): abs, sqrt, exp, log, ceil, floor, logical_not, and more
  - Binary (on two equal-sized arrays): +, -, /, *, **, min, max, mod, >, >=, <, <=, ==, !=, &, |, ^
- Mathematical and Statistical Functions/Methods
  - Aggregation (collection): mean(), sum(), std(), var(), min(), max(), argmin(), argmax()
  - Non-aggregation: cumsum(), cumprod()
  - Sort, concatenate, etc.

Pandas

- Pandas
  - Provides data processing and analysis capabilities
  - Built on top of Numpy functionality
- Two data structures: **Series** and **DataFrames**
- What can be done?
  - Creating Series and DataFrame objects
  - Basic Series and DataFrame methods
  - Indexing/reindexing, slicing, and filtering
  - Mathematical operations
  - Missing data handling

Pandas - Series

- A single column of DataFrame
- Similar to an ndarray
  - Easy to perform computation
  - Indexing, slicing, filtering
- With some additional features
  - Comes with an associated array of data labels, called an **index object**
  - Access values using integer indices (number: like an array) or specified indices (index name: like a dict)
  - Easy merging of data sets

Pandas - DataFrame

- 2-D tabular-like data structure
  - Similar to a dictionary of Series objects with the same indices
  - Hierarchical indexing or panel for higher dimensions
- Access rows by **index**, and columns by **column names**
- Built-in methods for data processing, computation, visualization, and aggregation

Pandas – DataFrames (example)

- From dictionary

In [149]:
```python
countries = ['CH','IN', 'US'] * 3
years = [1990, 2008, 2025] * 3
years.sort()
pop = [1141, 849, 250, 1333, 1140, 304, 1458, 1398, 352]
```

In [151]:
```python
D= {'country': countries, 'year':years, 'pop':pop}; D
```

Out[151]:
```
{'country': ['CH', 'IN', 'US', 'CH', 'IN', 'US', 'CH', 'IN', 'US'],
 'year': [1990, 1990, 1990, 2008, 2008, 2008, 2025, 2025, 2025],
 'pop': [1141, 849, 250, 1333, 1140, 304, 1458, 1398, 352]}
```

In [154]:
```python
frame = DataFrame(D, columns=['year','country','pop']); frame
```

Out[154]:

	year	country	pop
0	1990	CH	1141
1	1990	IN	849
2	1990	US	250
3	2008	CH	1333
4	2008	IN	1140
5	2008	US	304
6	2025	CH	1458
7	2025	IN	1398
8	2025	US	352

Pandas - DataFrames

- Basic DataFrame Methods
  - Indexing columns(features): either by column name or attribute (ex: df[‘year’] or df.year, df[[‘year’,’pop’]])
  - Indexing rows by index name or index number: df.loc() or df.iloc()
  - df.name, df.index, df.columns, and df.values (similar to Series)
- Functions
  - df.sort_index(), df.sort_index(axis=1) // sort by index or columns
  - df.sum(), df.mean()
  - df.idmax(), df.idmin() // index of max and min
  - df.value_counts() // counts of values
  - df.isin([‘b’,’c’]) // see if some elements are in df
  - df.fillna(), df.dropna() // remove or fill any columns of NaN

Data visualization - matplotlib

- Use:
  - %matplotlib inline magic command (once Jupyter is open)
  - import matplotlib.pyplot as plt
- Basic template
  - Create a new figure : (ex) fig = plt.figure(figsize = (12,8))
  - Add subplots (if necessary)
    - ax1 = fig.add_subplot(2,1,1)   # 2x1 arrangement, first figure
    - ax2 = fig.add_subplot(2,1,2)
  - Create plot (plt or ax1…axN methods)
  - Label, annotate, format plot
  - Copy or save plot

