COMP7409C Machine Learning in Trading and Finance

This course offers an overview of Machine Learning to students and prepares them with essential skills to apply it in problem-solving for Trading and Finance.

About this course

- Instructor
  - H.F. TING
- TAs:
  - Jolly CHENG
  - Karen CHOW

About this course

Assessment

- Three programming assignments (65%):
  - A1: 20%
  - A2: 20%
  - A3: 25%
- Final Exam (35%)

Topics to be covered

- Basics of Data Science
- Basics of Machine Learning
- Financial fault detection using ML
- Financial forecasting using ML
- NLP and Sentiment Analysis
- Trading bot with reinforcement learning
- Recommendation system
- Option pricing with deep learning

Assignment release schedule

L1 11/6
L2 14/6
L3 18/6
L4 25/6 release A1
L5 28/6
L6 2/7
L7 9/7 release A2
L8 12/7
L9 16/7
L10 23/7 release A3

Programming Languages: Python

Why Python?

- Free & open source
- Simple syntax, easy to self-learn and understand
- General-purpose programming language
- Has a rich set of libraries and packages designed for data science and machine learning
- Good memory management and high performance

Programming Languages for this course: Python

Why Python?

- Free & open source
- Simple syntax, easy to self-learn and understand
- General-purpose programming language
- Has a rich set of libraries and packages designed for data science and machine learning
- Good memory management and high performance

But,

- I don't know where to find these packages

and even if I know where to download the package I need,

- I don't know how to install it; different packages would have different installation procedures.

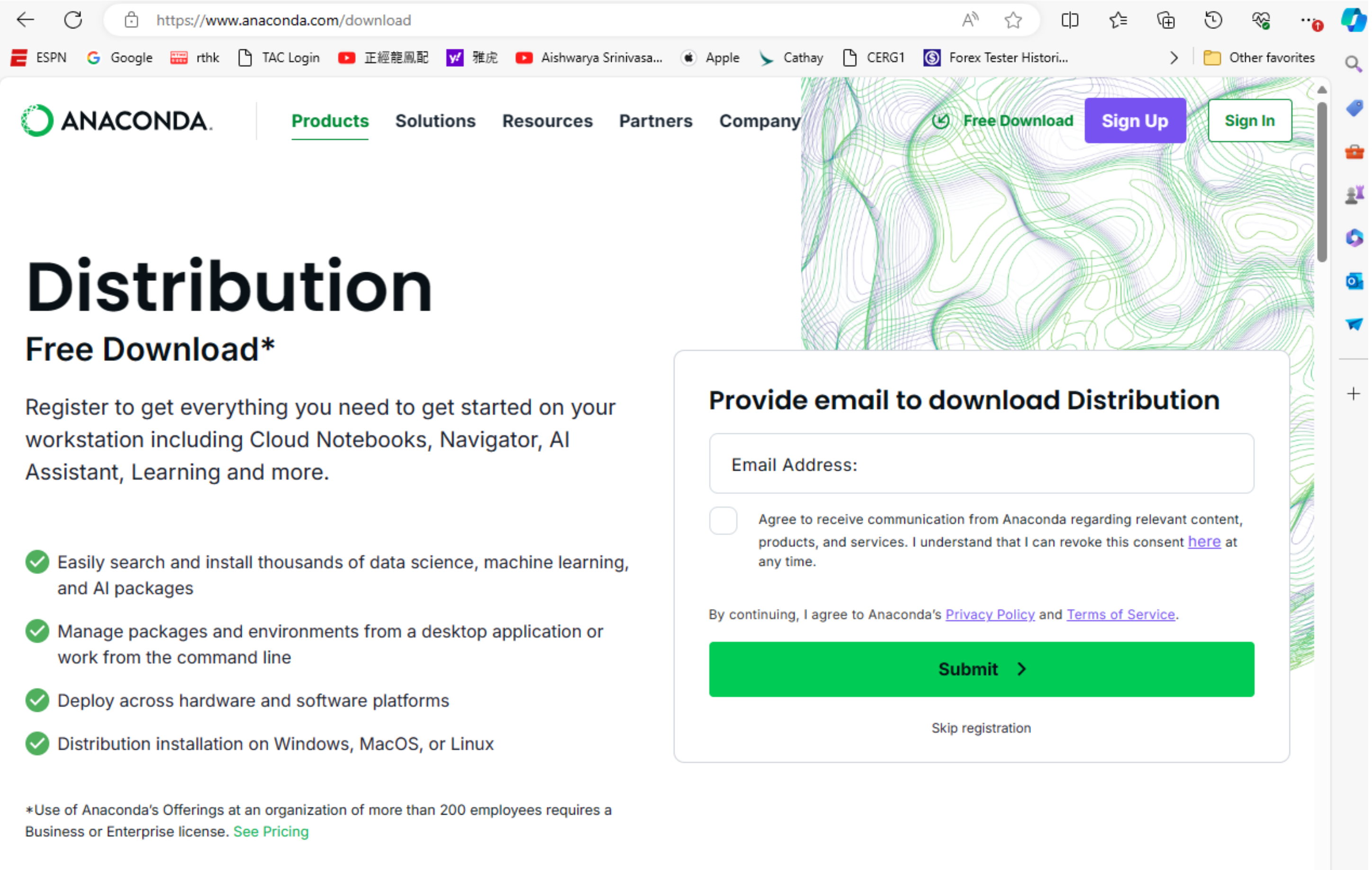
Anaconda Repository (Anaconda Repo)

- Anaconda repo is an open-source distribution of Python programs for scientific computing, data science, and machine learning.
- It is a cloud-based platform where users can find and share packages and programs.
- It includes thousands of packages for data science, machine learning, and scientific computing.
- It makes it easy for users to find and install the packages they need for their data analysis and modelling tasks.

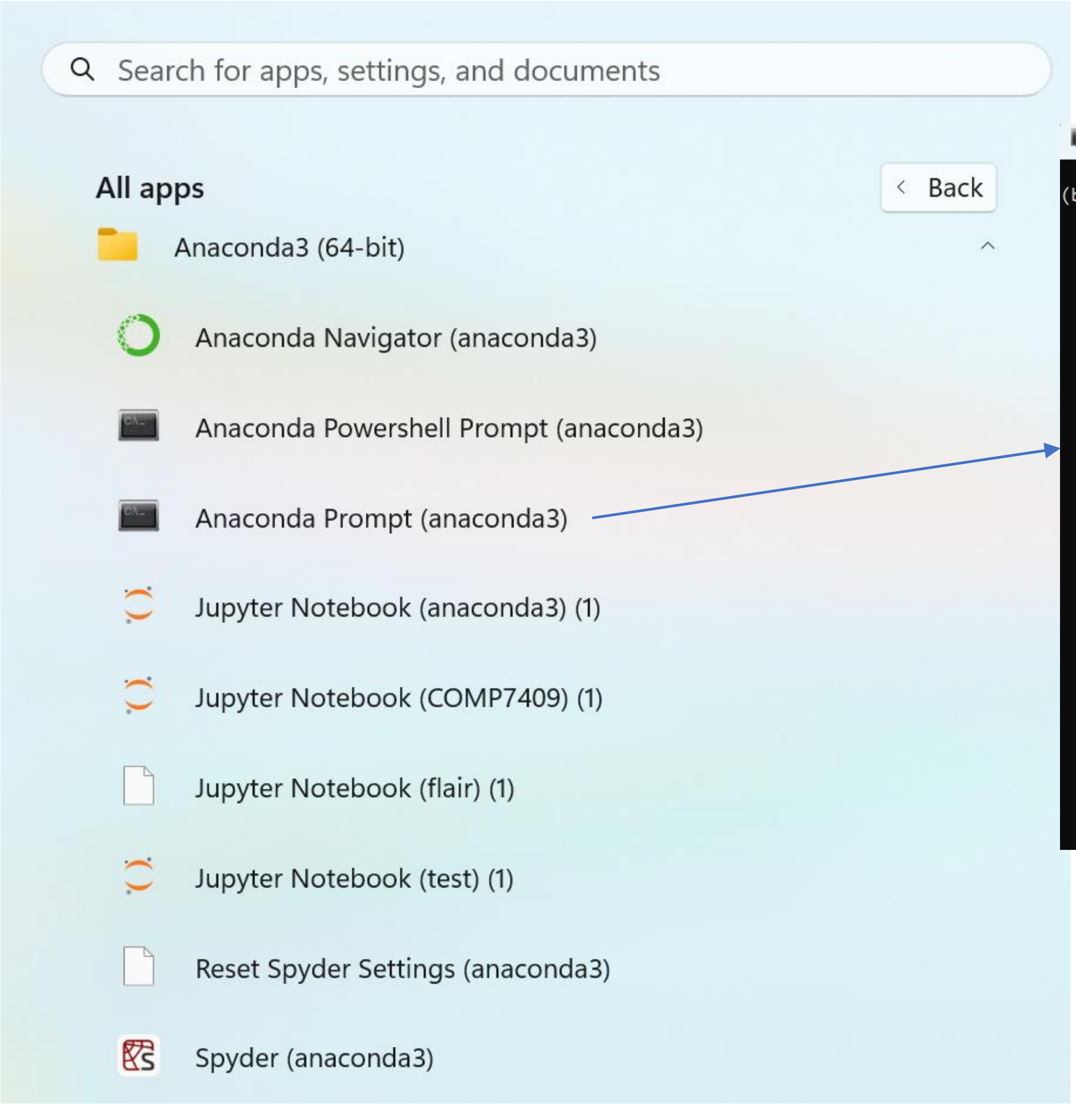
How to install Anaconda?

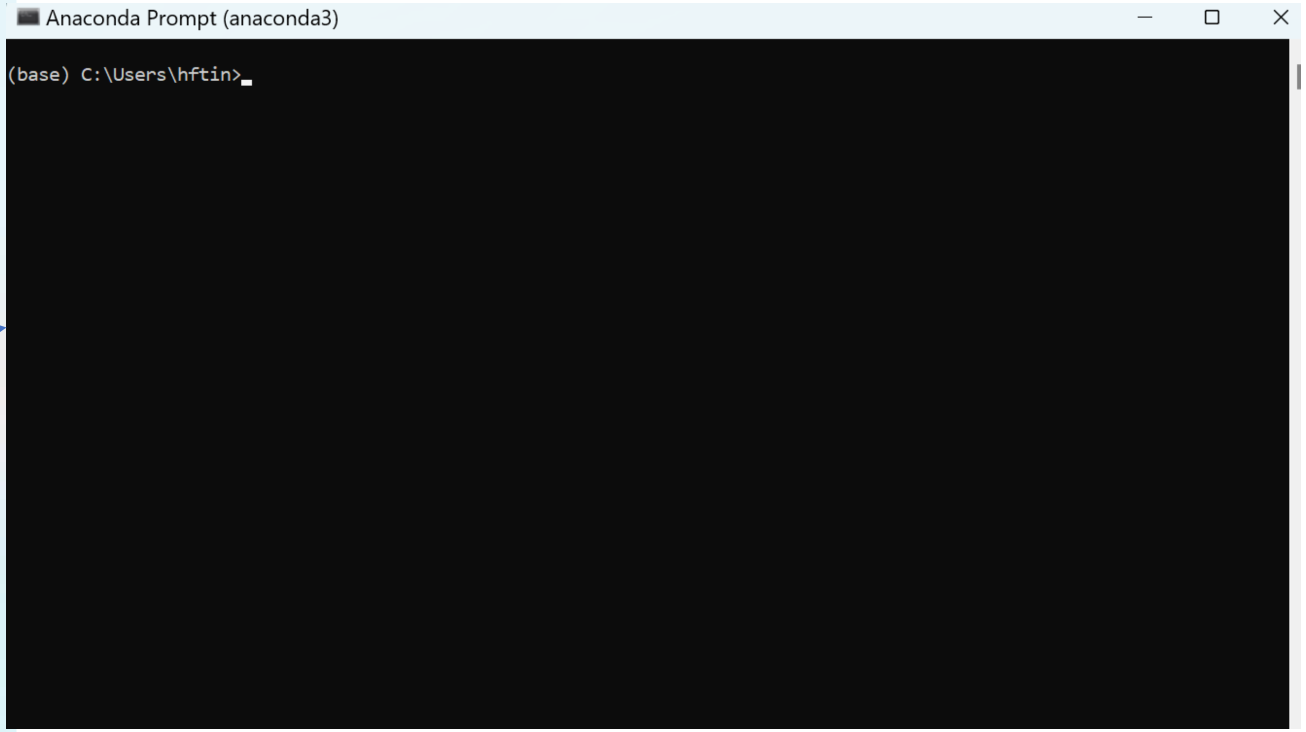
Simply download and install Anaconda Distribution via

https://www.anaconda.com/download

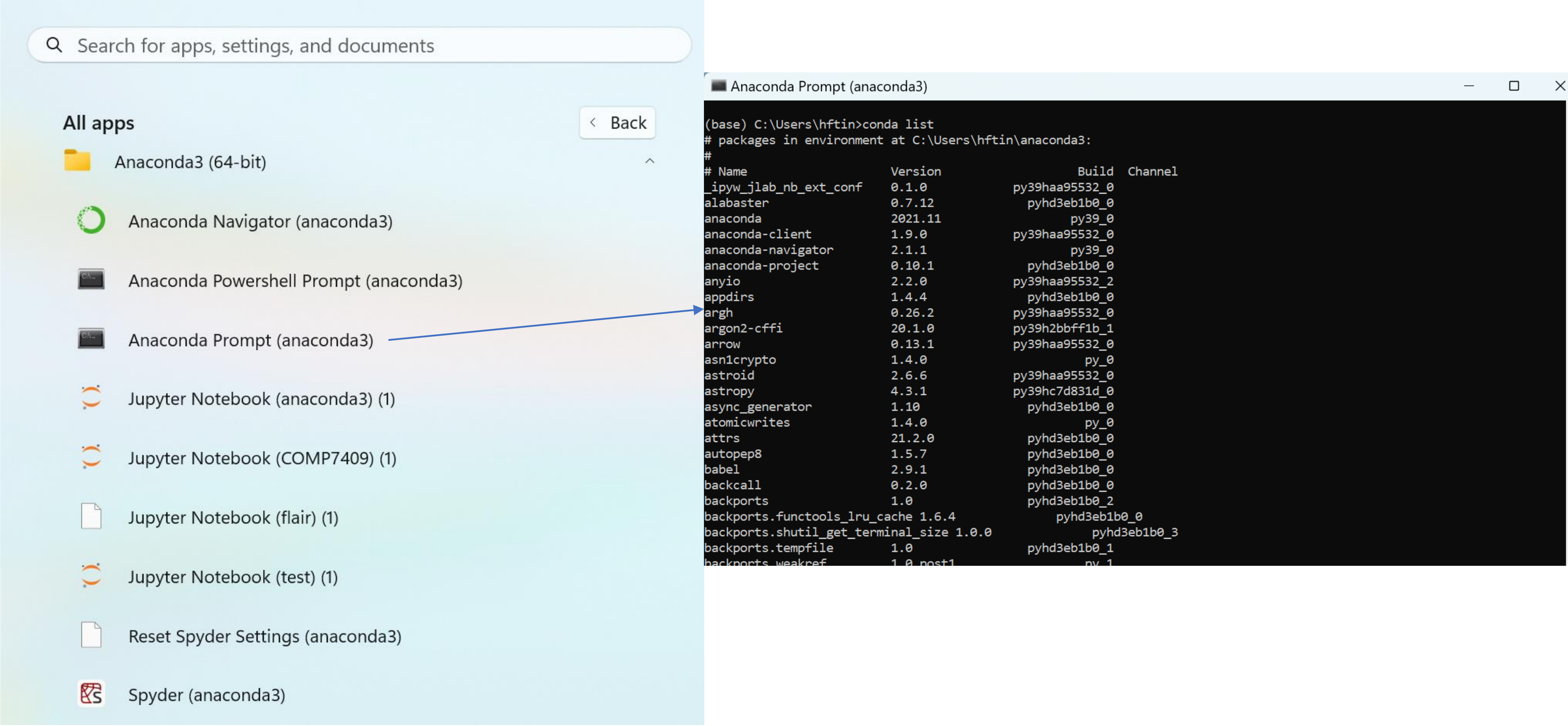


After installing Anaconda (on my Window 11 Home)

