

# Introduction to Data Science

Lecture 2.1 Python for data analysis

Instructor: Sangryul Jeon

School of Computer Science and Engineering

[srjeonn@pusan.ac.kr](mailto:srjeonn@pusan.ac.kr)



# Install Python

## ❖ Requirements

- Install Python programming language
- Install Integrated development environment (IDE) for the convenient development
- Install libraries needed for coding

## ❖ Anaconda

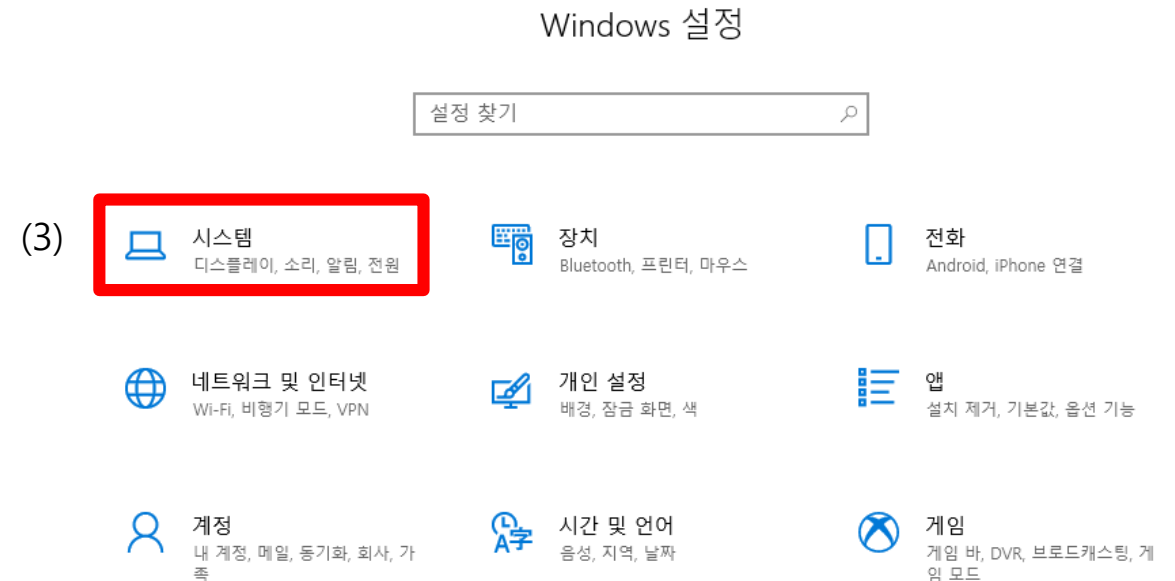
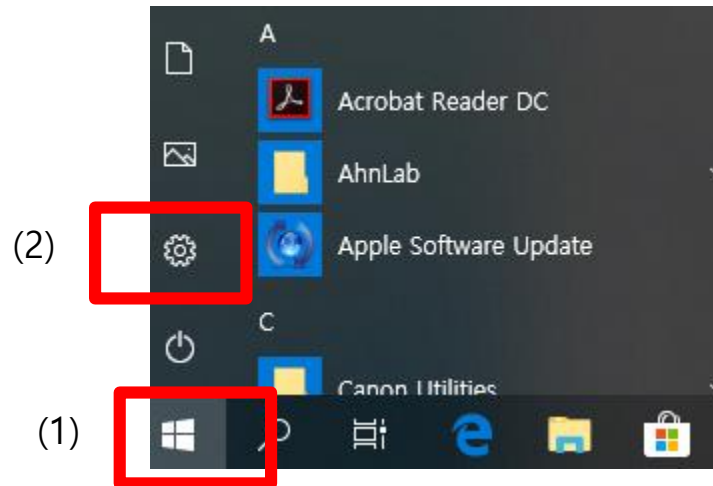
- Integrated development environment for Python development

# Check OS type: 64-bit/32-bit (1/2)

❖ (1) Click the Windows key → (2)

❖ (3) Click the System icon

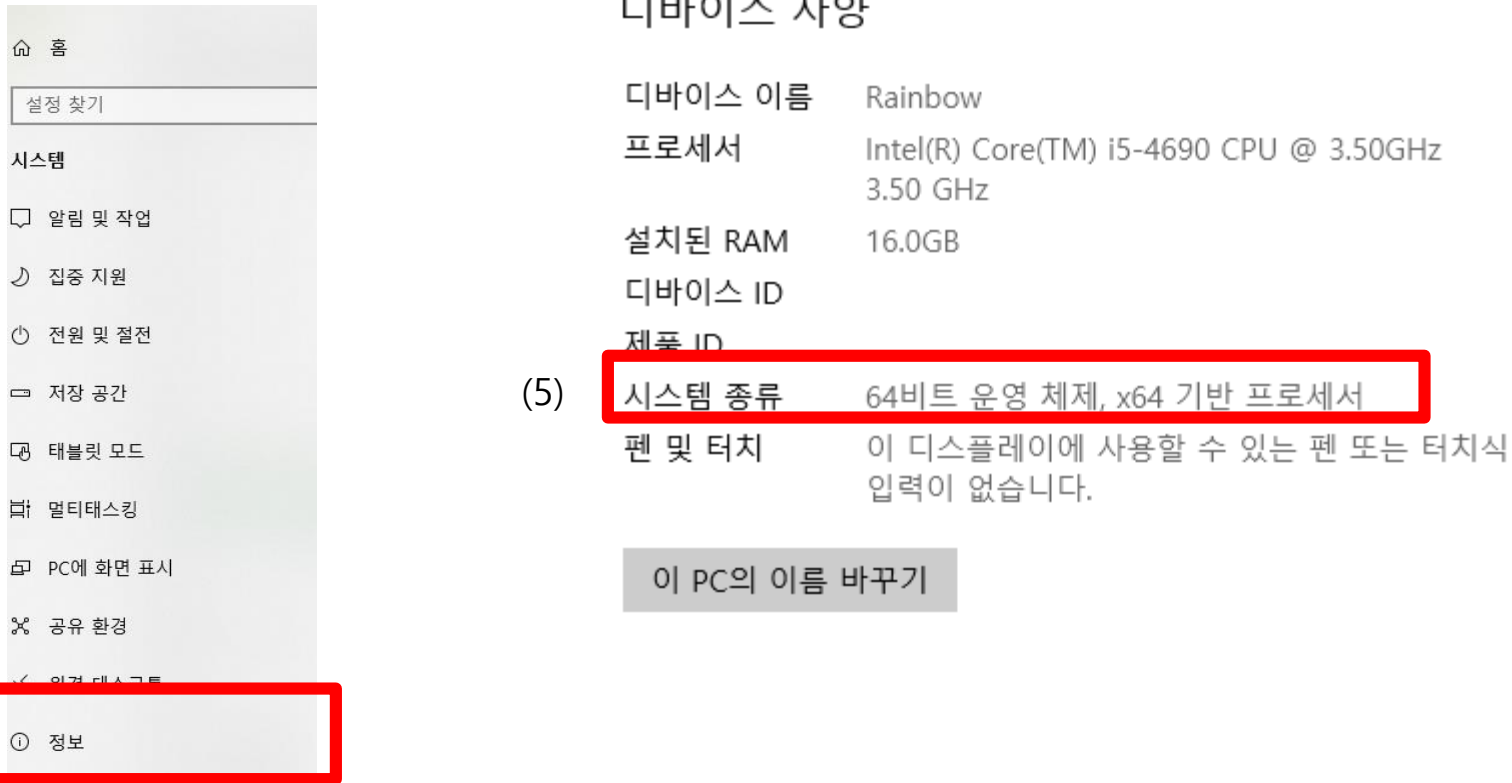
Click the Setting icon



## Check OS type: 64-bit/32-bit (2/2)

❖ (4) Click the Information menu on the left panel

❖ (5) Check the type



The image shows a Windows Settings window. On the left, the 'System' category is selected, and the 'Information' option at the bottom is highlighted with a red box and labeled (4). On the right, the 'Device specifications' page is shown. The 'System type' entry is highlighted with a red box and labeled (5), indicating '64-bit operating system, x64-based processor'. Other specifications listed include device name 'Rainbow', processor 'Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz', RAM '16.0GB', and device ID.

디바이스 사양

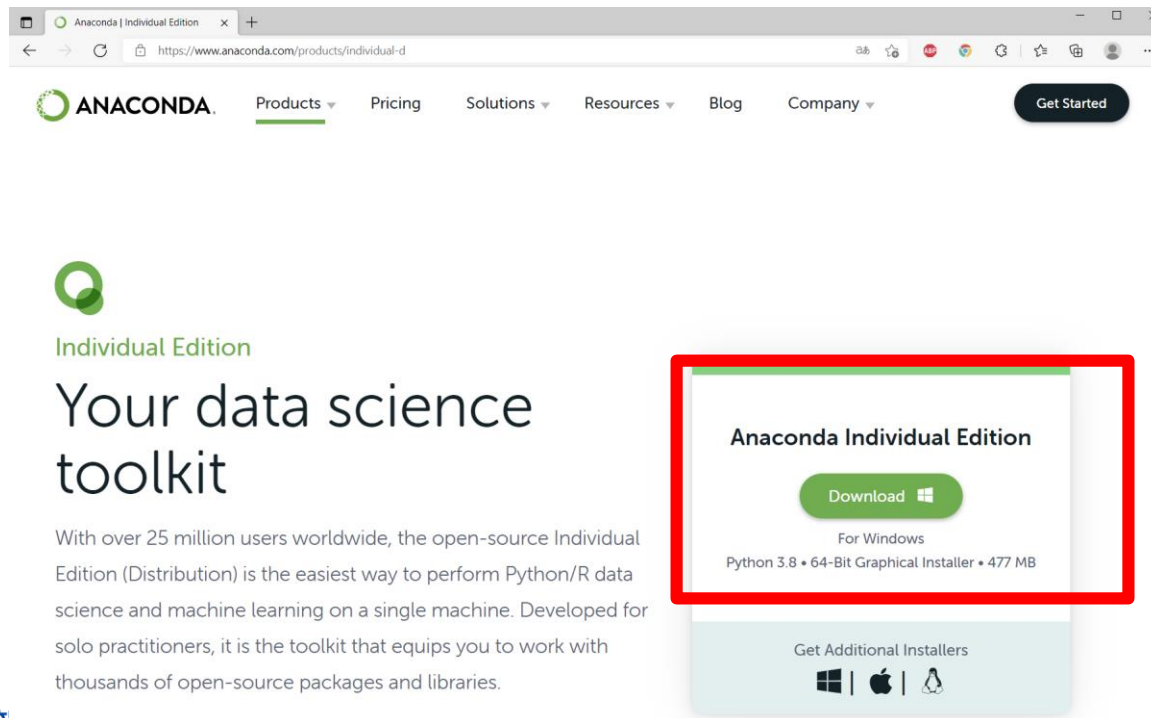
디바이스 이름	Rainbow
프로세서	Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz 3.50 GHz
설치된 RAM	16.0GB
디바이스 ID	
제품 ID	
(5) 시스템 종류	64비트 운영 체제, x64 기반 프로세서
펜 및 터치	이 디스플레이에 사용할 수 있는 펜 또는 터치식 입력이 없습니다.

이 PC의 이름 바꾸기

# Download the Anaconda

## ❖ Homepage

- <https://www.anaconda.com/download/>



## ❖ Checklist

- 64bit
- Python 3.8-based
- If you want to install a different version, click Get Additional Installers to select it.



