Introduction to IPython

Reference:

- Learning IPython for Interactive Computing and Data Visualization, Second Edition, by Cyrille Rossant
- *Python for Data Analysis,* by Wes McKinney
- Python Data Science Handbook: Essential Tools for Working with Data, by Jake VanderPlas, 2016

What is it?

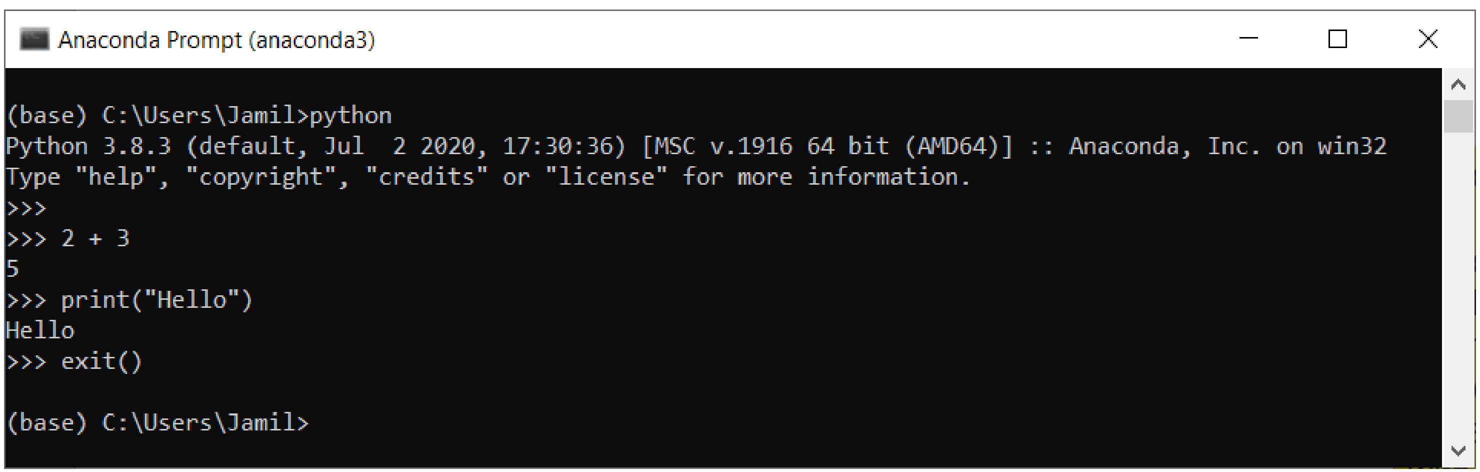
- IPython – Interactive Python
- Provides a convenient command-line interface to the scientific Python platform

Installing IPython

- Anaconda is a pre-packaged Python distribution that comes with many libraries for ML and data science
- Download from https://www.anaconda.com/download/
- We use Python 3

Starting Python

- On Windows, open Anaconda from the Start Menu and select **Anaconda Prompt**
- Type **python**
- Notice how prompt changes to >>>
- Type **exit()** to exit Python



Starting the IPython Console

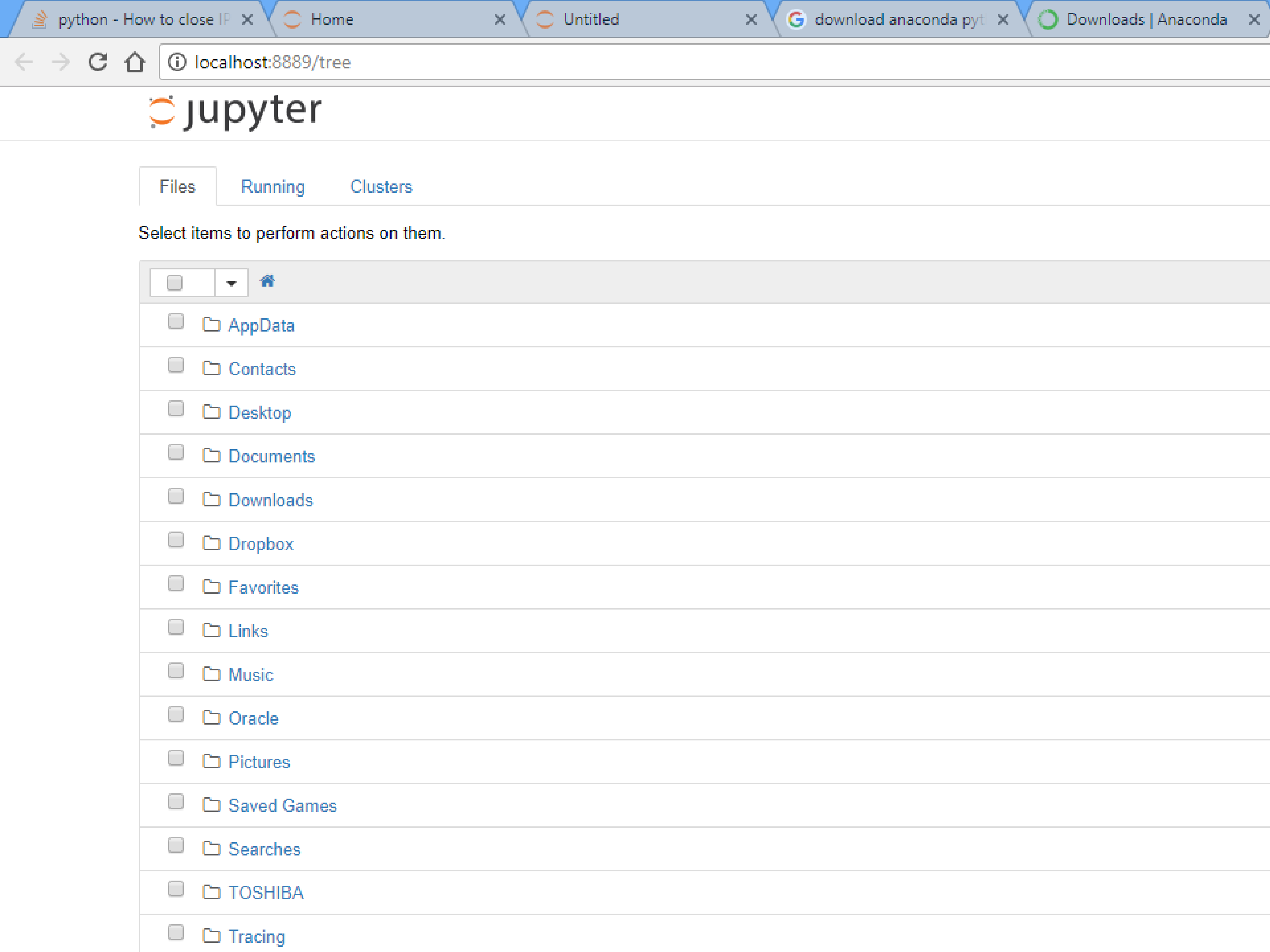
- At the command prompt, type **ipython**
- Notice the prompt
- Type **exit** to exit ipython

Anaconda Prompt (anaconda3)

```
(base) C:\Users\Jamil>ipython
Python 3.8.3 (default, Jul  2 2020, 17:30:36) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.16.1 -- An enhanced Interactive Python. Type '?' for help.

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

In [2]: print("Hello")
Hello

In [3]: exit

(base) C:\Users\Jamil>
```

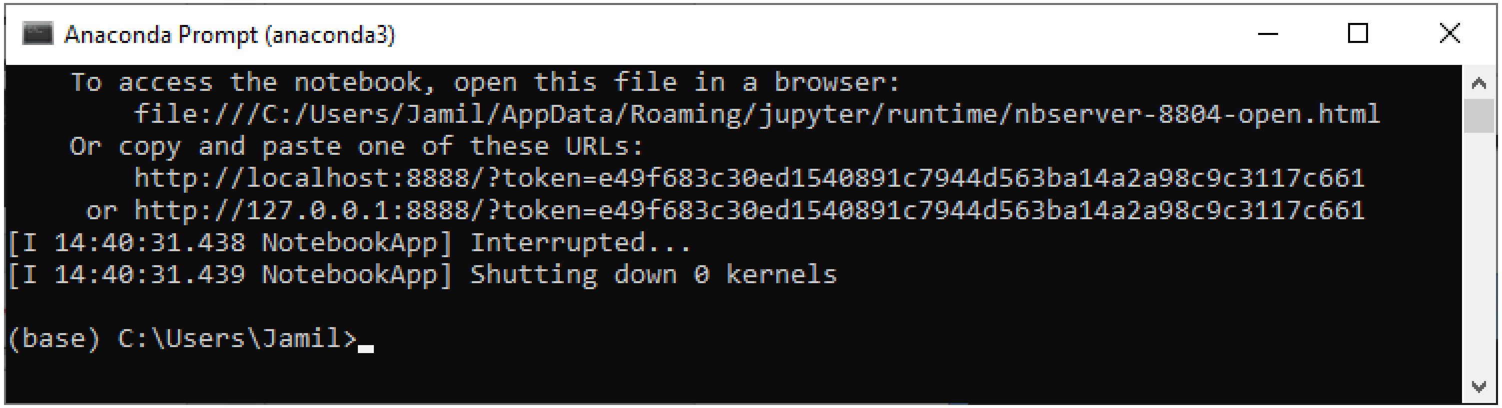
Jupyter Notebook

- It comes with Anaconda ipython
- To start, at the command prompt type jupyter notebook
- This starts the Jupyter server and opens a new window in the browser at address localhost:8888


python - How to close IP
Home
Untitled
download anaconda pyt
Downloads | Anaconda
localhost:8889/tree
jupyter
Files
Running
Clusters
Select items to perform actions on them.
AppData
Contacts
Desktop
Documents
Downloads
Dropbox
Favorites
Links
Music
Oracle
Pictures
Saved Games
Searches
TOSHIBA
Tracing

