Deep Learning for Science and Engineering Teaching Kit

Deep Learning for Scientists and Engineers

Lecture 2: A primer on Python, NumPy, SciPy and jupyter notebook

Instructors: George Em Karniadakis, Khemraj Shukla, Lu Lu
Teaching Assistants: Vivek Oommen and Aniruddha Bora

Course Roadmap

Module-1 (Basics)

- Lecture 1: Introduction
- Lecture 2: A primer on Python, NumPy, SciPy and *jupyter* notebooks
- Lecture 3: Deep Learning Networks
- Lecture 4: A primer on TensorFlow and PyTorch
- Lecture 5: Training and Optimization
- Lecture 6: Neural Network Architectures

Module-2 (PDEs and Operators)

- Lecture 7: Machine Learning using Multi-Fidelity Data
- Lecture 8: Physics-Informed Neural Networks (PINNs)
- Lecture 9: PINN Extensions
- Lecture 10: Neural Operators

Module-3 (Codes & Scalability)

- Lecture 11: Multi-GPU Scientific Machine Learning

Contents

Objectives

- ❑ Getting familiar with programming environment of the course
- ❑ Introduction of ***jupyter*** notebook and setting it up on your machine
- ❑ Basics of data structure and operation in NumpPy and SciPy
- ❑ Installation of deep learning frameworks TensorFlow and PyTorch
- ❑ Introduction to Nvidia's deep learning container and installation

Programming Languages

- ❑ A programming language is a set of rules that are used to communicate with the central brain of the computer and execute a single task or multiple tasks.

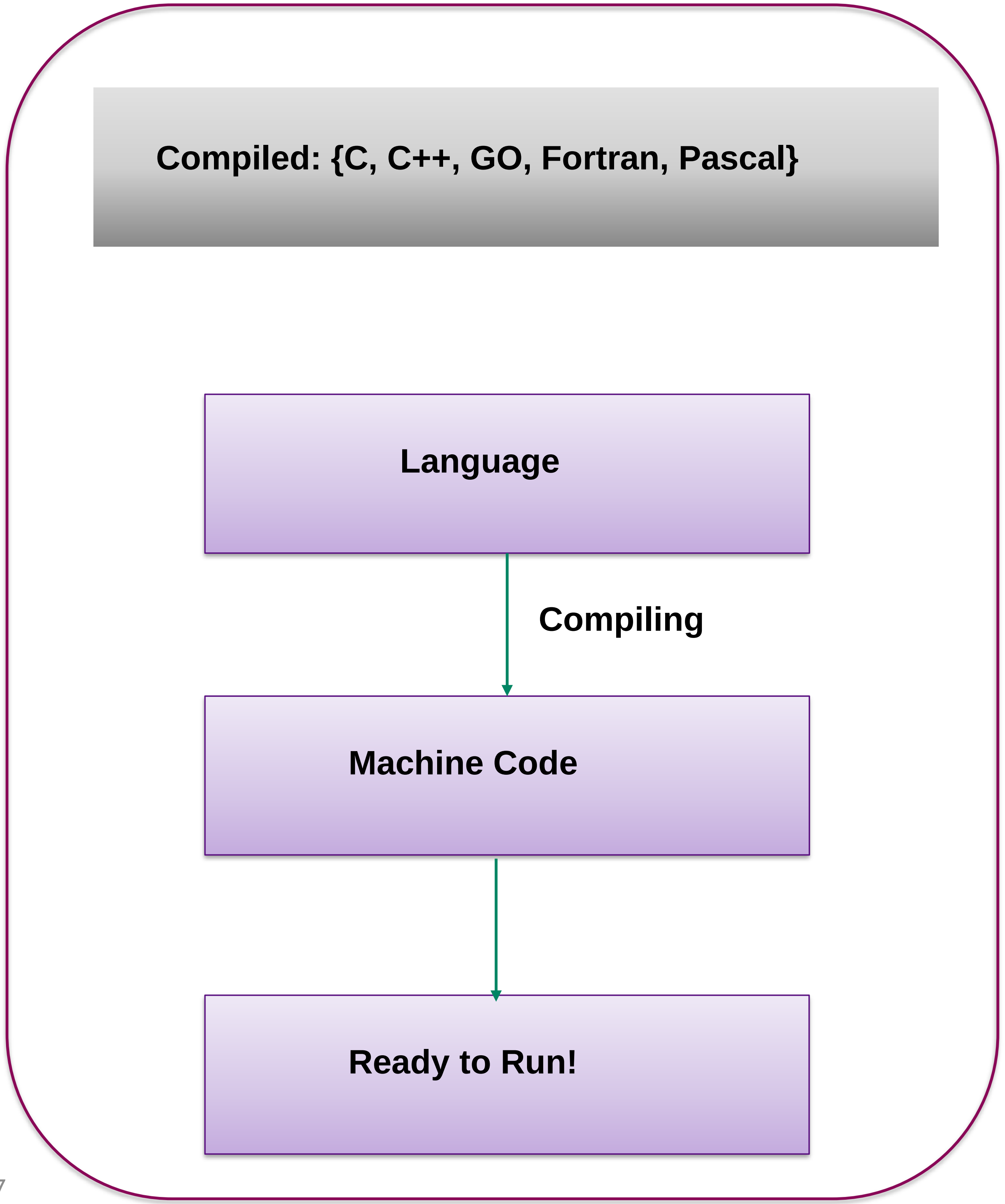
- ❑ Compiled languages vs. Interpreted language

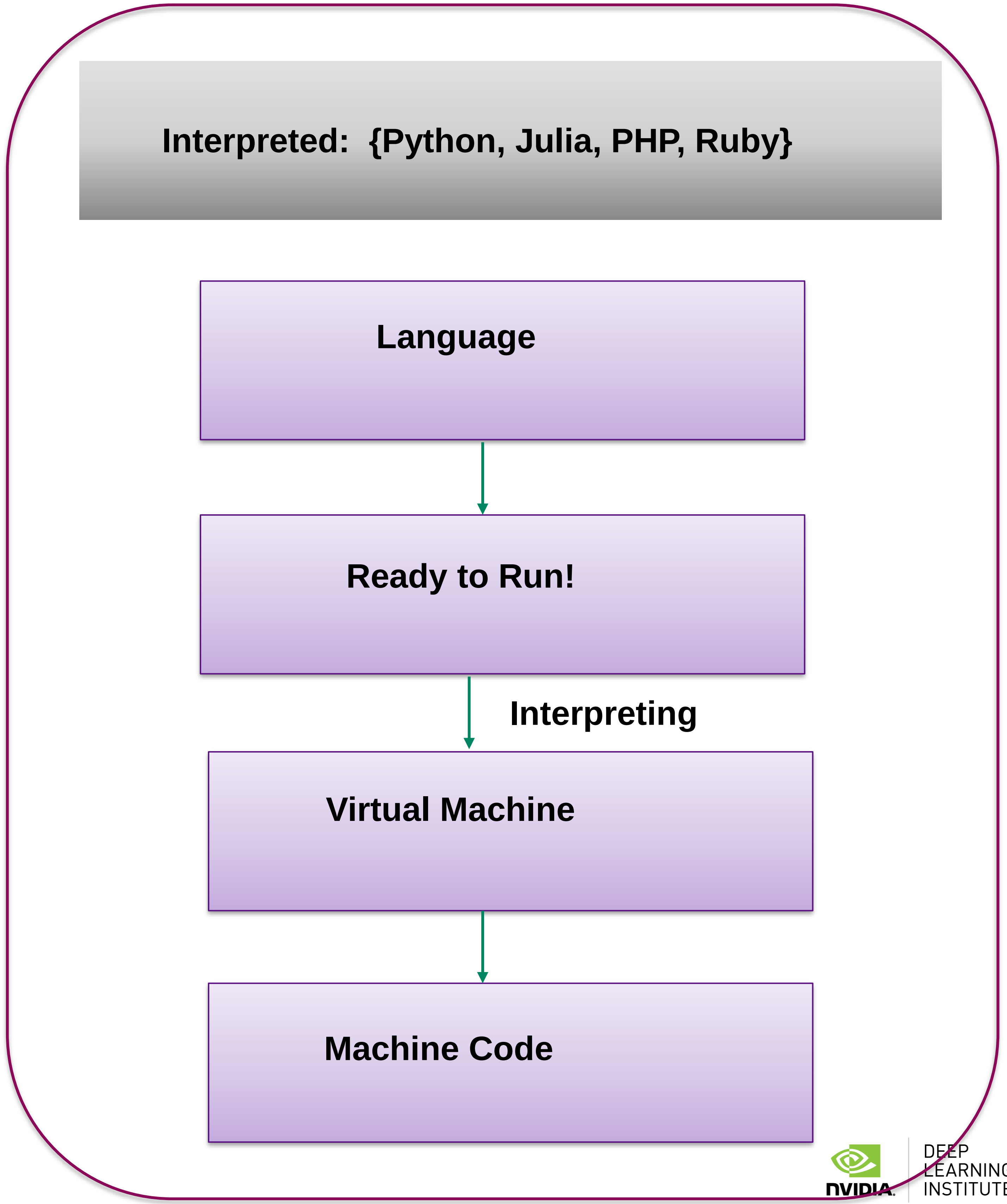
  - ❑ Compiled language: A program, once compiled, is expressed directly as instructions for the target machine. e.g., Addition “+” could be translated to “ADD” instructions in machine code. C,C++, Fortran etc.

  - ❑ Interpreted language: Instructions are not directly executed by target machine, but instead read and executed by some language on the native machine. e.g., Addition “+” is recognized by interpreter at run time and its own “add(a,b)” function is used to execute the machine code “ADD” instructions. MATLAB, JavaScript, Python, Ruby etc.

- ❑ In this course we will primarily use the Python3 language for all the implementations.

A Code: From High level to Machine level





Python

- Developed by **Guido Van Rossum** in the early 1990s.
- Named after **Monty Python.**
- Open-source web application: Jupyter Notebook.
- A good general-purpose, dynamic (interpreted), and object- oriented language.
- For a machine learning practitioner, Python is a first “choice”.
- Human readable programming language.

Python

- Interpreted language: work with an evaluator for language expressions
- Dynamically typed: variables do not have a predefined type
- Most important data types
  - Lists
  - Tuples
  - Dictionaries (maps) Sets or Hash
- Indentation instead of braces Sequence types
- Sequence types
  - Strings '...' : made of characters, immutable
  - Lists [...] : made of anything mutable
  - Tuples (...) : made of anything, immutable

Installing Python and Python packages

- All the demonstrations in the course will be carried out in jupyter notebooks.

- The best way to install jupyter notebook on your machine is to install the Anaconda Package, which will automatically install following packages.

jupyter SciPy SPYDER NumPy matplotlib

Essential Python Packages

jupyter is a web-based environment for writing the Python codes, generating the plots and analyzing the data.

SciPy is python library, which provide API for numerical algorithms, including signal processing, numerical integration, optimization, statistics, saving and loading the data.

NumPy is a core python library and used for array data formation and performing the operations on them. NumPy is also used to perform basic linear algebra operations.

SPYDER is an Integrated Development Environment for Python

Launching the jupyter

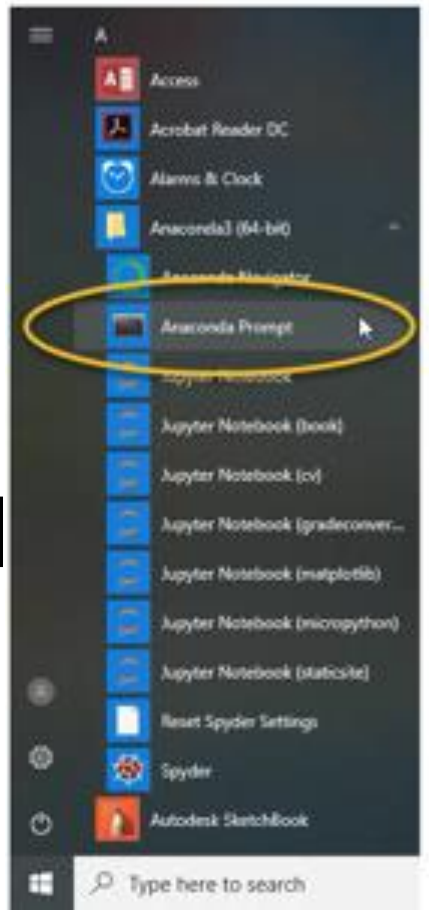
- Mac and Unix OS
  - Open a Terminal and go to any folder
  - Enter on terminal prompt

```
$ jupyter notebook
```

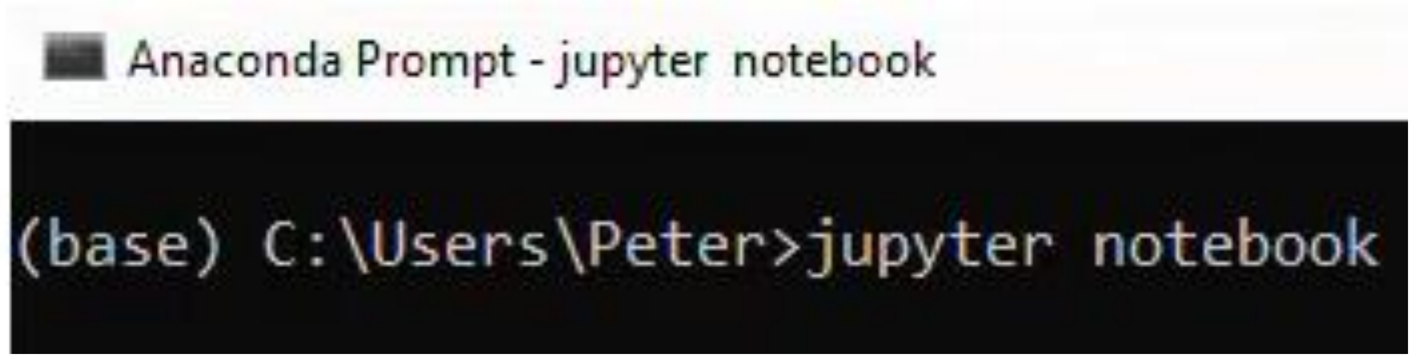
- Windows
  - Open a Terminal and go to any folder
  - Enter on terminal prompt

```
$ jupyter notebook
```

- Windows
  - Go to the Windows start menu and select **[Anaconda Prompt]** under **[Anaconda3]**