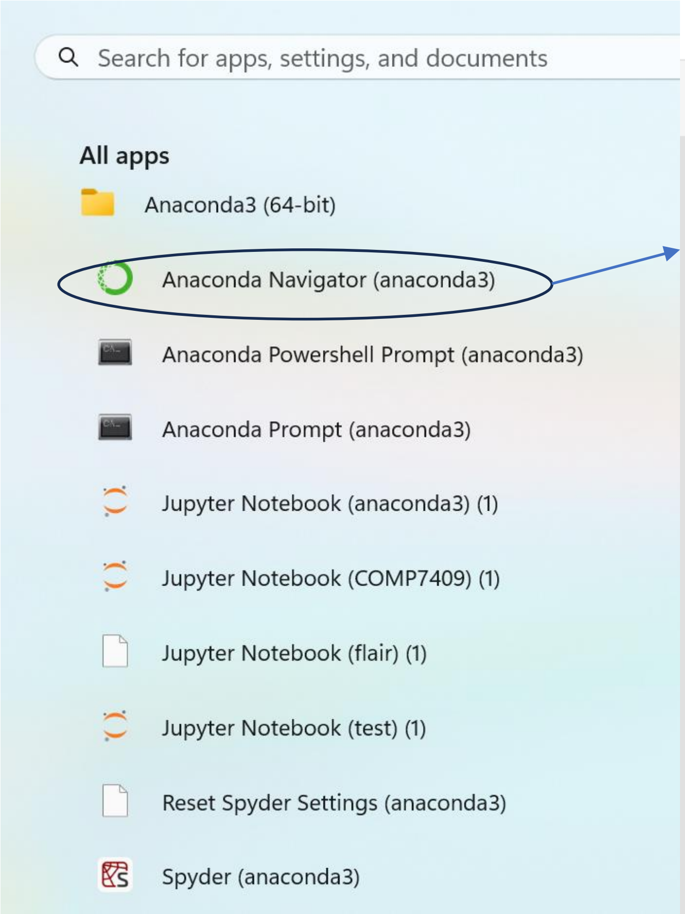


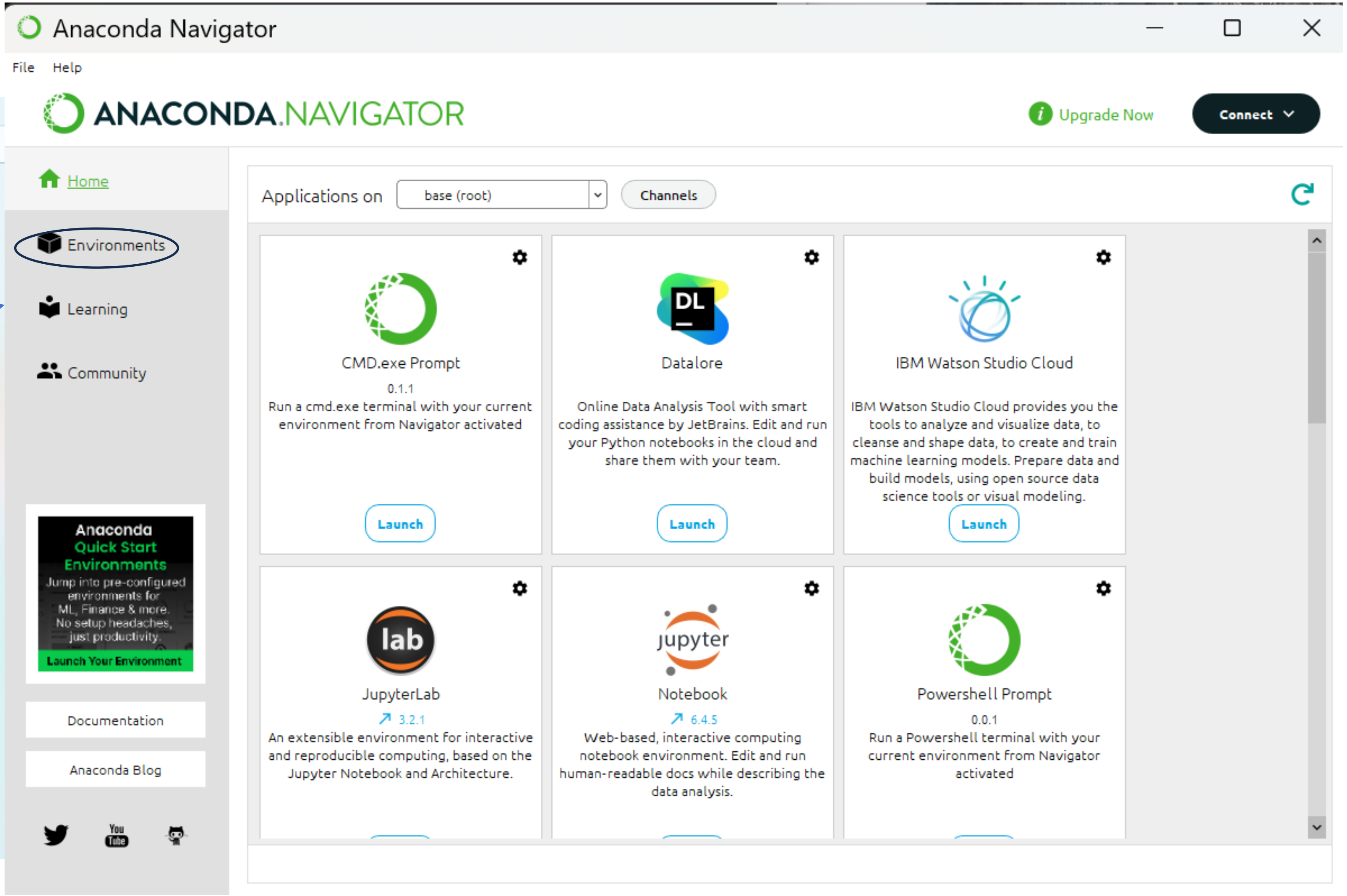


After installing Anaconda (on my Window 11 Home)

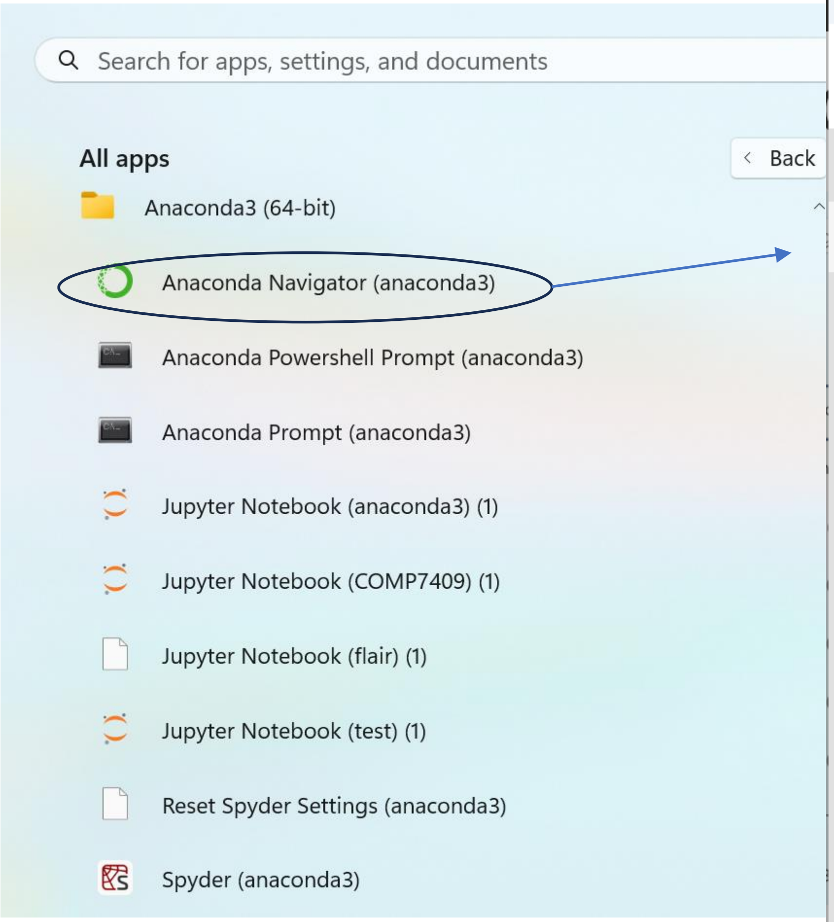


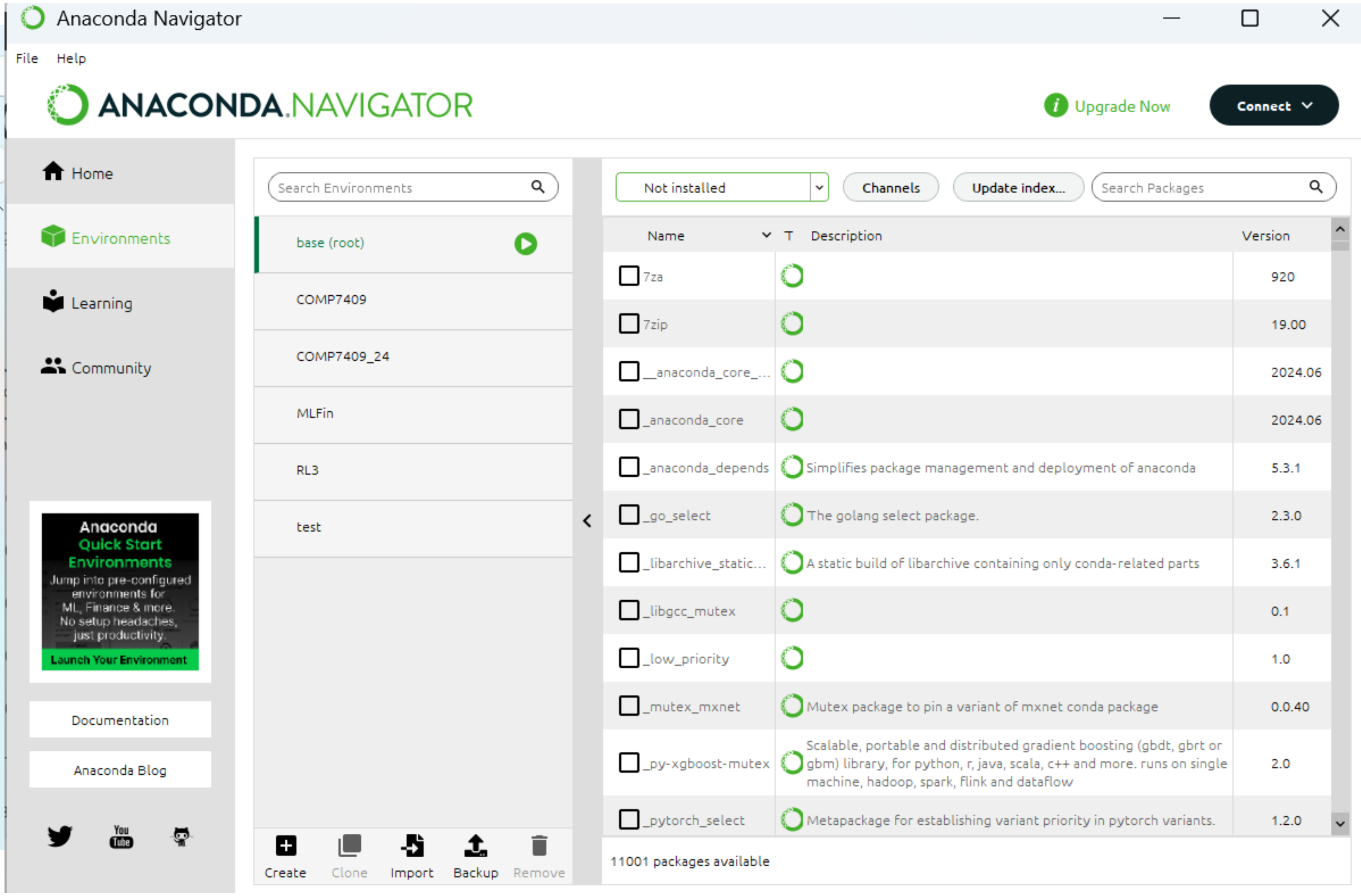
There are many other useful tools from Anaconda





There are many other useful tools from Anaconda





The Python Package Index (PyPI)

- In some of our examples we will use the pip command to install directly package from PyPI in our program. (You will know why in a minute)
- The Python Package Index is a respository of software packages maintain by the Python Software Foundation.
- pip is the package managers for PyPI (more info later).

IPython and Jupyter Notebook

- IPython, also known as Interactive Python, is a powerful tool that allows you to write Python program interactively; i.e., you can type a fragment of the program, then run it to see the results, make the necessary changes, and once satisfied, continue to input the following statements.

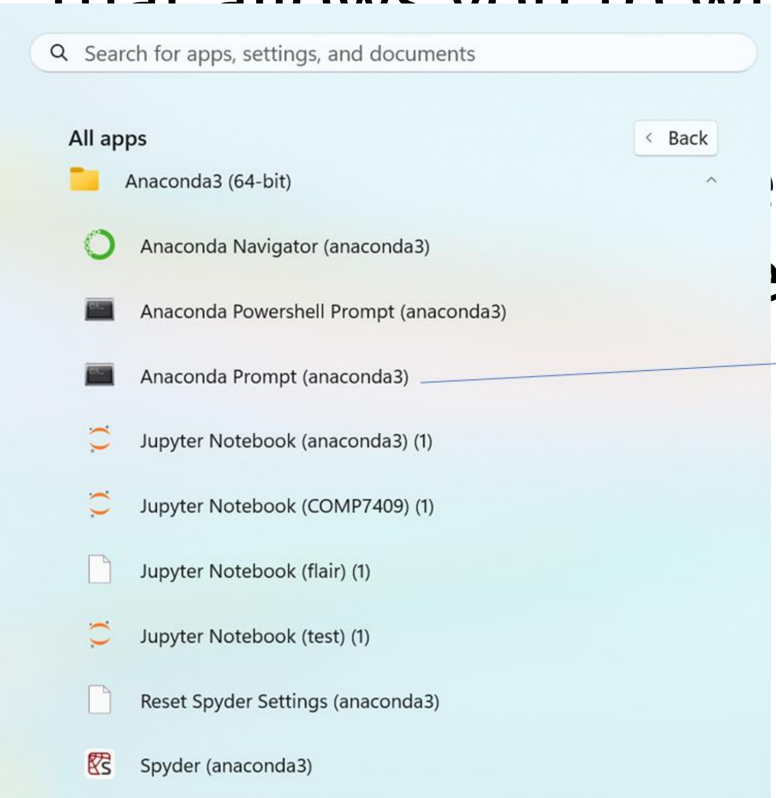
We will use iPython and Jupyter Notebook to write our Python programs.

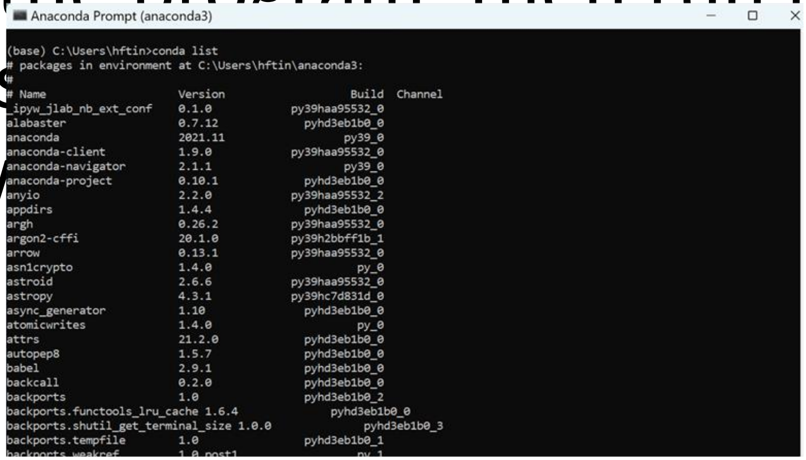
IPython and Jupyter Notebook

- IPython, also known as Interactive Python, is a powerful tool that allows you to write Python program interactively; i.e., you can type a fragment of the program, then run it to see the results, make necessary changes, and once satisfied, continue to input more statements.