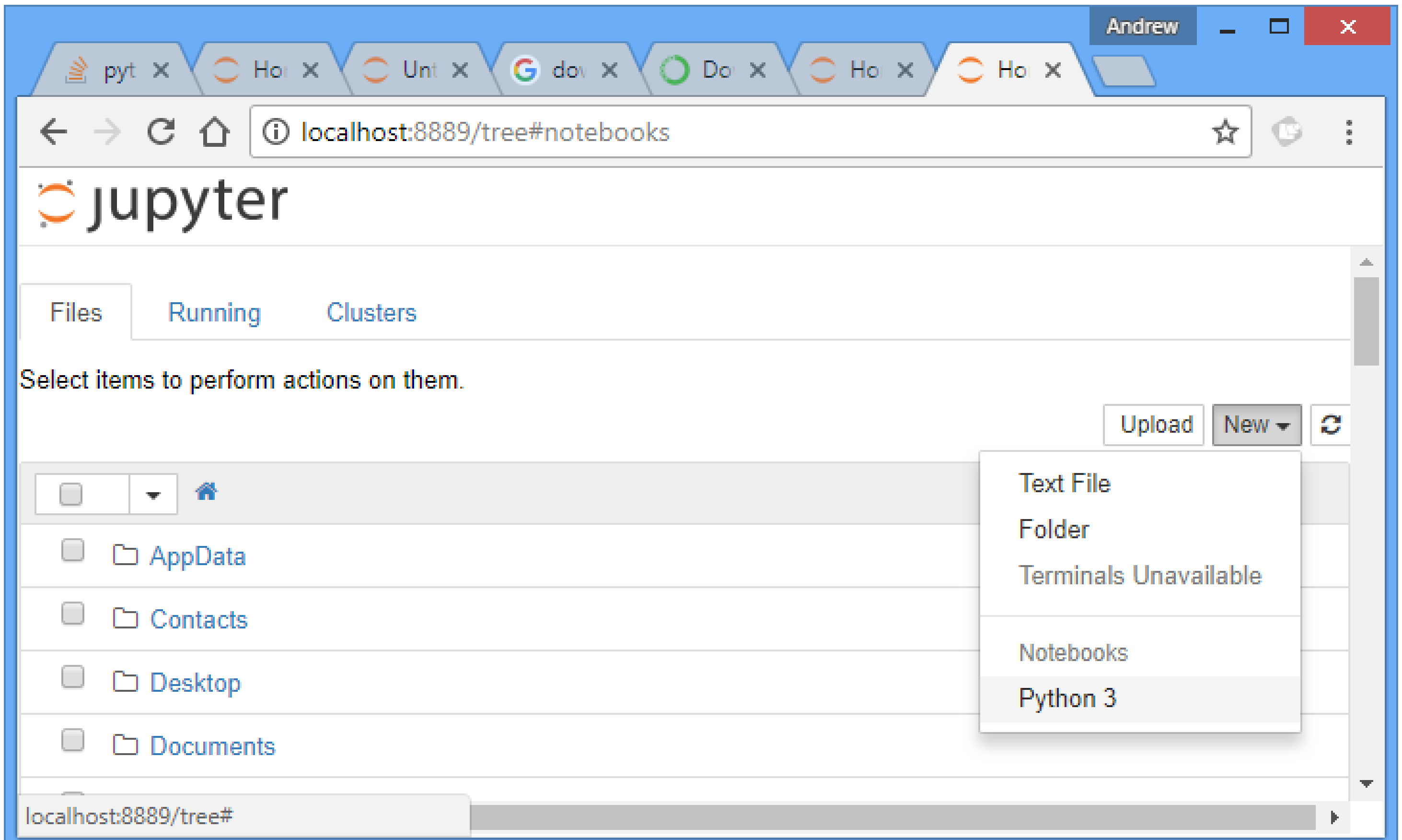
Closing the Notebook Server

- To close the notebook server, at the terminal window press **Ctrl + C**



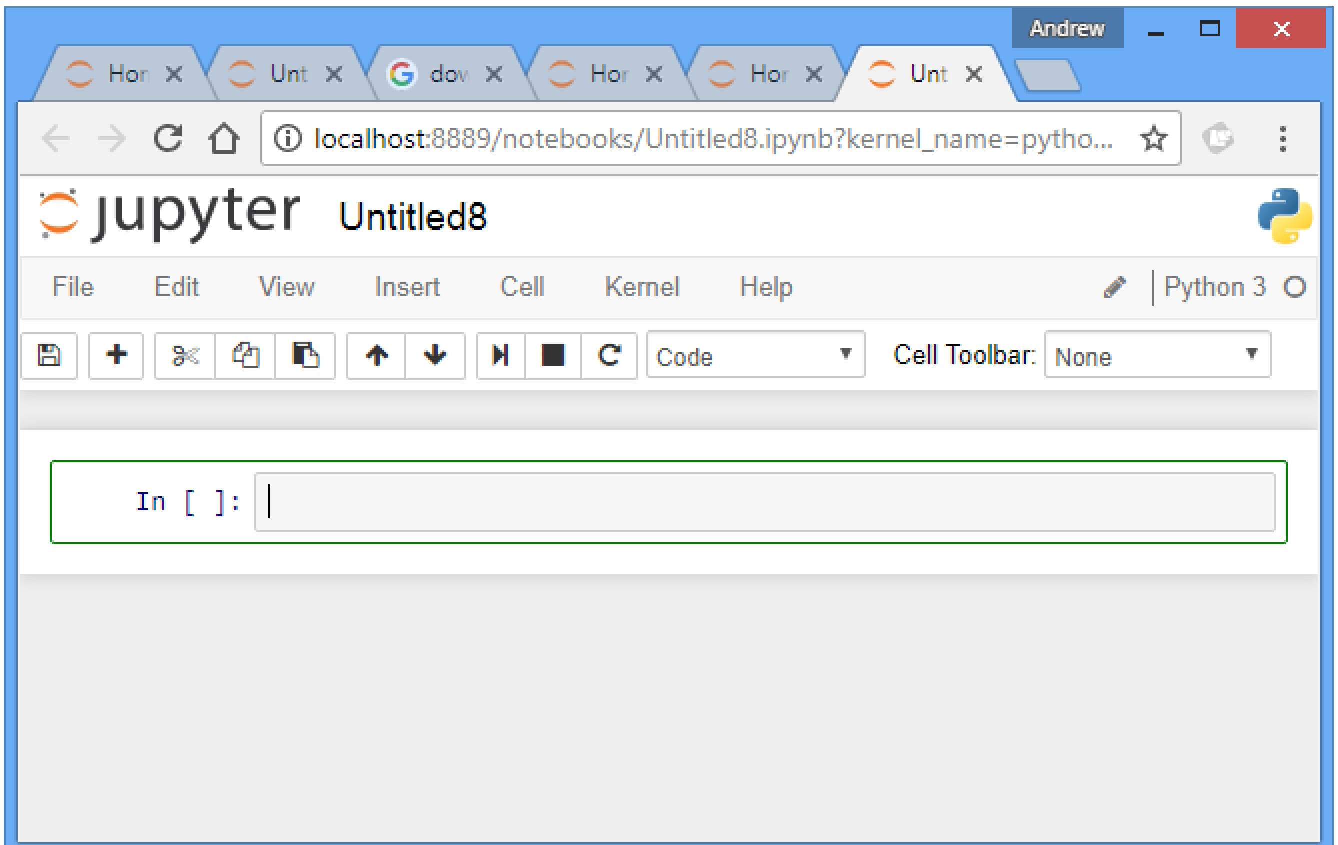
Starting a new notebook

- In the **Files** tab, click **New**, and click Notebooks **Python 3**



Starting a new notebook (cont)