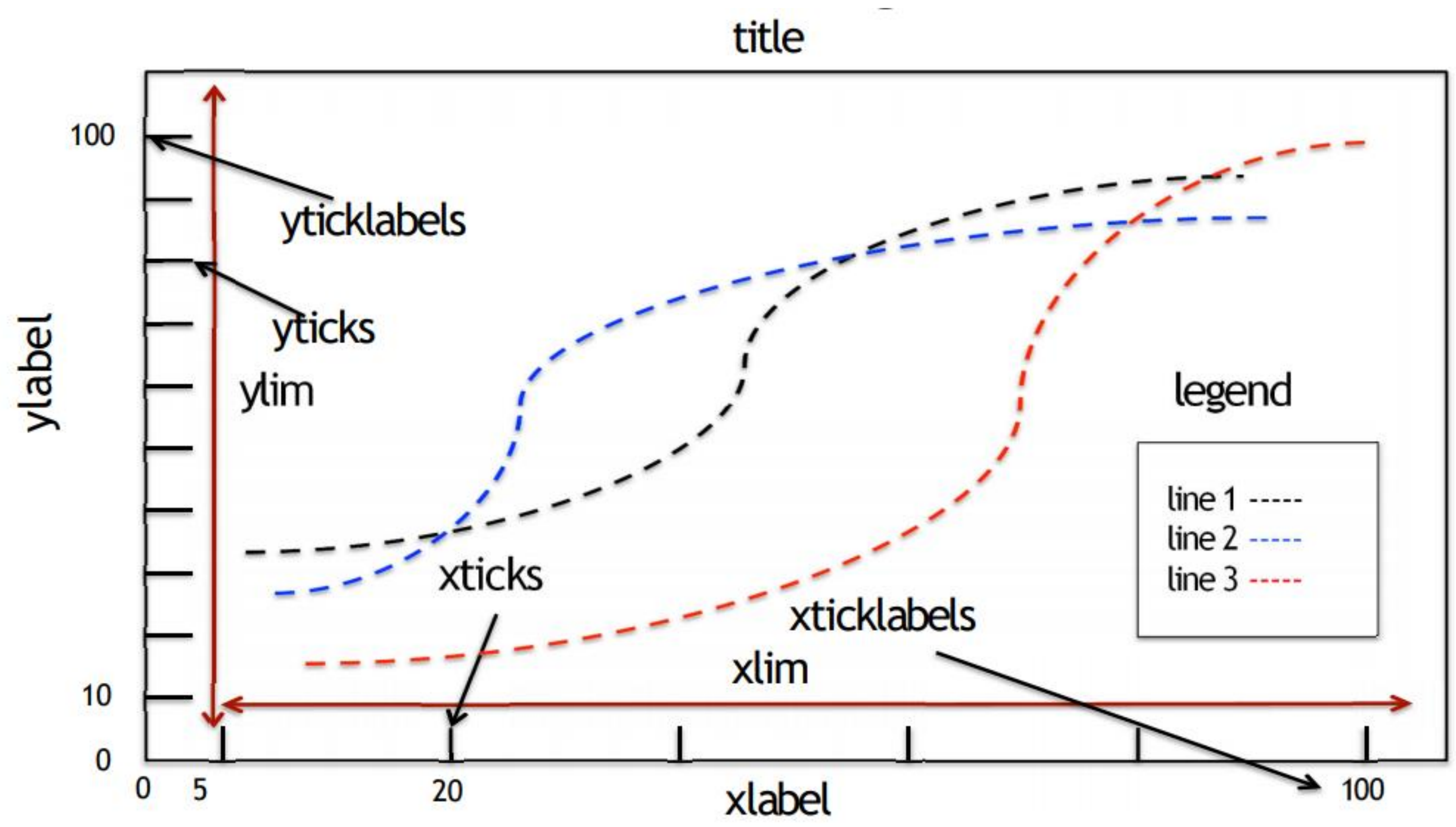
Matplotlib - Common plot types

- Line plots
  - plt.plot (x, y, '-')
- Scatter plots – comparison between lots of data
  - plt.scatter (x, y, '.')
- Bar plots – comparison between few data
  - Bar (horizontal): plt.barh (x, y, width)
  - Column (vertical): plt.bar (x, y, width)
- Histogram plots – single distributions
  - plt.hist (x, bins)
- Boxplots – one or more distributions
  - plt.boxplot (x)

Matplotlib - Colors, Markers, and Line Styles

- All specified as special string characters in plot call
- Colors - Many plot types
  - Basic colors: g(reen), r(ed), b(lue), (blac)k, m(agenta), y(ellow), c(yan), w(hite)
  - For more, see http://matplotlib.org/api/colors_api.html
- Markers and Line Styles - Mostly relate to plt.plot
  - Markers: ., o, +, * (star), 1, 2, 3, 4 (triangles), s(quare), D(iamond)
  - Line styles: solid (-), dashed (--), dotted (:), dash-dot (-.)
  - linewidth keyword (float value)
- Usage
  - Style string: Combines all three (e.g., ‘k.’, ‘g--’, ‘ro-’)
  - Separate keyword arguments: color, linestyle, marker