이름

▼ 오늘 (4)

Anaconda3-2021.05-Windows-x86\_64

# Install anaconda (1/5)

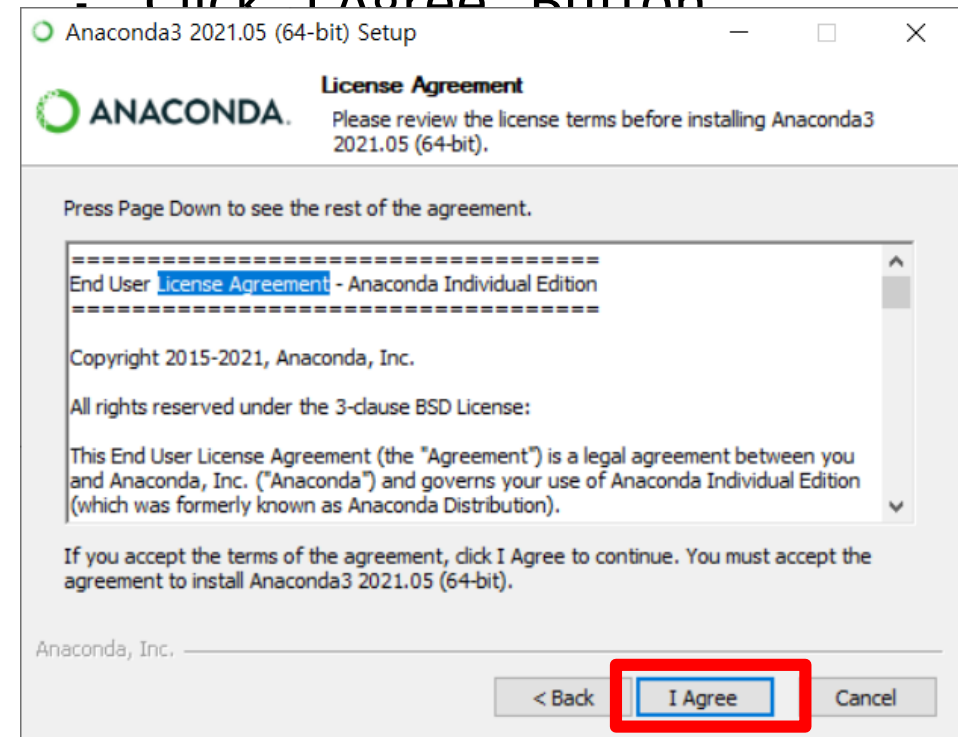
## ❖ Start

Click Next button



## ❖ License Agreement

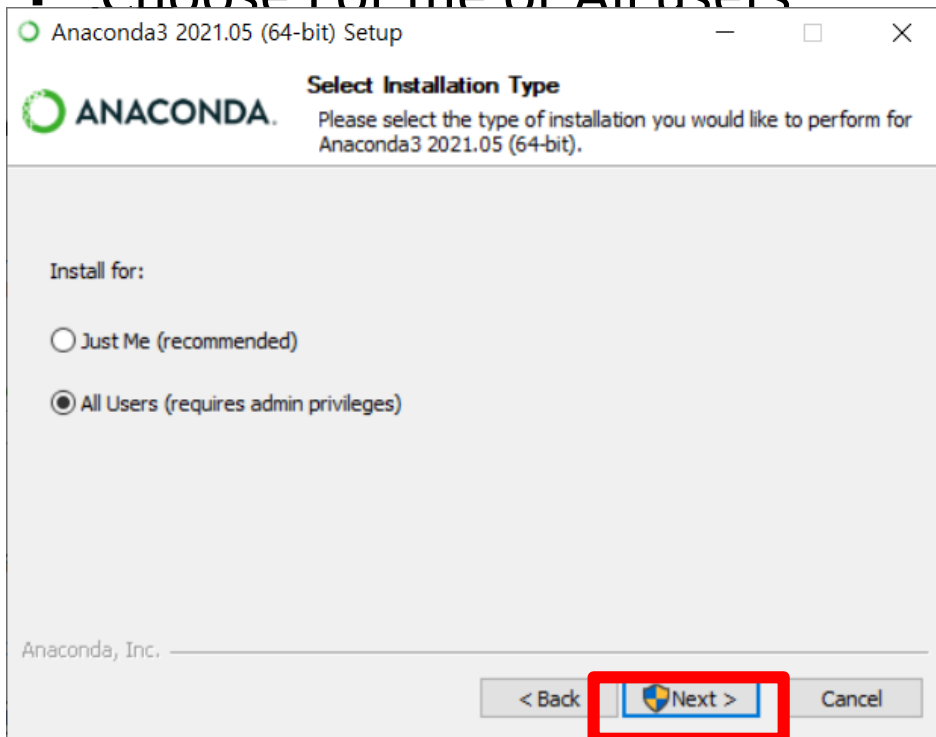
Click "I Agree" Button



## Install anaconda (2/5)

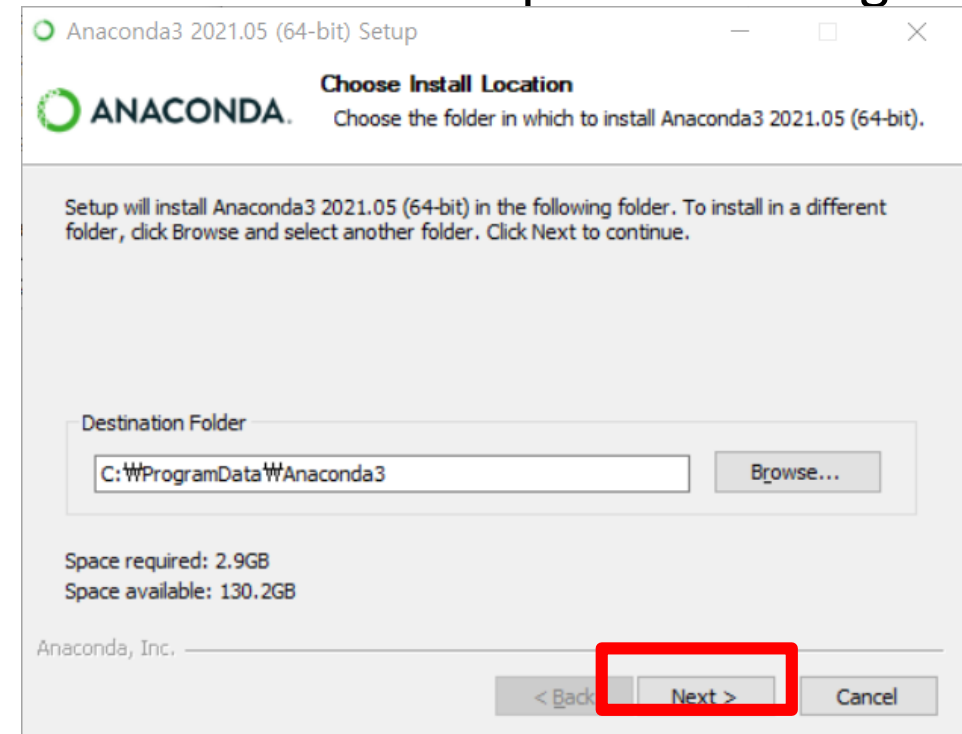
### ❖ Installation Type

- Choose For me or All users



### ❖ Install Location

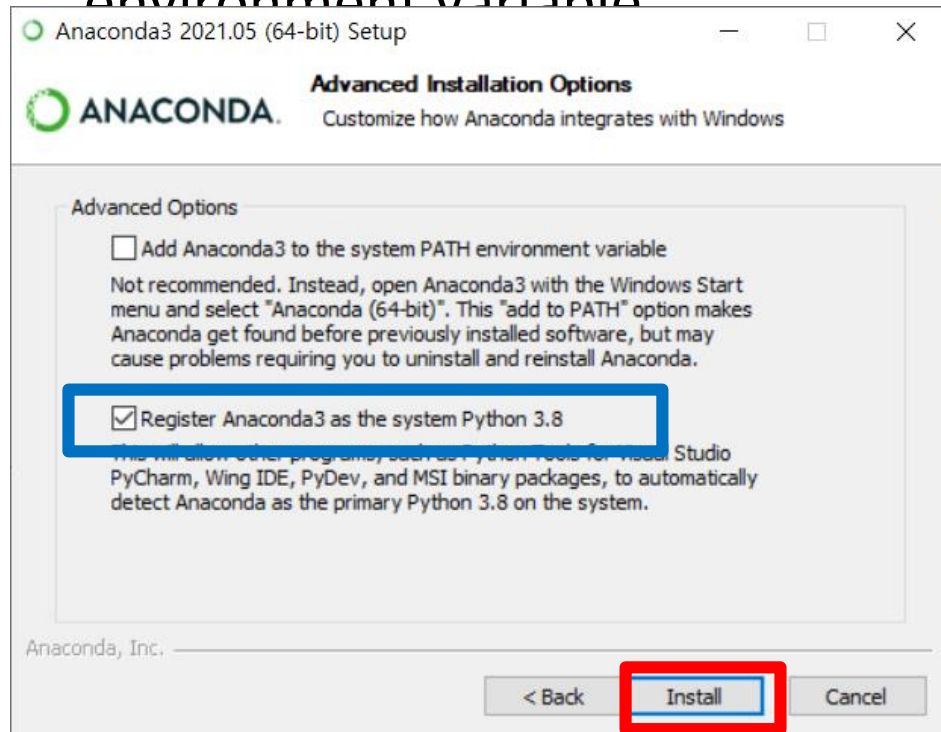
- Use the default path:  
C:\Python\Anaconda3
- Do not use the path including Korean



# Install anaconda (3/5)

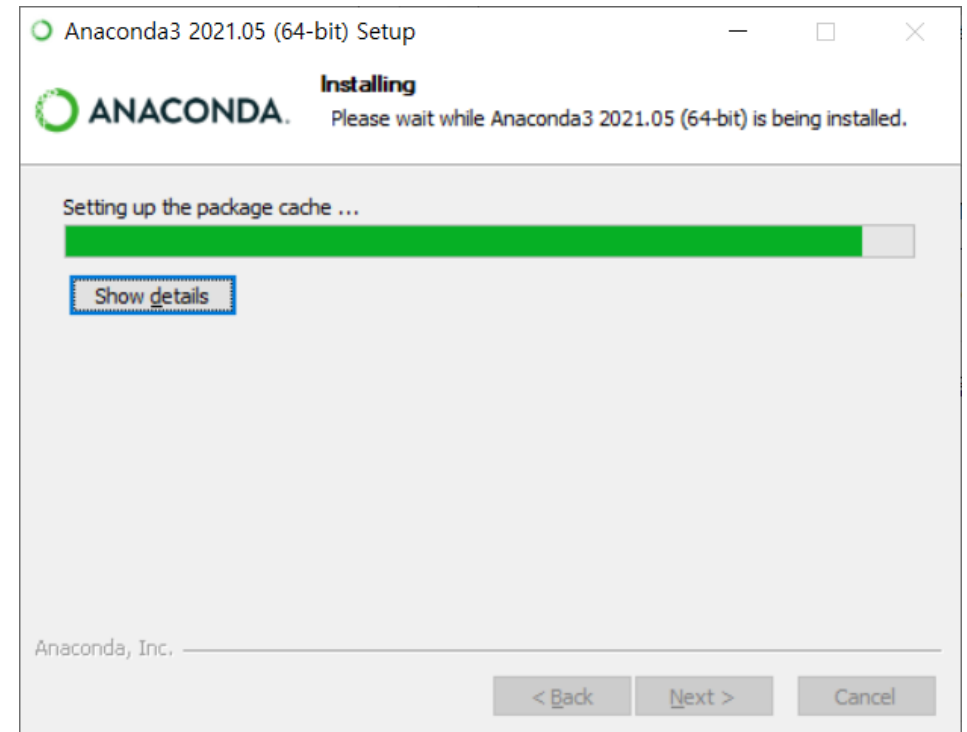
## ❖ Advanced Option

- Setup for Environment variables
- Uncheck "Add Anaconda to my PATH environment variable"



## ❖ Installing

- It takes time...