- This opens a new browser tab with the new notebook as follows



Types of Cells

- A notebook consists of a linear list of cells
- There are two main types of cells:
- **Markdown cells**: contain rich text that you can format and can contain images, HTML code, Latex math equations, and more
  - Ex: next slide
- **Code cells**: contain code to be executed by ipython
  - Ex: slide after next

Markdown Cell

````
### New paragraph

This is *rich* **text** with [links](http://ipython.org), equations:

$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x)\, \mathrm{e}^{-i \xi x} dx$$

code with syntax highlighting:

```python
print("Hello world!")
```

and images:

![This is an image](http://jupyter.org/images/jupyter-sq-text.svg)
````

New paragraph

This is *rich* **text** with links, equations:

$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x)\, \mathrm{e}^{-i \xi x} dx$$

code with syntax highlighting:

```python
print("Hello world!")
```

and images:

IP[y]: IPython
Interactive Computing

Code Cell

In [1]:
Prompt number

Input area

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
print("Hello world!")
plt.imshow(np.random.rand(20, 20), interpolation='none');
from IPython.display import display_html
from IPython.html.widgets import FloatSlider
display_html('<table><tr><td>some</td><td>table</td></tr></table>', raw=True)
FloatSlider(value=70)
```

Widget area

70

Output area

```
Hello world!
```

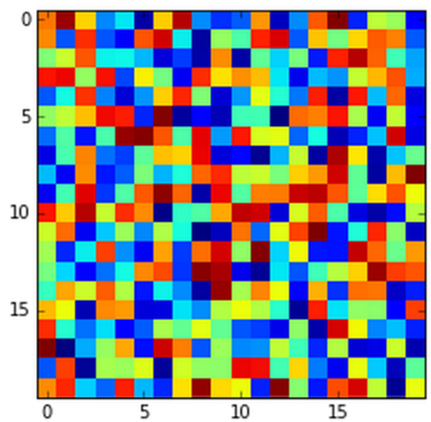
Standard output

Error output

```
:0: FutureWarning: IPython widgets are experimental and may change in the future.
```

some	table

Rich output



Magic commands

- Magic commands allow us to interact with the file system
- Magic commands start with %

```
In [1]:%pwd
Out[1]: 'c:\\TEACHING\\Summer\\17\\minibook-2nd\\chapter1‘
In [2]: %cd chapter1\facebook
Out[2]: c:\TEACHING\Summer\17\minibook-2nd\chapter1\facebook
In [3]: %ls
```

Running Python Scripts from IPython

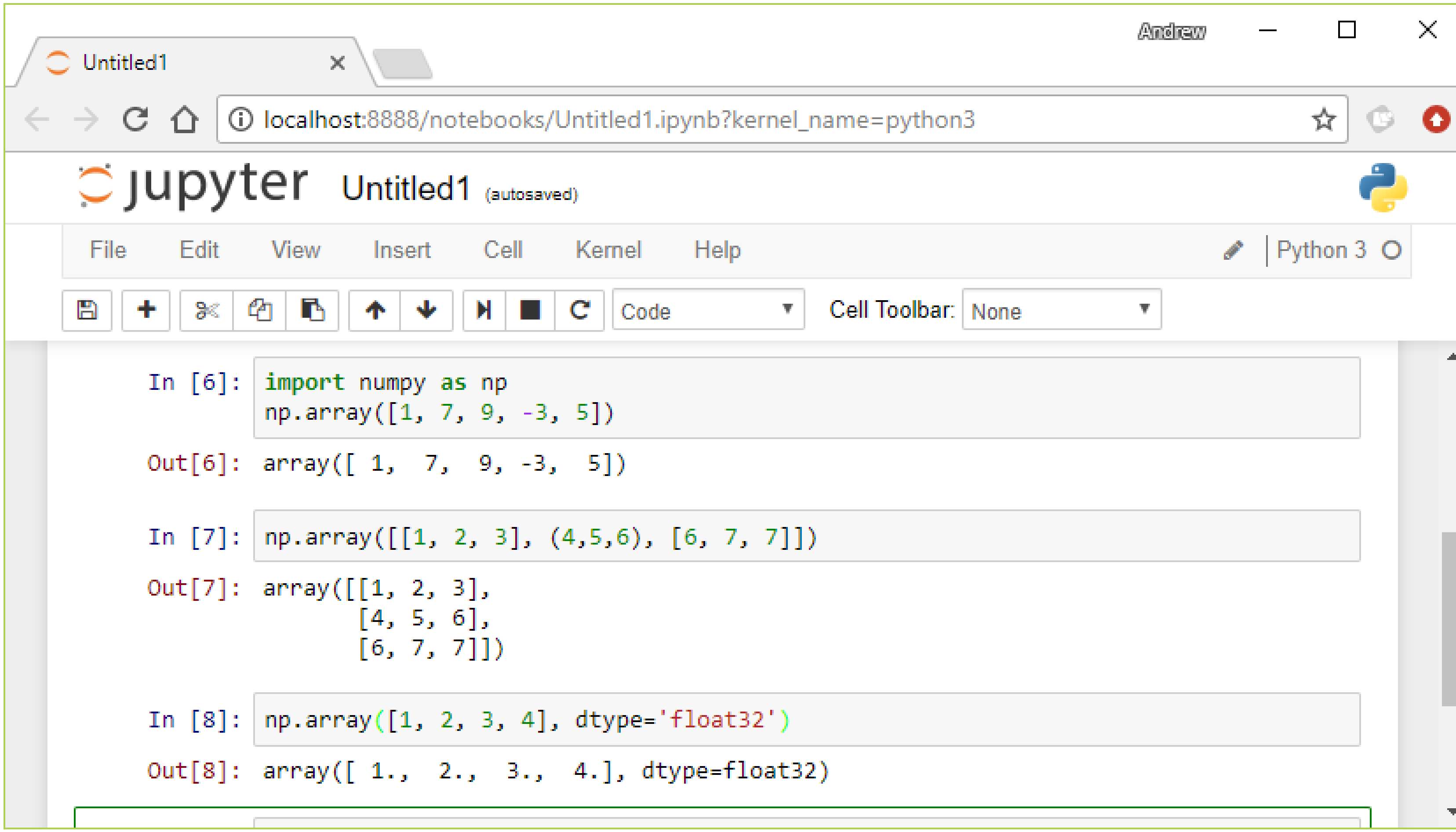
- Suppose we have a program saved in the file factorial.py that is located in the current directory. You can execute the file using any of these commands

```
In [3]: !ipython factorial.py
120
 3628800
In [4]: !python factorial.py
120
3628800
In [5]:%run factorial.py
120
3628800
```

NumPy

- NumPy stands for Numerical Python
- NumPy provides a multidimensional array data structure (ndarray)
- NumPy arrays are more efficient than Python's lists
- Many operations are performed on an element-wise basis
  - called vector (or vectorized) operations

Creating Arrays



Untitled1

localhost:8888/notebooks/Untitled1.ipynb?kernel_name=python3

jupyter Untitled1 (unsaved changes)

File Edit View Insert Cell Kernel Help Connecting to kernel Python 3

Code Cell Toolbar: None

```
In [9]:  # Create a length-10 integer array filled with zeros
         np.zeros(10, dtype=int)

Out[9]:  array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [11]: np.zeros(10)

Out[11]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [12]: # Create a 3x5 floating-point array filled with ones
         np.ones((3, 5), dtype=float)

Out[12]: array([[ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.]])