  - On the Anaconda Prompt type jupyter notebook like below



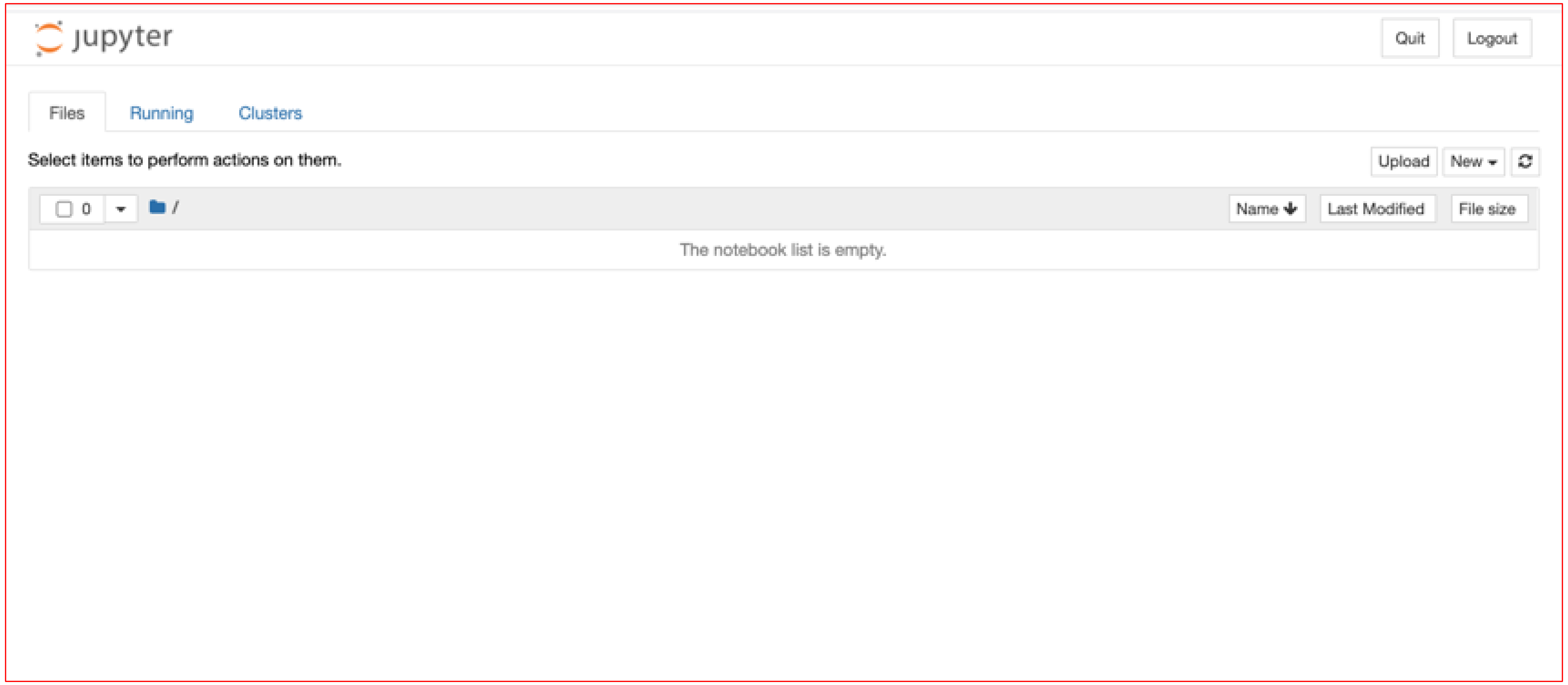
Anaconda Prompt - jupyter notebook

```
(base) C:\Users\Peter>jupyter notebook
```

  - Once you enter, jupyter web interface will appear

jupyter web interface

- jupyter launch will look like



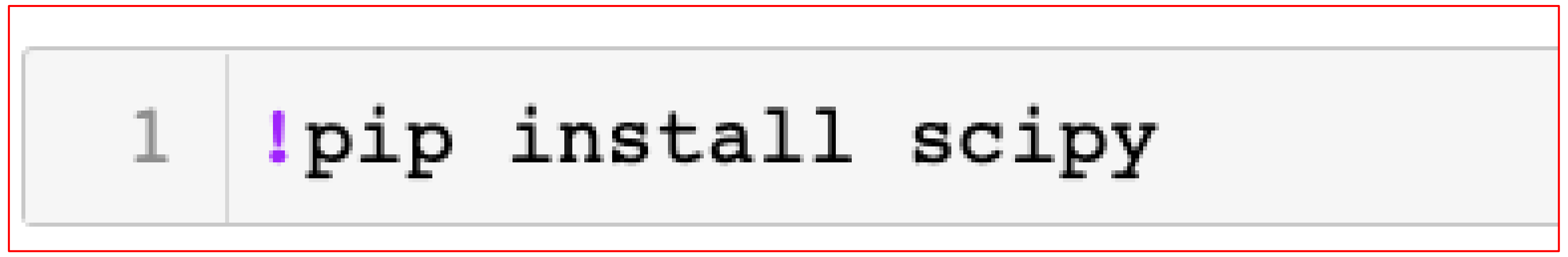
Select items to perform actions on them.

/

The notebook list is empty.

Some useful commands for jupyter

Install a python package via jupyter



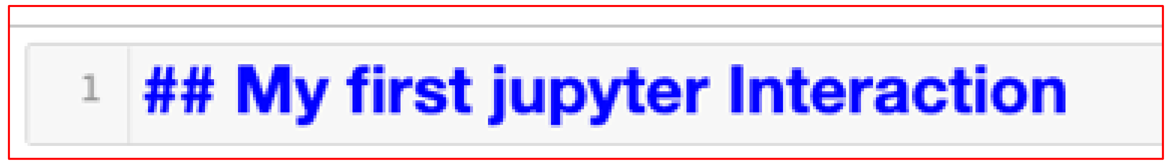
```
!pip install scipy
```

Executing a cell:

Select the cell and then press `shift + enter`

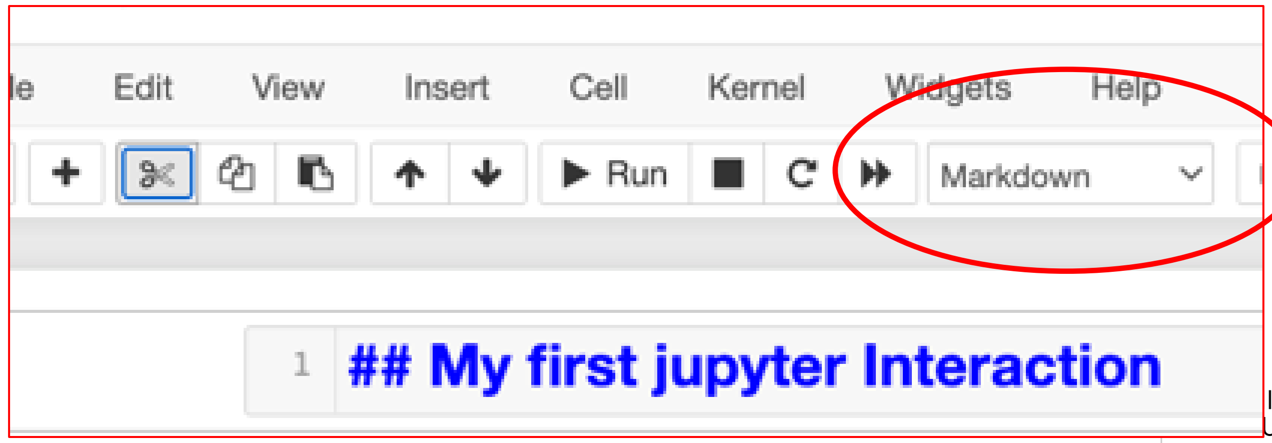
Adding documentation using Markdown:

1. Write the documentation in a cell ➔



```
## My first jupyter Interaction
```

2. Select Markdown from drop down list ➔



```
## My first jupyter Interaction
```

Basics of Python

- ❑ Indentation matters to the meaning of the code
- ❑ Variable types:
  - ❑ don’t need to be declared
  - ❑ Python figures out the variable type on its own
- ❑ Assignment uses = and comparison uses ==.
- ❑ For numbers + (add), – (subtract), * (Multiply), / (divide), // (Integer division) and % (remainder) are as expected
- ❑ Special use of + for string concatenation
- ❑ Special use of % for string formatting
- ❑ Logical operators are words (and, or, not) not symbols
- ❑ Comments start with #

Demo: Basic unary and binary operations

Sequence Types

❑ Tuple

A simple immutable ordered sequence of items. It cannot be modified once created Items can be mixed type

❑ Strings

- Immutable
- Conceptually similar to tuples
- Regular strings use 8-bit characters.

❑ List:

- Mutable ordered sequence of items of same or mixed types

Mutability: Tuples vs. Lists

- ❑ Lists are mutable

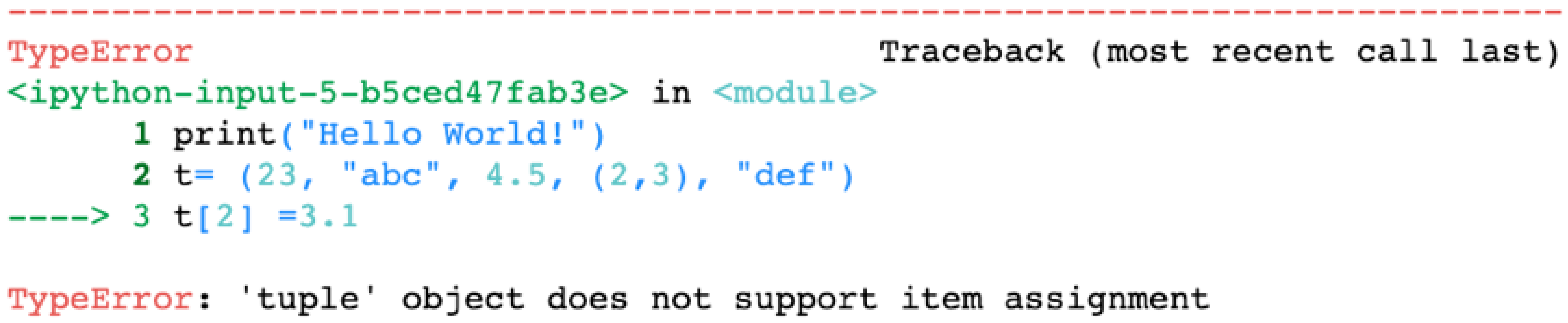
```
>> li = [‘abc’, 23, 4.5, 26] ## List of mix data type
>> li[1] = 45  ## Change second element: mutation
>> print(f “li: {li}”)
   li: [‘abc’, 45, 3.5, 26] ## mutated List
```

- ❑ We can change lists in place and `li` still points to the same memory reference

Mutability: Tuples vs Lists

- ❑ Consider a tuple

```
>> t= (23, “abc”, 4.5, (2,3), “def”) ## Tuple of mix data type
>> t[2] = 3.1 ## mutating tuple
```



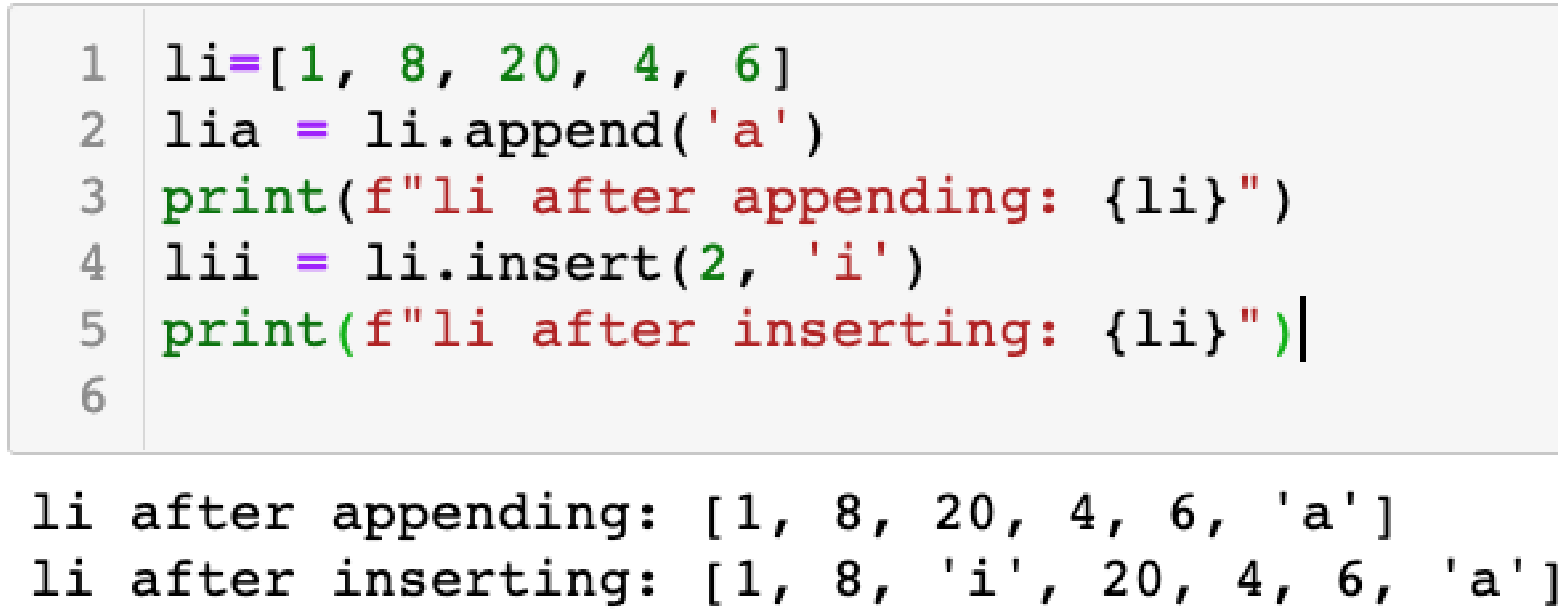
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-b5ced47fab3e> in <module>
      1 print("Hello World!")
      2 t= (23, "abc", 4.5, (2,3), "def")
----> 3 t[2] =3.1

TypeError: 'tuple' object does not support item assignment
```

- ❑ You can make a fresh tuple and assign its reference to a previously used name.

- ❑ The immutability of tuples means they’re faster than lists.

List Operations

- ❑ `append and insert`



```
li=[1, 8, 20, 4, 6]
lia = li.append('a')
print(f"li after appending: {li}")
lii = li.insert(2, 'i')
print(f"li after inserting: {li}")

```