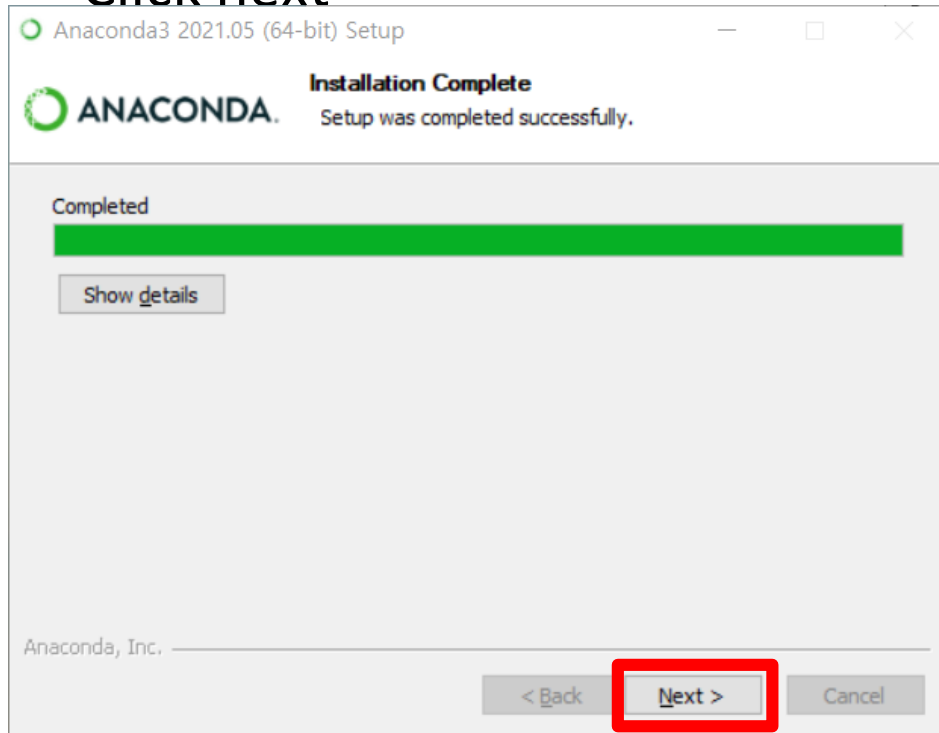




## Install anaconda (4/5)

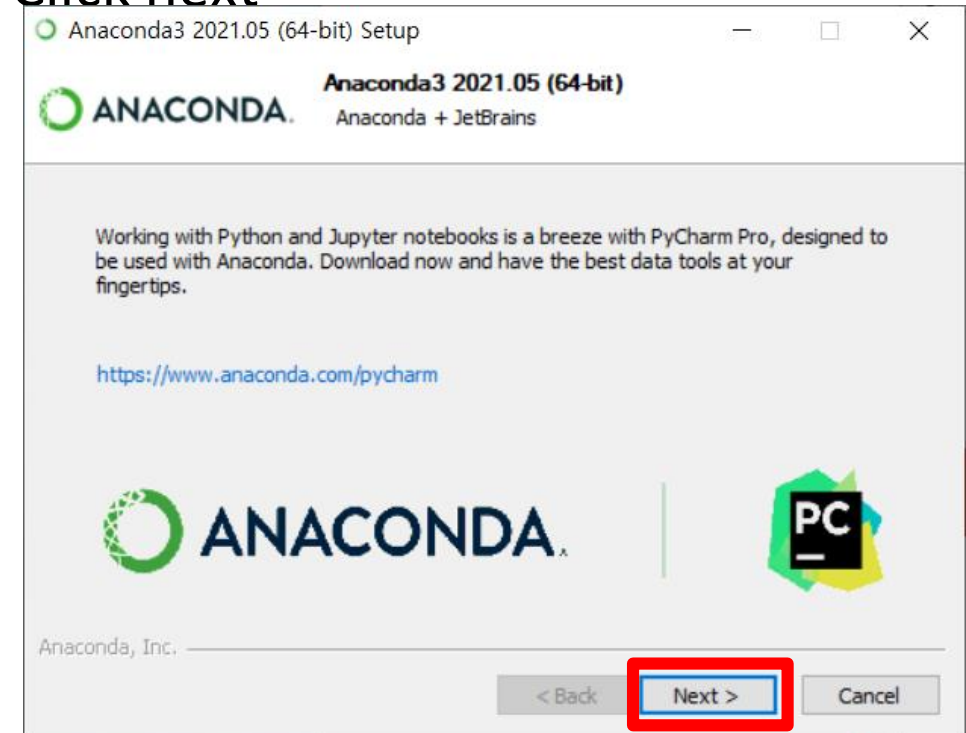
### ❖ Installation Complete

- Click next



### ❖ PyCharm Ad.

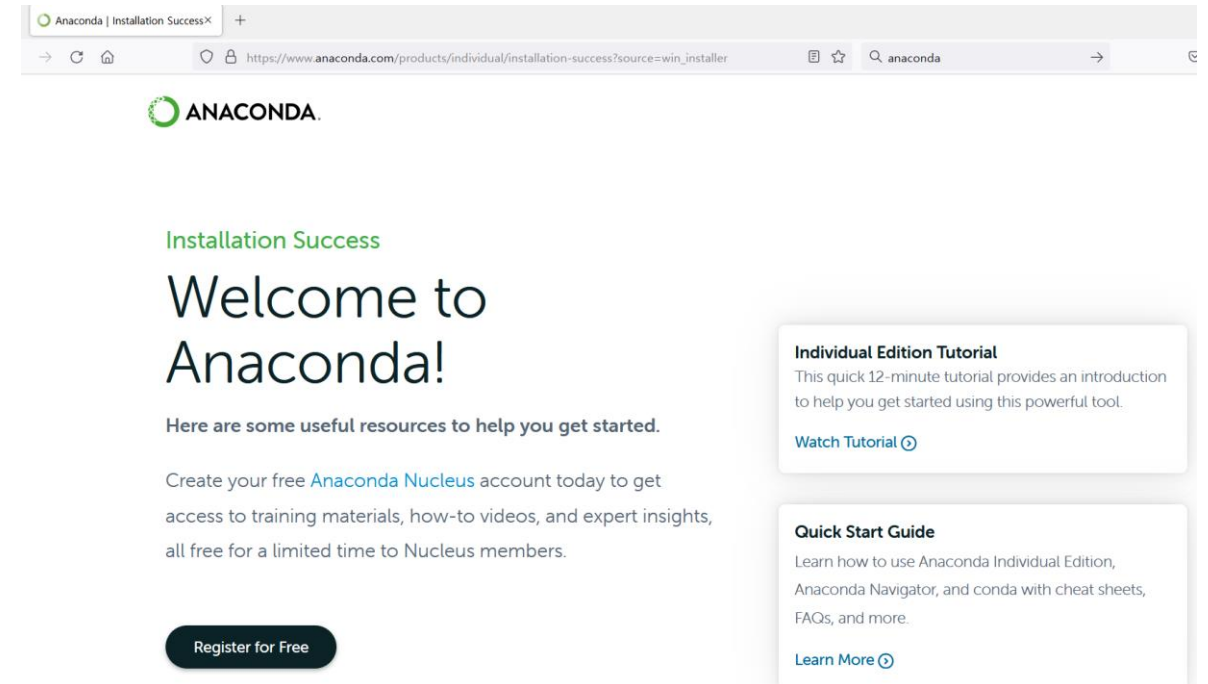
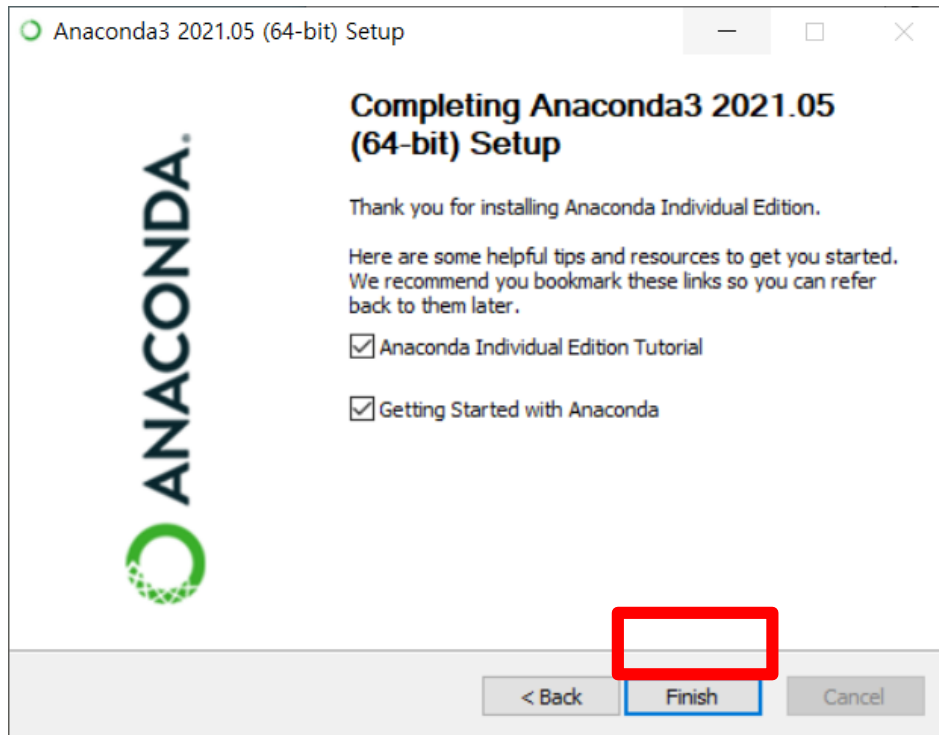
- Click next



# Install anaconda (5/5)

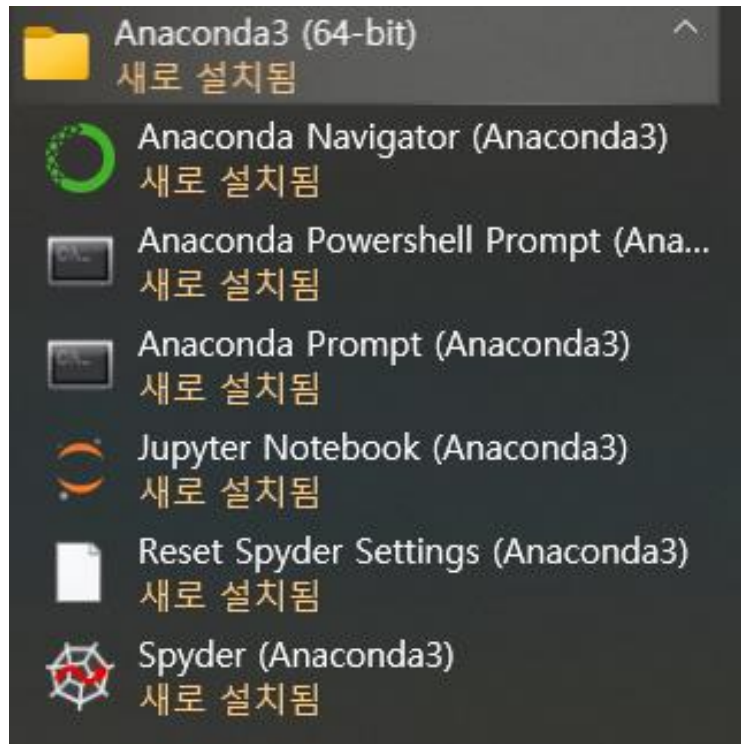
## ❖ Finally done

- Click Finish



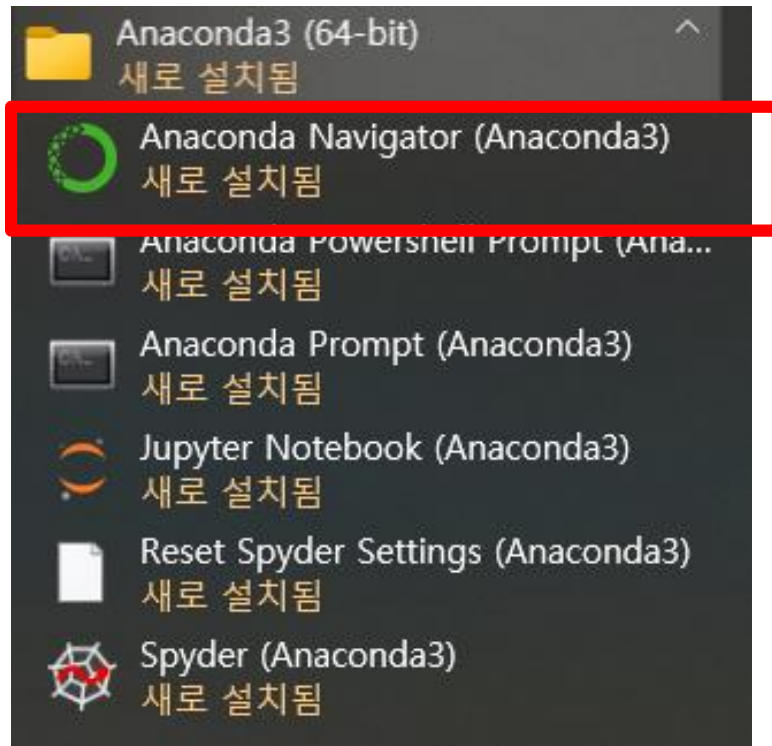
# Really Do we install Anaconda?