IPython and Jupyter Notebook

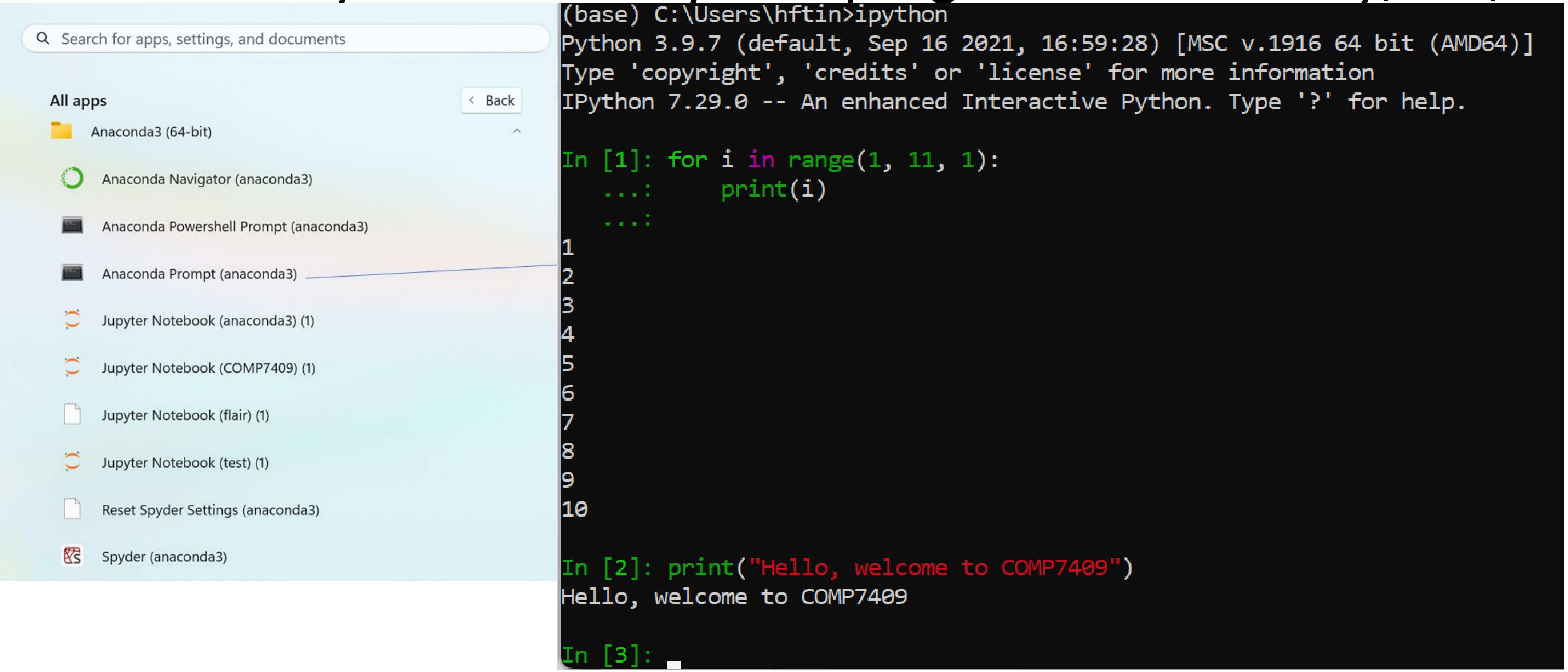
- IPython, also known as Interactive Python, is a powerful tool that allows you to write Python program interactively; i.e.,
ent of the program, then run it to see
necess satisfied,
follow





IPython and Jupyter Notebook

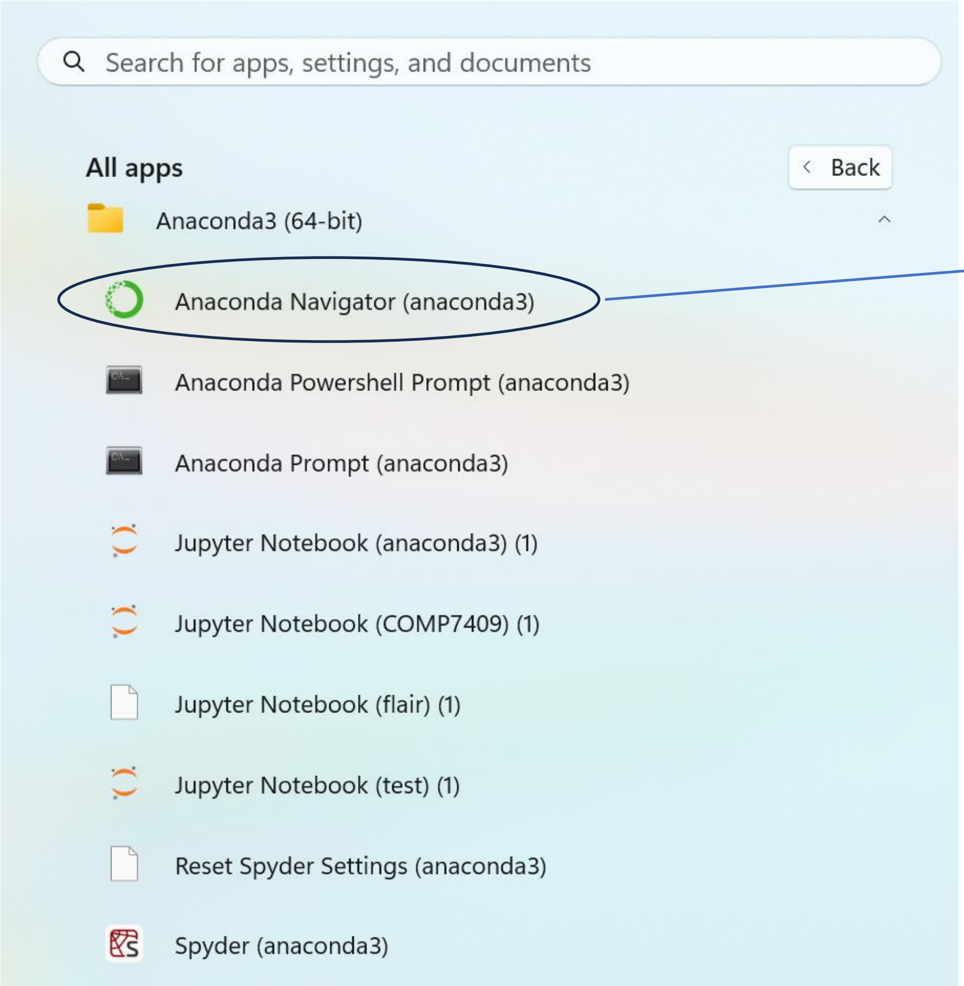
- IPython, also known as Interactive Python, is a powerful tool that allows you to write Python program interactively; i.e.,

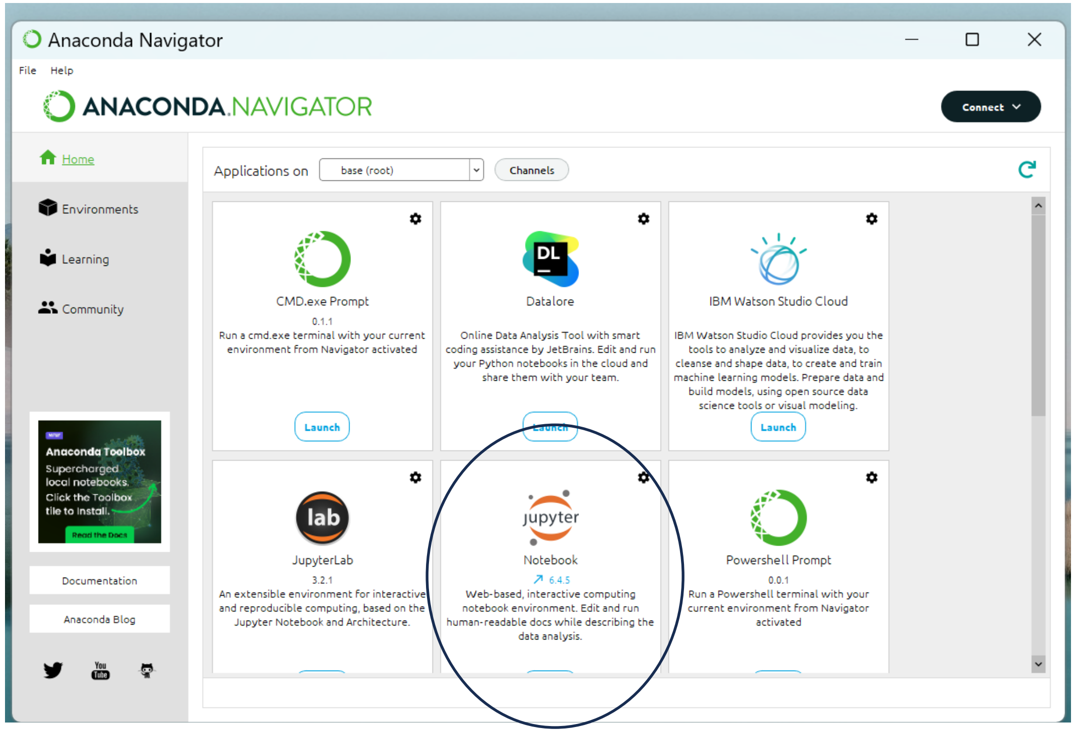


IPython and Jupyter Notebook

Jupyter Notebook is a browser-based tool that is the IDE (Integrated Development Environment) for IPython. Beside the standard output, it also support graphs, audio and video output. It contains an ordered list of input/output cells. You write a fragment of your program in an input cell, run it and the result will be displayed in the output cell.

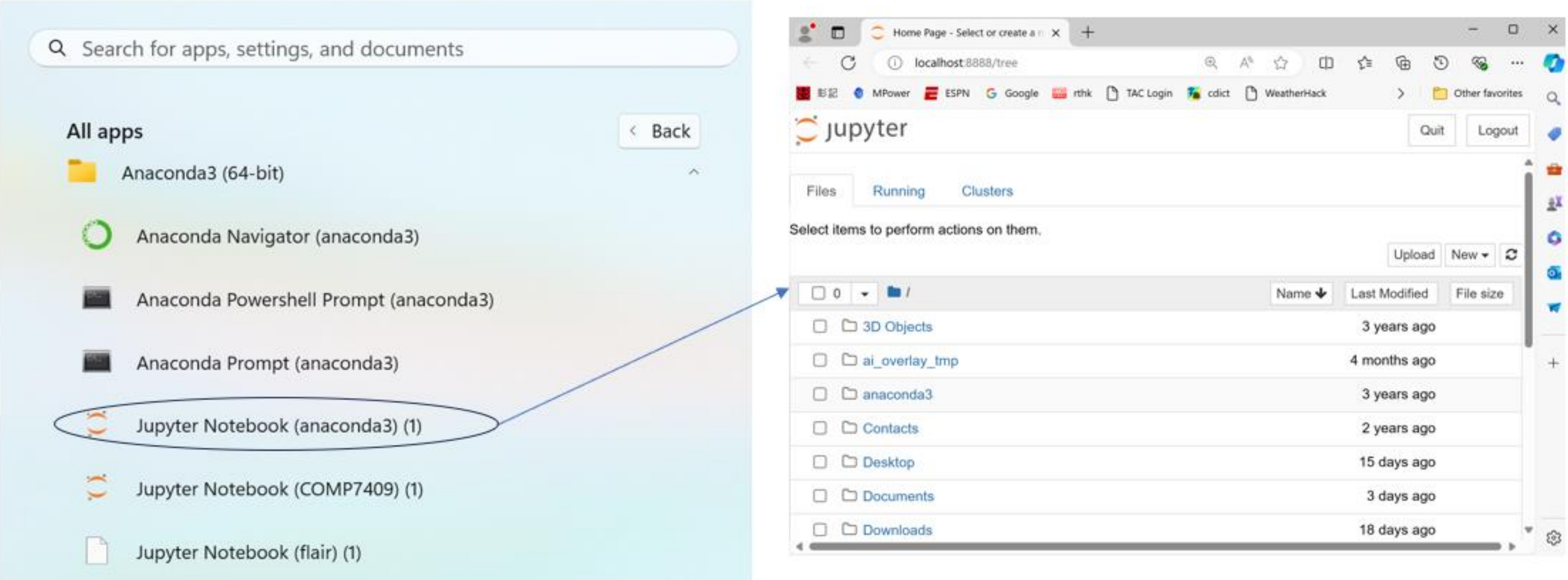
Another way for opening Jupyter Notebook



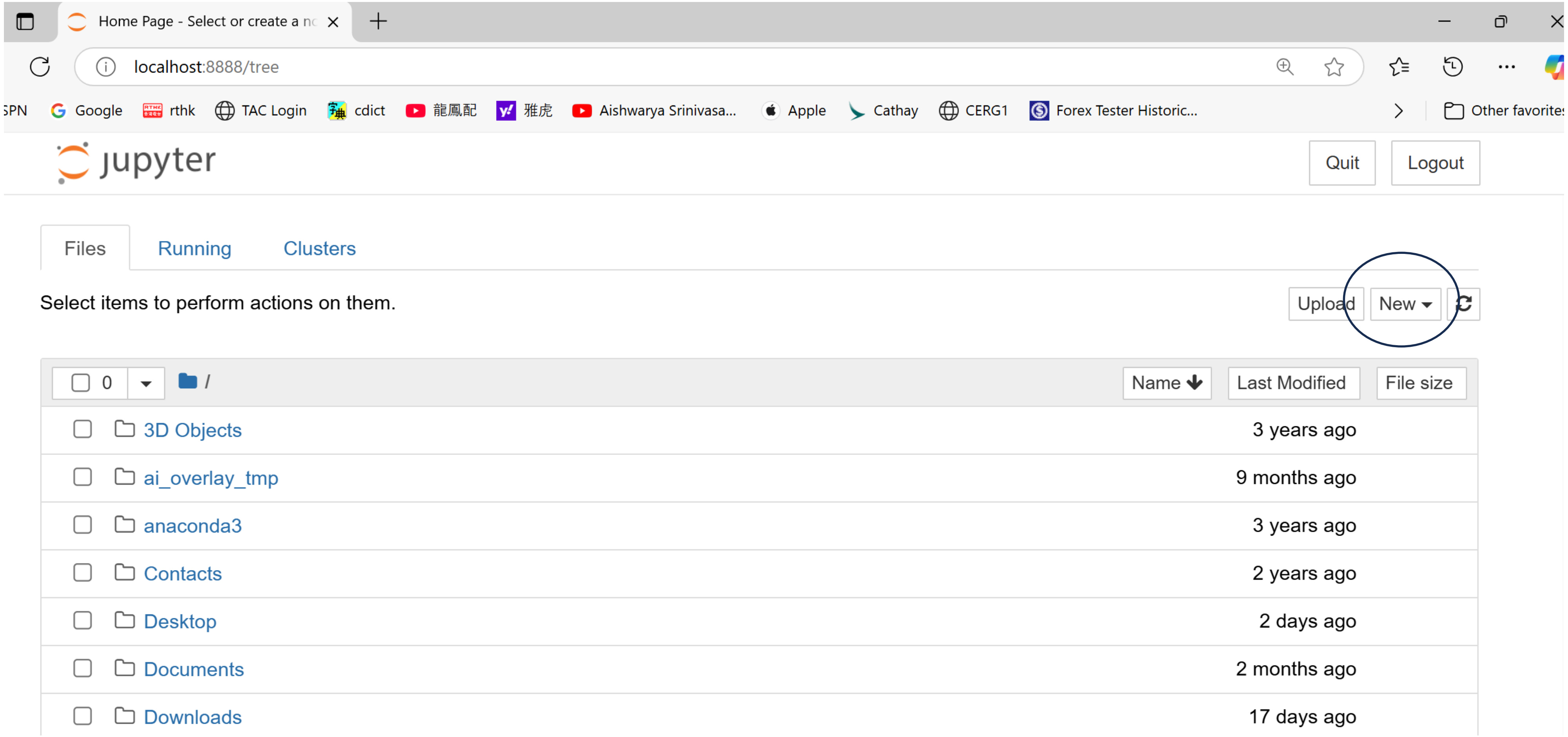


IPython and Jupyter Notebook

Another way to invoke Jupyter NB

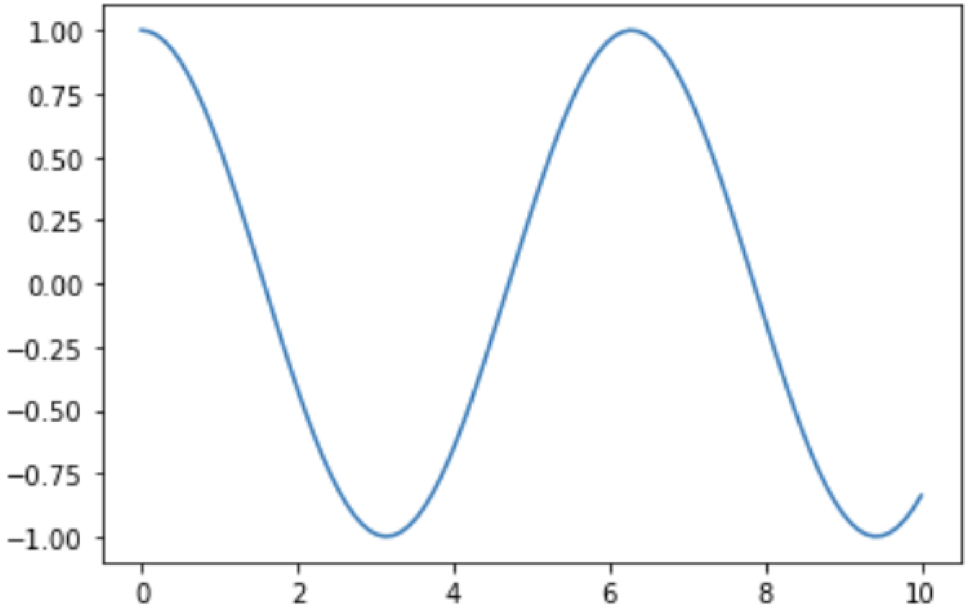


The browser-based interface of Jupyter NB



An example: JupyterOutput.ipynb

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

x = np.linspace(0, 10, 100)
y = np.cos(x)