In [13]: # Create a 3x5 array filled with 125.77
         np.full((3, 5), 125.77)

Out[13]: array([[ 125.77,  125.77,  125.77,  125.77,  125.77],
                [ 125.77,  125.77,  125.77,  125.77,  125.77],
                [ 125.77,  125.77,  125.77,  125.77,  125.77]])

In [14]: # Create an array filled with a linear sequence
         # Starting at 0, ending at 10, stepping by 2
         np.arange(0, 10, 2)

Out[14]: array([0, 2, 4, 6, 8])

In [15]: # Create an array of five values evenly spaced between 0 and 1
         np.linspace(0, 1, 5)

Out[15]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

Untitled

localhost:8889/notebooks/PythonDataScienceHandbook-master/notebooks/Untitle...

jupyter Untitled (unsaved changes)

File Edit View Insert Cell Kernel Help

Python 3

```
In [21]: np.random.random()

Out[21]: 0.18530580153107212

In [22]: np.random.random(size = 3)

Out[22]: array([ 0.26881064,  0.3728396 ,  0.94881514])

In [23]: np.random.random(size = (3,2))

Out[23]: array([[ 0.57267745,  0.97531188],
               [ 0.28060232,  0.9882659 ],
               [ 0.5138538 ,  0.1220976 ]])

In [24]: np.random.uniform(-5, 10)

Out[24]: 2.7762722806418028

In [25]: np.random.uniform(-2, 5, size = 10)

Out[25]: array([ 4.4639939 ,  2.97979323, -1.32641007,  1.753492  , -0.45617578,
                4.67564608, -1.10755548, -0.72563822, -0.71448746,  0.4445331 ])

In [26]: np.random.uniform(-2, 5, size = (2, 3) )

Out[26]: array([[ 3.97407648, -0.21518191, -1.73965714],
               [ 4.89360938,  4.70803748,  2.18301338]])

In [27]: np.random.uniform(-2, 5, (2, 3) )

Out[27]: array([[ 1.92280427,  1.14527645, -1.04494027],
               [ 1.92405513,  3.60977868,  0.78674202]])
```

np.random.randint(low, high = None, size = None)
generates random integers from [low, high)
If high is None (the default), then results are from [0, low)

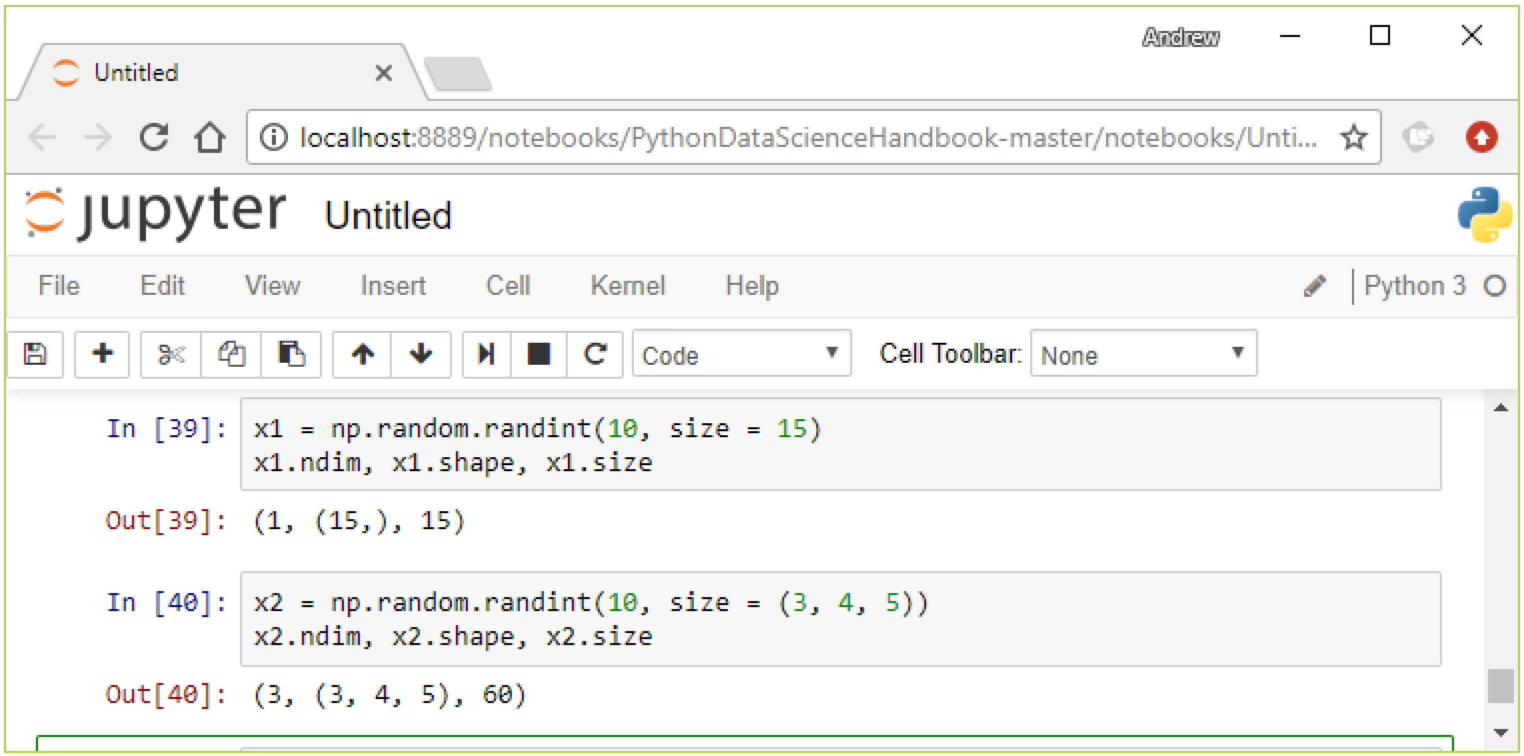
```
In [36]: np.random.randint(1, 5)

Out[36]: 3

In [37]: np.random.randint(1, 5, size = (3, 4))

Out[37]: array([[4, 2, 4, 4],
                [1, 2, 2, 4],
                [3, 3, 1, 1]])

In [38]: np.random.randint(5, size = 10)

Out[38]: array([0, 2, 3, 2, 2, 2, 3, 3, 0, 1])
```

NumPy Standard Data Types

Data type	**Description**
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

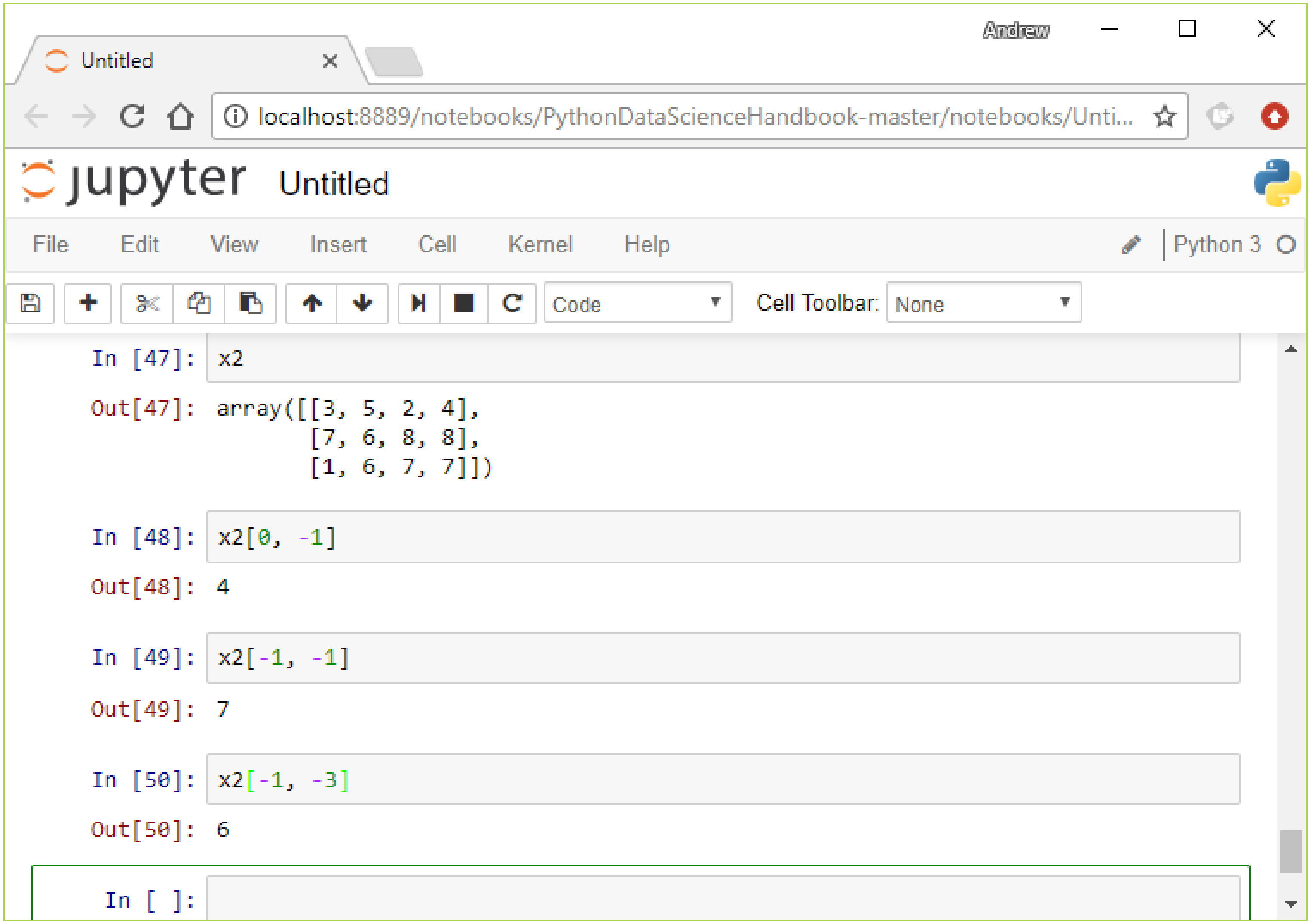
NumPy Array Attributes

- ndim: number of dimensions
- shape: the number of elements in each dimension
- size: the total number of elements in the array