```
li after appending: [1, 8, 20, 4, 6, 'a']
li after inserting: [1, 8, 'i', 20, 4, 6, 'a']
```

- ❑ Other operations include: count, remove, index, reverse, sort etc.

Demo: List and Tuples

- Demo

Dictionaries

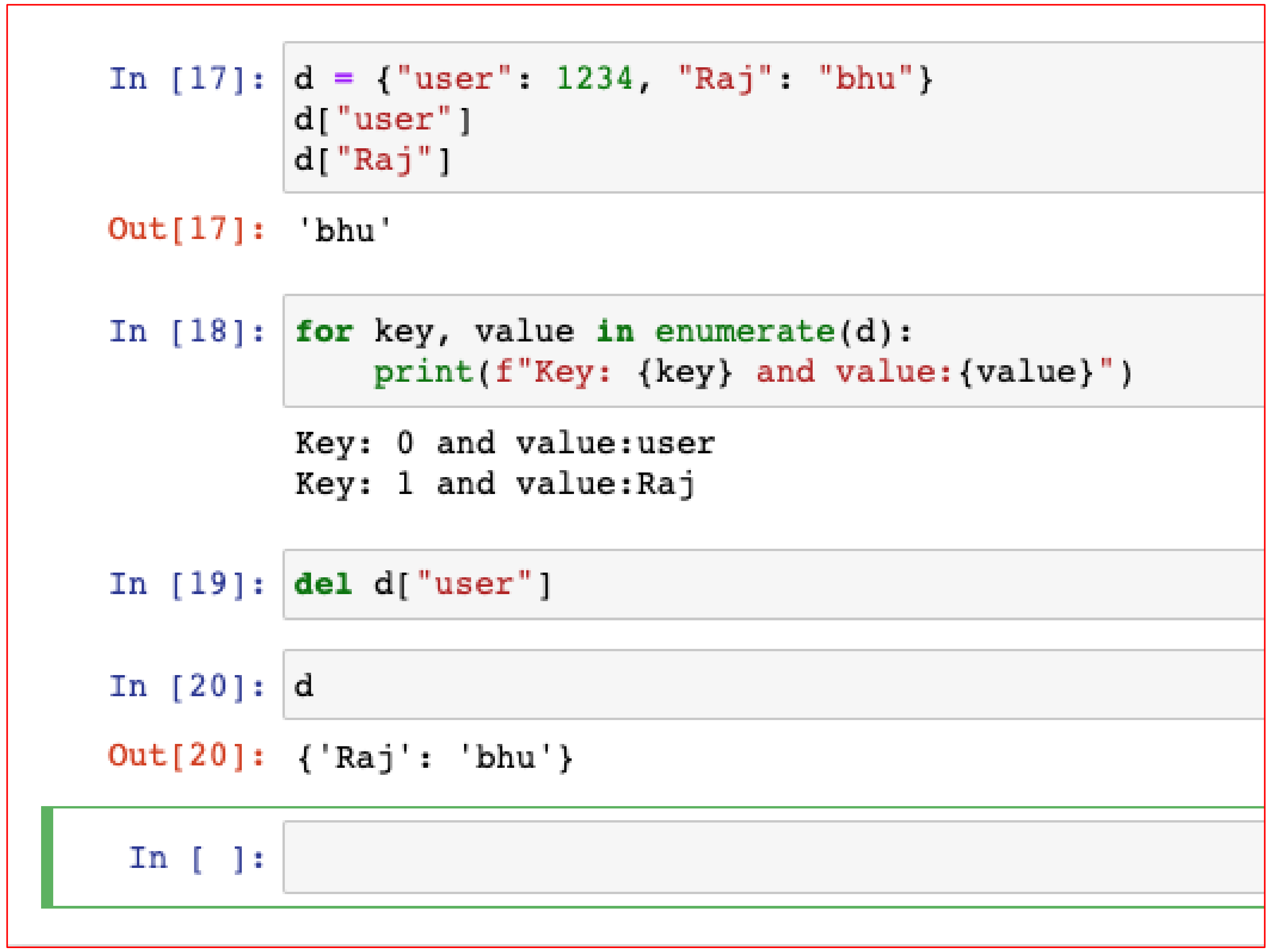
❑Dictionary is enclosed in curly brackets {}

❑Keys must be unique to avoid key Collison

❑Assigning to the existing key replaces the value

❑`del` is a function that can remove dictionary entries.

❑`clear()` removes all entries.



```
In [17]: d = {"user": 1234, "Raj": "bhu"}
         d["user"]
         d["Raj"]

Out[17]: 'bhu'

In [18]: for key, value in enumerate(d):
             print(f"Key: {key} and value:{value}")

         Key: 0 and value:user
         Key: 1 and value:Raj

In [19]: del d["user"]

In [20]: d

Out[20]: {'Raj': 'bhu'}

In [ ]:
```

Demo: Dictionaries

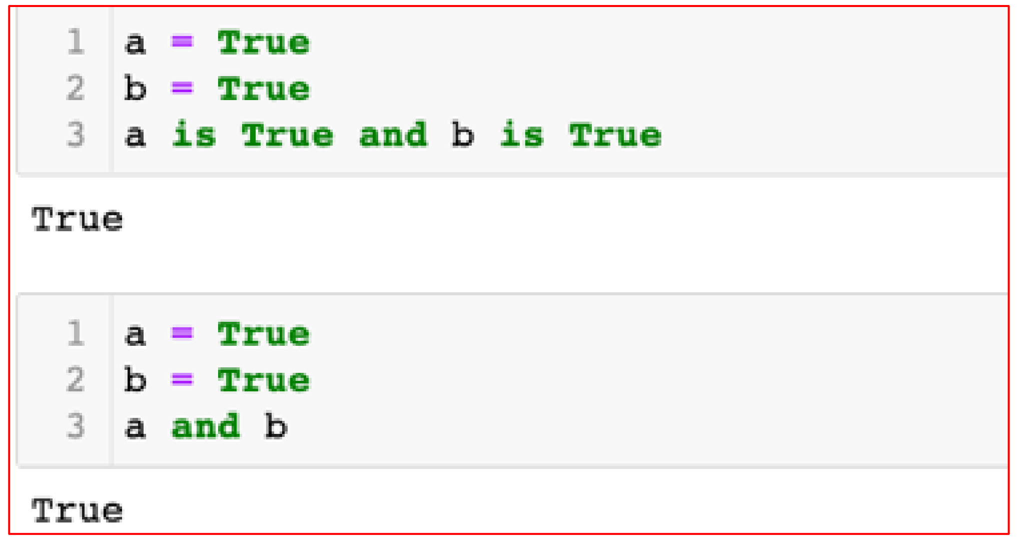
Booleans: True and False

- ❑ True and False are constants
- ❑ Comparison operators:

  `== (equal), != (not equal), < (less than), <= (less than or equal to), etc.`

  - ❑`X == Y ## X and Y have same value`
  - ❑X is Y: ## X and Y refer to the exact same object

Logical Operators: One can also combine Boolean expressions



```
a = True
b = True
a is True and b is True
```

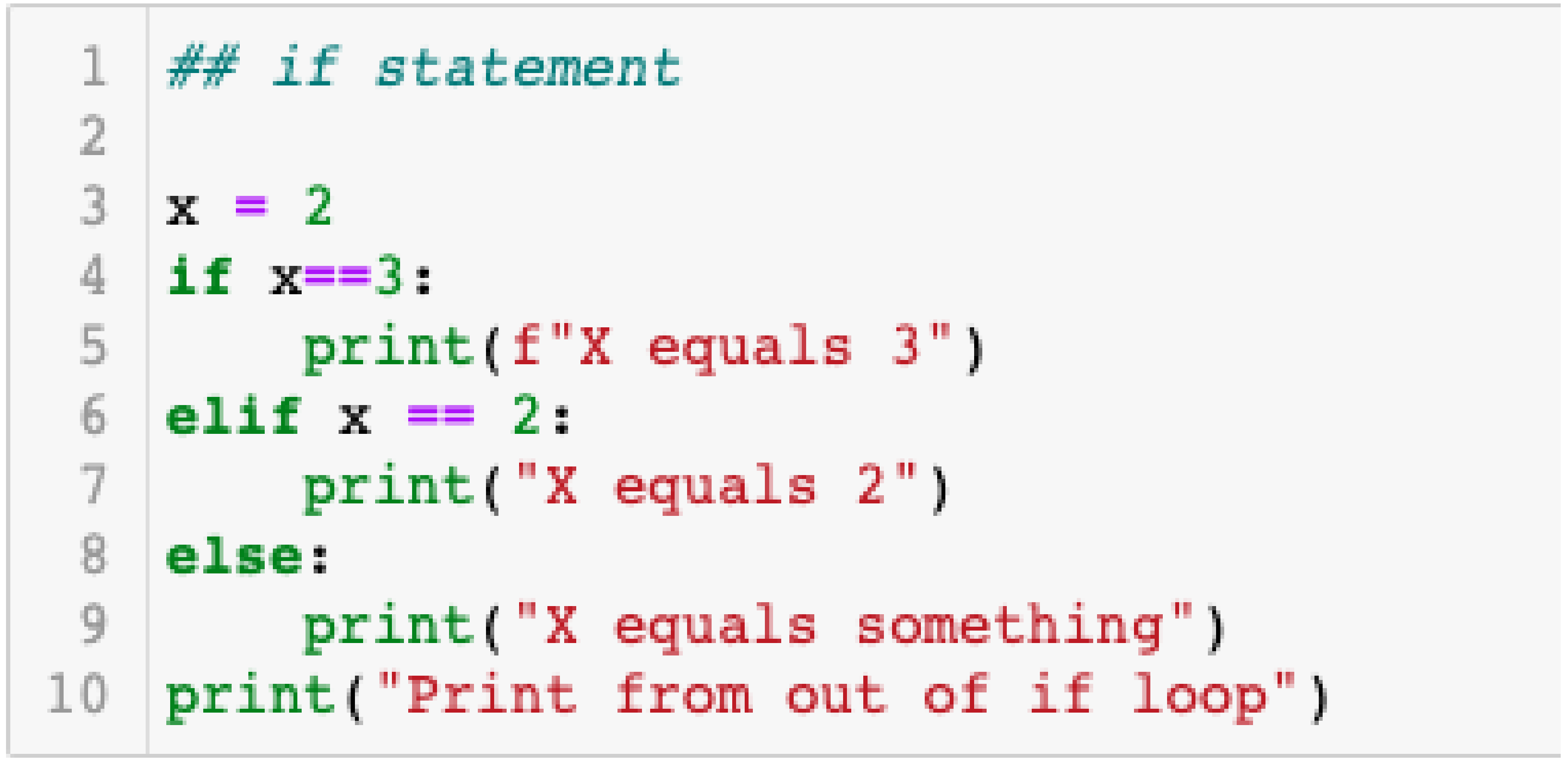
```
True
```

```
a = True
b = True
a and b
```

```
True
```

Demo: Booleans

Conditional: *if* statements



```python
## if statement

x = 2
if x==3:
    print(f"X equals 3")
elif x == 2:
    print("X equals 2")
else:
    print("X equals something")
print("Print from out of if loop")
```

- ❑ Please note use of indentation for blocks
- ❑ Colon (:) after Boolean expression

Demo: *if* statements

for loop

- Only form of for loop in Python

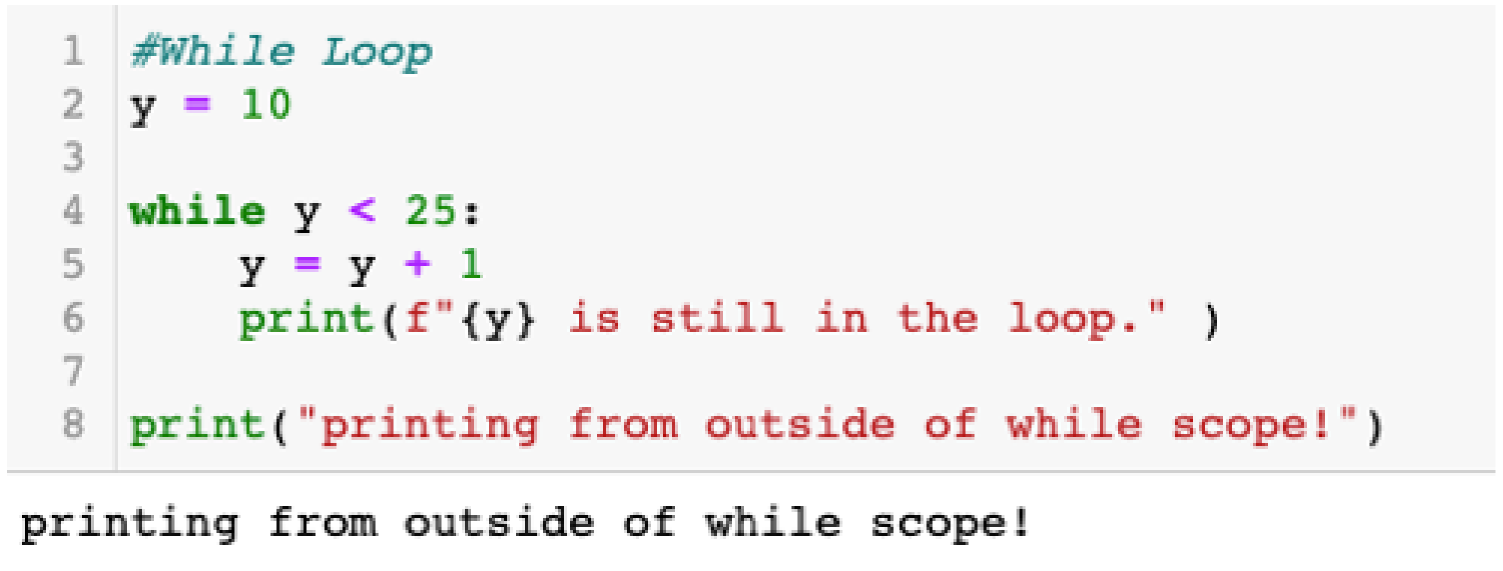
- For loop steps through each of the items in a collection type, or any other object type which is “iterable”