plt.plot(x, y)
plt.show()
```

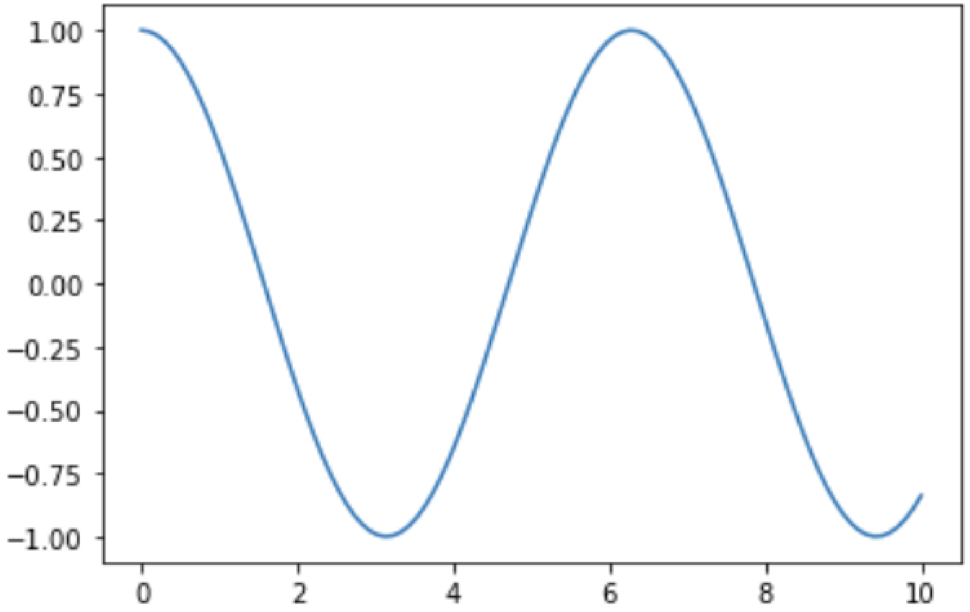


Note: A Jupyter notebook has extension .ipynb, not .py. You cannot run Jupyter notebook on a Python IDE, say IDLE. But Jupyter allows you to convert a .ipynb file to a .py file.

An example: JupyterOutput.ipynb

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

x = np.linspace(0, 10, 100)
y = np.cos(x)

plt.plot(x, y)
plt.show()
```

code cell



output cell

```python
import IPython.display as ipd
```

```python
ipd.Audio("https://www.soundjay.com/nature/rain-03.mp3")
```

0:00 / 0:30

```python
ipd.Video("big_buck_bunny_720p_5mb.mp4")
```

Colab: Running Jupyter NB on internet

- Jupyter notebook is a browser-based tool
- You open and execute it using any web browser.
- What we have done:

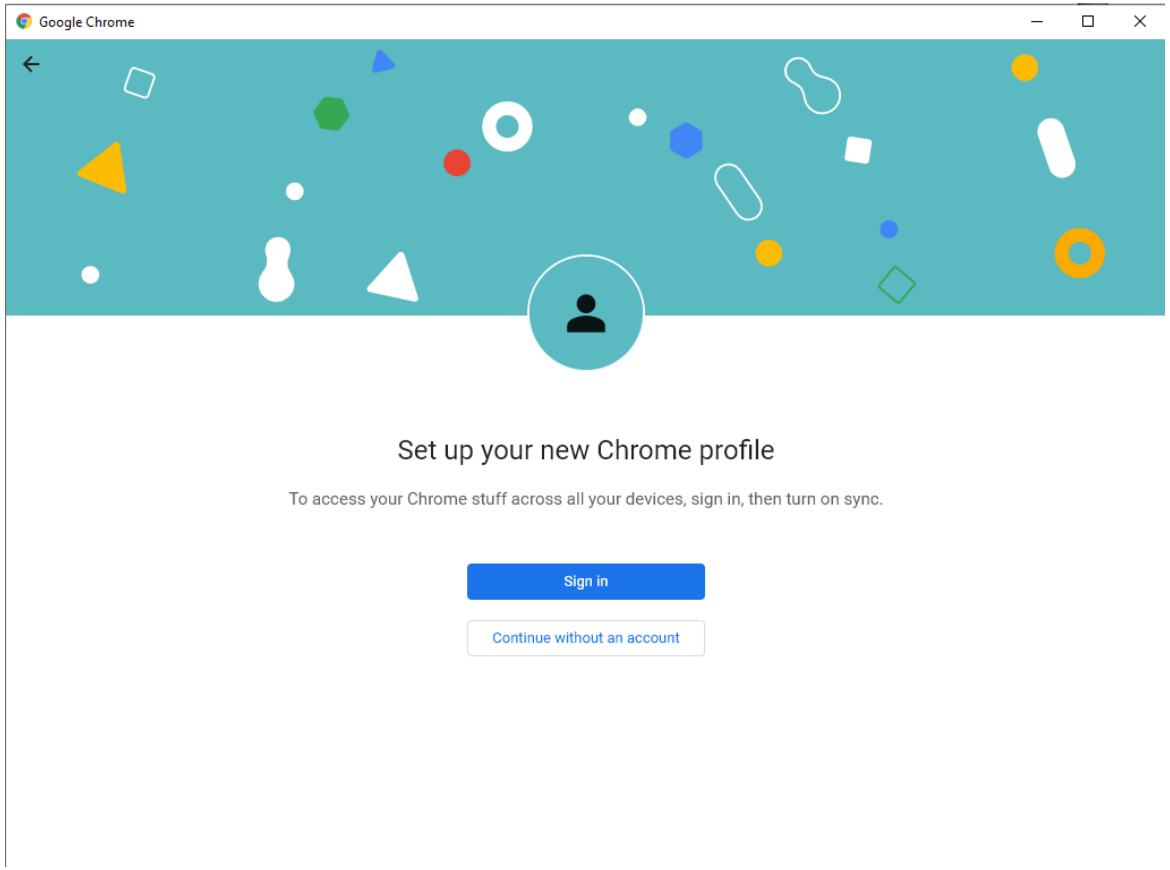
  We write our NB in our local computer: you open a web browser (e.g., Microsoft Edge) locally to run the notebook.
- But, we can also write and execute our NB on some remote servers on internet.

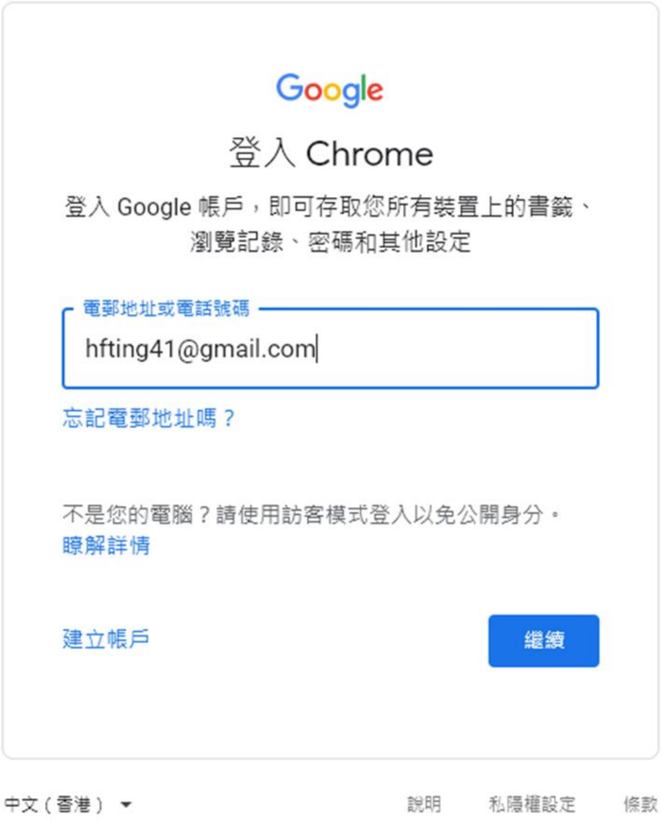
Colab: Running Jupyter NB on internet

- In the last few years, Google's Colaboratory (Colab) becomes one of the most popular cloud platform for data scientists to develop data science applications and perform data analytics.
- Colab supports a convenient environment that allows us to store our Jupyter notebooks on Google Drive and execute them using Google's cloud server.
- You don't need to worry about the installation of most packages as you run your programs in its cloud, though for some special ones, you still need to install it using pip, which is a package installer for installing packets in the repository Python Package Index (PyPI).

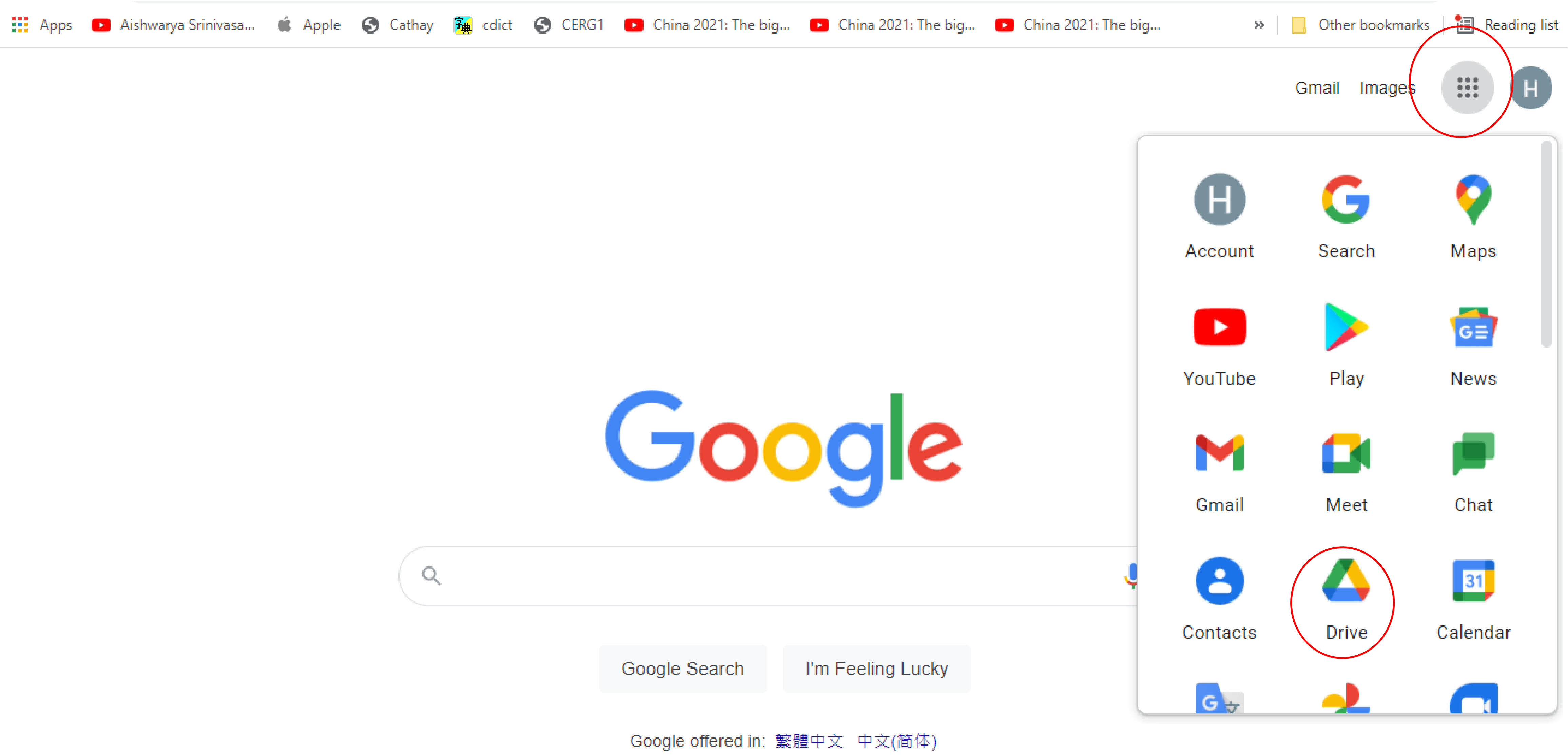
How to use Colab?