- ❖ Check Anaconda from the Windows menu



# Anaconda Navigator (1/3)

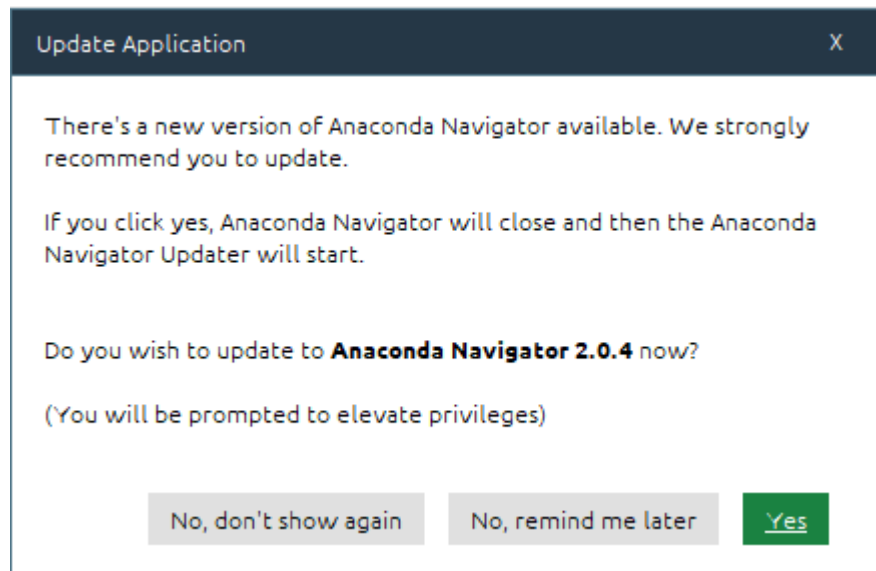
❖ Click Anaconda Navigator



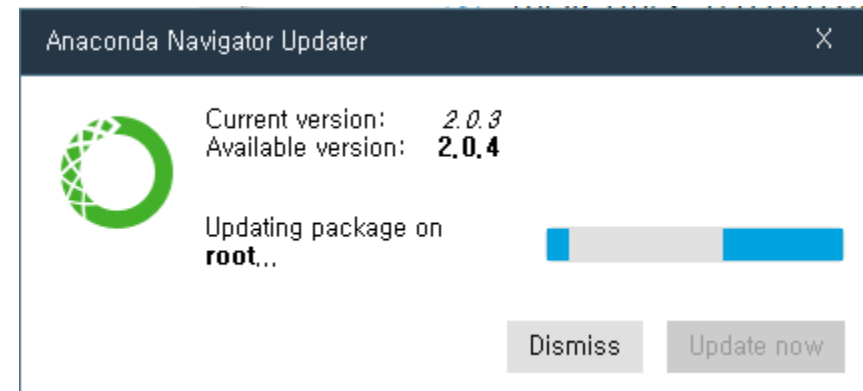
# Anaconda Navigator (2/3)

## ❖ Update question

- Click yes

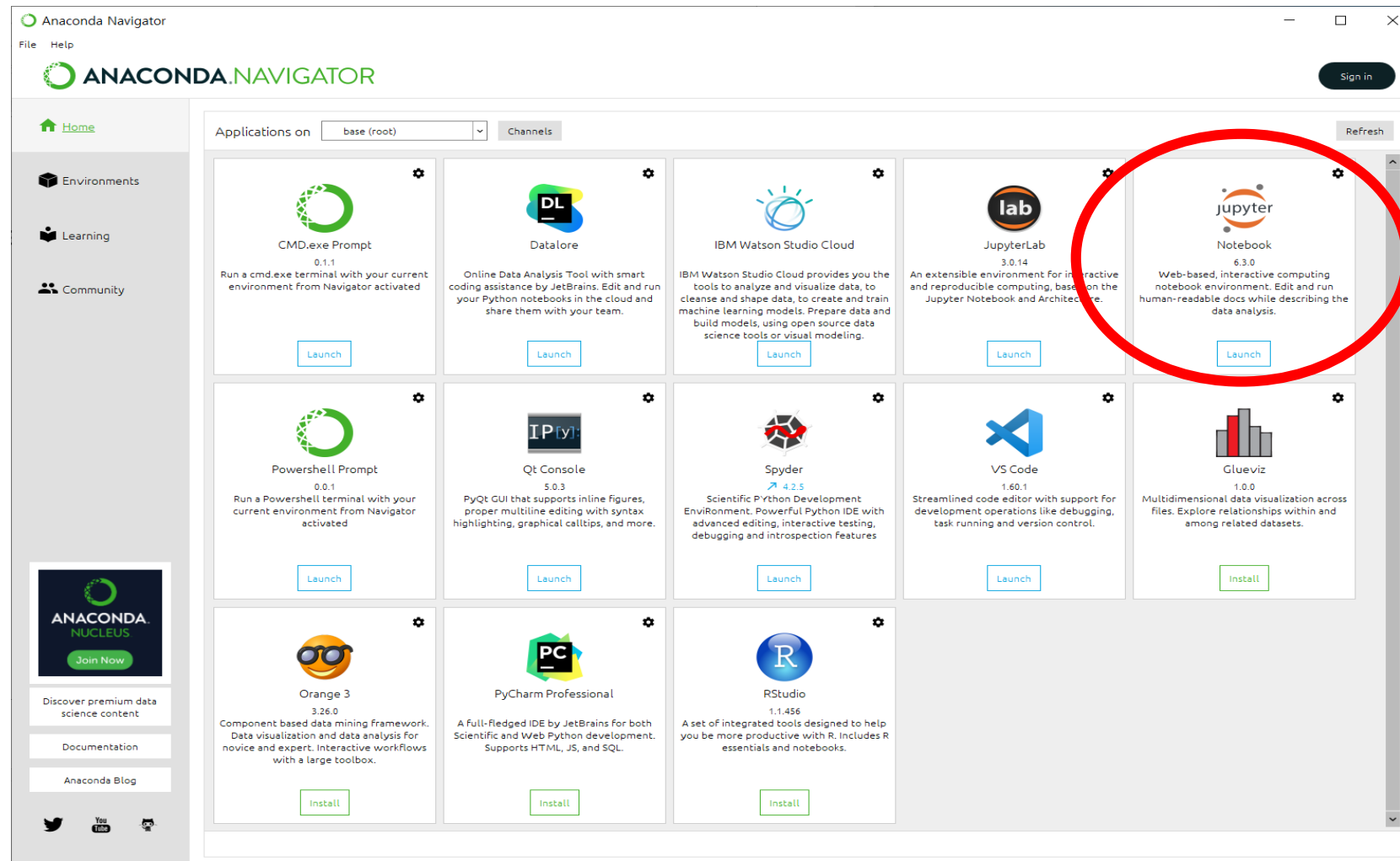


## ❖ Updater



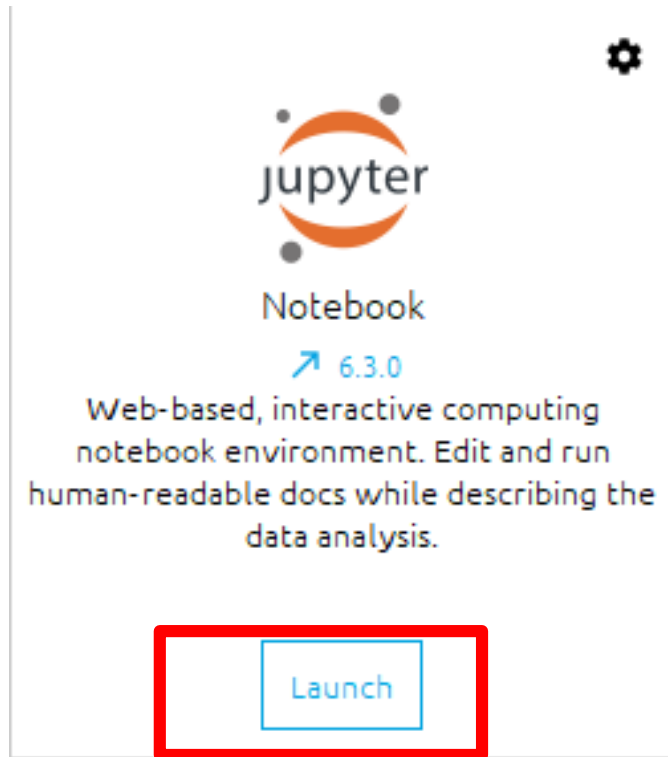
# Anaconda Navigator (3/3)