Array Indexing

```
In [41]: 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
         x1

Out[41]: array([5, 0, 3, 3, 7, 9])

In [42]: x2

Out[42]: array([[3, 5, 2, 4],
                [7, 6, 8, 8],
                [1, 6, 7, 7]])

In [43]: x1[0]

Out[43]: 5

In [44]: x1[-1]

Out[44]: 9

In [45]: x2[1, 2]

Out[45]: 8

In [46]: x2[(1, 2)]

Out[46]: 8
```


Untitled
Andrew
localhost:8889/notebooks/PythonDataScienceHandbook-master/notebooks/Unti...
jupyter Untitled
File Edit View Insert Cell Kernel Help
Python 3
Code
Cell Toolbar: None
In [47]: x2
Out[47]: array([[3, 5, 2, 4],
[7, 6, 8, 8],
[1, 6, 7, 7]])
In [48]: x2[0, -1]
Out[48]: 4
In [49]: x2[-1, -1]
Out[49]: 7
In [50]: x2[-1, -3]
Out[50]: 6
In [ ]:

Array Slicing: Accessing Subarrays

Code

Cell Toolbar: None

```
In [51]: x = np.arange(10)
         x

Out[51]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [52]: x[:5]

Out[52]: array([0, 1, 2, 3, 4])

In [53]: x[5:]

Out[53]: array([5, 6, 7, 8, 9])

In [54]: x[3:7]

Out[54]: array([3, 4, 5, 6])

In [55]: x[::2]

Out[55]: array([0, 2, 4, 6, 8])

In [56]: x[::-1]

Out[56]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

Multidimensional Arrays

jupyter Untitled

File Edit View Insert Cell Kernel Help Python 3

Cell Toolbar: None

```
In [64]: x2 = np.arange(12).reshape(3,4)
         x2

Out[64]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

In [65]: x2[0:2, 1:3]

Out[65]: array([[1, 2],
                [5, 6]])

In [66]: x2[2,:]

Out[66]: array([ 8,  9, 10, 11])

In [67]: x2[:,2]

Out[67]: array([ 2,  6, 10])

In [68]: x2[2]

Out[68]: array([ 8,  9, 10, 11])
```

Subarrays as no-copy views: Unlike in Standard Python, array slices return views rather than copies of the array data

Code

Cell Toolbar: None

```
In [72]: x2

Out[72]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

In [73]: x2_sub = x2[:,1:3]
         x2_sub

Out[73]: array([[ 1,  2],
                [ 5,  6],
                [ 9, 10]])

In [74]: x2_sub[0,0] = -125
         x2_sub

Out[74]: array([[-125,    2],
                [   5,    6],
                [   9,   10]])

In [75]: x2

Out[75]: array([[   0, -125,    2,    3],
                [   4,    5,    6,    7],
                [   8,    9,   10,   11]])
```

Creating copies of arrays: use the copy() method

```
In [76]: x2

Out[76]: array([[   0, -125,    2,    3],
                [   4,    5,    6,    7],
                [   8,    9,   10,   11]])

In [77]: x2_sub = x2[:,1:3].copy()

In [78]: x2_sub

Out[78]: array([[-125,    2],
                [   5,    6],
                [   9,   10]])

In [79]: x2_sub[0, 0] = 10
         x2_sub

Out[79]: array([[10,  2],
                [ 5,  6],
                [ 9, 10]])

In [80]: x2

Out[80]: array([[   0, -125,    2,    3],
                [   4,    5,    6,    7],
                [   8,    9,   10,   11]])
```

Reshaping of Arrays

Cell Toolbar: None

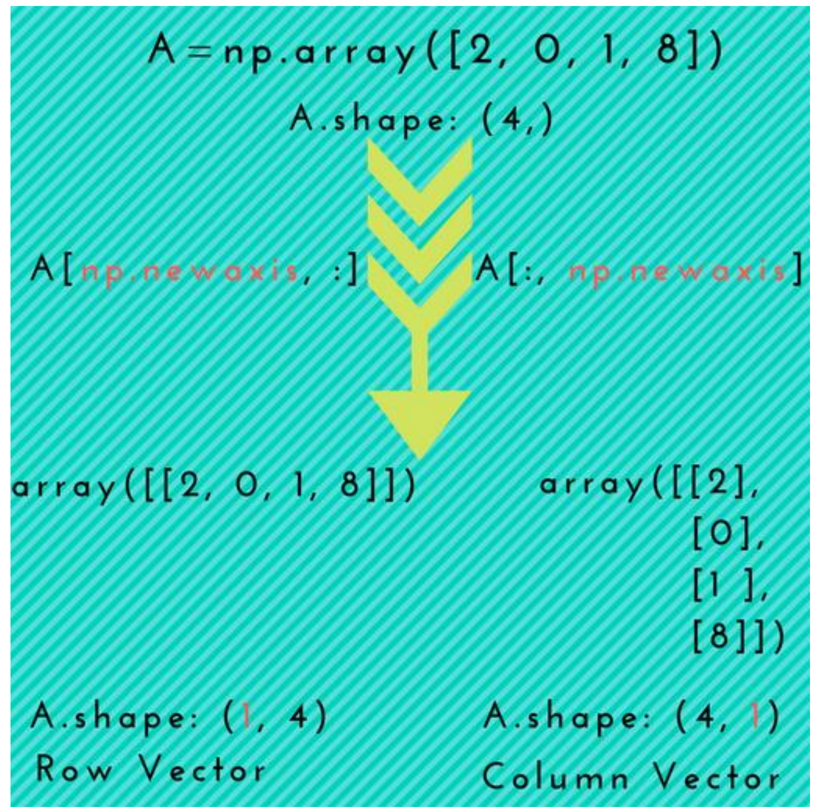
```
In [85]: x = np.arange(1,13).reshape(3,4)
         x

Out[85]: array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12]])

In [86]: vector = np.arange(1,6)
         vector

Out[86]: array([1, 2, 3, 4, 5])

In [87]: row_vector = vector.reshape(1,5)
         row_vector

Out[87]: array([[1, 2, 3, 4, 5]])

In [88]: row_vector.shape

Out[88]: (1, 5)

In [89]: column_vector = vector.reshape(5,1)
         column_vector

Out[89]: array([[1],
                [2],
                [3],
                [4],
                [5]])

In [90]: column_vector.shape

Out[90]: (5, 1)
```

Does reshape create a view or a copy of the original array?

```
In [25]: y = np.arange(1, 5)
         y.reshape(2,2)
         y

Out[25]: array([1, 2, 3, 4])

In [26]: x = y.reshape(2,2)
         x

Out[26]: array([[1, 2],
                [3, 4]])

In [27]: y

Out[27]: array([1, 2, 3, 4])

In [28]: x[0, 0] = -5
         x

Out[28]: array([[-5,  2],
                [ 3,  4]])

In [29]: y

Out[29]: array([-5,  2,  3,  4])

In [ ]:
```

- We can use np.newaxis to increase the dimension of an array by one dimension

```
Cell Toolbar: None

In [91]: vector
Out[91]: array([1, 2, 3, 4, 5])

In [92]: row_vector = vector[np.newaxis, :]
         row_vector
Out[92]: array([[1, 2, 3, 4, 5]])

In [93]: row_vector.shape
Out[93]: (1, 5)

In [96]: column_vector = vector[:, np.newaxis]
         column_vector
Out[96]: array([[1],
                [2],
                [3],
                [4],
                [5]])