1. Sign in your Google account

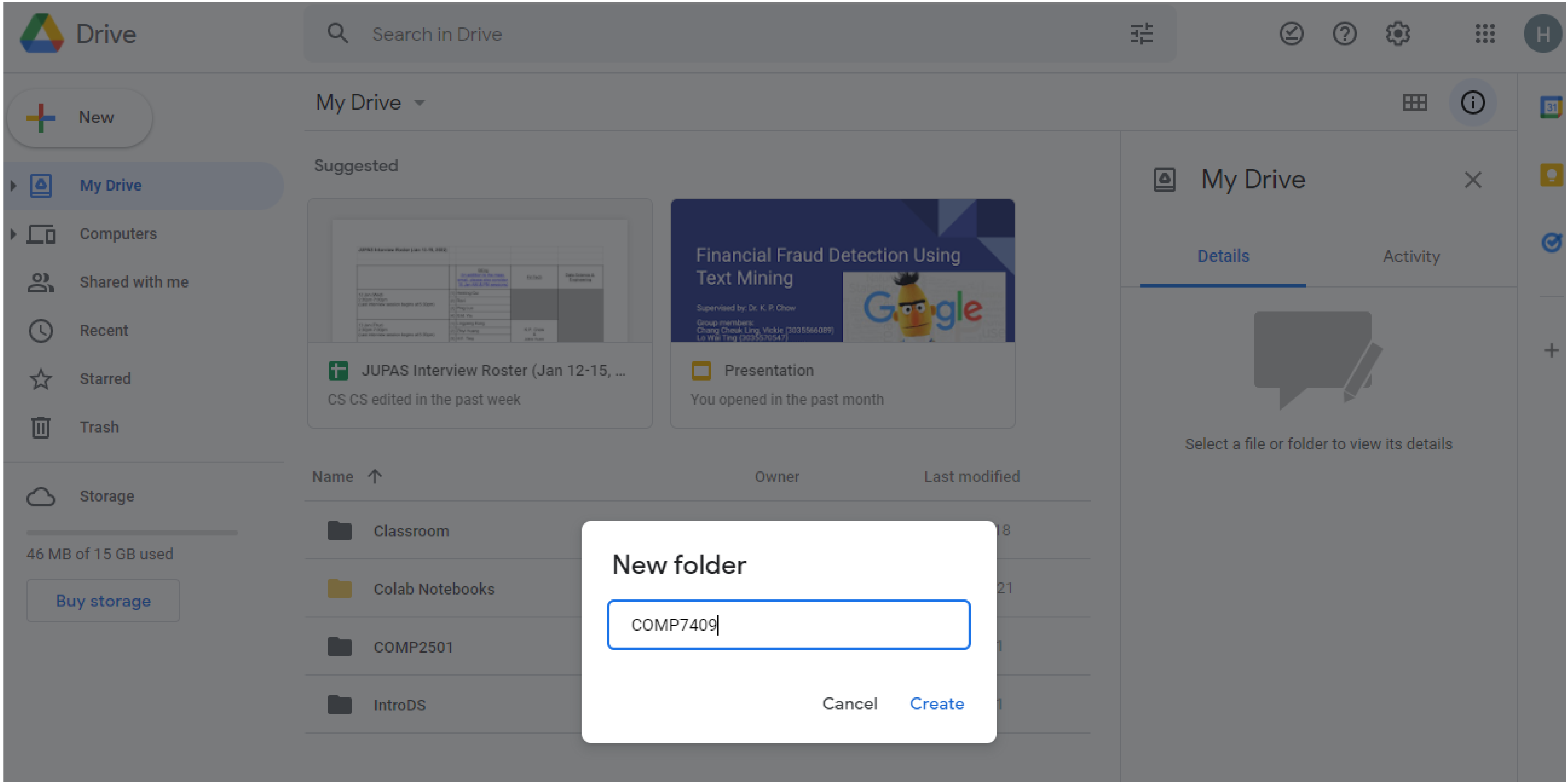




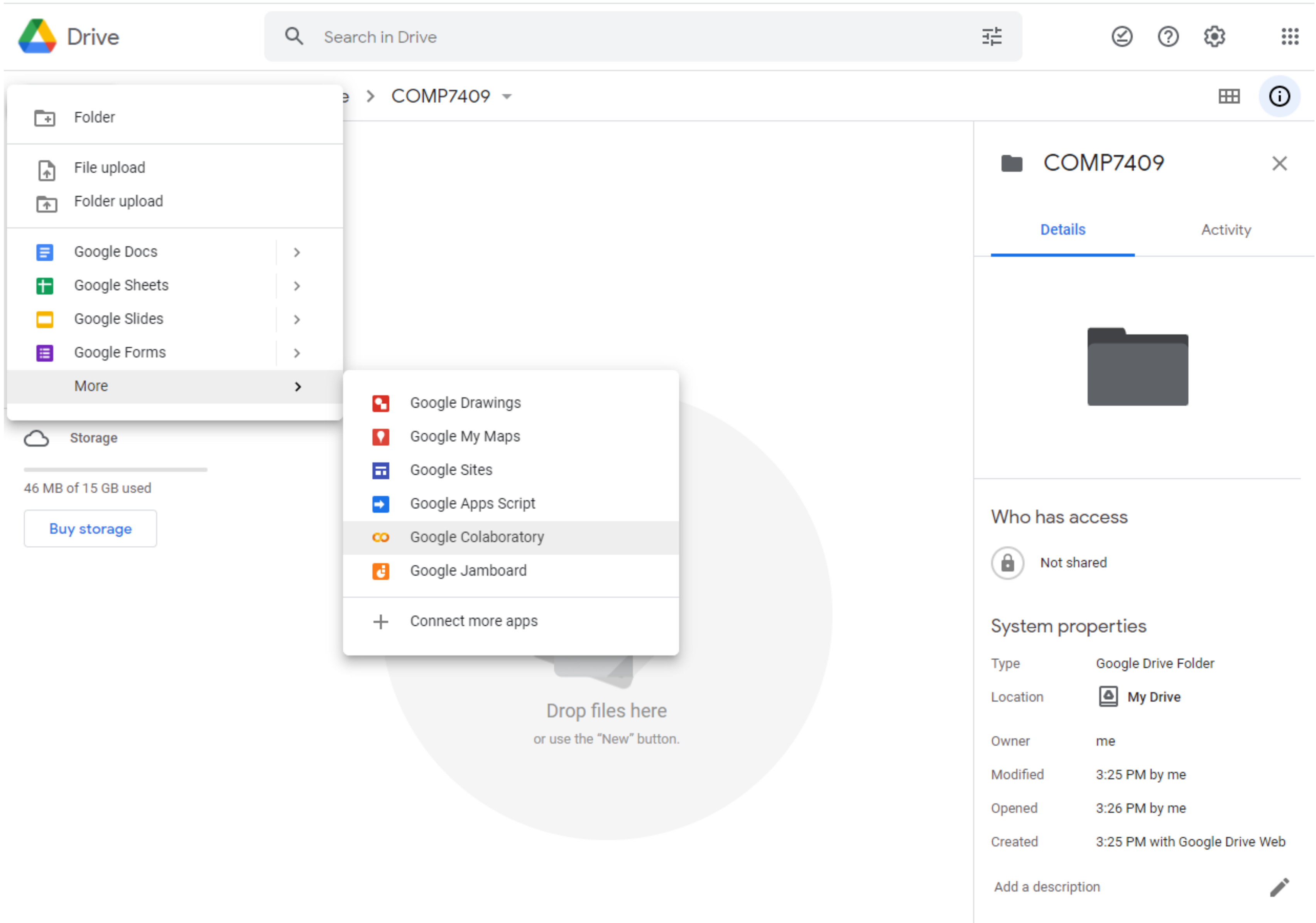
2. Open google drive



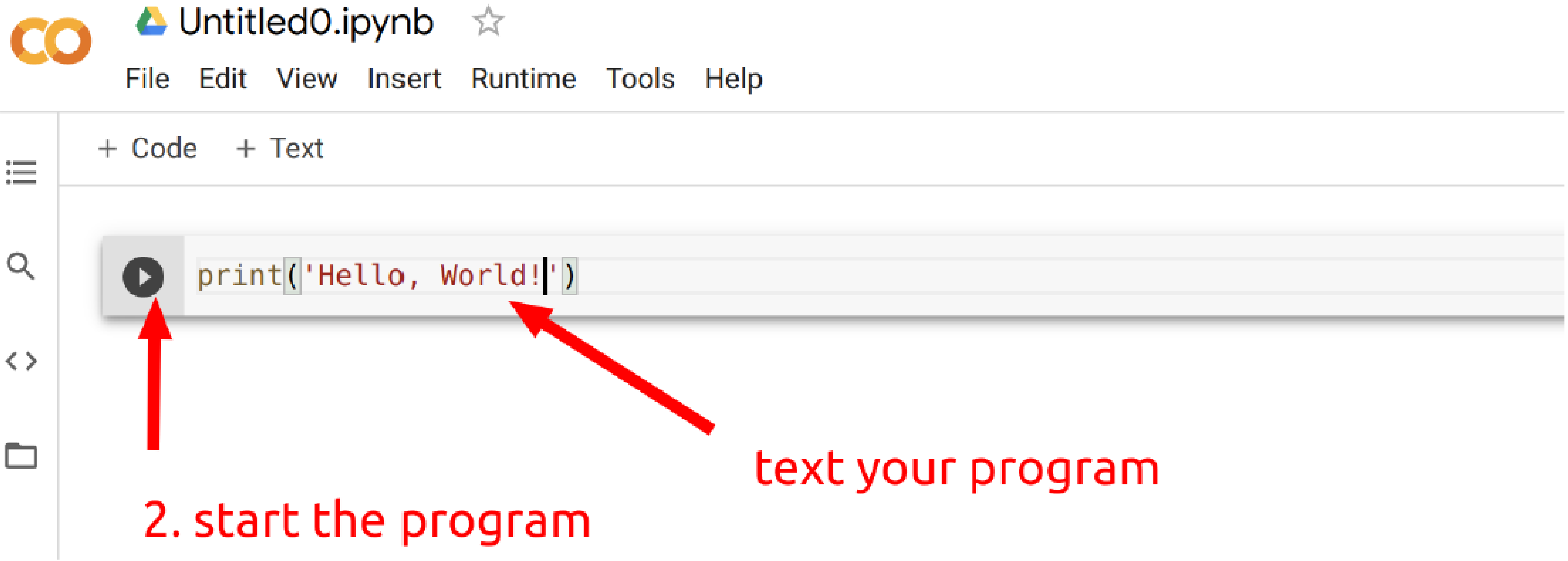
3. Create a folder



4. Move to that folder, and create a new “Colab” page



4. You can input your program now



How to install package when using colab

Like Anaconda, colab has pre-installed many popular libraries and packages like numpy, panda and matplotlib. For other packages, you can use the command pip to install those packages from Python Package Index.
For example,

```
!pip install numpy
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3
```

```
i = 0
while (i < 6):
  print("Hello")
  i = i + 1
```

```
Hello
Hello
Hello
```

What’s next?

- We will now cover some basic background:
  - Advanced Python
  - Numpy
  - Pandas
  - Machine Learning basics
- Important: You need to know how to program in Python. This course will not teach Python basics.

Advanced Python: Classes and Objects

- Python is an object-oriented programming language.
- The machine language packages available in popular libraries such as Scikit-learn and Tensorflow are all implemented using classes and objects.

Classes and Objects in Python

- Classes are the blueprint/recipe for object creation.
- Basically, classes are composed of two things:
  - Attributes: They are “variables” used to store date
  - Methods: They are functions defined within class.
- Objects are instances of classes.

An example: a simple class (simple_class.jpynb)

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

Name of the class

The __init__ function is compulsory. It is called every time a new object of this class is created.

An example: a simple class

```python
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```python
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

- The parameter **self** is very special. We use it to access the attributes and methods defined within the class . Note that it must be the first parameter in every method of the class.

An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

The other parameter a is used to pass value to the object to be created.

An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

This class has
- two attributes: self.a and self.b, and
- one method: f

An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

The statement is to create an instance of the class simple, and assign a reference of this instance to the variable q.
But, exactly what will happen when this statement is executed?

An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

Following are what will happen:

(1) Create an object of simple

objx

An example: a simple class

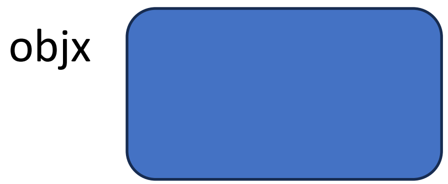
```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

Following are what will happen:

(1) Create an object of simple
(2) Execute __init__



An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

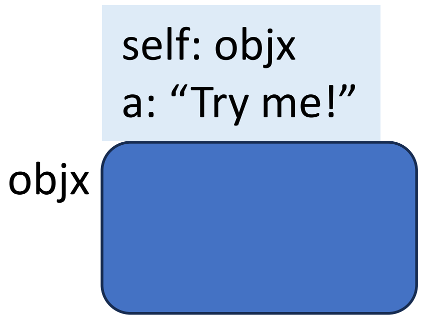
Following are what will happen:

(1) Create an object of simple

(2) Execute __init__:

  (i) arguments passing:

    self <= objx; a <= “Try me!”



An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

Following are what will happen:

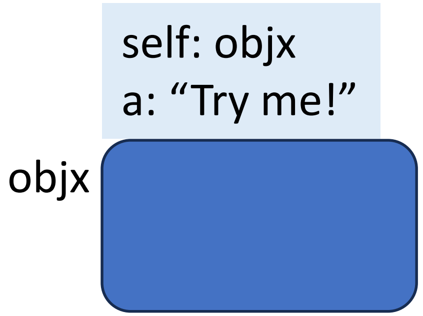
(1) Create an object of simple

(2) Execute __init__:

    (i) arguments passing:

        self <= objx; a <= “Try me!”

    (ii) execute stmt in the body:



An example: a simple class

```python
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```python
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

Following are what will happen:

(1) Create an object of simple

(2) Execute __init__:

  (i) arguments passing:

   self <= objx; a <= "Try me!"

  (ii) execute stmt in the body:

   self.a (== objx.a) 🡐 a (== "Try me!")

self: objx
a: "Try me!"

objx

objx.a = "Try me!"

An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

Following are what will happen:

(1) Create an object of simple
(2) Execute __init__:
    (i) arguments passing:
        self <= objx; a <= “Try me!”
    (ii) execute stmt in the body:
        self.a (== objx.a) 🡐 a (== “Try me!”)
        self.b (== objx.b) 🡐 13

self: objx
a: “Try me!”

objx

objx.a = “Try me!”
objx.b = 13

An example: a simple class

```python
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```python
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

Following are what will happen:

(1) Create an object of simple

(2) Execute __init__:

  (i) arguments passing:

    self <= objx; a <= "Try me!"

  (ii) execute stmt in the body:

    self.a (== objx.a) 🡐 a (== "Try me!")

    self.b (== objx.b) 🡐 13

  (iii) return self (== objx) and

    assign the returned value to q

self: objx
a: "Try me!"

objx

objx.a = "Try me!"
objx.b = 13

q

objx

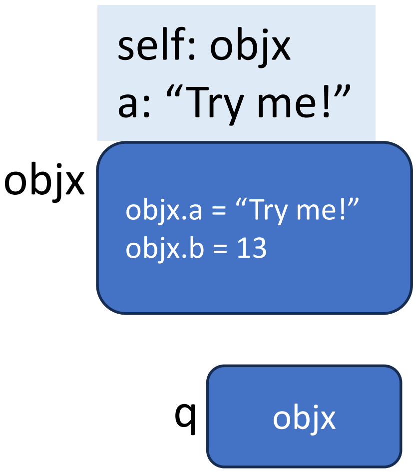
An example: a simple class

```python
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```python
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

print(q.a) == print(objx.a)
== print(“Try me!”)



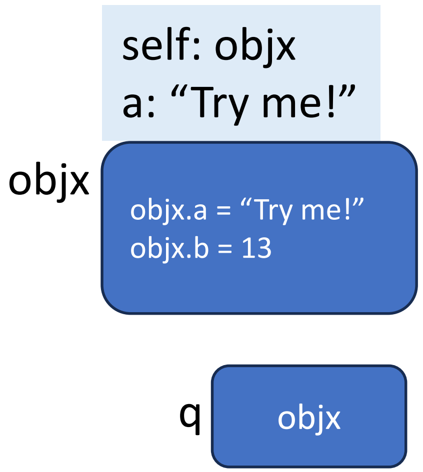
An example: a simple class

```python
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```python
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

q.f():
(i) argument passing:

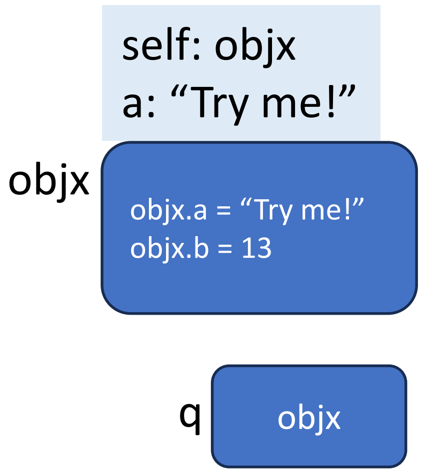


An example: a simple class

```python
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```python
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```



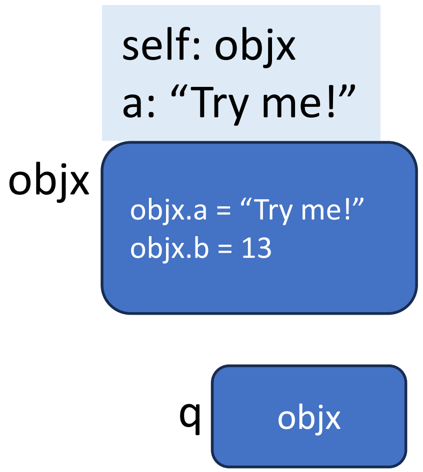
q.f():
(i) argument passing: ?? where is the first argument??

An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```



first argument

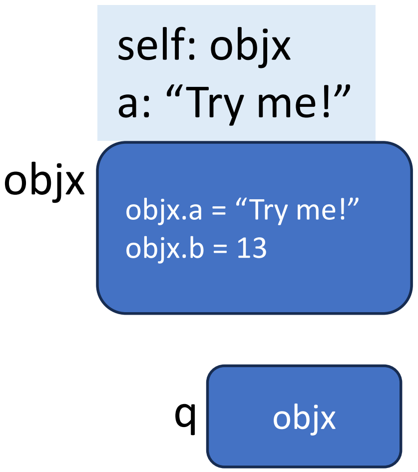
q.f():
(i) argument passing: ?? where is the first argument??

An example: a simple class