```
for <item> in <collection>:
     <statements>
```

- ***for* loop and *range()* function**

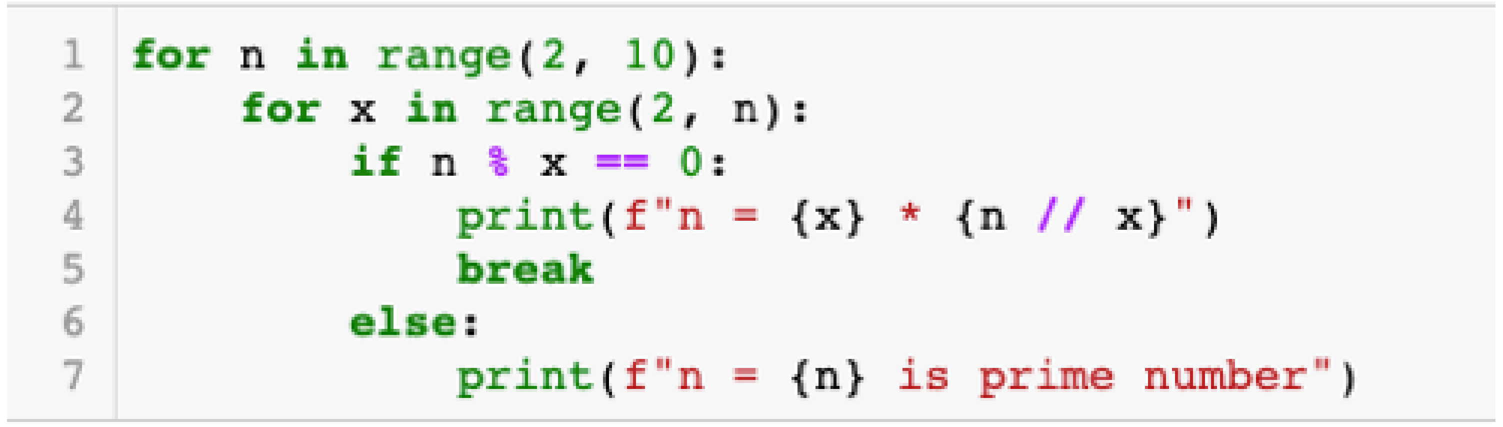
```
for x in range(5):
    print(x)
```

Conditional: *while* statements

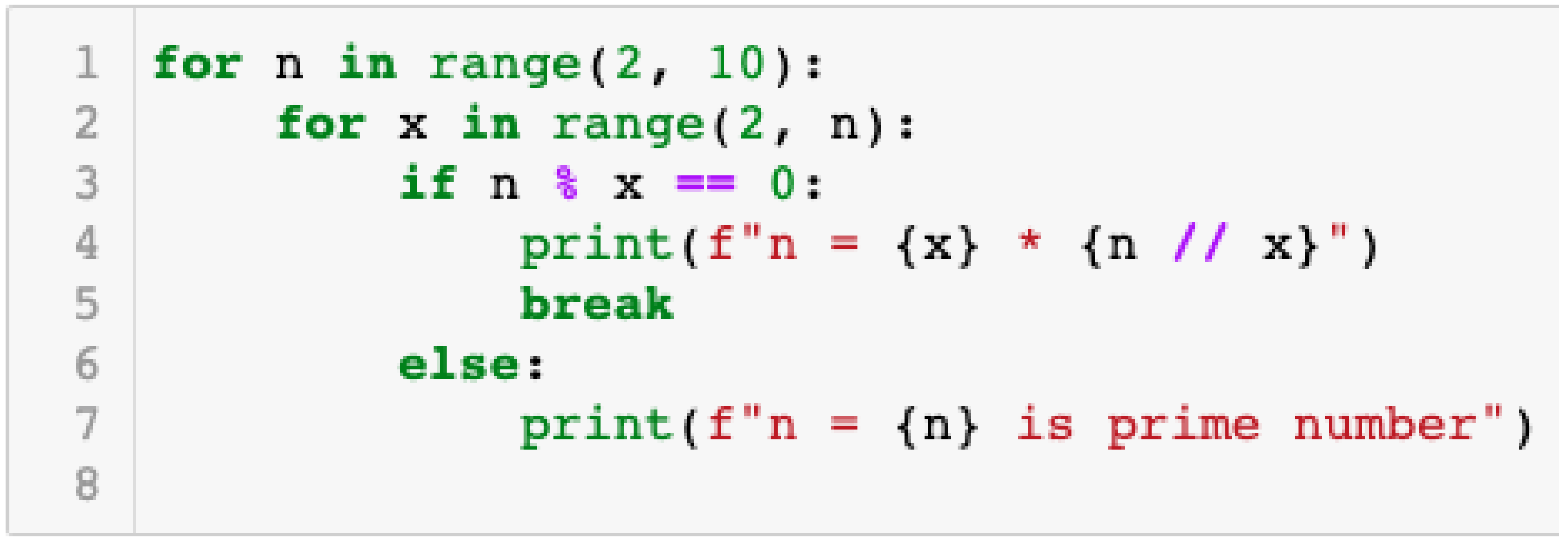


```python
#While Loop
y = 10

while y < 25:
    y = y + 1
    print(f"{y} is still in the loop." )

print("printing from outside of while scope!")
```

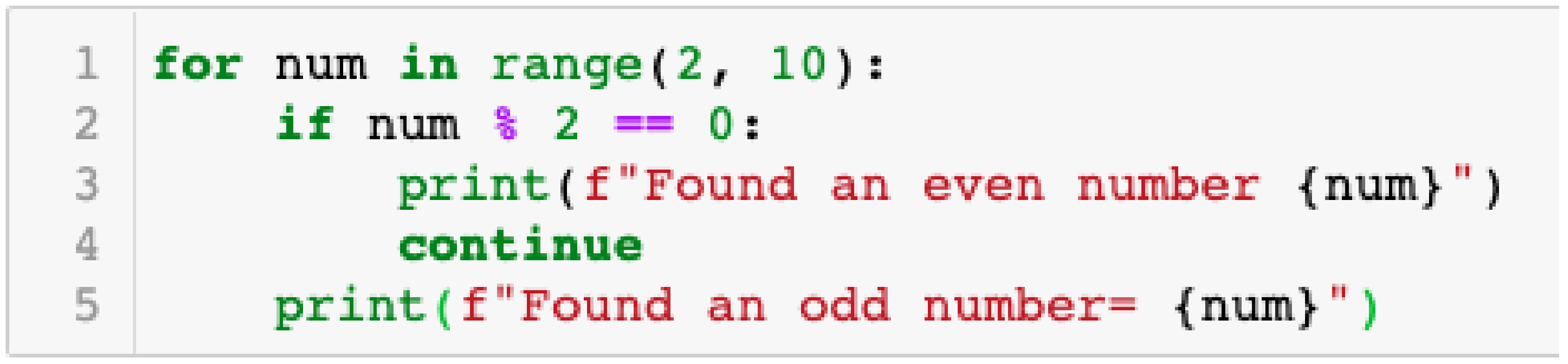
```
printing from outside of while scope!
```



```python
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"n = {x} * {n // x}")
            break
        else:
            print(f"n = {n} is prime number")
```



```python
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(f"n = {x} * {n // x}")
            break
        else:
            print(f"n = {n} is prime number")

```



```python
for num in range(2, 10):
    if num % 2 == 0:
        print(f"Found an even number {num}")
        continue
    print(f"Found an odd number= {num}")
```

- ❑ *break* inside a loop to leave the *while* loop entirely

- ❑ Can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and go to the next iteration

Demo: *`for, while, break and continue`*

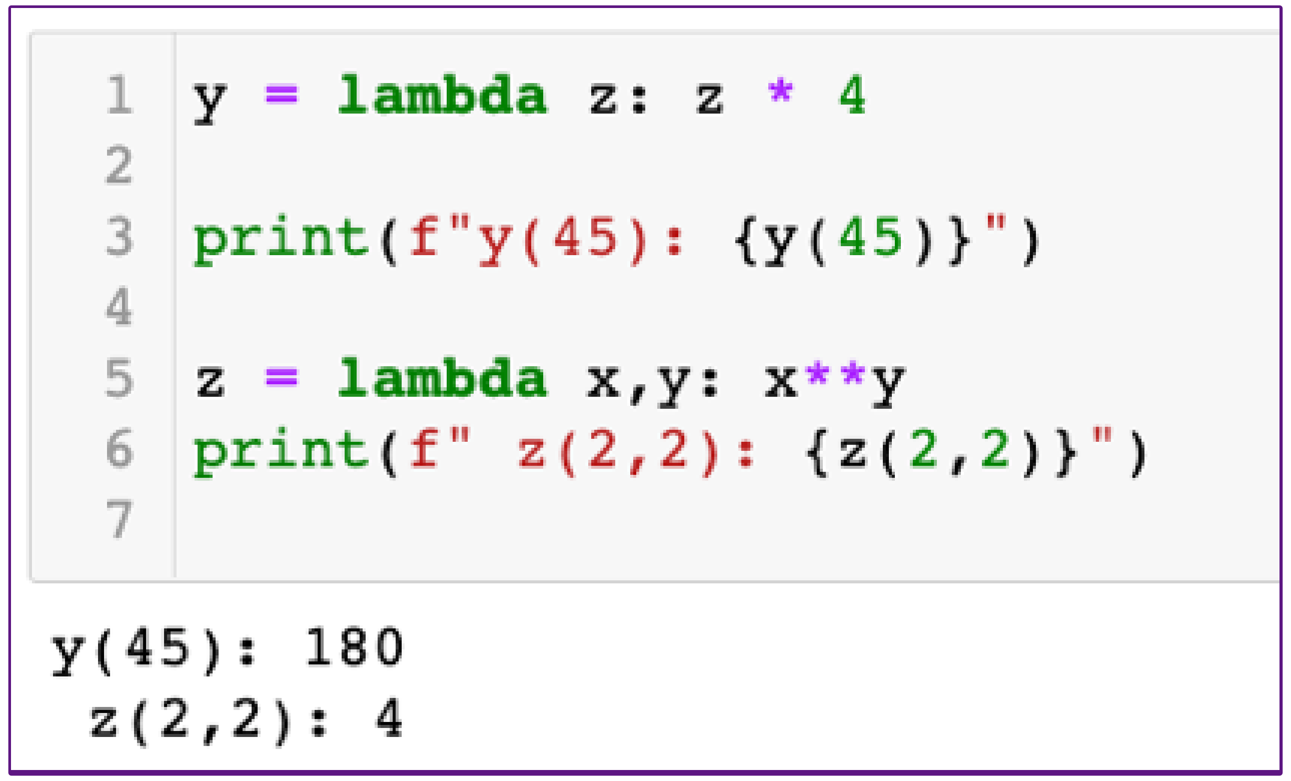
Functions

```
# Define a function

def myfun(x,y):

    return x * y

# Call the function myfun

>>> myfun(3, 4) ‘

>>> 12
```

- ❑ All functions in Python have a return value (even if no return line inside the code)
- ❑ Function without a return return the special value None

Special function: *Lambda*

- Lambda function are useful when passing a short function as an argument to another function.

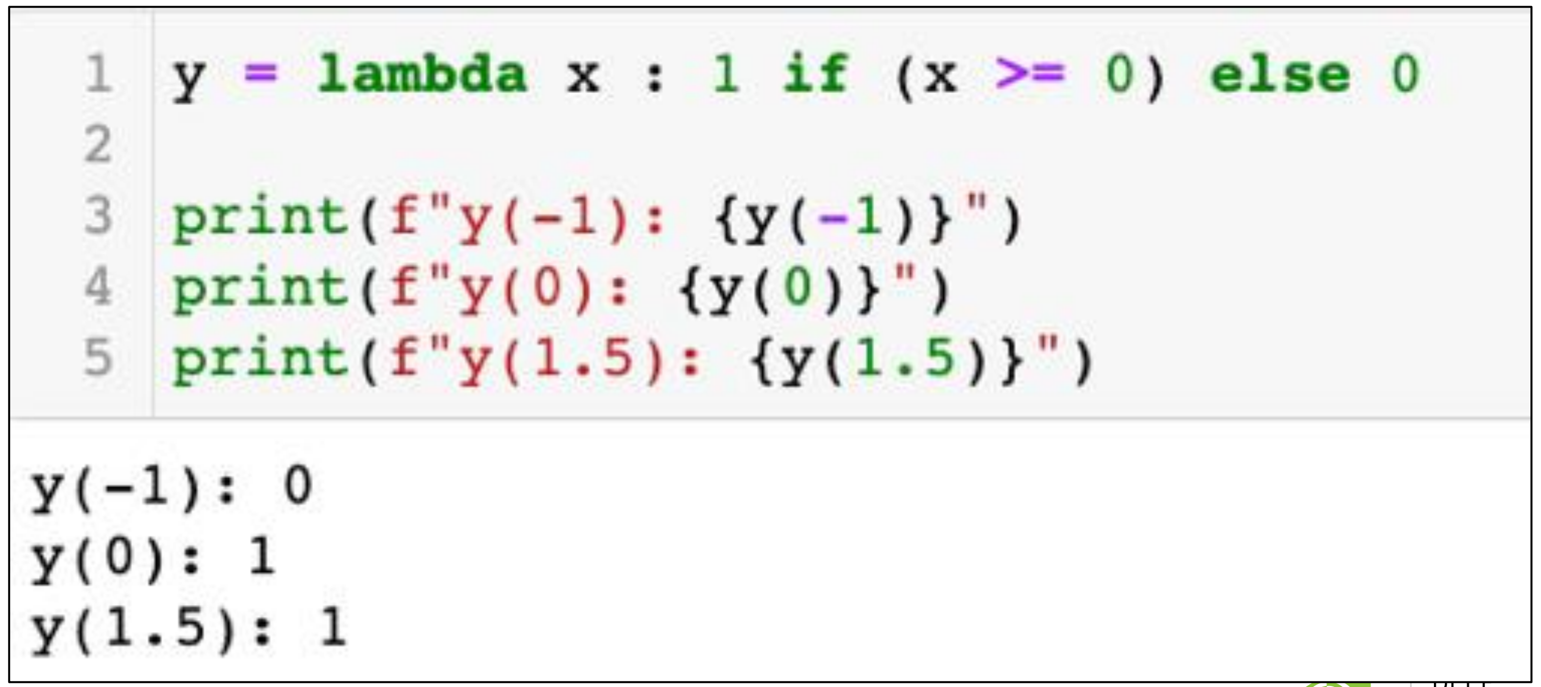


```python
y = lambda z: z * 4

print(f"y(45): {y(45)}")

z = lambda x,y: x**y
print(f" z(2,2): {z(2,2)}")

```

```
y(45): 180
 z(2,2): 4
```

- Define Heaviside Function

$$
\begin{aligned}
f(x) &:= 0 \qquad \text{if } x < 1 \\
&:= 1 \qquad \text{if } x \geq 1
\end{aligned}
$$