❖ Jupyter notebook is already installed



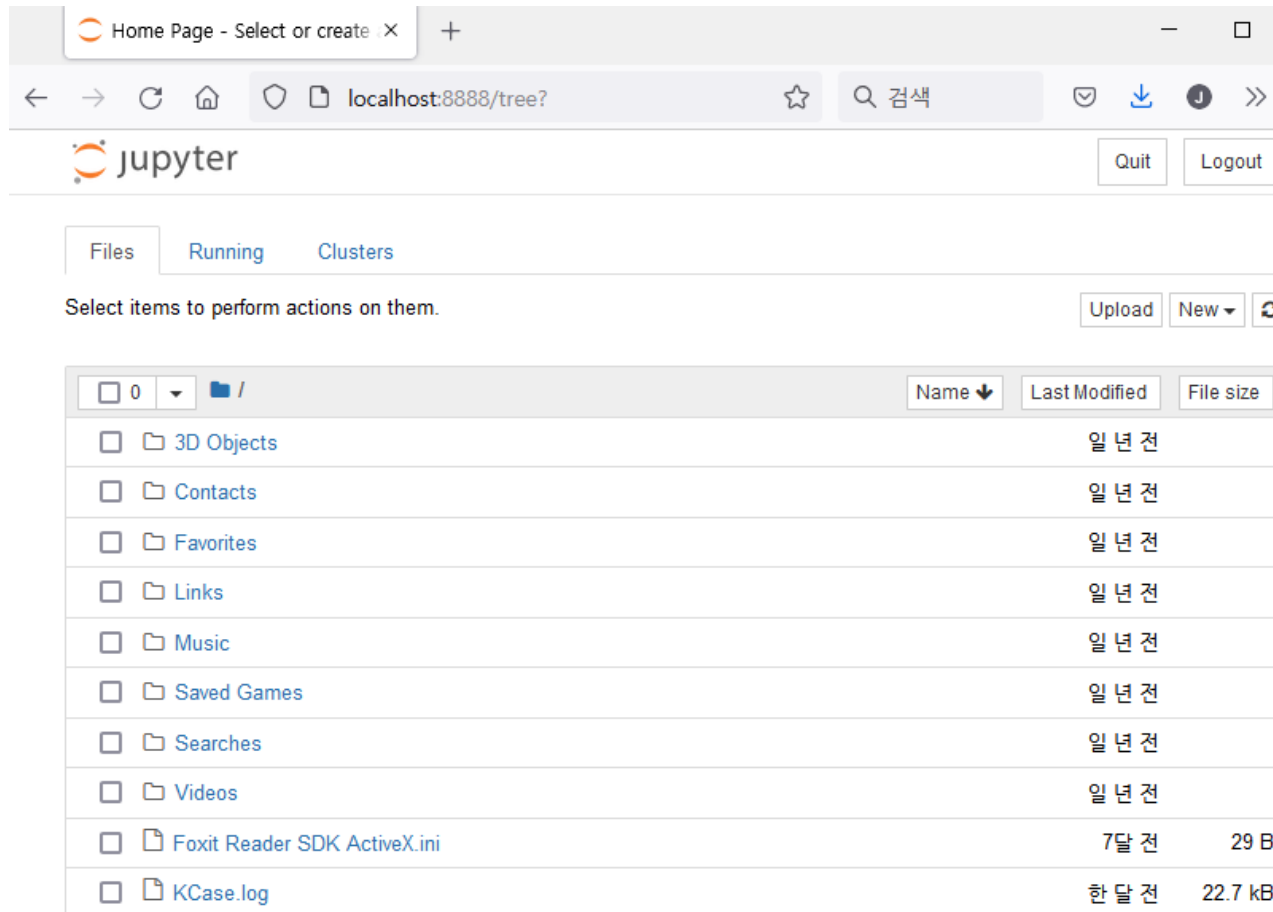
# Jupyter notebook

- ❖ Web-based, interactive computing note environment.
- ❖ Click launch button



# Jupyter notebook home screen

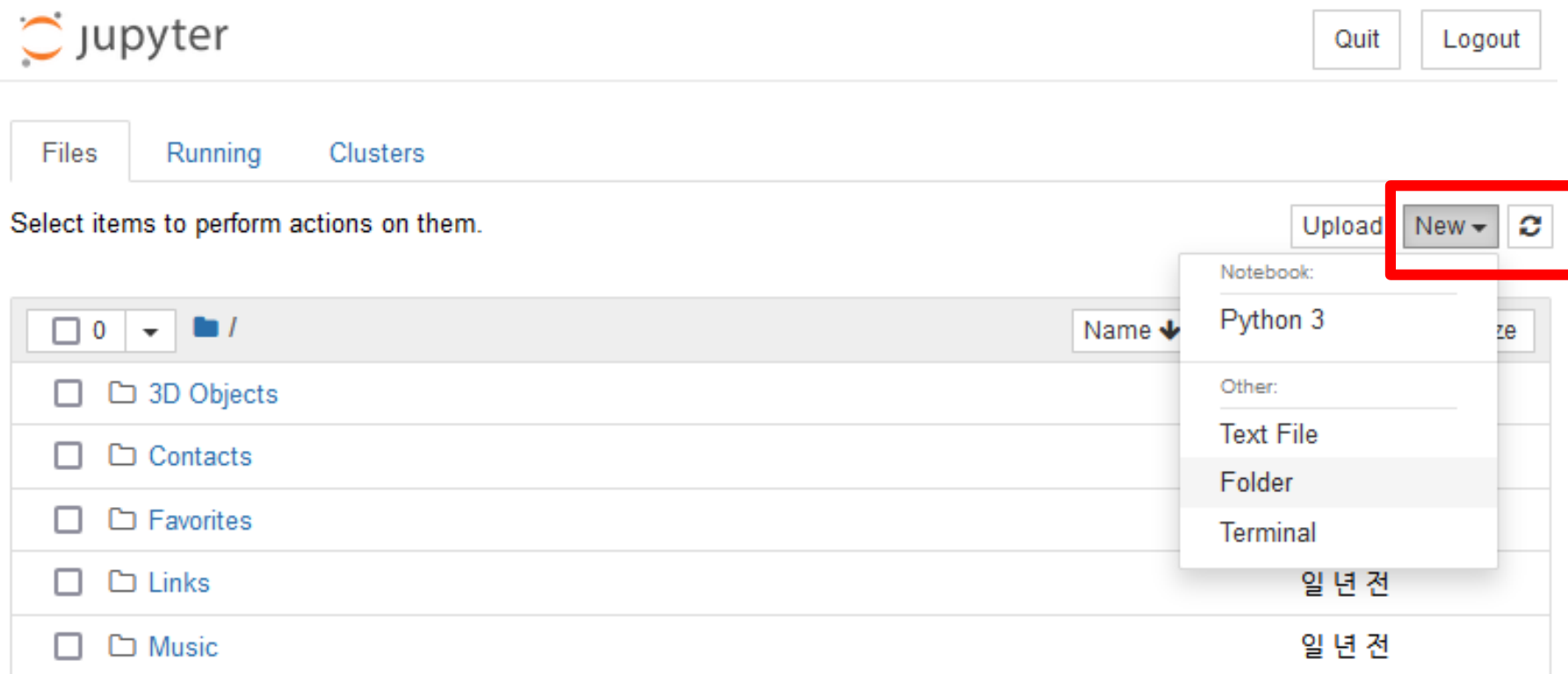
- ❖ By default, the Users folder (c:\users\username\) is displayed





# Create a folder and rename it (1/2)

- ❖ Click the new button in the upper right corner and then click Folder



## Create a folder and rename it (2/2)

### ❖ Untitled Folder is created




### ❖ After checking the folder, click Rename at the top to change it to an appropriate name.

- Example: ProbStat



# Go to the ProbStat folder

 jupyter

QuitLogout

FilesRunningClusters

Select items to perform actions on them.

UploadNew↺

☐ 0 ▾

/ ProbStat

Name ▾

Last Modified

File size

..

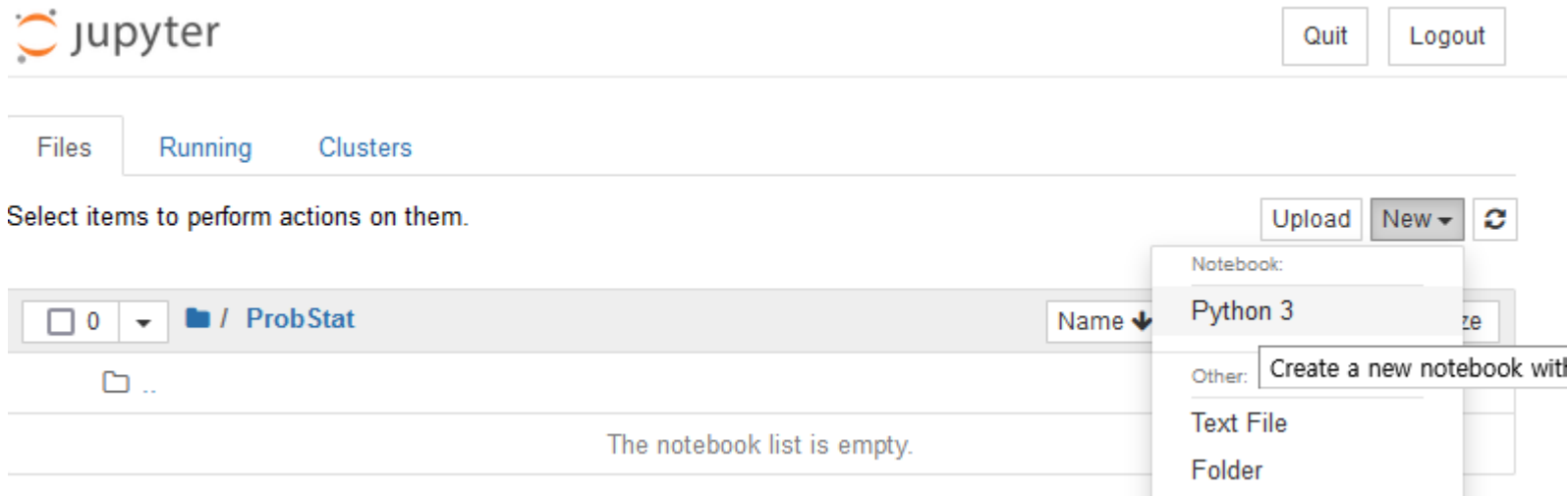
몇 초 전

The notebook list is empty.

# Create a new notebook(1/2)

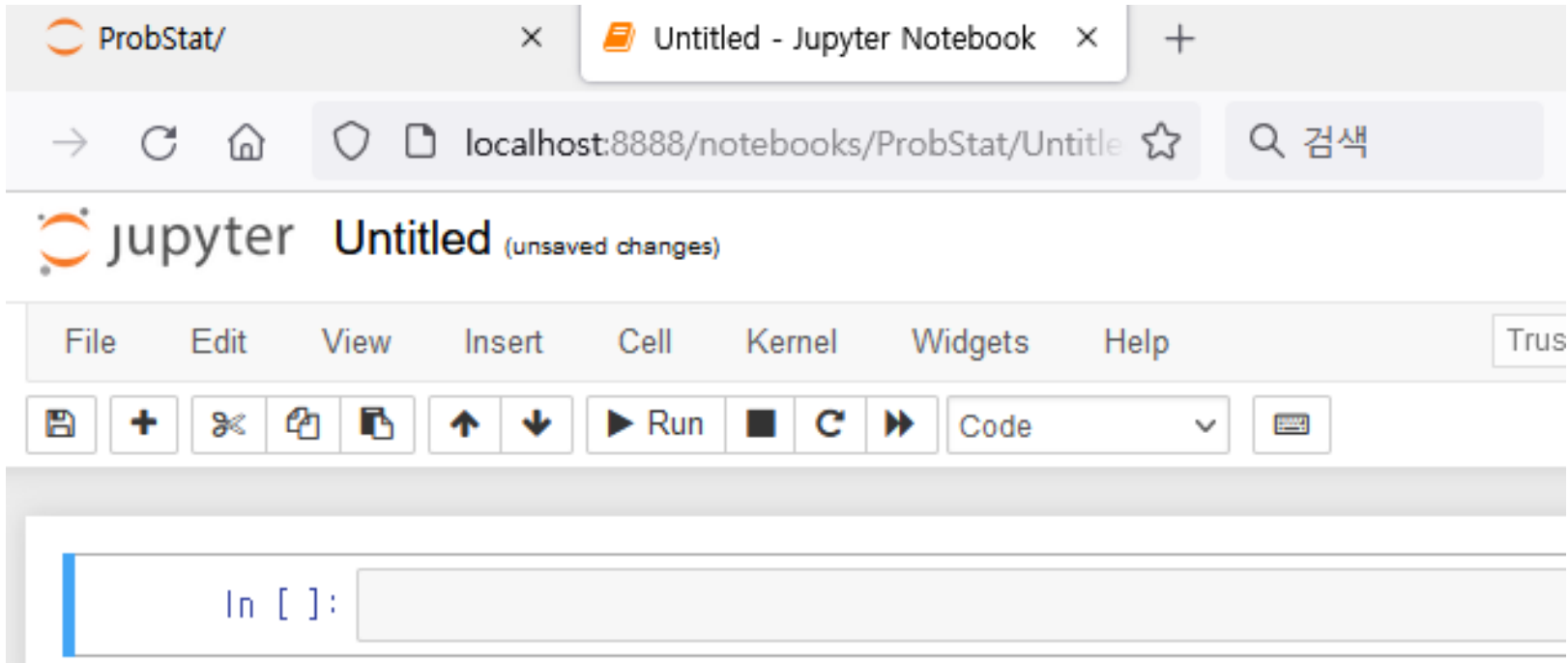
## ❖ Create a new notebook

- Click the New button at the top. click python 3



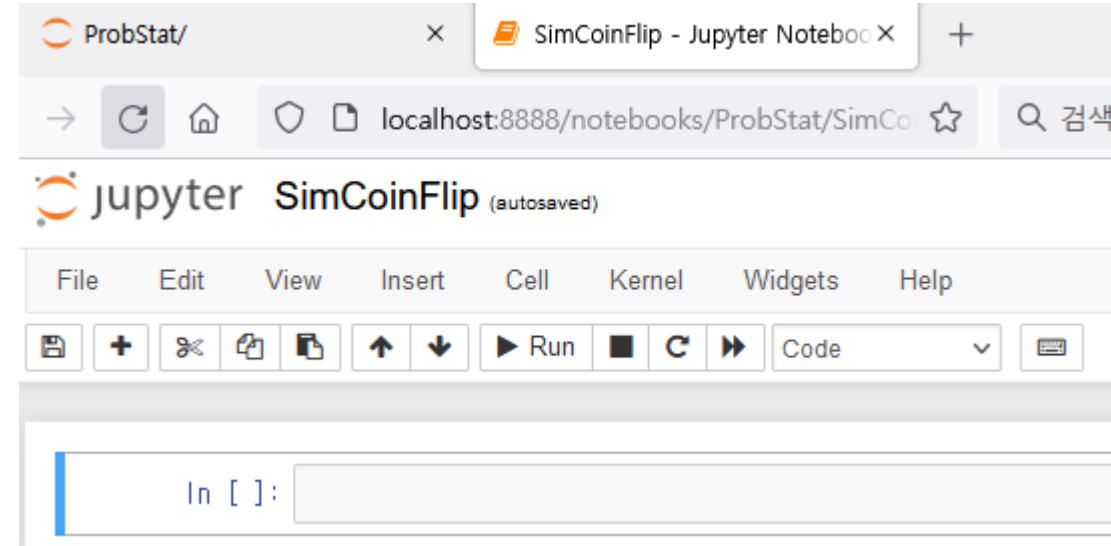
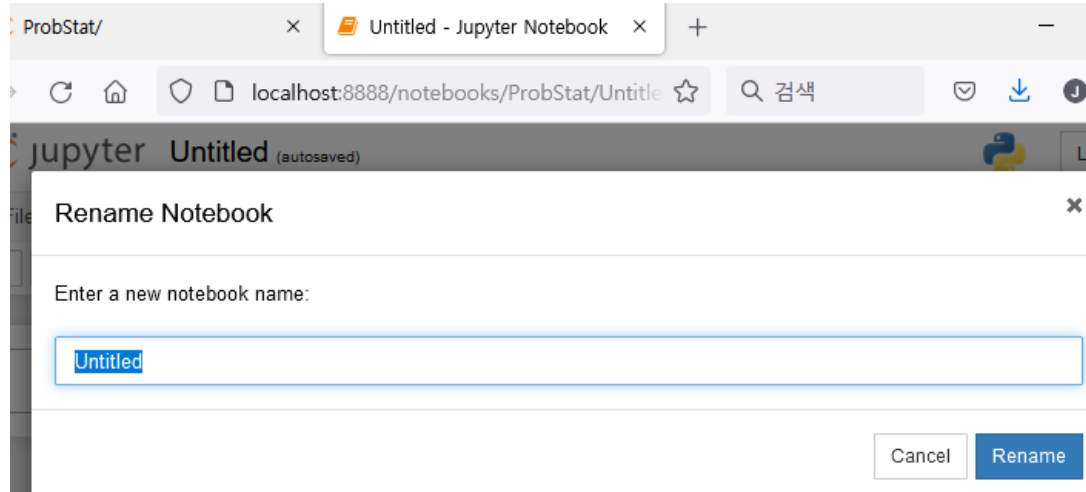
## Create a new notebook(2/2)