```python
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```python
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```



first argument

q.f():
(i) argument passing:
self ← q (== objx)

An example: a simple class

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

```
q = simple("Try me!")
print(q.a)
q.f()
```

```
Try me!
b =  13
```

self: objx
a: “Try me!”

objx

objx.a = “Try me!”
objx.b = 13

q

objx

first argument

q.f():
(i) argument passing:
self 🡐 q (== objx)
(ii) execute stmt in body:
print(“b = “, self.b (== objx.b ==13))

Using a class from another program

The file simple_class.py

```
class simple:
    def __init__(self, a):
        self.a = a
        self.b = 13

    def f(self):
        print("b = ", self.b)
```

The file module.jpynb

```
#The file is in the same folder that contains simple_class.py
```

```
import simple_class as sc
```

```
p = sc.simple("Good Morning")

print(p.a)
p.f()
```

```
Good Morning
b =  13
```

List comprehension

- A fast and elegant way to construct a list

Example – No List Comprehension	Example 1 – List Comprehension

`# Square all values`
`li = [2, 5, 3, 7]`
`r = []`
`for i in li:`
`  r.append(i ** 2)`
`print(r)`
	`# Square all values`
	`li = [2, 5, 3, 7]`
	`r = [i ** 2 for i in li]`
	`print(r)`

OUTPUT	OUTPUT
`[4, 25, 9, 49]`	`[4, 25, 9, 49]`

List comprehension

- A fast and elegant way to construct a list

Example – No List Comprehension

```
# Filter values above 3
li = [2, 5, 3, 7]
r = []
for i in li:
  if i > 3:
    r.append(i)
print(r)
```

OUTPUT

```
[5, 7]
```

Example 2 – List Comprehension

```
# Filter values above 3
li = [2, 5, 3, 7]
r = [i for i in li if i >
3]
print(r)
```

OUTPUT

```
[5, 7]
```

List comprehension

- A fast and elegant way to construct a list

```
coordinates = [(x, y) for x in range(3) for y in range(3)]

print(coordinates)
```

Output

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

List comprehension

- A fast and elegant way to construct a list

```
coordinates = [(x, y) for x in range(3) for y in range(3)]

print(coordinates)
```

$\cong$

```
coordinates = []
for x in range(3):
    for y in range(3):
        coordinates.append((x,y))
print(coordinates)
```

Dictionary comprehension

```
dc = dict()
for x in range(1,5):
    dc[x]=x**3
print(dc)
```

```
{1: 1, 2: 8, 3: 27, 4: 64}
```

$\cong$

EXAMPLE - Dict Comprehension

```
# Dict of cubes(values) of number(key) 1 to 4
dc = {x: x ** 3 for x in range(1, 5) }
print(dc)
```

OUTPUT

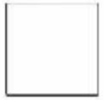
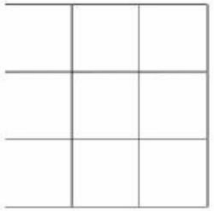
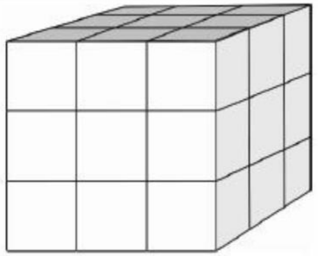
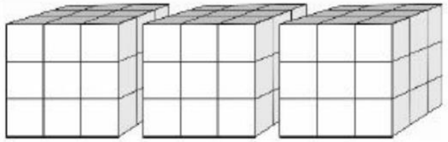
```
{1: 1, 2: 8, 3: 27, 4: 64}
```

Very brief introduction on NumPy

NumPy

Major data structures: Ndarray, i.e, Multidimensional array.

- A 1-D array is like a list of numbers in Python (but you will see later that the two are very different).
- A 2-D array corresponds to a matrix (table).
- A 3-D array corresponds to a cube

Illustration	Dimensions	Description
	0	Single value
	1	Multiple single values (List)
	2	Multiple List of values (Matrix)
	3	Multiple Matrices of values (Cube)
	4	Collection of Cubes
...	...	...

Very brief introduction on NumPy

NumPy

- Element-wise computation:

  NumPy provides a set of functions for performing element-wise computation with arrays and mathematical operations between arrays. (More details about element-wise computation will be given a few slides later.)

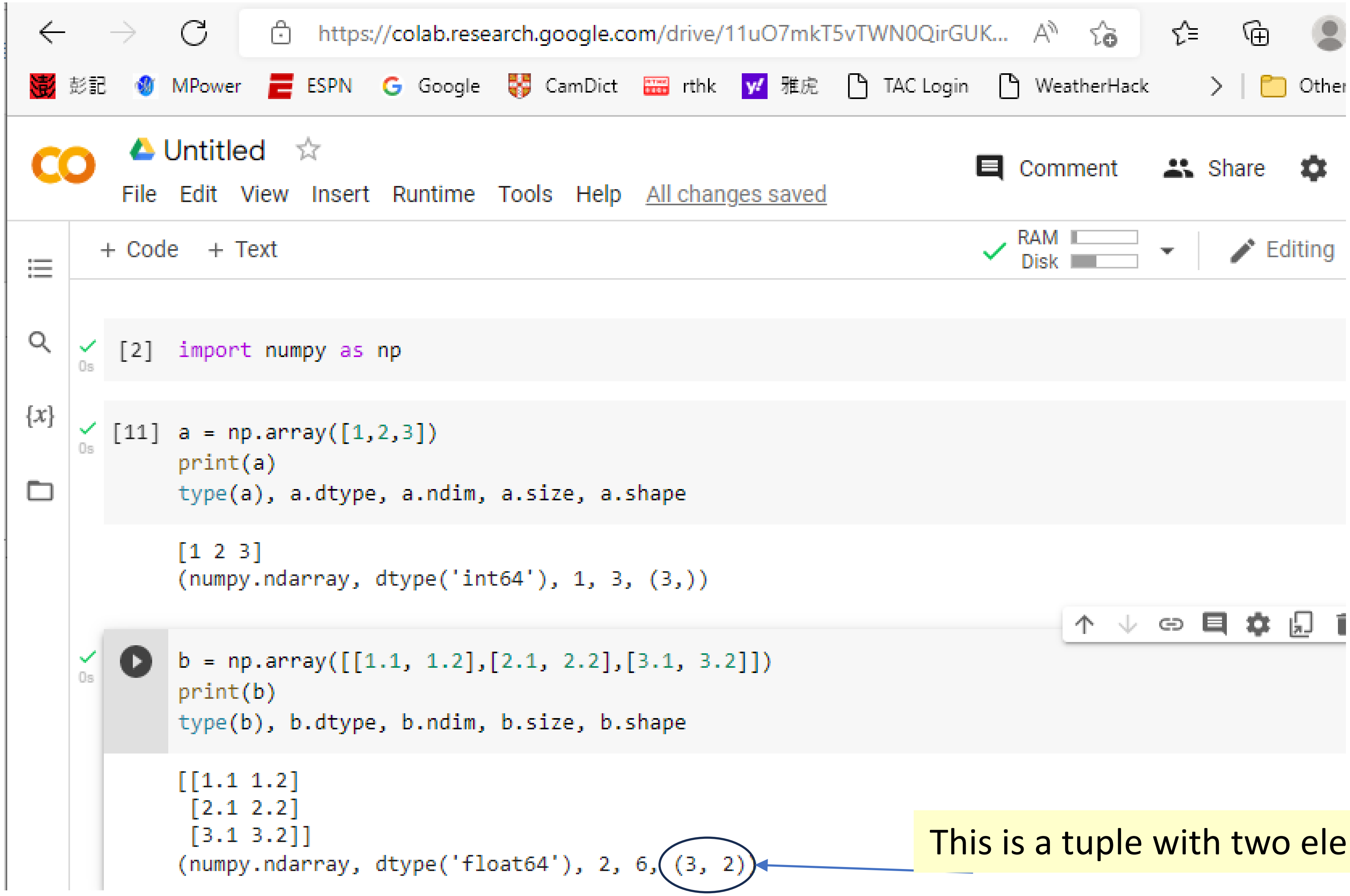
- Some algorithms can have a considerable performance increase after using numpy.

How to import numpy into your program?

```
import numpy as np
```

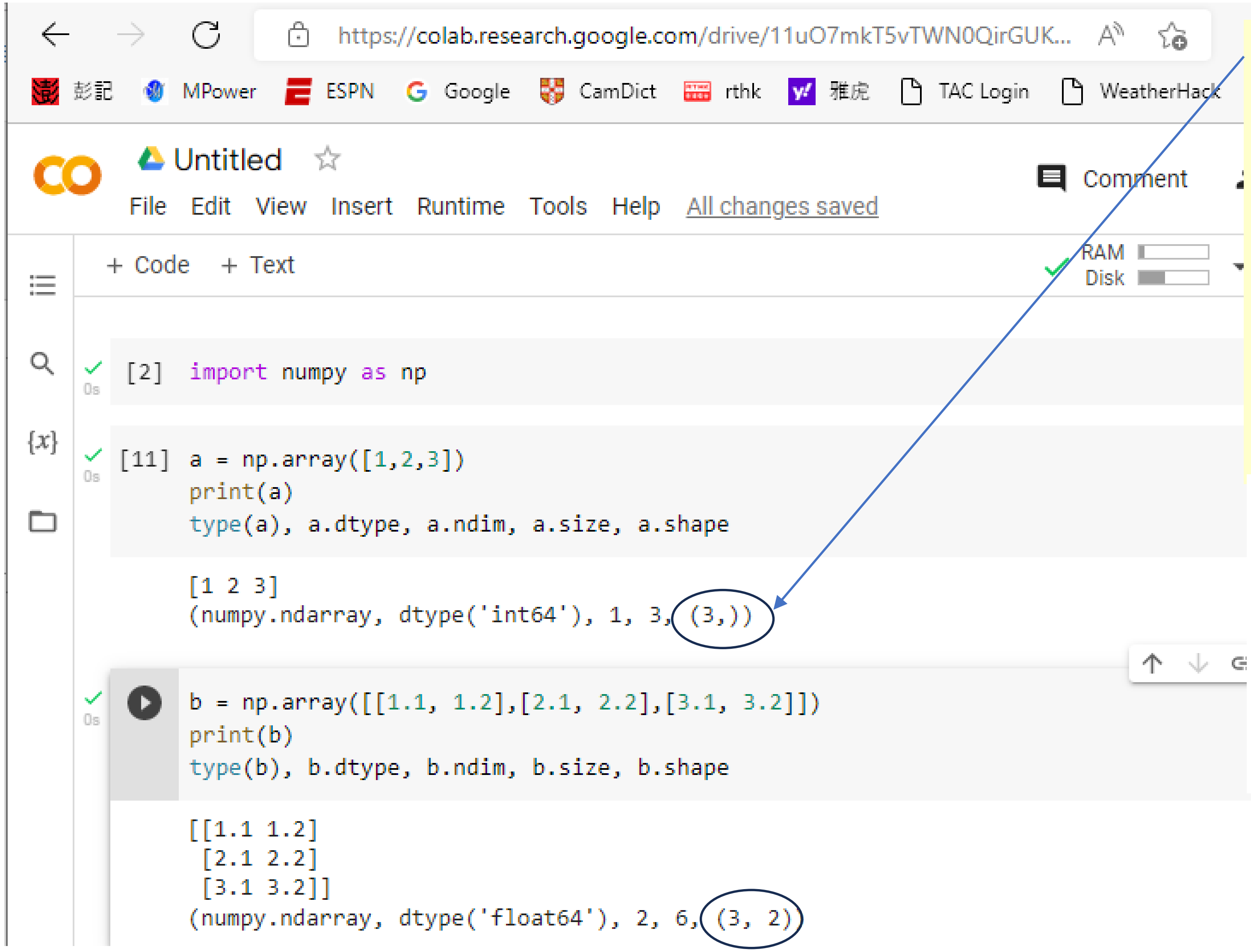
Not compulsory, but is very commonly used “name” for calling numpy

Constructing a numpy array from a Python list



This is a tuple with two elements.

Constructing a numpy array from a Python list



This is a tuple with one element.

Warning: (3,) and (3) are different:
(3,) is a tuple with one element
(3) is the integer 3

Example:
Remember the repetitive operator for list: [1, 2, 3] * 2 == [1, 2, 3, 1, 2, 3]

```
print([3] * 2)
print((3,) * 2)
print((3) * 2)
```

```
[3, 3]
(3, 3)
6
```

Simple ways to create filled ndarray

```
[18] np.zeros((2,3))

     array([[0., 0., 0.],
            [0., 0., 0.]])

[19] np.ones((4,2))

     array([[1., 1.],
            [1., 1.],
            [1., 1.],
            [1., 1.]])

[20] np.arange(0,10,2)

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

[21] np.arange(0,6,0.6)

     array([0. , 0.6, 1.2, 1.8, 2.4, 3. , 3.6, 4.2, 4.8, 5.4])

     np.linspace(0,10,5)

     array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

```
[24] np.random.random(3)

     array([0.77826755, 0.51211508, 0.48279984])

     np.random.random((4,2))

     array([[0.05478554, 0.46322134],
            [0.53737229, 0.11105537],
            [0.82457384, 0.96234194],
            [0.0962791 , 0.25828843]])
```

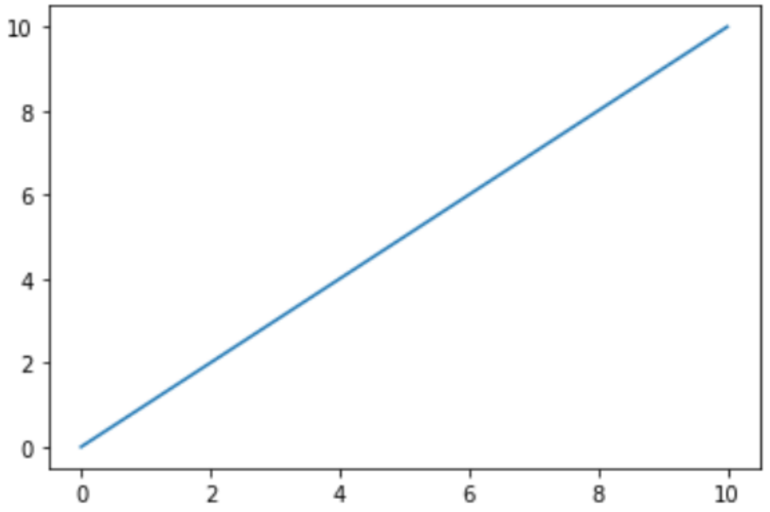
Simple ways to create filled ndarray

```
[18] np.zeros((2,3))

     array([[0., 0., 0.],
            [0., 0., 0.]])

[19] np.ones((4,2))

     array([[1., 1.],
            [1., 1.],
            [1., 1.],
            [1., 1.]])