```python
y = lambda x : 1 if (x >= 0) else 0

print(f"y(-1): {y(-1)}")
print(f"y(0): {y(0)}")
print(f"y(1.5): {y(1.5)}")
```

```
y(-1): 0
y(0): 1
y(1.5): 1
```

Basics of NumPy array

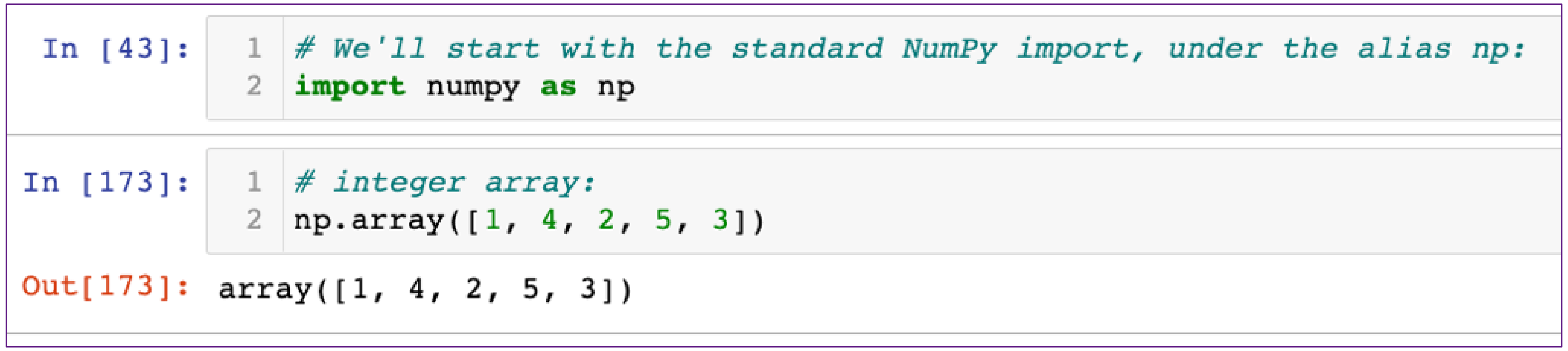
❑ Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas are built around the NumPy array.

❑ This section will present several examples of using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays.

❑ While the types of operations shown here may seem a bit dry and pedantic, they comprise the building blocks of many other examples used throughout the course. Get to know them well!

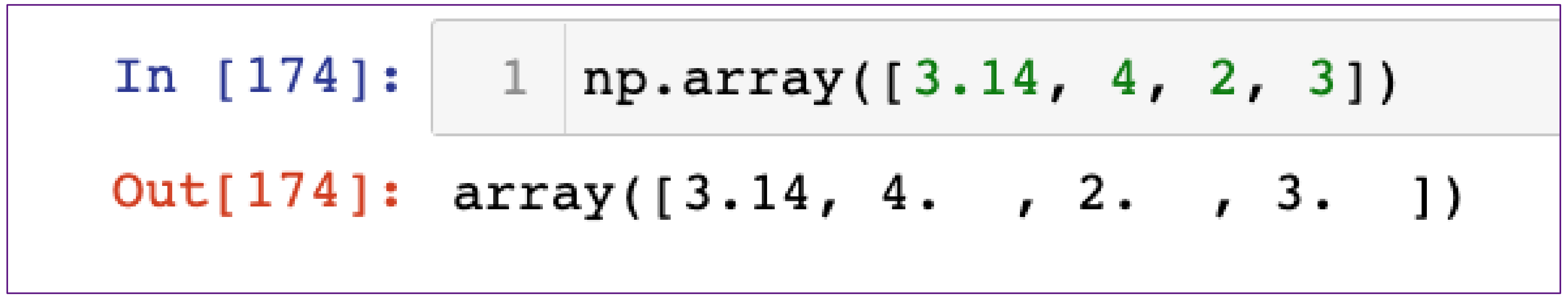
NumPy arrays

- Importing NumPy and creating an array:



```
In [43]:  # We'll start with the standard NumPy import, under the alias np:
          import numpy as np

In [173]: # integer array:
          np.array([1, 4, 2, 5, 3])

Out[173]: array([1, 4, 2, 5, 3])
```

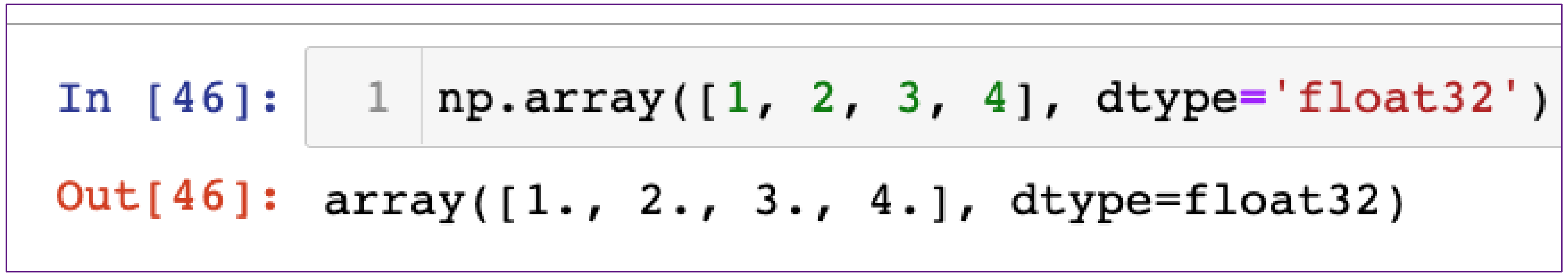
- Unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point). e.g.



```
In [174]:  np.array([3.14, 4, 2, 3])

Out[174]: array([3.14, 4.  , 2.  , 3.  ])
```

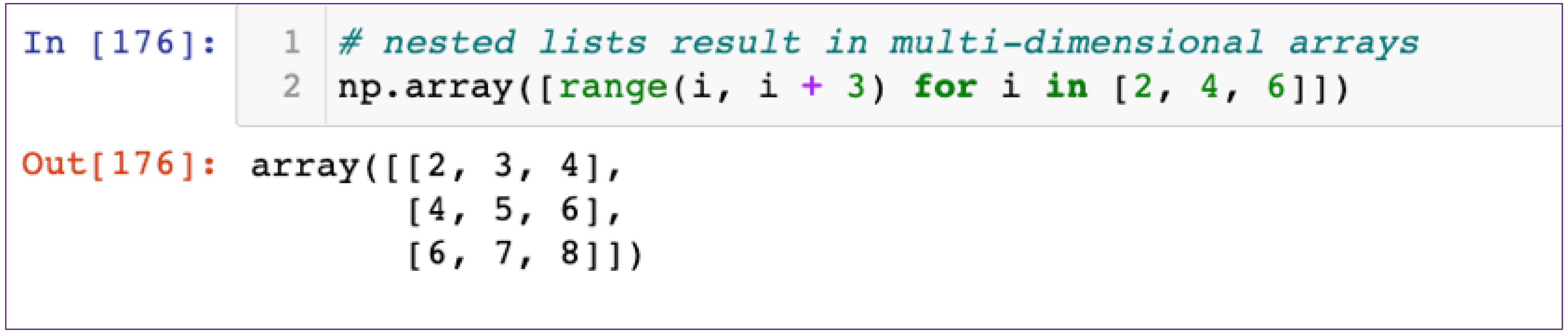
NumPy arrays

- If we want to explicitly set the data type of the resulting array, we can use the *dtype* keyword:



```
In [46]:  np.array([1, 2, 3, 4], dtype='float32')

Out[46]:  array([1., 2., 3., 4.], dtype=float32)
```

- Unlike Python lists, NumPy arrays can explicitly be multi-dimensional;



```
In [176]:  # nested lists result in multi-dimensional arrays
           np.array([range(i, i + 3) for i in [2, 4, 6]])

Out[176]:  array([[2, 3, 4],
                  [4, 5, 6],
                  [6, 7, 8]])
```

Demo: *`NumPy Array Basic`*

NumPy arrays pre-initializaton

- Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy., e.g.,
  - `np.zeros()`
  - `np.ones()`
  - `np.full()`
  - `np.arange()`
  - `np.linspace()`
  - `np.random.random()`
  - `np.random.normal()`
  - `np.random.randint()`
  - `np.eye()`
  - `np.empty`

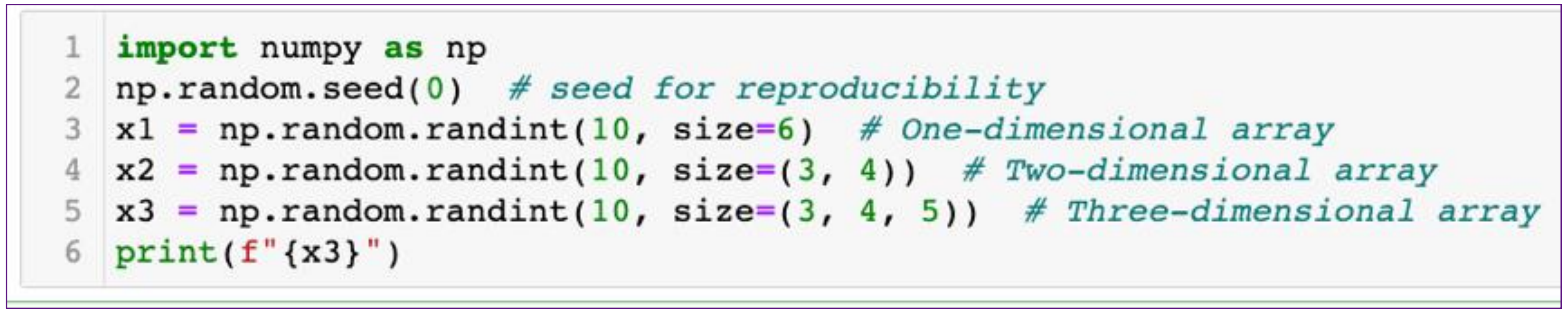
CRUNCH Group
Brown University

DEEP LEARNING INSTITUTE
BROWN

Demo: *`NumPy array pre-initialization`*

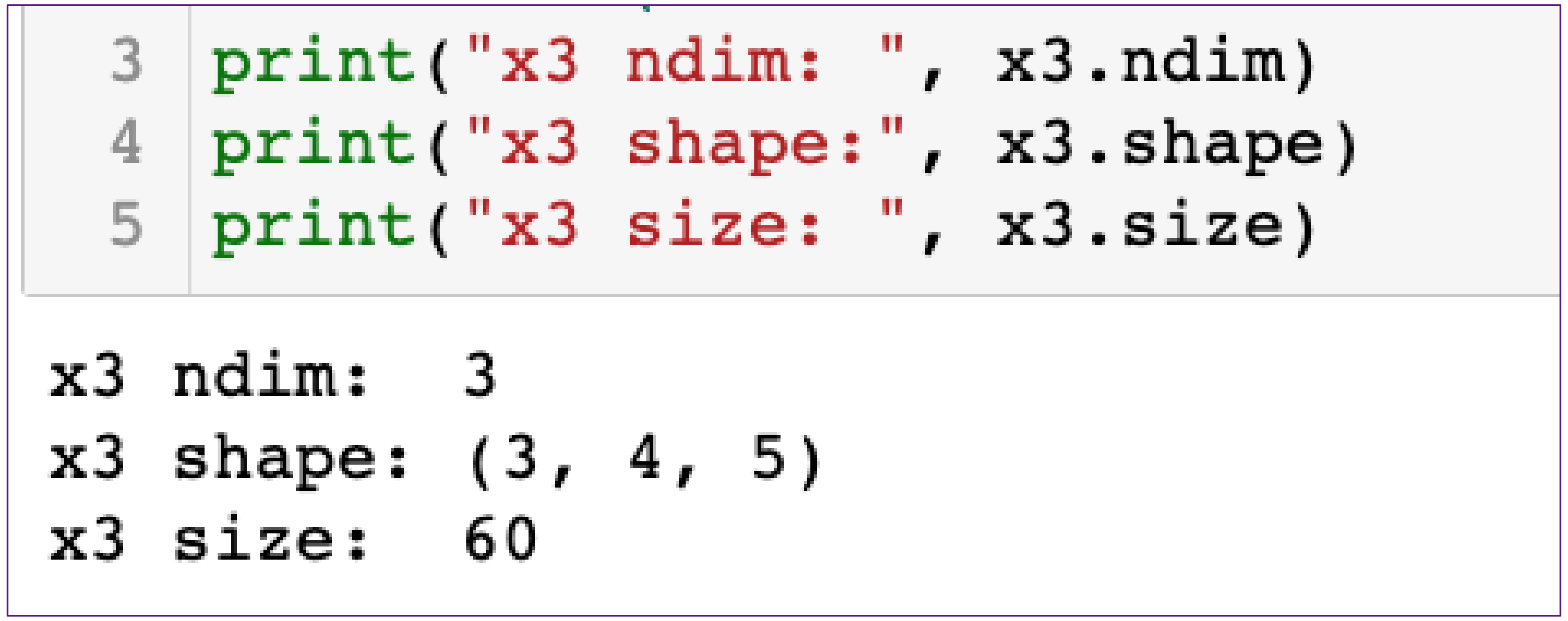
NumPy array attributes

- ❑ Generate Rank 1, 2 and 3 matrices by fixing the seed of pseudo random number generator:



```python
import numpy as np
np.random.seed(0)  # seed for reproducibility
x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
print(f"{x3}")
```

- ❑ `ndim`, `shape` and `size` of NumPy arrays:



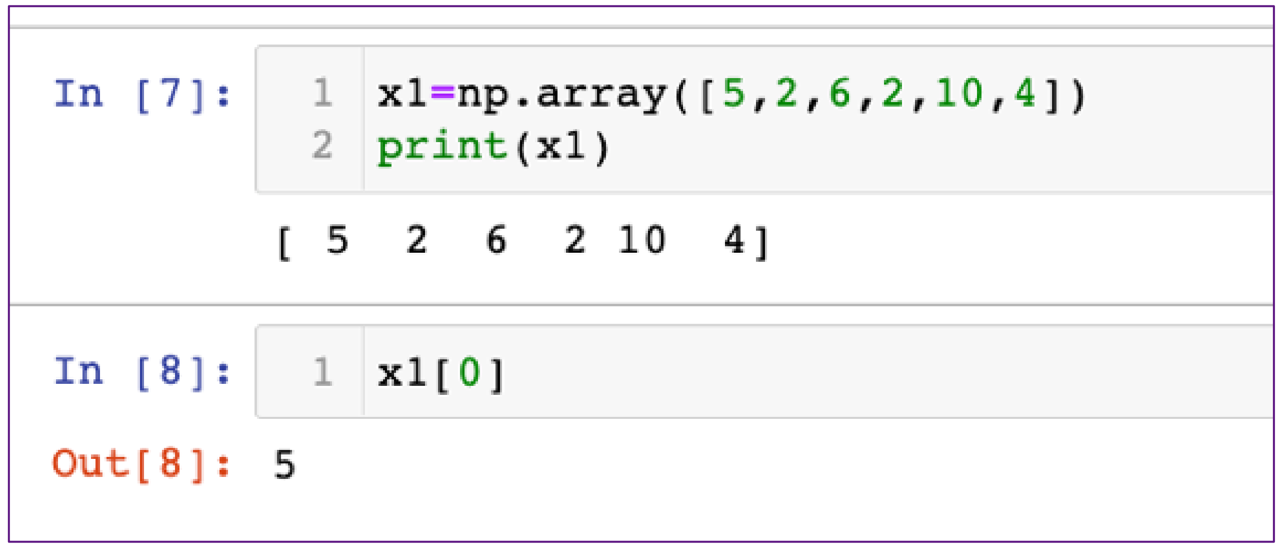
```python
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Demo: *`NumPy array attributes`*