- ❖ An empty notebook which is created in the new tab



# Naming your notebook

- ❖ Click the “untitled”
  - New name: SimCoinFlip

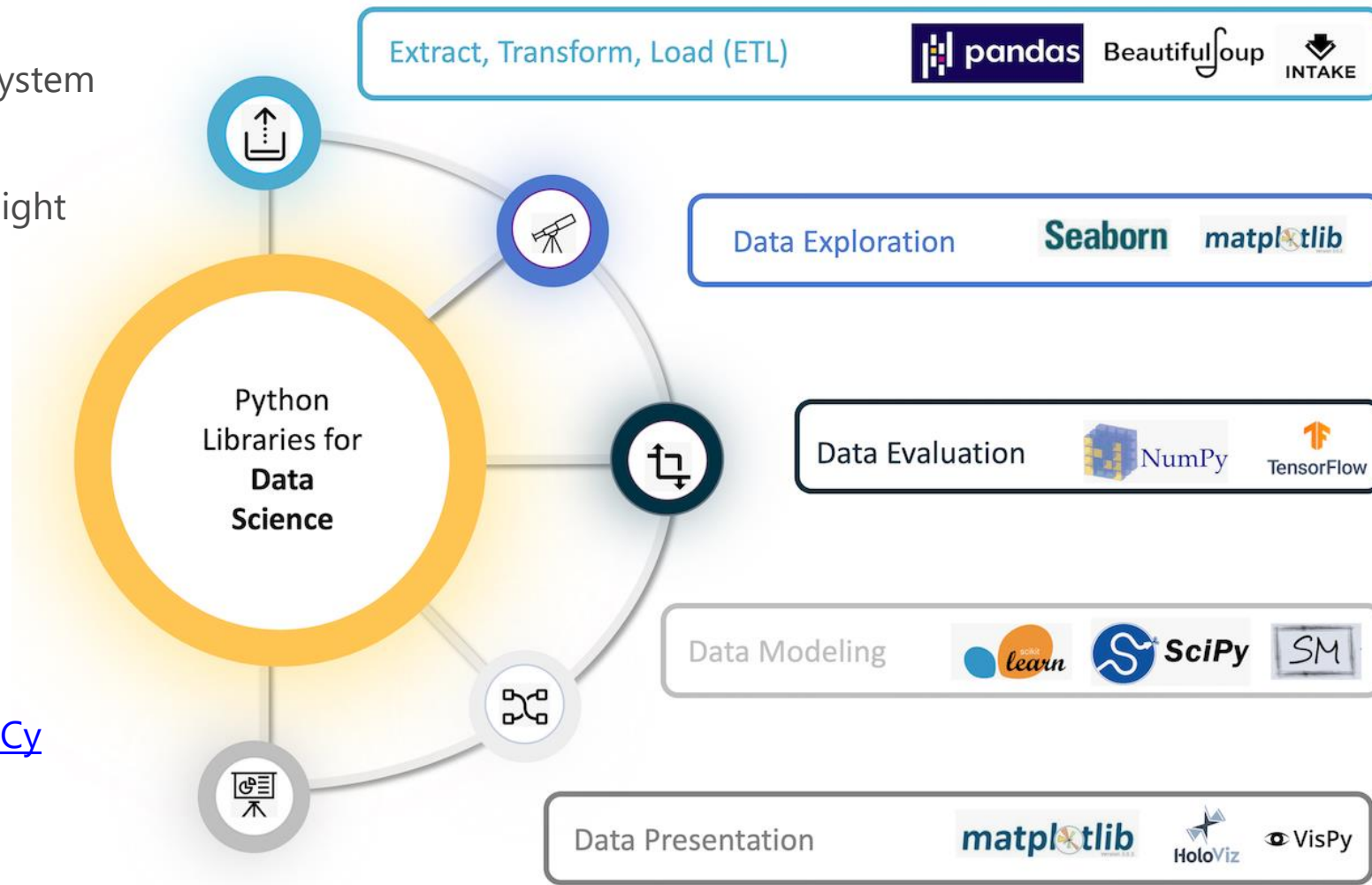


# Eco-system of Python data science libraries (1)

❖ Source : [numpy.org](https://numpy.org)

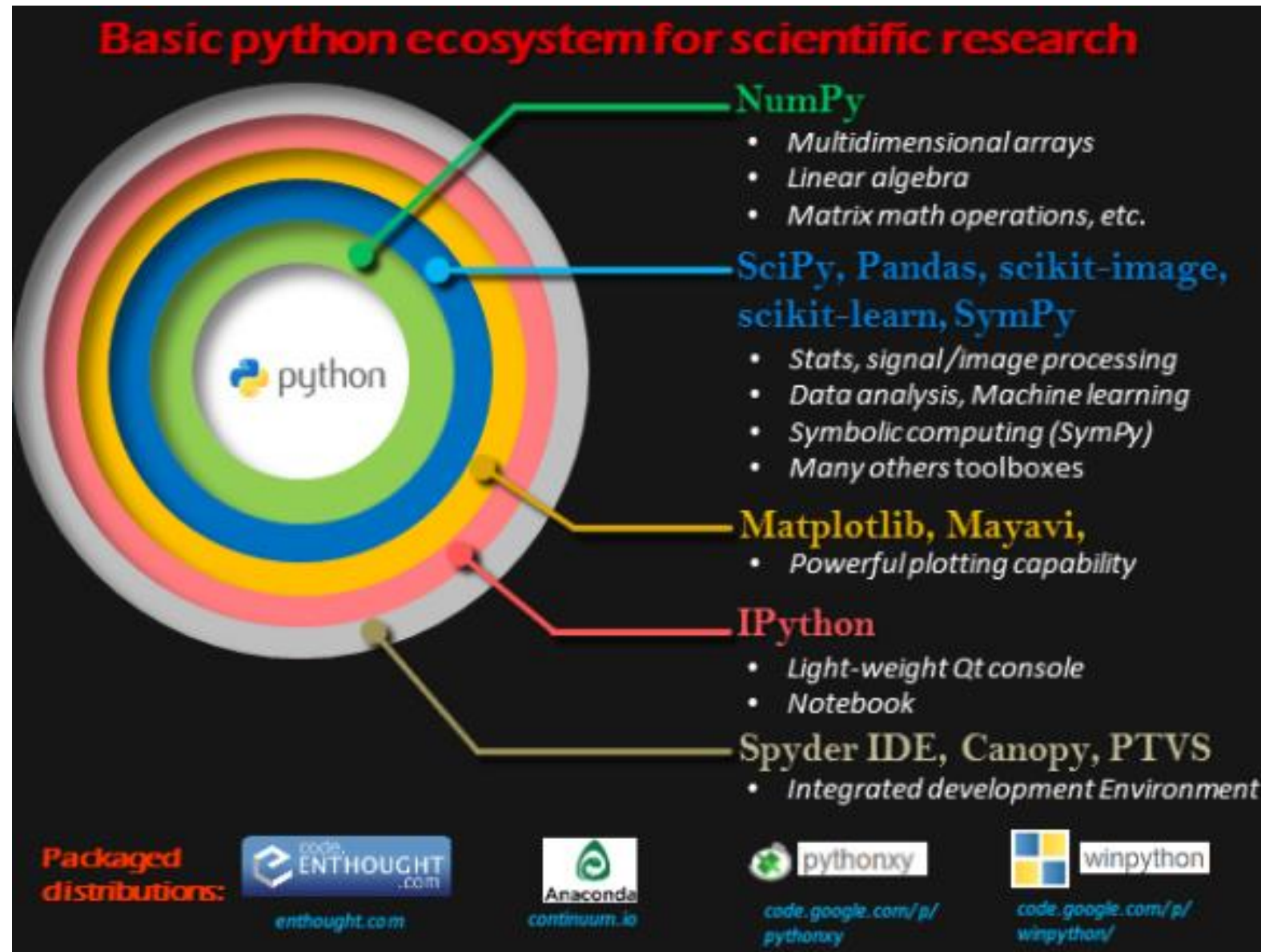
❖ NumPy lies at the core of a rich ecosystem of data science libraries. A typical exploratory data science workflow might look like:

- **Extract, Transform, Load:**  
[Pandas](#), [Intake](#), [PyJanitor](#)
- **Exploratory analysis:**  
[Jupyter](#), [Seaborn](#), [Matplotlib](#), [Altair](#)
- **Model and evaluate:**  
[scikit-learn](#), [statsmodels](#), [PyMC3](#), [spaCy](#)
- **Report in a dashboard:**  
[Dash](#), [Panel](#), [Voila](#)



# Python ecosystem for scientific computing

❖ [\[source\]](#)





# Essential Libraries



## ❖ NumPy:

- introduces objects for multidimensional arrays and matrices, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects
- provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance
- many other python libraries are built on NumPy
- <http://www.numpy.org/>



## ❖ Pandas:

- adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)
- provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.
- allows handling missing data
- <http://pandas.pydata.org/>



## ❖ matplotlib:

- python 2D **plotting library** which produces publication quality figures in a variety of hardcopy formats
- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization
- <https://matplotlib.org/>



## ❖ SciPy:

- **collection of algorithms** for linear algebra, differential equations, numerical integration, optimization, statistics and more
- part of SciPy Stack; built on NumPy
- <https://www.scipy.org/scipylib/>



## ❖ SciKit-Learn:

- provides **machine learning algorithms**: classification, regression, clustering, model validation etc.
- built on NumPy, SciPy and matplotlib
- <http://scikit-learn.org/>

## Anatomy of a Call Expression (1/2)

What function to call

Argument to the function

**f**(**27**)

"Call f on 27."

## Anatomy of a Call Expression (2/2)

What function  
to call

First argument

Second argument

**max**( **15** , **27** )

# Review of Function Concepts

❖ Some functions require a particular number of arguments  
(e.g., **abs** must be called on one value)

❖ Arguments can be named in the call expression:

**round(number=12.34)**

But the names must match the documentation

# Ints and Floats

## ❖ Python has two real number types

- int: an integer of any size
- float: a number with an optional fractional part

An int never has a decimal point; a float always does  
A float might be printed using scientific notation

## ❖ Three limitations of float values:

- They have limited size (but the limit is huge)
- They have limited precision of 15-16 decimal places
- After arithmetic, the final few decimal places can be wrong

# Text and Strings

- ❖ A string value is a snippet of text of any length
  - 'a'
  - 'word'
  - "there can be 2 sentences. Here's the second!"
- ❖ Strings consisting of numbers can be converted to numbers
  - `int('12')`
  - `float('1.2')`
- ❖ Any value can be converted to a string
  - `str(5)`

# Conversions

## ❖ Strings that contain numbers can be converted to numbers

- `int('12')`
- `float('1.2')`
- `float('one point two')` # Not a good idea!

## ❖ Any value can be converted to a string

- `str(5)`

## ❖ Numbers can be converted to other numeric types

- `float(1)`
- `int(1.2)` # DANGER: loses information!

# Discussion Question

❖ Assume you have run the following statements:

- `x = 3`
- `y = '4'`
- `z = '5.6'`

❖ What's the source of the error in each example?