Formatting plots



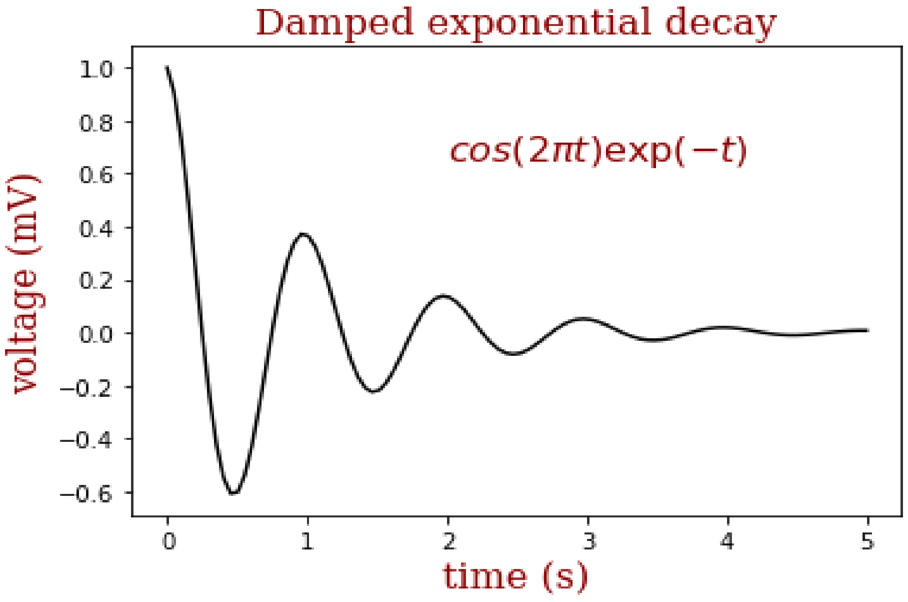
Formatting plots

- Title: title('title')
- Axis labels: xlabel ('Time'), ylabel ('Price)
- Axis limits: xlim([0,10]), ylim
- Ticks: xticks([0,60,70,80,90,100]), yticks
- Tick labels: xticklabels(['F','D','C','B','A']), yticklebals
- Legends: legend(('one','two','three'))
- Text
  - text(x, y, text, fontsize)
  - arrow(x, y, dx, dy) # draws arrow from (x,y) to (x+dx, y+dy)
  - annotate (text, xy, xytext) # annotate the xy point with text positioned at xytext
- shapes
  - Rectangles, circles, polygons
  - Location, size, color, transparency (alpha)

Matplotlib - Example(1)

In [27]:

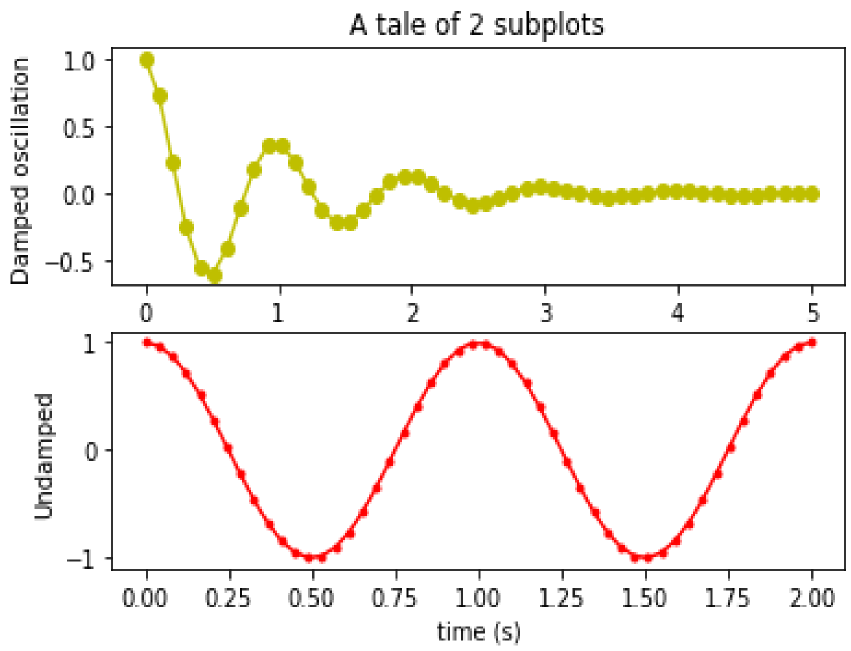
```
x = np.linspace(0.0,5.0,100)
y = np.cos(2*np.pi*x) * np.exp(-x)

plt.plot (x,y,'k')
plt.title('Damped exponential decay', fontdict=font)
plt.text(2, 0.65, r'$cos(2 ₩pi t) ₩exp(-t)$', fontdict=font)

plt.xlabel('time (s)', fontdict=font)
plt.ylabel('voltage (mV)', fontdict=font)

plt.subplots_adjust(left=0.15)
```