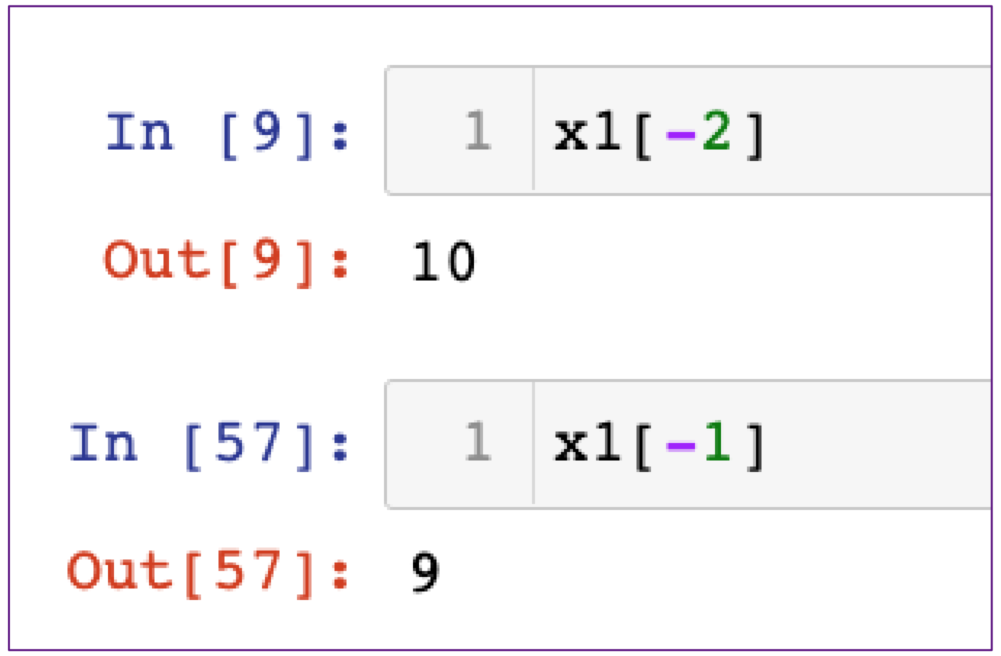
Indexing: accessing elements within an array

- ❑NumPy array indexing starts with 0 while moving forward



```
In [7]:  x1=np.array([5,2,6,2,10,4])
         print(x1)

         [ 5  2  6  2 10  4]

In [8]:  x1[0]

Out[8]:  5
```

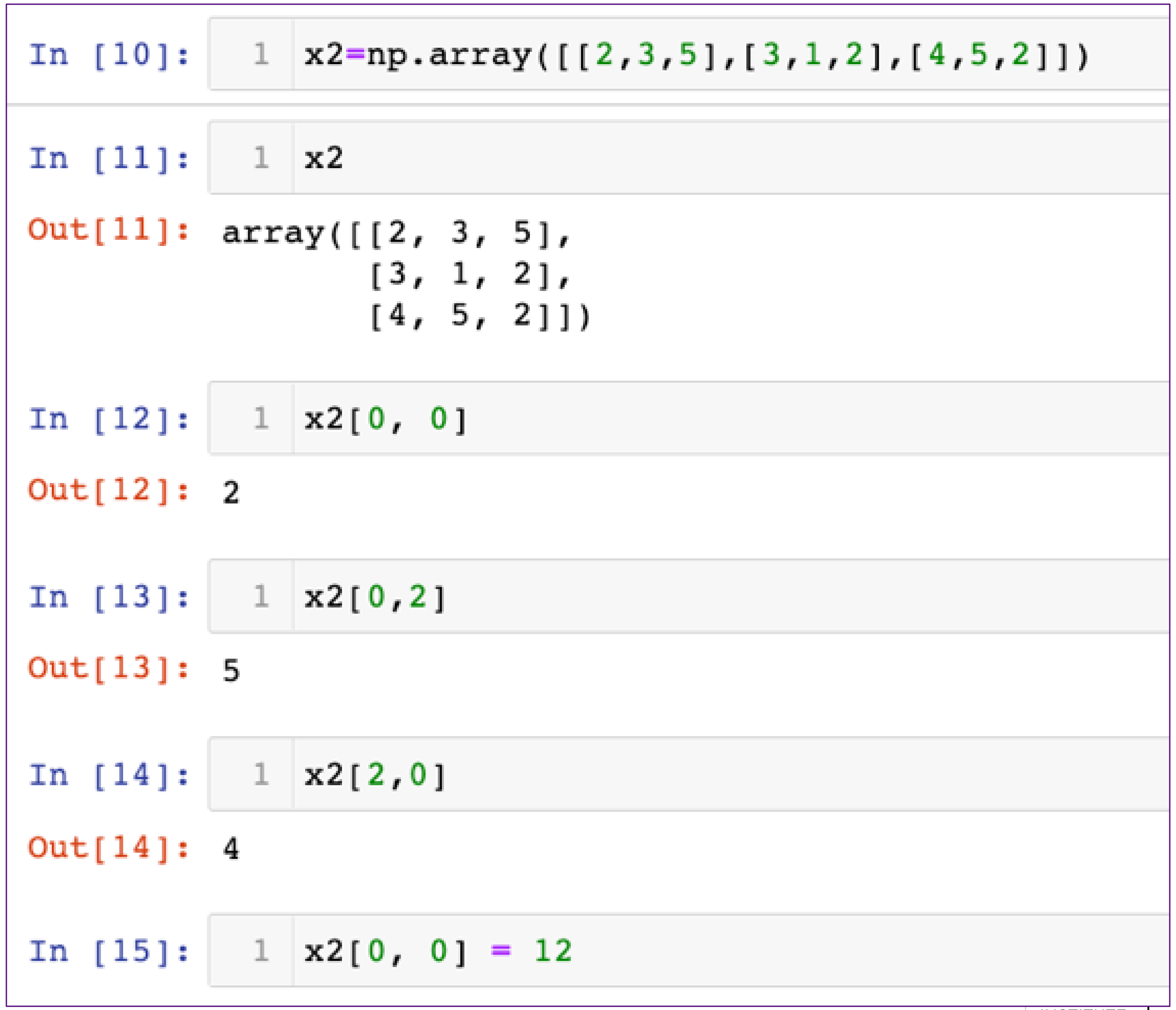
- ❑Reverse mapping:



```
In [9]:   x1[-2]

Out[9]:   10

In [57]:  x1[-1]

Out[57]:  9
```

Indexing: Accessing elements within an N-d Array

- Similar to 1-D array the elements of N-d array can be accessed by passing two indices



```
In [10]: x2=np.array([[2,3,5],[3,1,2],[4,5,2]])

In [11]: x2

Out[11]: array([[2, 3, 5],
                [3, 1, 2],
                [4, 5, 2]])

In [12]: x2[0, 0]

Out[12]: 2

In [13]: x2[0,2]

Out[13]: 5

In [14]: x2[2,0]

Out[14]: 4

In [15]: x2[0, 0] = 12
```

Demo: *`NumPy array indexing`*

NumPy array slicing

- ❑ Just as we can use square brackets to access individual array elements, we can also use them to access subarrays.

- ❑ The slice notation, marked by the colon (:) character.

- ❑ The NumPy slicing syntax follows that of the standard Python list.

- ❑ To access a slice of an array x, use this:

  `x[start:stop:step]`

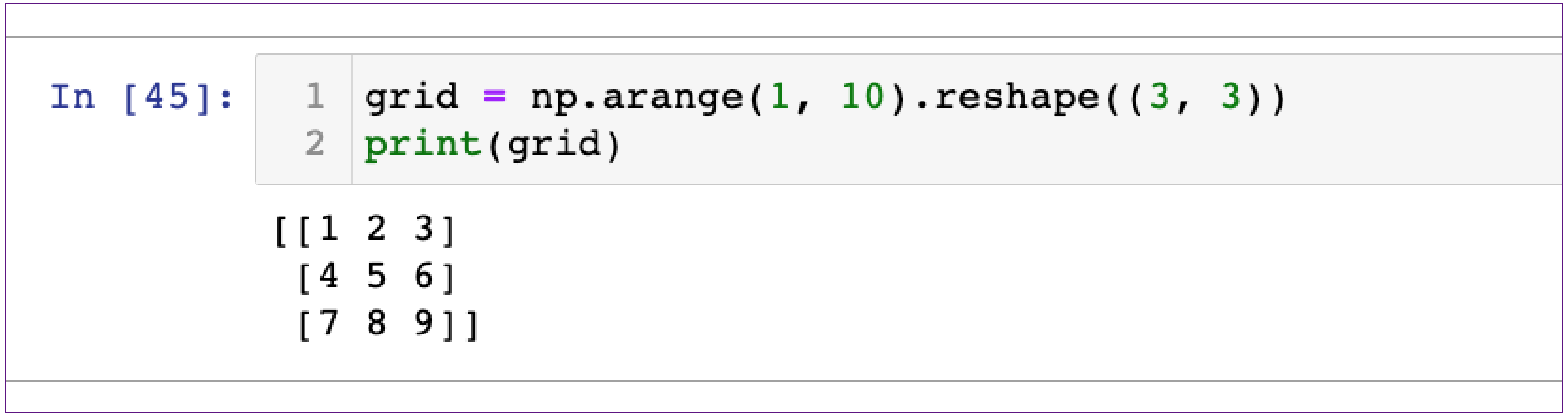
- ❑ If any of these are unspecified, they default to the values
  - `start=0, stop=size of dimension, step=1.`

- ❑ We will take a look at accessing sub-arrays in one dimension and in multiple dimensions.

Demo: NumPy array slicing

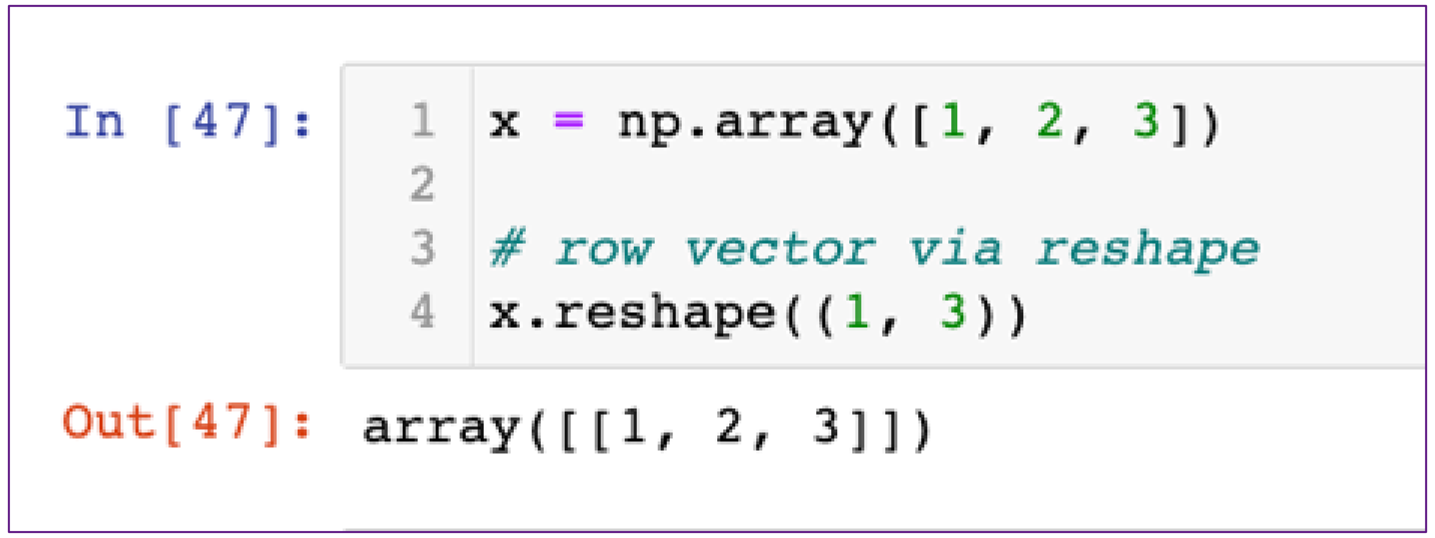
Reshaping NumPy arrays

- ❑ Reshaping a numpy array can be done by using *`reshape`* method.
- ❑ For example, if you want to put the numbers 1 through 9 in a 3 × 3 grid, you can do the following:



```
In [45]: grid = np.arange(1, 10).reshape((3, 3))
         print(grid)

[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

- ❑ Another method

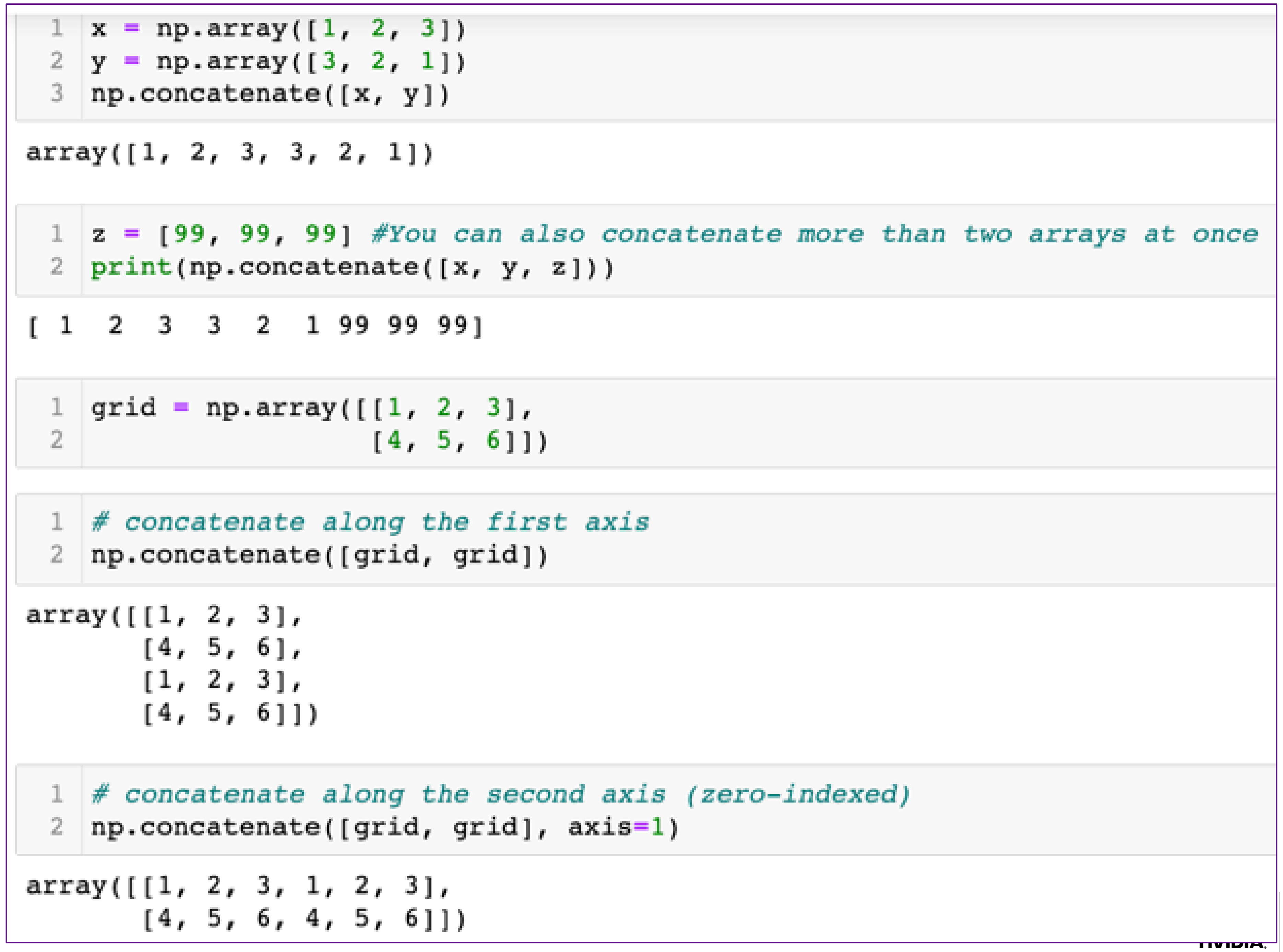