- 1. `x + y`
- 2. `x + int(y + z)`
- 3. `str(x) + int(y)`
- 3. `y + float(z)`



# What is Numpy (Numerical Python) ?

- ❖ NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a **multidimensional array object**, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.
- ❖ At the core of the NumPy package, is the **ndarray** object. This encapsulates *n-dimensional arrays 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 at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
  - The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
  - NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
  - A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

# Ndarray basics

## ❖ Attributes

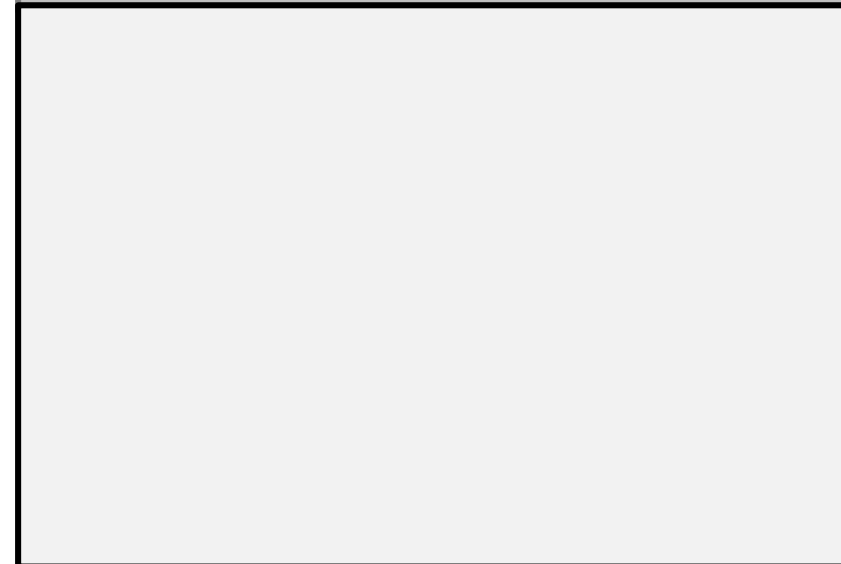
- ndarray.**ndim** : *the number of axes (dimensions)* of the array.
- ndarray.**shape** : the dimensions of the array. This is *a tuple of integers indicating the size of the array in each dimension*. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.
- ndarray.**size** : *the total number of elements* of the array. This is equal to the product of the elements of shape.
- ndarray.**dtype** : an object describing *the type of the elements* in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
- ndarray.**itemsize** : *the size in bytes of each element* of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is 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.

```
import numpy as np

a = np.arange(15).reshape(3, 5)
print('a :=\n', a)
print('a.shape :=', a.shape)
print('a.ndim :=', a.ndim)
print('a.dtype :=', a.dtype)
print('a.itemsize :=', a.itemsize)
print('a.size := ', a.size)
print('type(a) := ', type(a))

b = np.array([6, 7, 8])
print('\nb :=\n', b)
print('type(b) := ', type(b))
```

Execution Result



# Array Creation (1)

## ❖ Narray – n-dimensional array

- Can create an array from a regular Python list or tuple using the `array` function
- `array` transforms [sequences of sequences](#) into two-dimensional arrays, [sequences of sequences of sequences](#) into three-dimensional arrays, and so on.

```
import numpy as np

a1 = np.array([1,2,3])
a2 = np.array([1,2,3],[2,3,4])
a3 = np.array([[1,2,3],[2,3,4]],[[3,4,5],[4,5,6]])
a4 = np.array([[[1,2,3],[2,3,4]],[[3,4,5],[4,5,6]],
               [[5,6,7],[6,7,8]],[[7,8,9],[8,9,0]]])

x1 = [a1,a2,a3,a4]
for i in range(len(x1)) :
    print("\n{} dimensional array a{{{}} :=\n".format(
x1[i].ndim, i+1, x1[i].shape), x1[i])
```

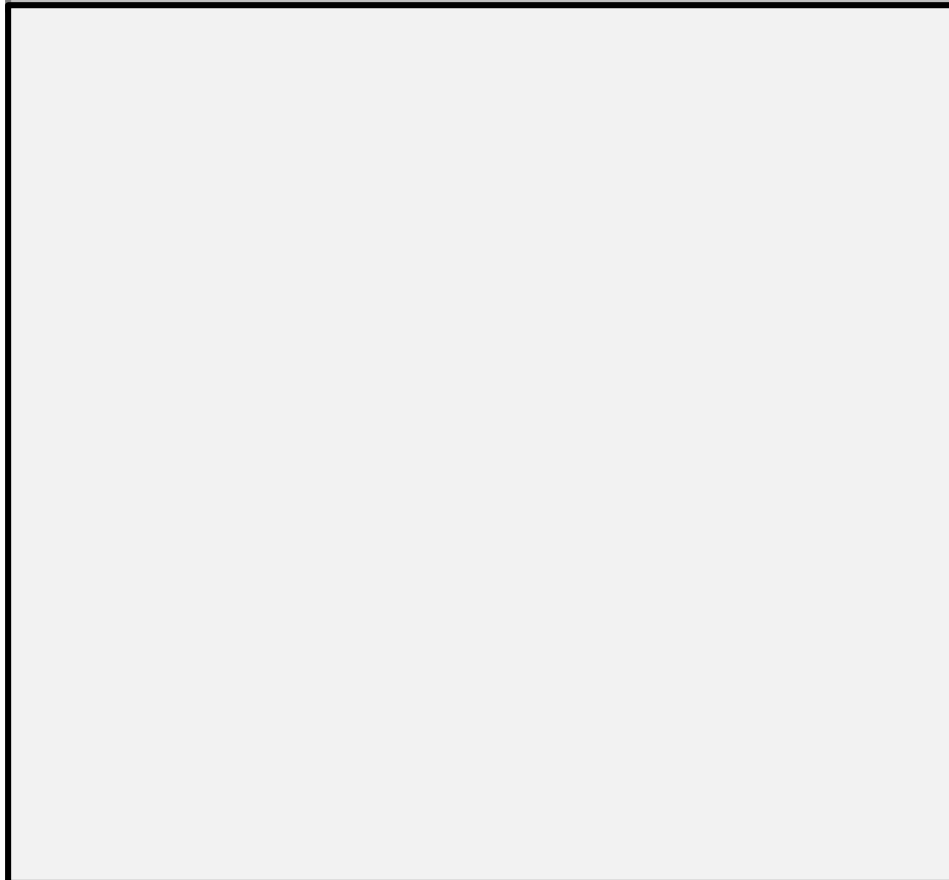
Execution Result

# Array Creation (2)

- ❖ **zeros**(shape, dtype=float, order='C')
  - Return a new array of given shape and type, filled with 0s.
- ❖ **ones**(shape, dtype=None, order='C')
  - Return a new array of given shape and type, filled with 1s
- ❖ **full**(shape, fill\_value, dtype=None, order='C')
  - Return a new array of given shape and type, filled with 'fill\_value'.
- ❖ **eye**(N, M=None, k=0, dtype=<class 'float'>, order='C')
  - Return a 2-D array with ones on the diagonal and zeros elsewhere.
- ❖ **random**()
  - Return random floats in the half-open interval [0.0, 1.0)
- ❖ **arange**([start,] stop[, step,], dtype=None)
  - Returns evenly spaced values within a given interval. Values are generated within the half open interval [start, stop). For integer arguments the function is equivalent to the Python built-in *range* function, but returns an ndarray rather than a list.
- ❖ **linspace**(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)
  - Returns "num" evenly spaced samples, calculated over the interval [start, stop].

```
a5 = np.zeros((2,3))
a6 = np.ones((1,1,3))
a7 = np.full((2,2),5)
a8 = np.eye(2)
a9 = np.random.random((2,2))
a10 = np.arange(1,10,2)
a11 = np.linspace(0, 2, 9)
a12 = np.arange(0, 2, 0.25)
```

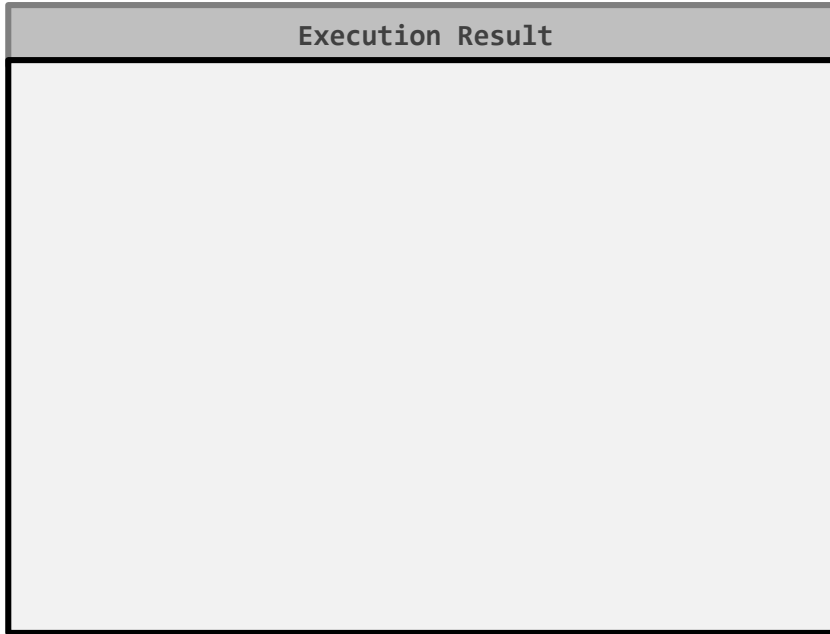
Execution Result



# Python list vs. Numpy Array

❖ +1, \*2

❖ List + List vs. Array + Array



```
import numpy as np

plist1 = [1,2,3]
plist2 = [4,5,6]

arr1 = np.array(plist1)
arr2 = np.array(plist2)
```

```
### add 1 to each item in array
#print('plist1+1 :=', plist1+1) #producing an error
plist3 = []
for i in range(len(plist1)) :
    plist3.append(plist1[i]+1)
print('plist3 :=', plist3)
print('arr1+1 :=', arr1+1)
```

```
### multiply by 2 to each item in array
print('\nplist1*2 :=', plist1*2)
print('arr1*2 :=', arr1*2)
```

```
### element wise add of two arrays
print('\nplist1+plist2 :=', plist1+plist2)
plist3 = []
for i in range(len(plist1)) :
    plist3.append(plist1[i]+plist2[i])
print('plist3 := ', plist3)
```

```
print('arr1+arr2 :=', arr1 + arr2)
```

```
### element wise multiply of two arrays
#print('plist1*plist2 :=', plist1*plist2) #producing
an error
plist3 = []
for i in range(len(plist1)) :
    plist3.append(plist1[i]*plist2[i])
print('\nplist3 := ', plist3)
print('arr1*arr2 :=', arr1*arr2)
```

# Basic Operations

- ❖ Arithmetic operators on arrays apply **elementwise**.

A new array is created and filled with the result.

- ❖ Unlike in many matrix languages, the product operator \*

operates **elementwise** in NumPy arrays.

- The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

- ❖ Universal Functions

- NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called “universal functions”(ufunc). Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
>>> 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])

>>> 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]])
>>> 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.] )
```

# Array Indexing & Slicing

## ❖ Slicing:

- Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:
- You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array.

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])    # Prints "2"
b[0, 0] = 2       # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])    # Prints "2"

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                              #          [ 6]
                              #          [10]] (3, 1)"
```

# Boolean Array Indexing

- ❖ Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

```
import numpy as np

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

bool_idx = (a > 2)    # Find the elements of a that are bigger than 2;
                      # this returns a numpy array of Booleans of the same
                      # shape as a, where each slot of bool_idx tells
                      # whether that element of a is > 2.

print(bool_idx)       # Prints "[[False False]
                      #      [ True  True]
                      #      [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])    # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])       # Prints "[3 4 5 6]"
```



# Pandas

## ❖ Pandas is mainly used for data analysis

- Pandas allows importing data from various file formats such as CSV(Comma Separated Values), JSON, SQL, Microsoft Excel
- Pandas allows various data manipulation operations such as merging, reshaping, selecting, as well as data cleaning and data wrangling features

## ❖ Data Structures in Pandas

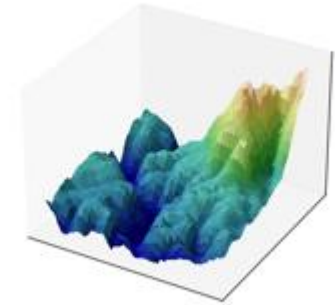
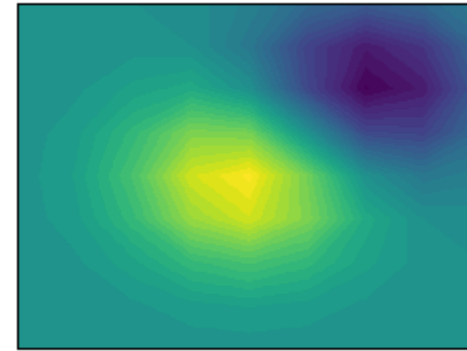
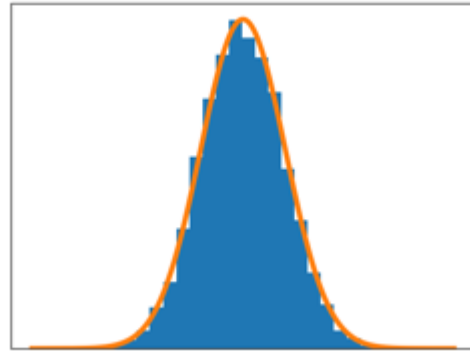
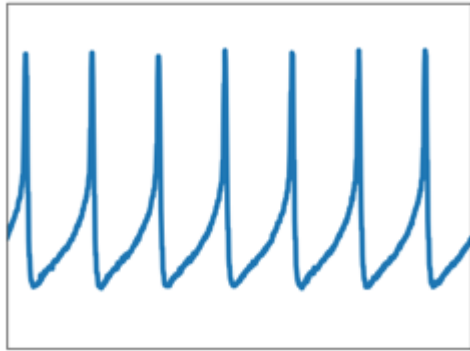
Dim	Name	Description
1	<b>Series</b>	1D labeled homogeneously-typed array
2	<b>DataFrame</b>	General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column