[20] np.arange(0,10,2)

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

[21] np.arange(0,6,0.6)

     array([0. , 0.6, 1.2, 1.8, 2.4, 3. , 3.6, 4.2, 4.8, 5.4])

     np.linspace(0,10,5)

     array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

very useful for plotting graph

```
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 5)
y = x
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x2ced4a9a310>]
```



Reshaping an array

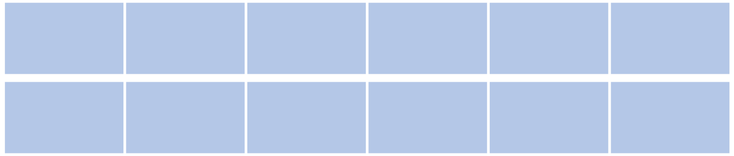
```
b
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
c = b.reshape(2,6)
c
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

Create a empty 2D array c with shape (2, 6)



and pick the entries from b from left to right, top to bottom, and put them in c one by one from left to right, top to bottom

Reshaping an array

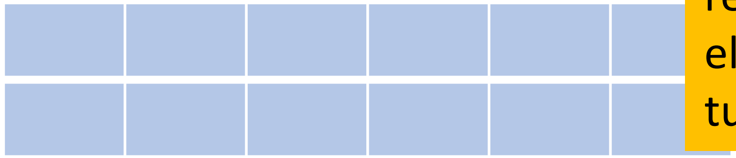
```
b
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
c = b.reshape(2,6)
c
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

Create a empty 2D array c with shape (2, 6)



and pick the entries from b from left to right, top to bottom, and put them in c one by one from left to right, top to bottom

Note: Recall that shape is a tuple, so the call reshape(2, 6) should be reshape((2, 6)). But in Python, we can eliminate the parathesis enclosing the tuple, i.e., (2, 6) is the same as 2, 6.

Reshaping an array

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

```
(array([1, 2, 3, 4, 5, 6]), (6,))
```

```
a.reshape(3,2)
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
a.reshape(3,-1)
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Reshaping an array

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

```
(array([1, 2, 3, 4, 5, 6]), (6,))
```

- If we set a dimension to -1, the actual value of this dimension is computed (inferred) from the size and the remaining dimensions.
- reshape() will not modify the array, instead, it returns another array with the new shape.

```
a.reshape(3,2)
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
a.reshape(3,-1)
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Element-wise operations

An ndarray is very different from the standard Python list:

```
[26] a = [1,2,3]
     b = [10, 20, 30]
     a+b

     [1, 2, 3, 10, 20, 30]
```

```
a = np.array([1,2,3])
b = np.array([10,20,30])
a+b

array([11, 22, 33])
```

Element-wise operations

Some simple examples:

```
>>> a = np.array([1, 2, 3], dtype=float)
>>> b = np.array([-1, 1, 3], dtype=float)
>>> a + b                                   # Addition
array([0., 3., 6.])
>>> a - b                                   # Subtraction
array([2., 1., 0.])
>>> a * b                                   # Multiplication
array([-1,  2,  9])
>>> a / b                                   # Division
array([-1.,  2.,  1.])
>>> a//b                                    # Integer division
array([-1.,  2.,  1.])
>>> a % b                                   # Modulus
array([-0.,  0.,  0.])
>>> a ** b                                  # Power
array([ 1.,  2., 27.])
```

Element-wise on functions

Consider the following example:

```
b = np.arange(12).reshape(2,6)
print(b)
print(b+2)
print(np.sqrt(b))
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
[[ 2  3  4  5  6  7]
 [ 8  9 10 11 12 13]]
[[0.         1.         1.41421356 1.73205081 2.         2.23606798]
 [2.44948974 2.64575131 2.82842712 3.         3.16227766 3.31662479]]
```

Element-wise on functions

Consider the following example:

```
b = np.arange(12).reshape(2,6)
print(b)
print(b+2)
print(np.sqrt(b))
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
[[ 2  3  4  5  6  7]
 [ 8  9 10 11 12 13]]
[[0.         1.         1.414213     98]
 [2.44948974 2.64575131 2.828427     79]]
```

??? the two operands not with the same shape???
Because of the "Broadcasting" capability of numpy.
Broadcasting is a great feature that allows great flexibility.
Shortly, broadcasting is the ability to perform an operation between arrays that do not have the exact same size or shape!

Broadcasting

```
a = np.array([1., 0., -1.,
2.])
b = a + 1
c = a + np.ones(4)
print(b)
print(c)
```

OUTPUTS

```
[2. 1. 0. 3.]
[2. 1. 0. 3.]
```

```
a = np.array([[1., 0.],
              [-1., 2.]])
b = np.array([3., 1.])
c = a + b
print(c)
```

OUTPUTS

```
[[4. 1.]
 [2. 3.]]
```

Broadcasting

```
a = np.array([1., 0., -1.,
2.])
b = a + 1
c = a + np.ones(4)
print(b)
print(c)
```

OUTPUTS

```
[2. 1. 0. 3.]
[2. 1. 0. 3.]
```

```
a = np.array([[1., 0.],
              [-1., 2.]])
([3., 1.])
```

```
[[4. 1.]
 [2. 3.]]
```

The general rules for numpy array broadcasting are rather complicated. Later, we will only broadcast a value, or an 1D array to some multi-dimensional array

Conditions, Boolean Arrays and Selection

```
[46] A = np.random.random((4,4))
     A

     array([[0.35660508, 0.96118156, 0.54066201, 0.60913444],
            [0.77205799, 0.92219434, 0.85651754, 0.76912115],
            [0.19362348, 0.09213051, 0.90208505, 0.669729  ],
            [0.78381514, 0.32448186, 0.94835817, 0.21323069]])
```

```
[48] cond = A < 0.5
     cond

     array([[ True, False, False, False],
            [False, False, False, False],
            [ True,  True, False, False],
            [False,  True, False,  True]])
```

```
[49] A[cond]

     array([0.35660508, 0.19362348, 0.09213051, 0.32448186, 0.21323069])
```

```
[50] A[ A < 0.5 ]

     array([0.35660508, 0.19362348, 0.09213051, 0.32448186, 0.21323069])
```

Indexing and slicing an ndarray

Just like a Python list, we can access an ndarray by indexing or by slicing.

```
A = np.arange(0, 18).reshape(3,6)
A
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17]])
```

```
A[0,0], A[1,1], A[2,2]
```

```
(0, 7, 14)
```

```
A[1, 1:4]
```

```
array([7, 8, 9])
```

Indexing and slicing an ndarray

Just like a Python list, we can access an ndarray by indexing or by slicing.

row with index 0, 1, 2
column with index
0, 1, 2, 3, 4, 5

```
A = np.arange(0, 18).reshape(3,6)
A
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17]])
```

```
A[0,0], A[1,1], A[2,2]
```

```
(0, 7, 14)
```

```
A[1, 1:4]
```

the stopping index is not included in the "output".

```
array([7, 8, 9])
```

```
A[1]
```

```
array([ 6,  7,  8,  9, 10, 11])
```

Indexing and slicing an ndarray

Another example:

```
[37] A = np.arange(0,18).reshape(3,6)
     A

     array([[ 0,  1,  2,  3,  4,  5],
            [ 6,  7,  8,  9, 10, 11],
            [12, 13, 14, 15, 16, 17]])
```

```
A[0:2,0:4]

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

Indexing and slicing an ndarray

- Positive and Negative indices
  - We can access any specific element in an ndarray by its positive index *p* and its negative index *n*

a[3] = a[-6] = 4.4

a =

1.1	**2.2**	**3.3**	**4.4**	**5.5**	**6.6**	**7.7**	**8.8**	**9.9**
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

- Relationship between n and p: p = n + len(a)

Indexing and slicing an ndarray

	1.1	**2.2**	**3.3**	**4.4**	**5.5**	**6.6**	**7.7**	**8.8**	**9.9**
a =	0	1	2	3	4	5	6	7	8
	-9	-8	-7	-6	-5	-4	-3	-2	-1

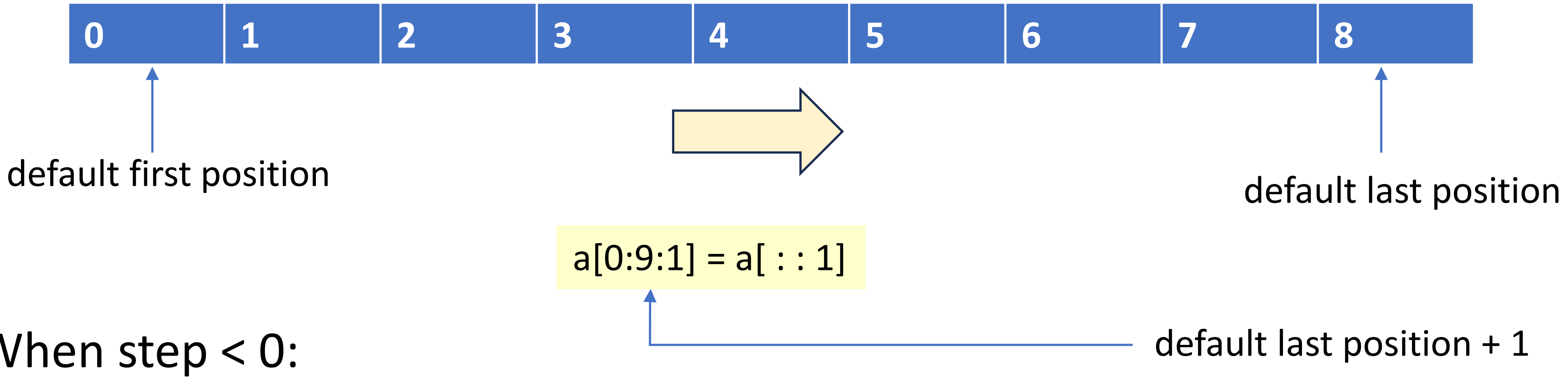
- A slice can be (i) a[start:stop:step] or (ii) a[start:stop].
  ((ii) is (i) with step having the default value 1.)
- As in Python list, the resulting slice is

  [start, start+step, …, start + k * step]

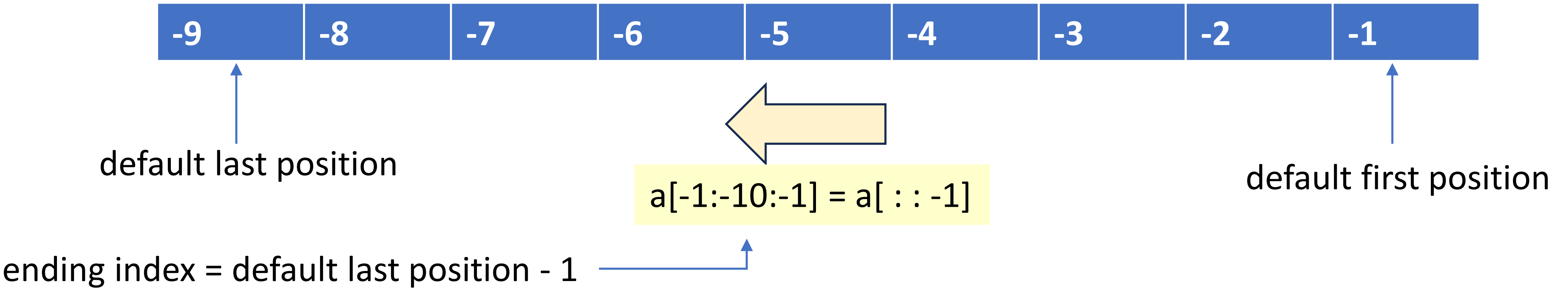
  where is the largest k such that (start + k * step) < stop
  (Note: the slice does not include the stopping index).

Indexing and slicing an ndarray

- Default values for the start and stop index.
  - When step > 0:



  - When step < 0:



Indexing and slicing an ndarray

a =

1.1	**2.2**	**3.3**	**4.4**	**5.5**	**6.6**	**7.7**	**8.8**	**9.9**
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

Basic steps for determining a slice:

- determine the value of step
- If step > 0
  - fill in, if missing, the default start and stop index.
  - convert all negative index to the equivalent positive index
- If step < 0
  - fill in, if missing, the default start and stop index.
  - convert all positive index to the equivalent negative index
- Determine the slice

Exercises:

a[::-1]

a[1::-1]

a[-3:-1]

a[-2:]

a[:-2]

The Ellipsis

```
x = np.array([[[1,10,100],[2,20,200],[3,30,300]],
              [[4,40,400],[5,50,500],[6,60,600]]])
x.shape, x
```

```
((2, 3, 3),
 array([[[  1,  10, 100],
         [  2,  20, 200],
         [  3,  30, 300]],

        [[  4,  40, 400],
         [  5,  50, 500],
         [  6,  60, 600]]]))
```

```
y = x[:,:, 0:2]
y.shape, y
```

```
((2, 3, 2),
 array([[[ 1, 10],
         [ 2, 20],
         [ 3, 30]],

        [[ 4, 40],
         [ 5, 50],
         [ 6, 60]]]))
```

The Ellipsis

```
x = np.array([[[1,10,100],[2,20,200],[3,30,300]],
              [[4,40,400],[5,50,500],[6,60,600]]])
x.shape, x
```

```
((2, 3, 3),
 array([[[  1,  10, 100],
         [  2,  20, 200],
         [  3,  30, 300]],

        [[  4,  40, 400],
         [  5,  50, 500],
         [  6,  60, 600]]]))
```

```
y = x[:,:, 0:2]
y.shape, y
```

```
((2, 3, 2),
 array([[[ 1, 10],
         [ 2, 20],
         [ 3, 30]],

        [[ 4, 40],
         [ 5, 50],
         [ 6, 60]]]))
```

```
y = x[..., 0:2]
y.shape, y
```

```
((2, 3, 2),
 array([[[ 1, 10],
         [ 2, 20],
         [ 3, 30]],

        [[ 4, 40],
         [ 5, 50],
         [ 6, 60]]]))
```

The ellipsis automatically expands the necessary number of ‘:’.

Very brief introduction of Pandas

- Pandas: A powerful Python Data Analysis Library
- The library was designed and developed primarily by Wes McKinney starting in 2008. In 2012, Sien Chang, one of his colleagues, was added to the development. Together they set up one of the most used libraries in the Python community.
- Pandas supports two major data structures:
  - Series: Extension of Numpy's 1D-array
  - Dataframes: A 2D table
-
```
import pandas as pd
```

Series

- Series is the Pandas 1-dimensional structure. It is a useful extension of Numpy array.
- Extension of the notion of index: index and index label
- index: position of the values: 0, 1, 2, …
- index label: give a name to the each index in a Series.
- If index labels are not provided in its creation, they follows the usual index sequence, i.e., 0, 1, 2, …

Creating Series: using the pd.Series method

Create a Series from a Python list

```python
s = pd.Series([12, -4, 7, 9], index=['a','b','c','d'])
s
```

```
a    12
b    -4
c     7
d     9
dtype: int64
```

```python
s = pd.Series([12, -4, 7, 9])
s
```

```
0    12
1    -4
2     7
3     9
dtype: int64
```

index label

Creating Series: using the pd.Series method

- Create a series from a Python dictionary

```
tdict = {'Tom':12, 'Peter':-4, 'Mary':7, 'Zoe':9}
type(tdict), tdict
```

```
(dict, {'Mary': 7, 'Peter': -4, 'Tom': 12, 'Zoe': 9})
```

```
t = pd.Series(tdict)
t
```

```
Tom      12
Peter    -4
Mary      7
Zoe       9
dtype: int64
```

Creating Series: using the pd.Series method

- Create a series from a Python dictionary

```
tdict = {'Tom':12, 'Peter':-4, 'Mary':7, 'Zoe':9}
type(tdict), tdict
```

```
(dict, {'Mary': 7, 'Peter': -4, 'Tom': 12, 'Zoe': 9})
```

```
t = pd.Series(tdict)
t
```

```
Tom      12
Peter    -4
Mary      7
Zoe       9
dtype: int64
```

caution: dict does not have order
➔you cannot be sure the order of the series constructed

Obtain the values only (i.e., without row labels)

```
[11] tdict = {'Tom':12, 'Peter':-4, 'Mary':7, 'Zoe':9}
     type(tdict), tdict

     (dict, {'Mary': 7, 'Peter': -4, 'Tom': 12, 'Zoe': 9})
```

```
[12] t = pd.Series(tdict)
     t

     Tom      12
     Peter    -4
     Mary      7
     Zoe       9
     dtype: int64
```

```
t.values

array([12, -4,  7,  9])
```

Access the elements in a Series

We have two ways to access the elements of a Series:
(1) using the index as if it is a numpy array; or
(2) using the series's index labels

```
[11] tdict = {'Tom':12, 'Peter':-4, 'Mary':7, 'Zoe':9}
     type(tdict), tdict
```