```
In [47]: x = np.array([1, 2, 3])

         # row vector via reshape
         x.reshape((1, 3))

Out[47]: array([[1, 2, 3]])
```

Demo: *`Reshaping NumPy arrays`*

Concatenation of NumPy arrays

- All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one.
- Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, e.g.,



```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

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

```
z = [99, 99, 99] #You can also concatenate more than two arrays at once
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

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

```
# concatenate along the first axis
np.concatenate([grid, grid])
```

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

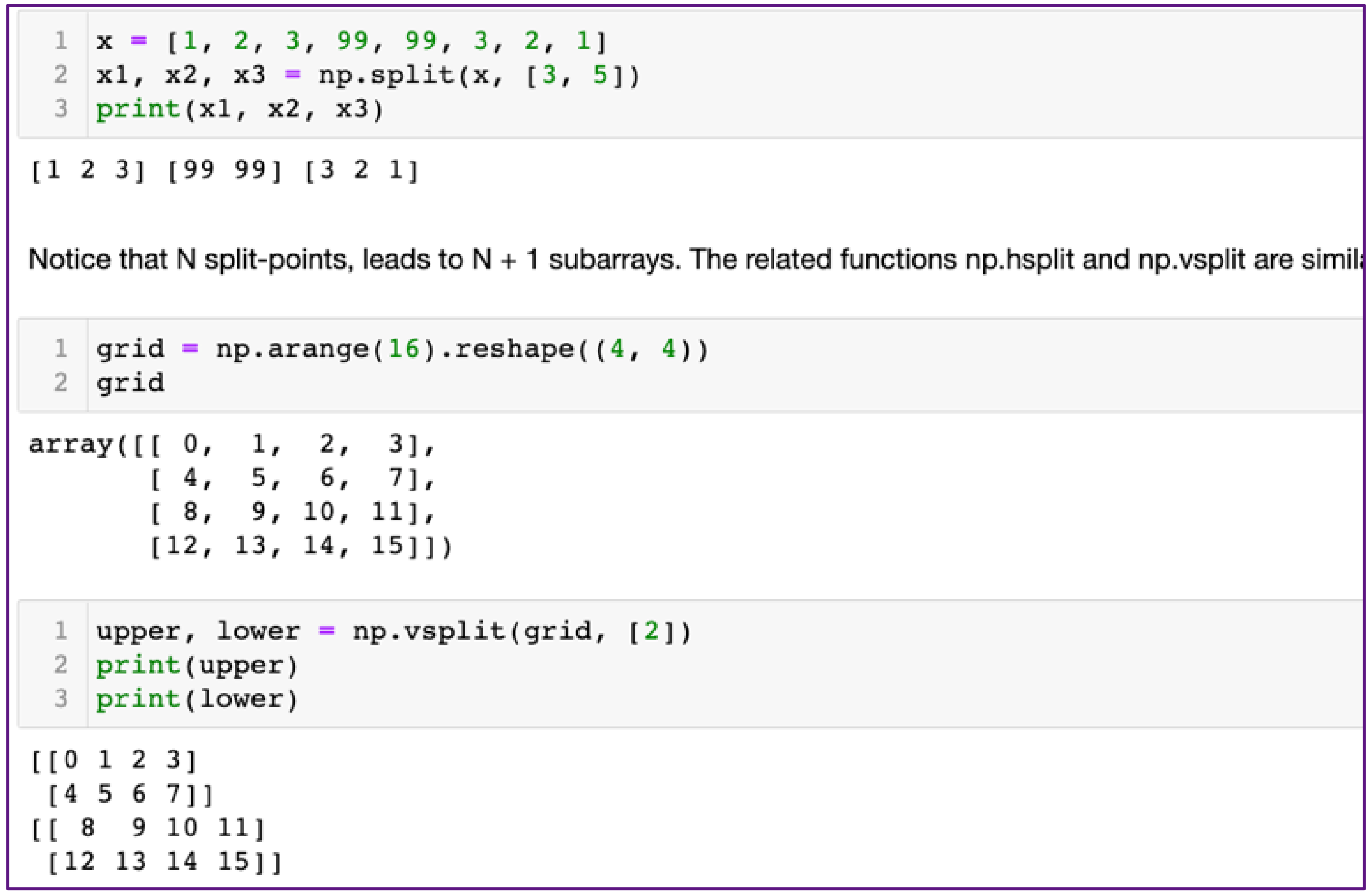
```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

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

Demo: *`Concatenation of NumPy arrays`*

Splitting of NumPy arrays

- We can also do the opposite of concatenation and split the array instead.
- This can be done by implementing functions *np.split*, *np.hsplit*, *np.vsplit*.
- Examples



```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

Notice that N split-points, leads to N + 1 subarrays. The related functions np.hsplit and np.vsplit are simil

```
grid = np.arange(16).reshape((4, 4))
grid
```

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

```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

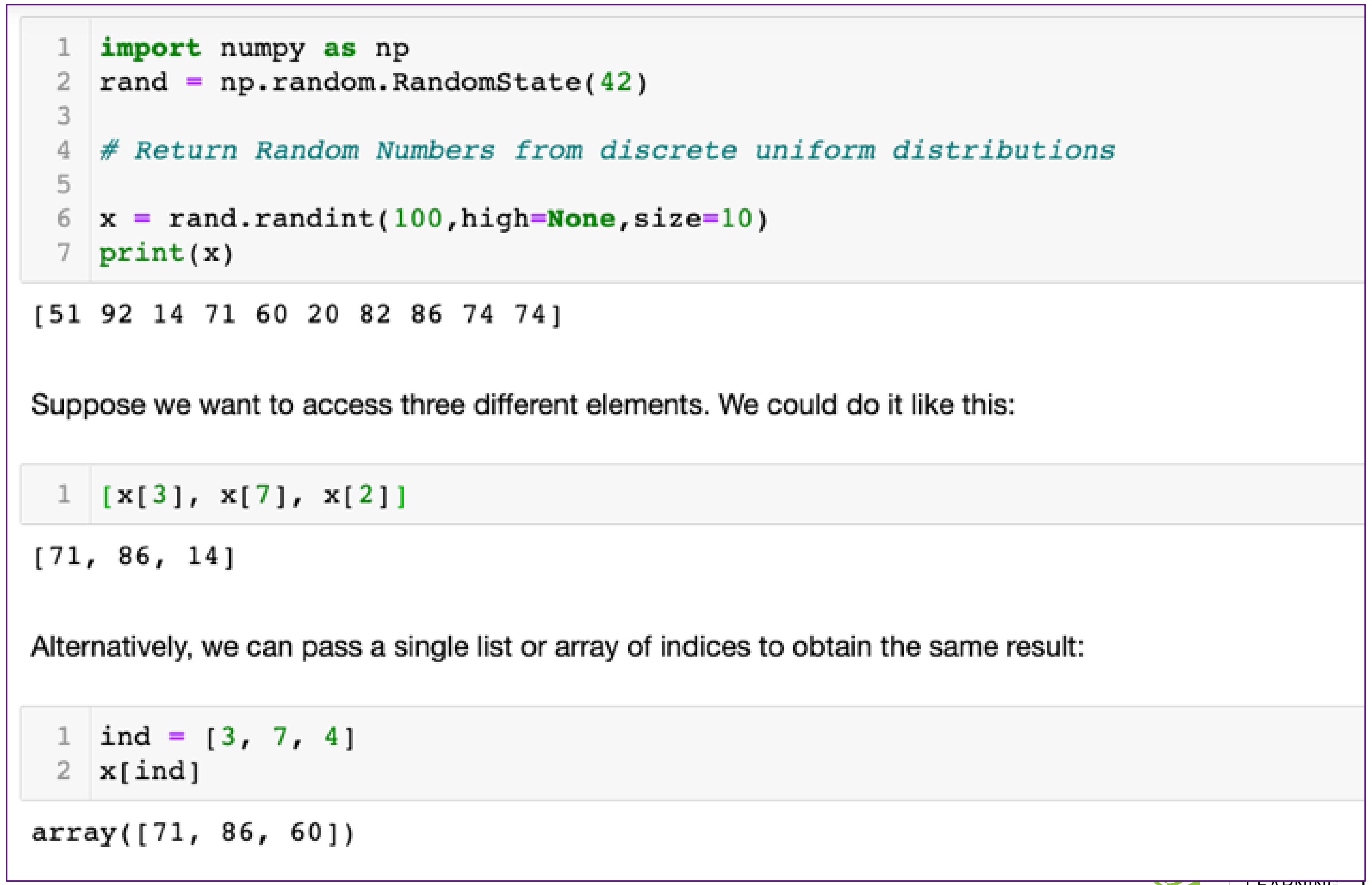
```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

Demo: *`Splitting of NumPy arrays`*

Fancy indexing of NumPy arrays

- ❑ Fancy indexing is conceptually simple; it means passing an array of indices to access multiple array elements at once.

- ❑ For example, we want to access three different elements of an array an



```python
import numpy as np
rand = np.random.RandomState(42)

# Return Random Numbers from discrete uniform distributions

x = rand.randint(100,high=None,size=10)
print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

Suppose we want to access three different elements. We could do it like this:

```python
[x[3], x[7], x[2]]
```

```
[71, 86, 14]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```python
ind = [3, 7, 4]
x[ind]
```

```
array([71, 86, 60])
```

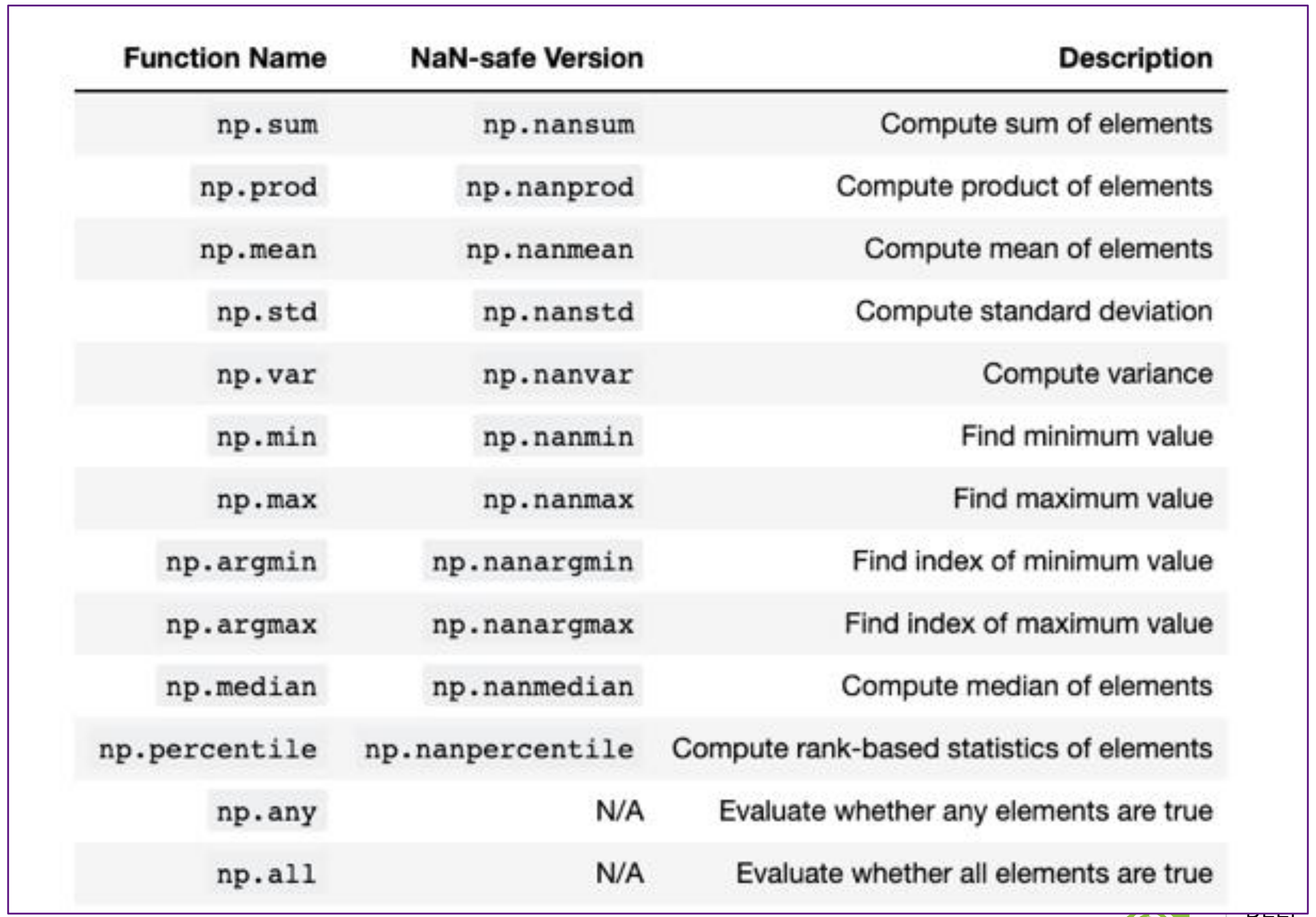
Demo: *`Fancy indexing of NumPy arrays`*

Aggregations on NumPy arrays

- ❑ Computing summary statistics of a large amount of data is usually the first step. Important summary statistics are mean and standard deviation but other aggregates are useful as well (sum, product, median, minimum, maximum, quantiles, etc.)