## ❖ [Tutorial - 10 minutes to Pandas](#) Contents

- Object Creation : Series, DataFrame
- **Viewing Data**
- **Selection** : Getting; Selection by label, by position; Boolean Indexing, Setting;
- Missing Data
- Operations : Stats, Apply, Histogramming, String Methods
- Merge : Concat, Join,
- Grouping
- Reshaping : Stack, Pivot Tables
- Time Series
- Categoricals
- Plotting
- **Getting data in/out** : CSV, HDF5, MS Excel

# Matplotlib - <https://matplotlib.org/index.html>

- ❖ Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python



- ❖ basic plotting will be covered in this lecture.
  - <https://cs231n.github.io/python-numpy-tutorial/#jupyter-and-colab-notebooks>

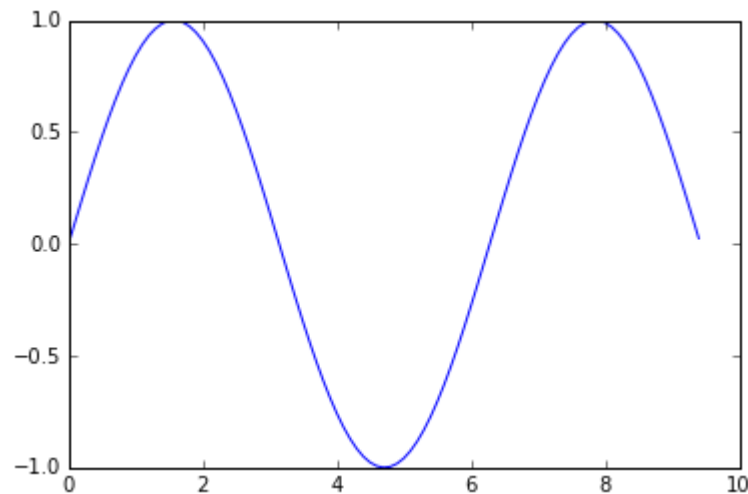
# Line Plot

- ❖ The most important function in matplotlib is `plot`, which allows you to plot 2D data.

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

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show() # You must call plt.show() to make graphics appear.
```



# Line Plot

- ❖ With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

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

```
# Compute the x and y coordinates for points on sine and cosine curves
```

```
x = np.arange(0, 3 * np.pi, 0.1)
```

```
y_sin = np.sin(x)
```

```
y_cos = np.cos(x)
```

```
# Plot the points using matplotlib
```

```
plt.plot(x, y_sin)
```

```
plt.plot(x, y_cos)
```

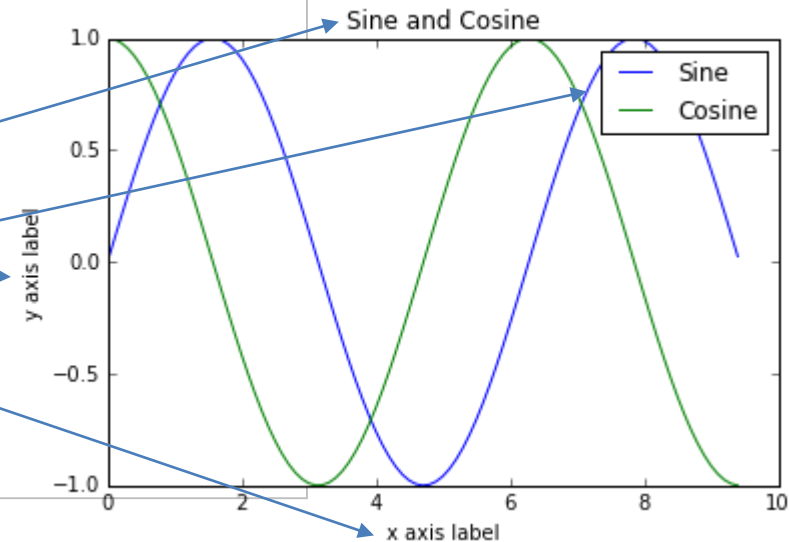
```
plt.xlabel('x axis label')
```

```
plt.ylabel('y axis label')
```

```
plt.title('Sine and Cosine')
```

```
plt.legend(['Sine', 'Cosine'])
```

```
plt.show()
```



# Subplots

- ❖ You can plot different things in the same figure using the **subplot** function.

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

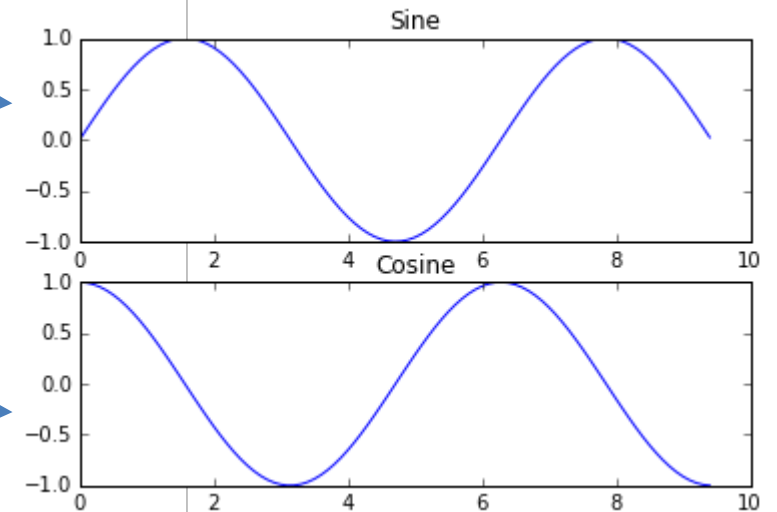
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

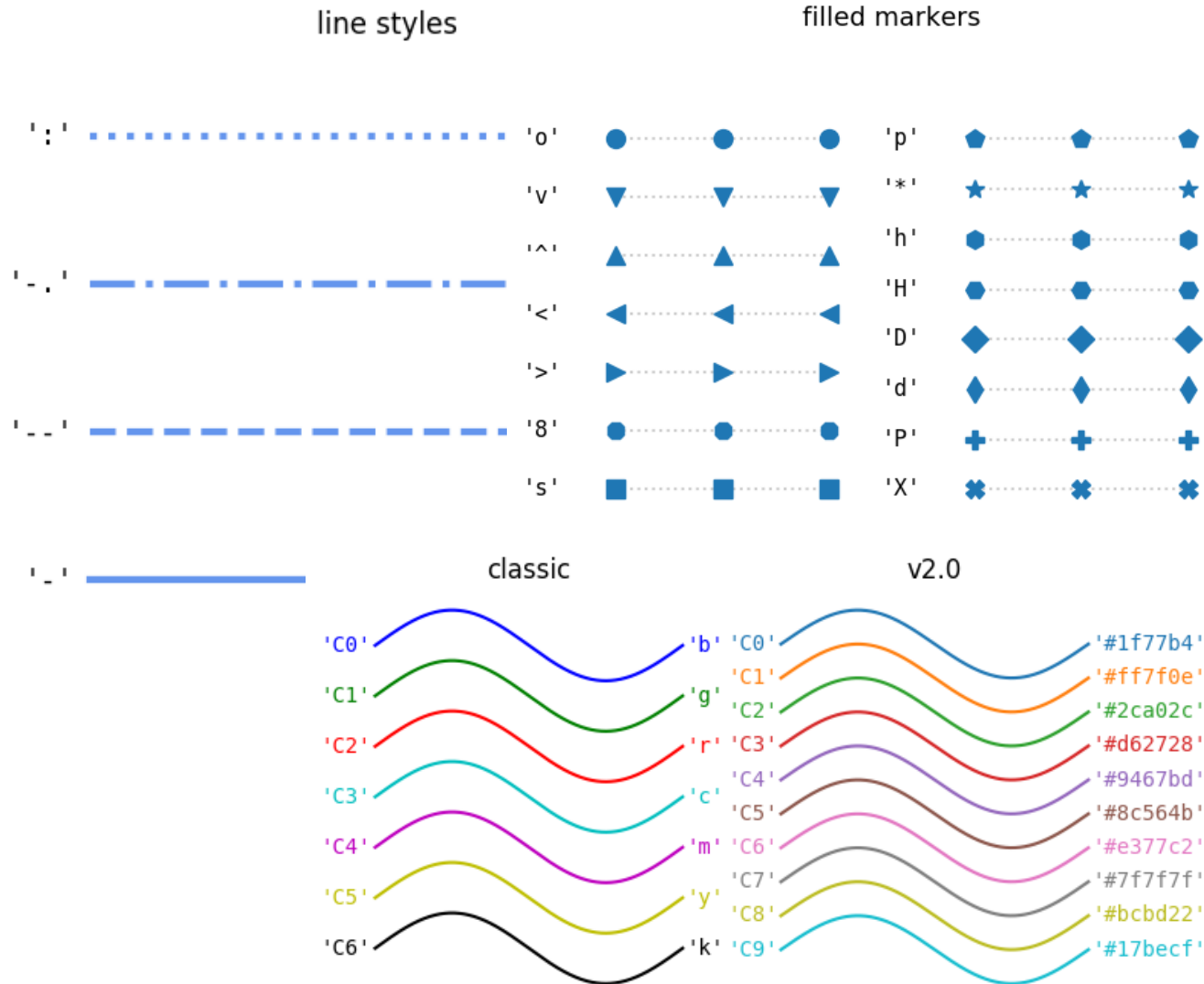
# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



# Style

## ❖ Line Style, Line Color, Maker



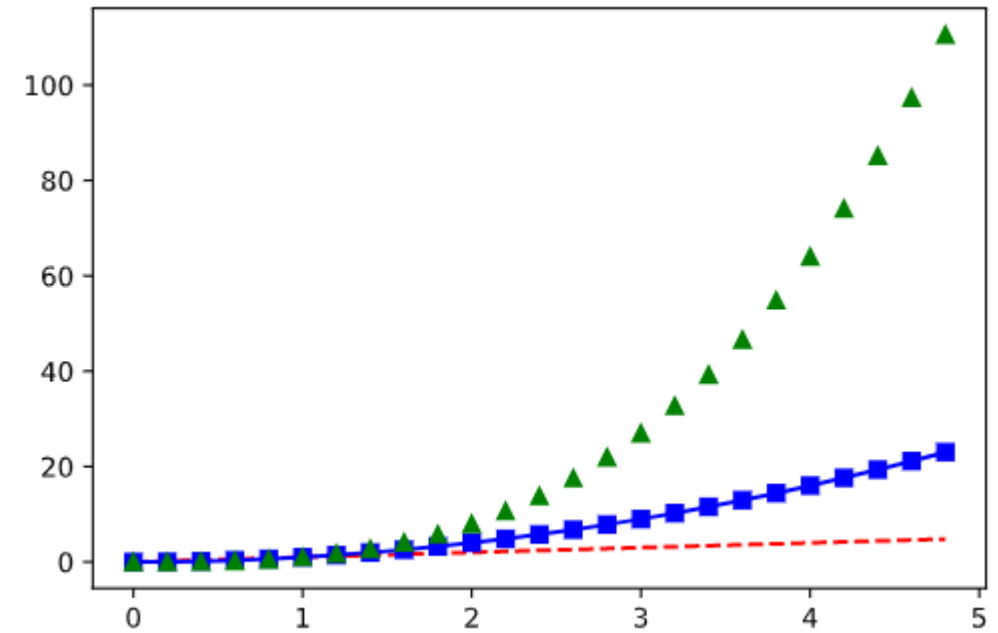
```
import numpy as np
```

```
# evenly sampled time at 200ms intervals
```

```
t = np.arange(0., 5., 0.2)
```

```
# red dashes, blue squares and green triangles
```

```
plt.plot(t, t, 'r--', t, t**2, 'b-s', t, t**3, 'g^')
plt.show()
```



# Q&A

## **Prof. Jeon, Sangryul**

Computer Vision Lab.

Pusan National University, Korea

Tel: +82-51-510-2423

Web: <http://sr-jeon.github.io/>

E-mail: [srjeonn@pusan.ac.kr](mailto:srjeonn@pusan.ac.kr)