```
(dict, {'Mary': 7, 'Peter': -4, 'Tom': 12, 'Zoe': 9})
```

```
[12] t = pd.Series(tdict)
     t
```

```
Tom      12
Peter    -4
Mary      7
Zoe       9
dtype: int64
```

```
t.values
```

```
array([12, -4,  7,  9])
```

```
[14] t[3], t['Zoe']
```

```
(9, 9)
```

```
[17] t[0:3]
```

```
Tom      12
Peter    -4
Mary      7
dtype: int64
```

```
t['Tom':'Mary']
```

```
Tom      12
Peter    -4
Mary      7
dtype: int64
```

```
[20] t[[0,2]]
```

```
Tom     12
Mary     7
dtype: int64
```

```
[21] t[['Tom','Mary']]
```

```
Tom     12
Mary     7
dtype: int64
```

Access the elements in a Series

We have two ways to access the elements of a Series:
(1) using the index as if it is a numpy array; or
(2) using the series's index labels

```
[11] tdict = {'Tom':12, 'Peter':-4, 'Mary':7, 'Zoe':9}
     type(tdict), tdict

(dict, {'Mary': 7, 'Peter': -4, 'Tom': 12, 'Zoe': 9})
```

```
[12] t = pd.Series(tdict)
     t

Tom      12
Peter    -4
Mary      7
Zo...
dt...
```

Be very careful

The stopping index is not included in the output

The stopping index label is included in the output

```
t.
array([12, -4,  7,  9])
```

```
[14] t[3], t['Zoe']

(9, 9)
```

```
[17] t[0:3]

Tom      12
Peter    -4
Mary      7
dtype: int64
```

```
t['Tom':'Mary']

Tom      12
Peter    -4
Mary      7
dtype: int64
```

```
[20] t[[0,2]]

Tom     12
Mary     7
dtype: int64
```

```
[21] t[['Tom','Mary']]

Tom     12
Mary     7
dtype: int64
```

Like Numpy Array, element-wise operations are supported

```
[11] tdict = {'Tom':12, 'Peter':-4, 'Mary':7, 'Zoe':9}
     type(tdict), tdict

     (dict, {'Mary': 7, 'Peter': -4, 'Tom': 12, 'Zoe': 9})
```

```
[12] t = pd.Series(tdict)
     t

     Tom      12
     Peter    -4
     Mary      7
     Zoe       9
     dtype: int64
```

```
t.values

array([12, -4,  7,  9])
```

```
[3] t + 2

    Tom      14
    Peter    -2
    Mary      9
    Zoe      11
    dtype: int64
```

```
[4] t / 3

    Tom      4.000000
    Peter   -1.333333
    Mary     2.333333
    Zoe      3.000000
    dtype: float64
```

```
[5] np.square(t)

    Tom      144
    Peter     16
    Mary      49
    Zoe       81
    dtype: int64
```

Conditions and Boolean indexing are also supported

```
[11] tdict = {'Tom':12, 'Peter':-4, 'Mary':7, 'Zoe':9}
     type(tdict), tdict
```

```
(dict, {'Mary': 7, 'Peter': -4, 'Tom': 12, 'Zoe': 9})
```

```
[12] t = pd.Series(tdict)
     t
```

```
Tom      12
Peter    -4
Mary      7
Zoe       9
dtype: int64
```

```
t.values
```

```
array([12, -4,  7,  9])
```

```
[11] t <= 7
```

```
Tom      False
Peter     True
Mary      True
Zoe      False
dtype: bool
```

```
[12] t[t<=7]
```

```
Peter   -4
Mary     7
dtype: int64
```

Some useful functions on Series

```
[14] s = pd.Series([1,0,2,1,2,3])
     s
```

```
0    1
1    0
2    2
3    1
4    2
5    3
dtype: int64
```

```
[17] s.unique(), s.sum(), s.mean(), s.max(), s.min()
```

```
(array([1, 0, 2, 3]), 9, 1.5, 3, 0)
```

```
[18] s.value_counts()
```

```
1    2
2    2
0    1
3    1
dtype: int64
```

DataFrame

- A DataFrame can be thought of as a table like data structure that each column is expressed as a Series.
- It can also be seen as an excel spreadsheet which offers very flexible ways of working with it.

Column names

Out[13]:

	year	team	wins	draws	losses
0	2018	HKU	30	6	2
1	2019	HKU	28	7	3
2	2020	HKU	32	4	2
3	2018	CU	29	5	4
4	2019	CU	32	4	2
5	2020	CU	26	7	5
6	2018	UST	21	8	9
7	2019	UST	17	10	11
8	2020	UST	19	8	11

Row labels

Create a DataFrame using pd.DataFrame

```
[4] data = {'color': ['blue','green','yellow','red','white'],
            'object': ['ball','pen','pencil','paper','mug'],
            'price': [1.2,1.0,0.6,0.9,1.7]}
    frame = pd.DataFrame(data)
    frame
```

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

Create a DataFrame using pd.DataFrame

```
[4] data = {'color': ['blue','green','yellow','red','white'],
            'object': ['ball','pen','pencil','paper','mug'],
            'price': [1.2,1.0,0.6,0.9,1.7]}
    frame = pd.DataFrame(data)
    frame
```

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

Sometimes, the dictionary contains many columns that are not useful, and we can select explicitly useful columns to create the DataFrame

```
frame = pd.DataFrame(data, columns=['object','price'])
frame
```

	object	price
0	ball	1.2
1	pen	1.0
2	pencil	0.6
3	paper	0.9
4	mug	1.7

Create a DataFrame using pd.DataFrame

And of course we can give labels to the indexes.

```
data = {'color': ['blue','green','yellow','red','white'],
        'object': ['ball','pen','pencil','paper','mug'],
        'price': [1.2,1.0,0.6,0.9,1.7]}
frame2 = pd.DataFrame(data, index=['one','two','three','four','five'])
frame2
```

	color	object	price
one	blue	ball	1.2
two	green	pen	1.0
three	yellow	pencil	0.6
four	red	paper	0.9
five	white	mug	1.7

Create a DataFrame using pd.DataFrame

Assigning index labels and selecting columns at the same time:

```
[10] data = {'color': ['blue','green','yellow','red','white'],
             'object': ['ball','pen','pencil','paper','mug'],
             'price': [1.2,1.0,0.6,0.9,1.7]}
     frame2 = pd.DataFrame(data, index=['one','two','three','four','five'],
                           columns=['object','price'])
     frame2
```

	object	price
one	ball	1.2
two	pen	1.0
three	pencil	0.6
four	paper	0.9
five	mug	1.7

Create a DataFrame using pd.DataFrame

In addition to dictionary, there are many other ways to create a dataframe, e.g., list of lists, 2D numpy arrays, …

```
[13] frame3 = pd.DataFrame(np.arange(16).reshape((4,4)),
                           index=['red','blue','yellow','white'],
                           columns=['ball','pen','pencil','paper'])
     frame3
```

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yellow	8	9	10	11
white	12	13	14	15

Create a DataFrame using pd.DataFrame

Creating a dataframe from a list of lists

```
import pandas as pd

t = pd.DataFrame([[1,2,3],[4,5,6]])
t
```

	0	1	2
0	1	2	3
1	4	5	6

```
import pandas as pd

t = pd.DataFrame([[1,2,3],[4,5,6]],columns=['A','B','C'])
t
```

	A	B	C
0	1	2	3
1	4	5	6

Some functions for information

```
[16] data = {'color':['blue','green','yellow','red','white'],
             'object': ['ball','pen','pencil','paper','mug'],
             'price':[1.2,1.0,0.6,0.9,1.7]}
     frame = pd.DataFrame(data)
     frame
```

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

```
[17] frame.columns

     Index(['color', 'object', 'price'], dtype='object')
```

```
[18] frame.index

     RangeIndex(start=0, stop=5, step=1)
```

```
[19] frame.values

     array([['blue', 'ball', 1.2],
            ['green', 'pen', 1.0],
            ['yellow', 'pencil', 0.6],
            ['red', 'paper', 0.9],
            ['white', 'mug', 1.7]], dtype=object)
```

Some statistical functions

```
[16] data = {'color':['blue','green',
             'object': ['ball','pen',
             'price':[1.2,1.0,0.6,0.9
     frame = pd.DataFrame(data)
     frame
```

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

```
frame.sum()

color       bluegreenyellowredwhite
object        ballpenpencilpapermug
price                           5.4
dtype: object
```

```
frame.mean()

/usr/local/lib/python3.7/dist-packa
  """Entry point for launching an 
price    1.08
dtype: float64
```

```
frame.max()

color      yellow
object     pencil
price         1.7
dtype: object
```

```
[8]  frame.min()

color      blue
object     ball
price       0.6
dtype: object
```

Some statistical functions

```
[16] data = {'color':['blue','green','yellow','red','white'],
             'object': ['ball','pen','pencil','paper','mug'],
             'price':[1.2,1.0,0.6,0.9,1.7]}
     frame = pd.DataFrame(data)
     frame
```

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

```
frame.sort_values(by='price')
```

	color	object	price
2	yellow	pencil	0.6
3	red	paper	0.9
1	green	pen	1.0
0	blue	ball	1.2
4	white	mug	1.7

```
frame.median()
```

```
/usr/local/lib/python3.7/dist-packa
  """Entry point for launching an I
price    1.0
dtype: float64
```

Selecting rows and columns

```
data = {'color' : ["blue", "green", "yellow", "red", "white"],
        'object' : ["ball", "pen", "pencil", "paper", "mug"],
        'price' : [1.2, 1.0, 0.6, 0.9, 1.7]}
frame = pd.DataFrame(data, index = ['zero', 'one', 'two', 'three', 'four'])
frame
```

	color	object	price
zero	blue	ball	1.2
one	green	pen	1.0
two	yellow	pencil	0.6
three	red	paper	0.9
four	white	mug	1.7

Selecting columns

```
frame['color']
```

```
zero       blue
one       green
two      yellow
three       red
four      white
Name: color, dtype: object
```

```
frame[['object','price']]
```

	object	price
zero	ball	1.2
one	pen	1.0
two	pencil	0.6
three	paper	0.9
four	mug	1.7

Selecting rows and columns

- Selecting rows

```
: frame[1]

---------------------------------------
KeyError
~\anaconda3\lib\site-packages\pandas\core
   3360                try:
-> 3361                    return self._engi
   3362                except KeyError as er

~\anaconda3\lib\site-packages\pandas\_lib

~\anaconda3\lib\site-packages\pandas\_lib
```

Use the methods
iloc is for selecting by index, and
loc is for selecting by index label.

Selecting rows and columns

```
frame
```

	color	object	price
zero	blue	ball	1.2
one	green	pen	1.0
two	yellow	pencil	0.6
three	red	paper	0.9
four	white	mug	1.7

```
frame.iloc[1]
```

```
color       green
object        pen
price         1.0
Name: one, dtype: objec
```

```
frame.iloc[3,2]
```

```
0.9
```

```
frame.iloc[1:4]
```

	color	object	price
one	green	pen	1.0
two	yellow	pencil	0.6
three	red	paper	0.9

```
frame.iloc[:,2]
```

```
zero     1.2
one      1.0
two      0.6
three    0.9
four     1.7
Name: price, dtype: float64
```

```
frame.iloc[1:3,2]
```

```
one    1.0
two    0.6
Name: price, dtype: float64
```

```
frame.iloc[[1,3], [2]]
```

	price
one	1.0
three	0.9

```
frame.iloc[[1,3], [0,2]]
```

	color	price
one	green	1.0
three	red	0.9

Selecting rows and columns

```
frame
```

	color	object	price
zero	blue	ball	1.2
one	green	pen	1.0
two	yellow	pencil	0.6
three	red	paper	0.9
four	white	mug	1.7

```
frame.loc[['one', 'two'], ['object', 'price']]
```

	object	price
one	pen	1.0
two	pencil	0.6

```
frame.loc['one':'four',['object', 'price']]
```

	object	price
one	pen	1.0
two	pencil	0.6
three	paper	0.9
four	mug	1.7

Selecting rows and columns

Again, be careful when slicing a DataFrame

```
f = pd.DataFrame([[1,2,3],[4,5,6],[7,8,9]], columns=['A','B','C'], index=[0,1,2]
f
```

	A	B	C
0	1	2	3
1	4	5	6
2	7	8	9

```
f.iloc[0:2]
```

	A	B	C
0	1	2	3
1	4	5	6

stopping index not included

```
f.loc[0:2]
```

	A	B	C
0	1	2	3
1	4	5	6
2	7	8	9

stopping index label included

Reading and writing files

- Most of public data are delivered in plain text format, excel format, CSV (comma-separated- value) format, or any other delimiter-separated value format.
- Pandas provides many methods for reading these files, e.g., read_csv(), read_excel(), read_table(), read_clipboard();
- and methods for writing these files, e.g., to_csv(), to_excel, to_table(), ...

An example

In [7]:
```python
import pandas as pd

t = pd.read_excel("educ_uoe_fine06.xls")
t
```

Out[7]:

	GEO/TIME	2012	2013	2014	2015	2016	2017	2018
0	European Union - 27 countries (from 2020)	:	:	4.96	4.81	:	4.64	:
1	European Union - 28 countries (2013-2020)	:	:	:	:	:	4.73	:
2	Belgium	6.43	6.56	6.49	6.43	6.33	6.3	[illegible]
3	Bulgaria	3.68	4.06	4.22	3.93	3.86	4.09	4.05
4	Czechia	4.33	3.95	3.84	3.79	3.56	3.77	4.23
5	Denmark	:	:	:	:	6.84	6.5	6.24
6	Germany (until 1990 former territory of the FRG)	4.64	4.61	4.57	4.52	4.5	4.51	4.59
7	Estonia	4.7	4.85	:	:	:	:	:
8	Ireland	6.16	5.32	4.92	3.77	:	:	:

for
missing values

In [8]:

```
import pandas as pd

t = pd.read_excel("educ_uoe_fine06.xls",na_values=':')
t
```

Out[8]:

	GEO/TIME	2012	2013	2014	2015	2016	2017	2018
0	European Union - 27 countries (from 2020)	NaN	NaN	4.96	4.81	NaN	4.64	NaN
1	European Union - 28 countries (2013-2020)	NaN	NaN	NaN	NaN	NaN	4.73	NaN
2	Belgium	6.43	6.56	6.49	6.43	6.33	6.30	6.29
3	Bulgaria	3.68	4.06	4.22	3.93	3.86	4.09	4.05
4	Czechia	4.33	3.95	3.84	3.79	3.56	3.77	4.23
5	Denmark	NaN	NaN	NaN	NaN	6.84	6.50	6.24
6	Germany (until 1990 former territory of the FRG)	4.64	4.61	4.57	4.52	4.50	4.51	4.59
7	Estonia	4.7	4.85	NaN	NaN	NaN	NaN	NaN
8	Ireland	6.16	5.32	4.92	3.77	NaN	NaN	NaN

<u>Not a Number</u>
Pandas has many methods to deal with this value

Note: We cannot access local file from Colab directly, we need to upload the file first

```
from google.colab import files
```

```
uploaded = files.upload()
```

Choose Files educ_uoe_fine06.xls

- **educ_uoe_fine06.xls**(application/vnd.ms-excel) - 30208 bytes, last modified: 14/1/2025 - 100% done

```
Saving educ_uoe_fine06.xls to educ_uoe_fine06.xls
```

```
t = pd.read_excel("educ_uoe_fine06.xls", na_values = ':')
t
```

	GEO/TIME	2012	2013	2014	2015	2016	2017	2018
0	European Union - 27 countries (from 2020)	NaN	NaN	4.96	4.81	NaN	4.64	NaN
1	European Union - 28 countries (2013-2020)	NaN	NaN	NaN	NaN	NaN	4.73	NaN
2	Belgium	6.43	6.56	6.49	6.43	6.33	6.30	6.29

Additional resources for Python's scientific computing stack

If you are not yet familiar with Python's scientific libraries or need a refresher, please see the following resources:

- **NumPy**: https://sebastianraschka.com/pdf/books/dlb/appendix_f_numpy-intro.pdf
- **pandas**: https://pandas.pydata.org/pandas-docs/stable/10min.html
- **Matplotlib**: https://matplotlib.org/tutorials/introductory/usage.html