In [97]: column_vector.shape
Out[97]: (5, 1)
```

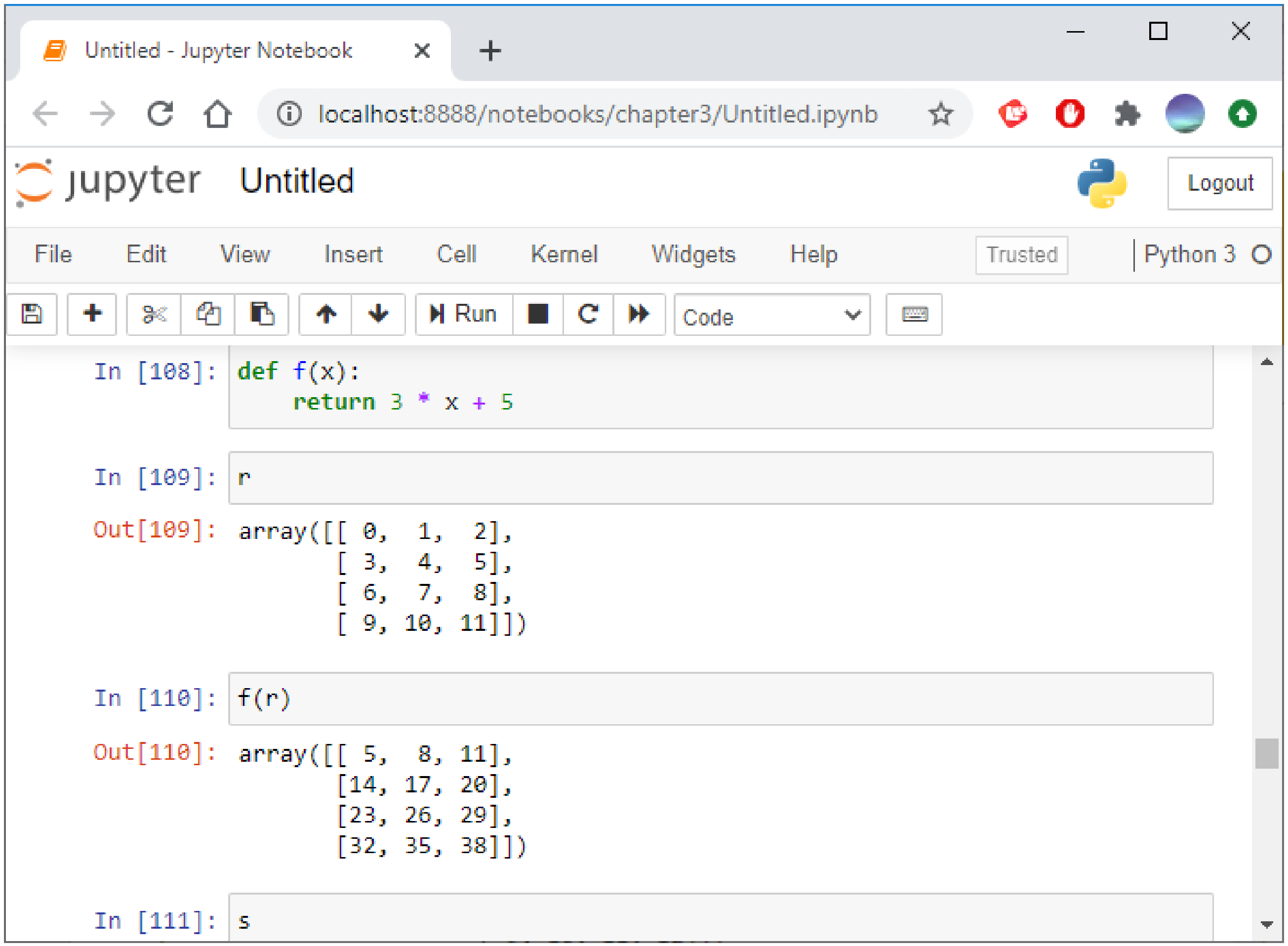


Vector Operations on Arrays

Cell Toolbar: None

```
In [98]: a = np.ones(12).reshape(3, 4)
         a

Out[98]: array([[ 1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.]])

In [99]: b = np.arange(1,13).reshape(3,4)
         b

Out[99]: array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12]])

In [100]: a + b

Out[100]: array([[  2.,   3.,   4.,   5.],
                 [  6.,   7.,   8.,   9.],
                 [ 10.,  11.,  12.,  13.]])

In [101]: a * 5

Out[101]: array([[ 5.,  5.,  5.,  5.],
                 [ 5.,  5.,  5.,  5.],
                 [ 5.,  5.,  5.,  5.]])

In [102]: 2 ** b

Out[102]: array([[   2,    4,    8,   16],
                 [  32,   64,  128,  256],
                 [ 512, 1024, 2048, 4096]], dtype=int32)
```

Using User-Defined Functions with ndarray Objects



Boolean Operations on Arrays

- Comparison and logical operations in ndarray objects work on an element-wise basis
- Evaluating conditions yield by default a Boolean ndarray object
- Use the bitwise logical operators (&, |, ^, ~) to perform element-wise logical operations

Untitled - Jupyter Notebook

localhost:8888/notebooks/chapter3/Untitled.ipynb

jupyter Untitled (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [40]: g = np.arange(15)
         g

Out[40]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

In [41]: h = g.reshape((3, 5))
         h

Out[41]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14]])

In [42]: h > 5

Out[42]: array([[False, False, False, False, False],
                [False,  True,  True,  True,  True],
                [ True,  True,  True,  True,  True]])

In [43]: (h > 5) & ( h <= 12)

Out[43]: array([[False, False, False, False, False],
                [False,  True,  True,  True,  True],
                [ True,  True,  True, False, False]])
```

Boolean Arrays as Masks

- Boolean arrays can be used as masks, to select particular subsets of the data jupyter Untitled (unsaved changes) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [46]: h

Out[46]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14]])

In [47]: h > 5

Out[47]: array([[False, False, False, False, False],
                [False,  True,  True,  True,  True],
                [ True,  True,  True,  True,  True]])

In [48]: h[h>5]

Out[48]: array([ 6,  7,  8,  9, 10, 11, 12, 13, 14])

In [49]: (h > 5) & ( h <= 12)

Out[49]: array([[False, False, False, False, False],
                [False,  True,  True,  True,  True],
                [ True,  True,  True, False, False]])

In [50]: h[(h > 5) & ( h <= 12)]

Out[50]: array([ 6,  7,  8,  9, 10, 11, 12])

In [ ]:
```

Untitled - Jupyter Notebook

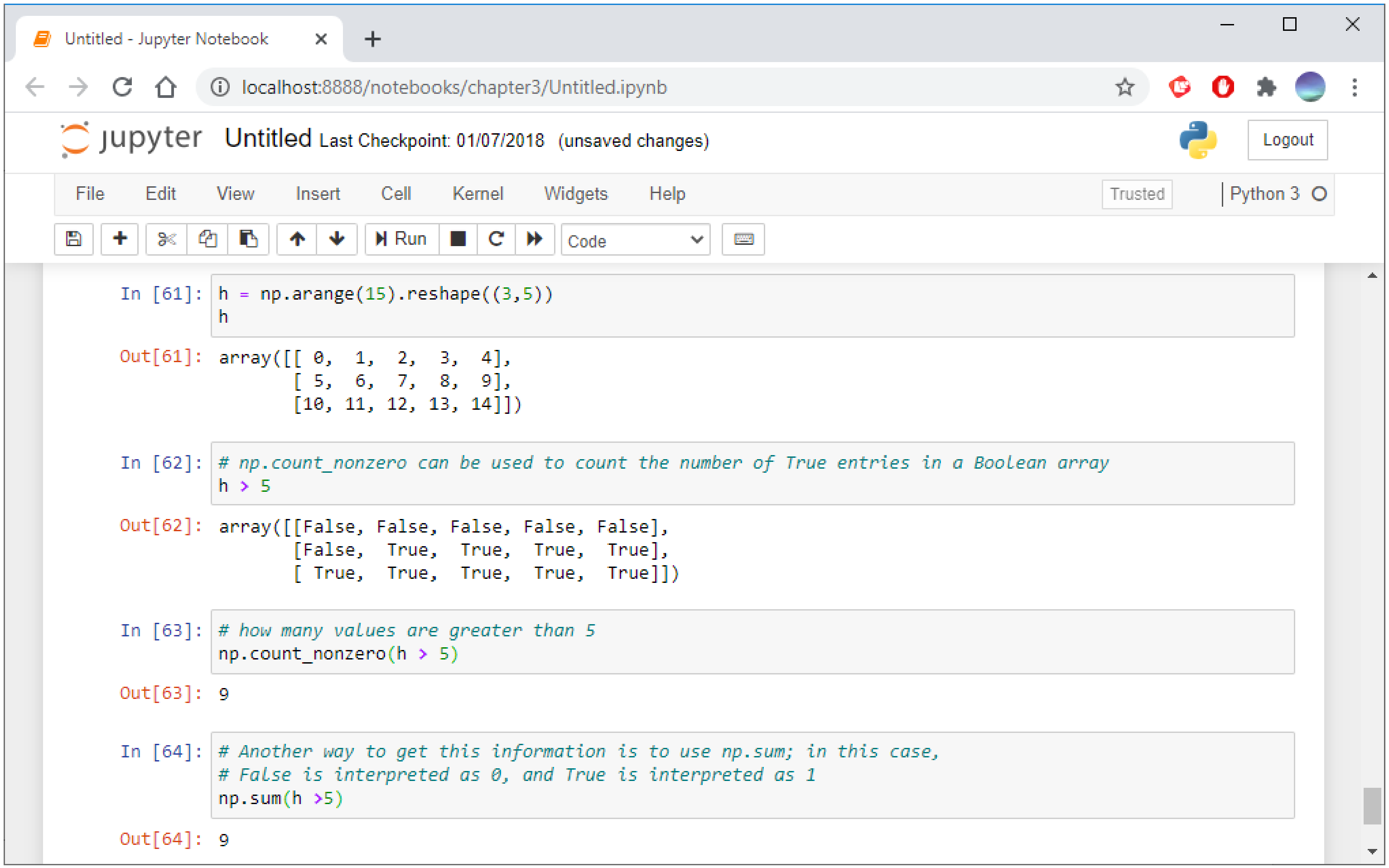
localhost:8888/notebooks/chapter3/Untitled.ipynb

jupyter Untitled

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```
In [45]: r

Out[45]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14]])

In [46]: def even(x): return x % 2 == 0

In [47]: even(2)

Out[47]: True

In [48]: even(r)

Out[48]: array([[ True, False,  True, False,  True],
                [False,  True, False,  True, False],
                [ True, False,  True, False,  True]])

In [50]: r[even(r)]

Out[50]: array([ 0,  2,  4,  6,  8, 10, 12, 14])

In [ ]:
```