Matplotlib - Example(2)

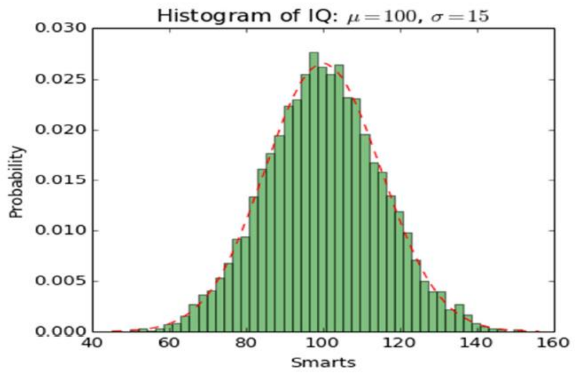
```
In [39]: x1 = np.linspace(0.0,5.0)
         x2 = np.linspace(0.0,2.0)
         y1 = np.cos(2*np.pi*x1) * np.exp(-x1)
         y2 = np.cos(2* np.pi* x2)

         plt.subplot(2, 1, 1)
         plt.plot(x1,y1,'yo-')
         plt.title('A tale of 2 subplots')
         plt.ylabel('Damped oscillation')

         plt.subplot(2, 1, 2)
         plt.plot(x2, y2,'r.-')
         plt.xlabel('time (s)')
         plt.ylabel('Undamped')
```

```
Out[39]: Text(0, 0.5, 'Undamped')
```

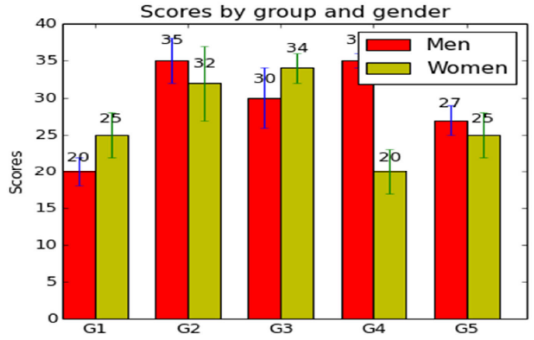


Many more examples...

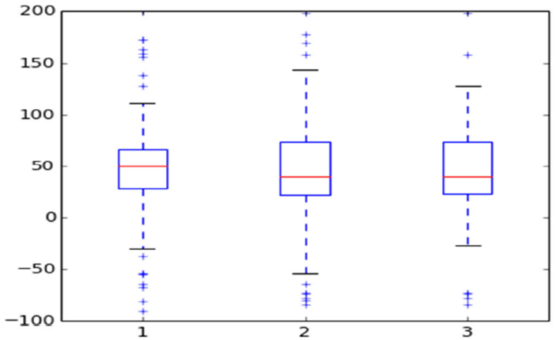
Histogram

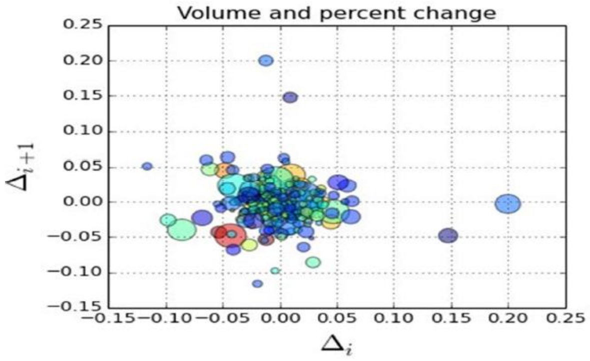


Bar Chart (with error bars and legend)



Boxplots





Scatter + Histogram