- ❑ NumPy has fast built-in aggregation functions for working with arrays

- ❑ NumPy Built In Aggregations

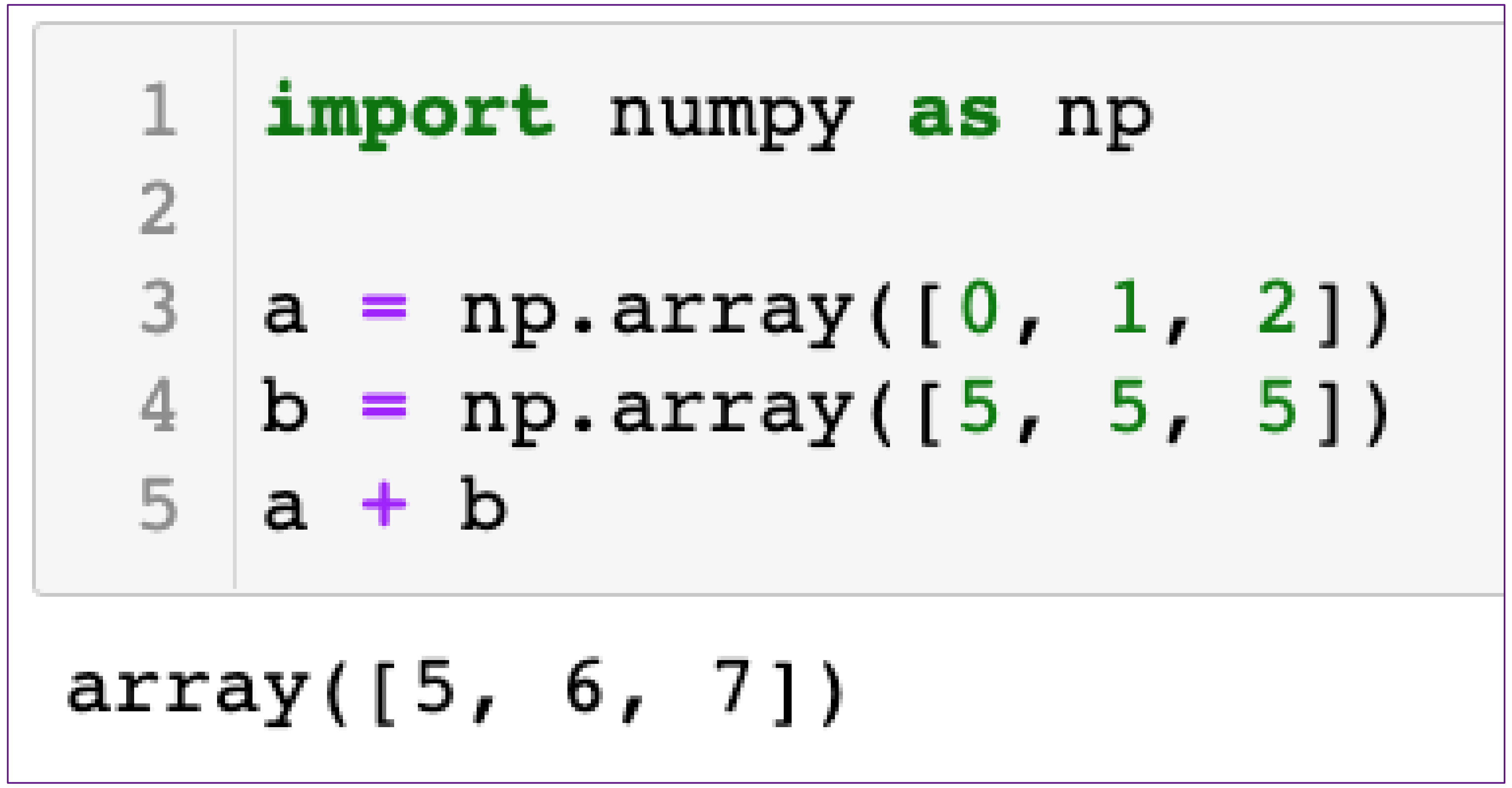


Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Demo: *`Aggregations of NumPy arrays`*

Broadcasting on NumPy arrays

- Broadcasting performs binary operations an element-by-element basis in vectorized function; e.g.,



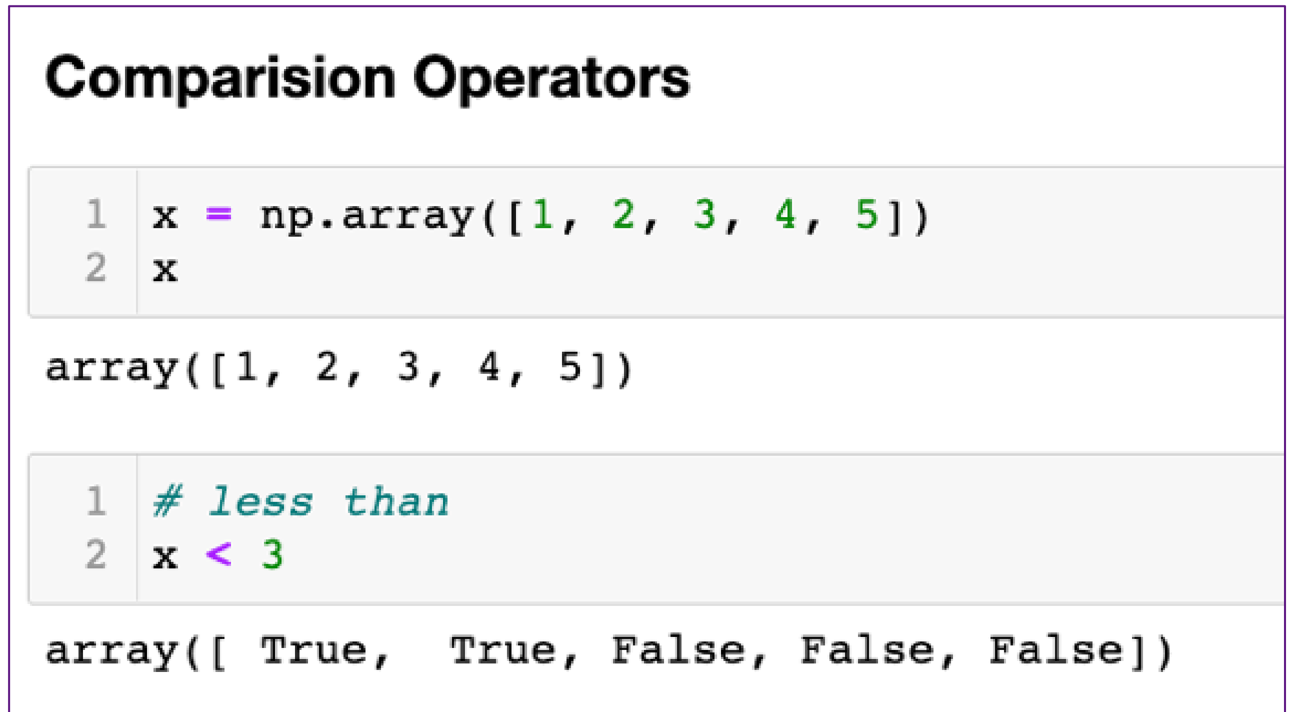
```python
import numpy as np

a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

```
array([5, 6, 7])
```

Demo: Broadcasting of NumPy arrays

Comparisons, Masks and Boolean Logic

- This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays.
- Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion.
- For example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold.
- In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.


Comparision Operators

```
x = np.array([1, 2, 3, 4, 5])
x
```

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

```
# less than
x < 3
```

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

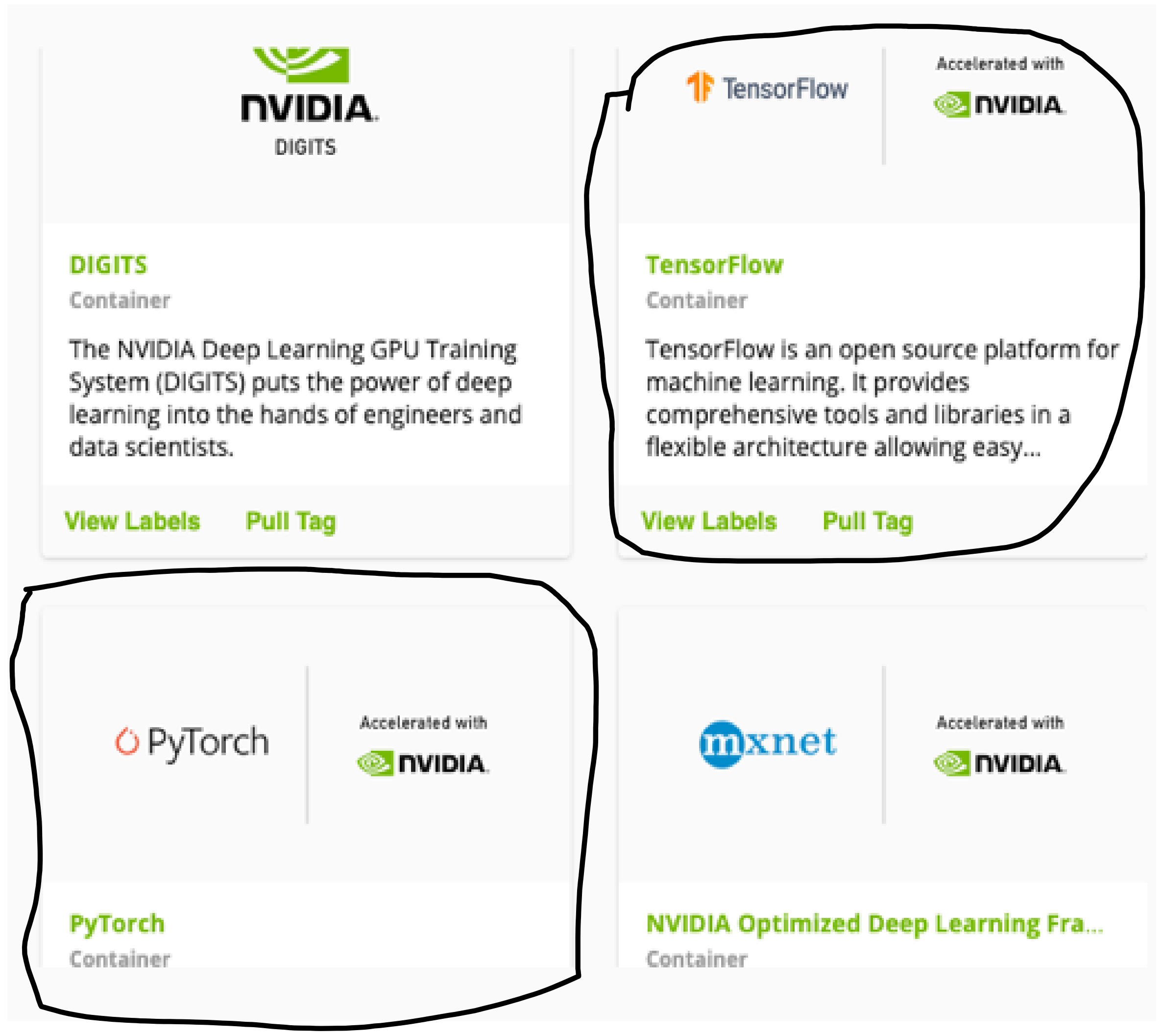
Demo: *Comparisons, Masks and Boolean Logic*

Nvidia Deep Learning Container

- The NGC catalog hosts containers for the top AI and data science software, tuned, tested and optimized by NVIDIA, as well as fully tested containers for HPC applications and data analytics.

- NGC catalog containers provide powerful and easy-to-deploy software proven to deliver the fastest results, allowing users to build solutions from a tested framework, with complete control.

- Containers are hosted on : [NGC Container]



DIGITS
Container
The NVIDIA Deep Learning GPU Training System (DIGITS) puts the power of deep learning into the hands of engineers and data scientists.
View Labels Pull Tag

TensorFlow
Container
TensorFlow is an open source platform for machine learning. It provides comprehensive tools and libraries in a flexible architecture allowing easy...
View Labels Pull Tag

PyTorch
Container

NVIDIA Optimized Deep Learning Fra...
Container

Installation of TensorFlow NGC Container

- ❑ The docker image of TF2 can be downloaded by entering the command on terminal

```
docker pull nvcr.io/nvidia/21.08-tf2-py3
```

- ❑ Build the image by entering the following command

```
singularity build crunch_tf2.simg
docker://nvcr.io/nvidia/tensorflow:21.03-tf1-py3
```

- ❑ Invoke the environment along with filesystem mount

```
singularity exec -B /lus:/lus --nv ContainerName bash
```

Installation of PyTorch NGC Container

- ❑ The docker image of TF2 can be downloaded by entering the command on terminal

```
docker pull nvcr.io/nvidia/pytorch:21.08-py3
```

- ❑ Build the image by entering the following command

```
singularity build crunch_pyt.simg docker:nvcr.io/nvidia/pytorch:21.08-py3
```

- ❑ Invoke the environment along with filesystem mount

```
singularity exec -B /lus:/lus --nv ContainerName bash
```

Demo: *`Installing the Nvidia Containers`*

jupyter

Summary

- ❑ Getting familiar with programming environment of the course
- ❑ Introduction of ***jupyter*** notebook and setting it up on your machine
- ❑ Basics of data structure and operation in NumpPy and SciPy
- ❑ Installation of deep learning frameworks TensorFlow and PyTorch
- ❑ Introduction to Nvidia's deep learning container and installation

Exercise-

- ❑ Write a python code for competing L_2-Norm of a vector of length n. Please check the accuracy of your computation using in built routine of *numpy* which is `np.linalg.norm()`

References

- Guttag JV. Introduction to computation and programming using Python: With application to understanding data. MIT Press; 2016 Aug 12.
- Altenkirch T, Triguero I. Conceptual Programming with Python. Lulu. com; 2020 Apr 8.

Deep Learning for Science and Engineering Teaching Kit

Thank You