Counting Entries



np.any() and np.all()

```
In [91]: h

Out[91]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14]])

In [92]: # are there any values greater than 5?
         np.any(h > 5)

Out[92]: True

In [93]: # are all values <= 10?
         np.all(h <= 10)

Out[93]: False

In [94]: # is any value == 100?
         np.any(h == 100)

Out[94]: False

In [95]: # are all values != 15?
         np.all(h != 15)

Out[95]: True

In [96]: # is an value between 10 and 20?
         np.any( (h >= 10) & (h<=20))

Out[96]: True
```

np.where()

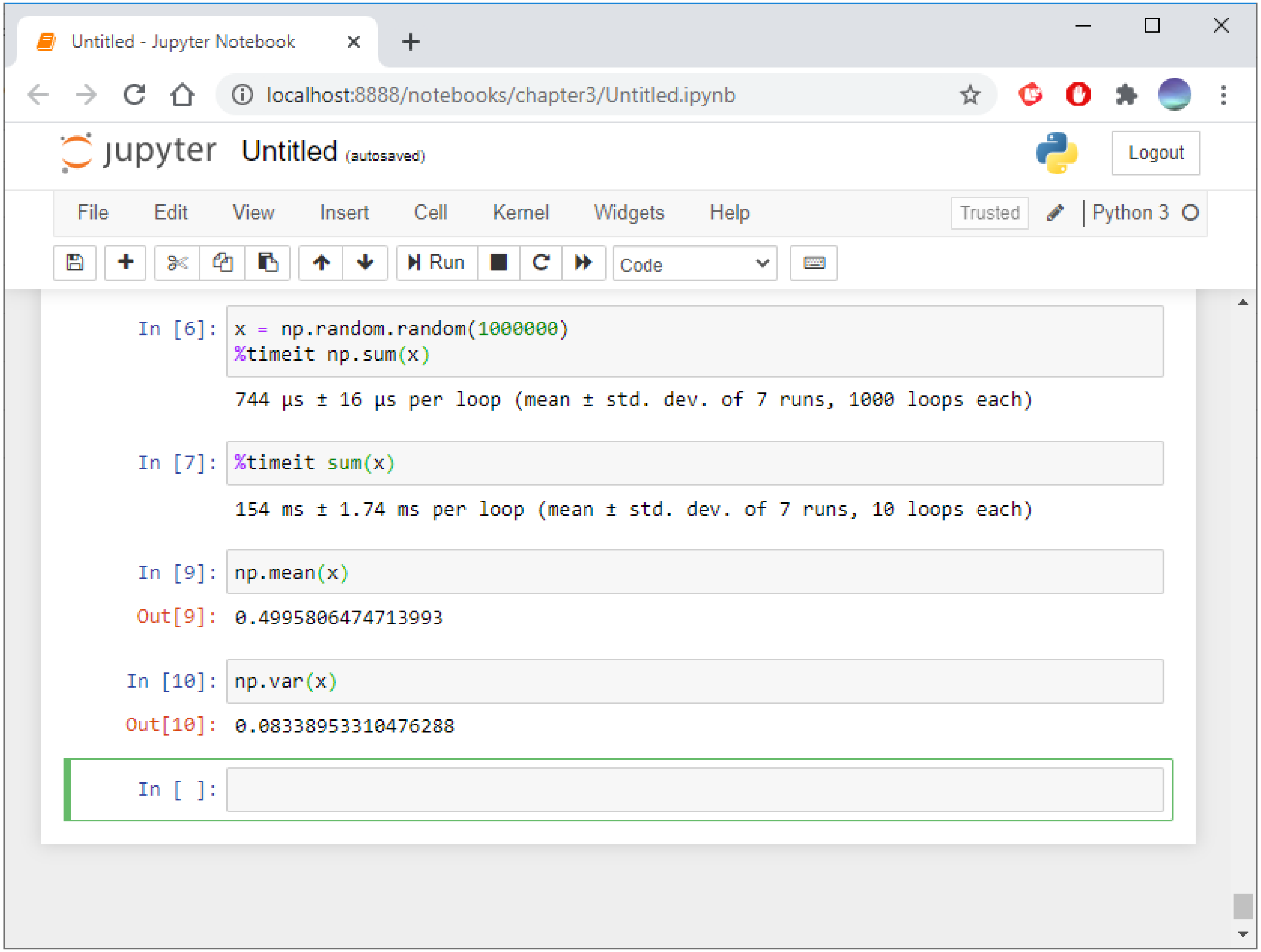
- The np.where() function, allows the definition of actions depending on whether a condition is True or False
- The result of applying np.where() is a new ndarray object of the same shape as the original one

```
In [33]: a = np.arange(12).reshape(3, -1) #same as reshape(3, 4)
         a

Out[33]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

In [34]: np.where(a > 5, 1, 0)

Out[34]: array([[0, 0, 0, 0],
                [0, 0, 1, 1],
                [1, 1, 1, 1]])

In [35]: np.where(a %2 == 0, 'even', 'odd')

Out[35]: array([['even', 'odd', 'even', 'odd'],
                ['even', 'odd', 'even', 'odd'],
                ['even', 'odd', 'even', 'odd']], dtype='<U4')

In [36]: np.where(a <= 5, 2 * a,  a / 2)

Out[36]: array([[ 0. ,  2. ,  4. ,  6. ],
                [ 8. , 10. ,  3. ,  3.5],
                [ 4. ,  4.5,  5. ,  5.5]])
```

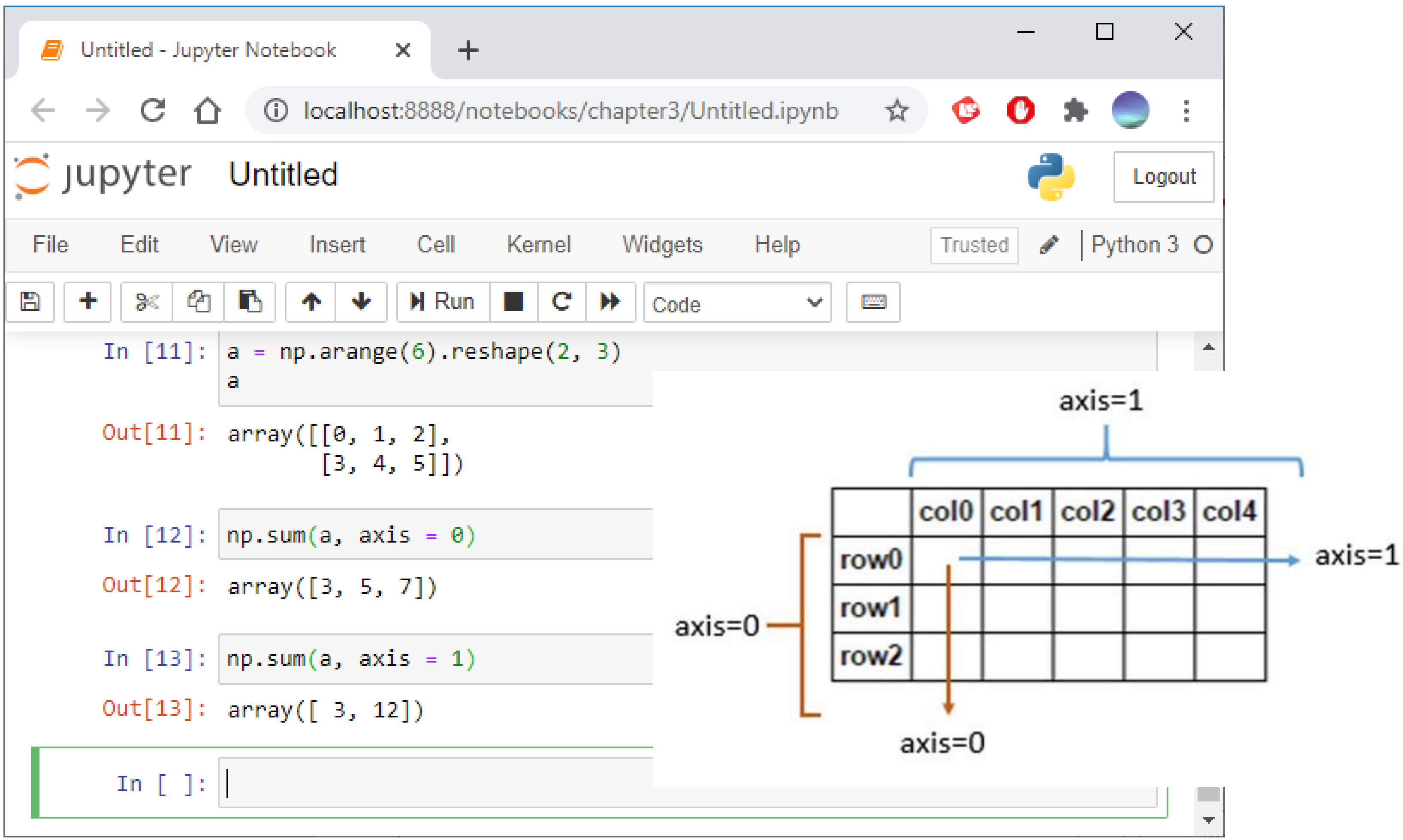
Aggregate Functions

- They are useful to compute summary statistics for large data
- NumPy provides many functions for this including: sum, max, min, mean, std, var, median
- These functions are faster than their Python counterparts

NumPy functions are faster than their Python counterparts

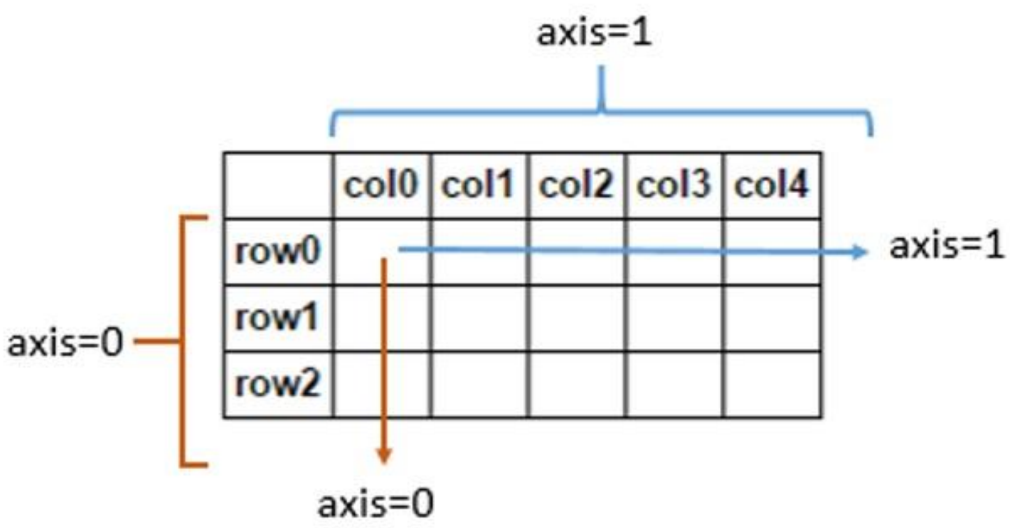


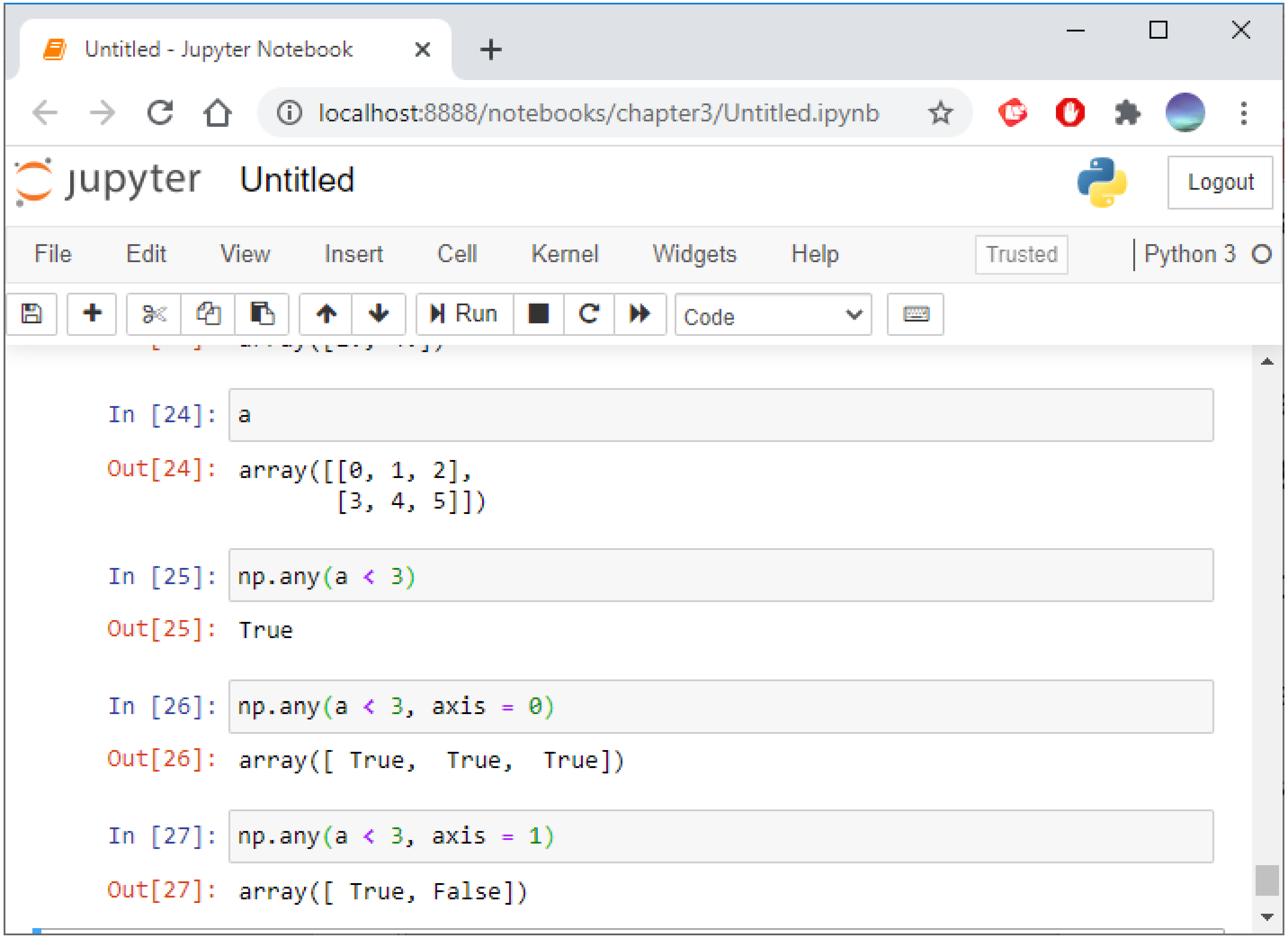
Applying Aggregate Functions to Rows/Columns of 2D Array



More on Axis/Axes

- In multidimensional arrays, there is an axis per dimension (ex: a[3,1, 2])
- First axis runs across the rows, second axis runs across the columns
- When an aggregate function is be applied along an axis, think about the axis as the dimension of the array that will be collapsed, rather than the dimension that will be returned
  - In case of axis=0, 1st dimension is collapsed; i.e., NumPy performs the action on elements of each column
  - If axis=1, NumPy performs the action on rows
- Another way to think about axis is as the direction along which the operation is performed




Untitled - Jupyter Notebook
localhost:8888/notebooks/chapter3/Untitled.ipynb
jupyter Untitled
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
In [24]: a
Out[24]: array([[0, 1, 2],
[3, 4, 5]])
In [25]: np.any(a < 3)
Out[25]: True
In [26]: np.any(a < 3, axis = 0)
Out[26]: array([True, True, True])
In [27]: np.any(a < 3, axis = 1)
Out[27]: array([True, False])

Broadcasting

- For arrays of the same shape, binary operations are performed on an element-by-element basis

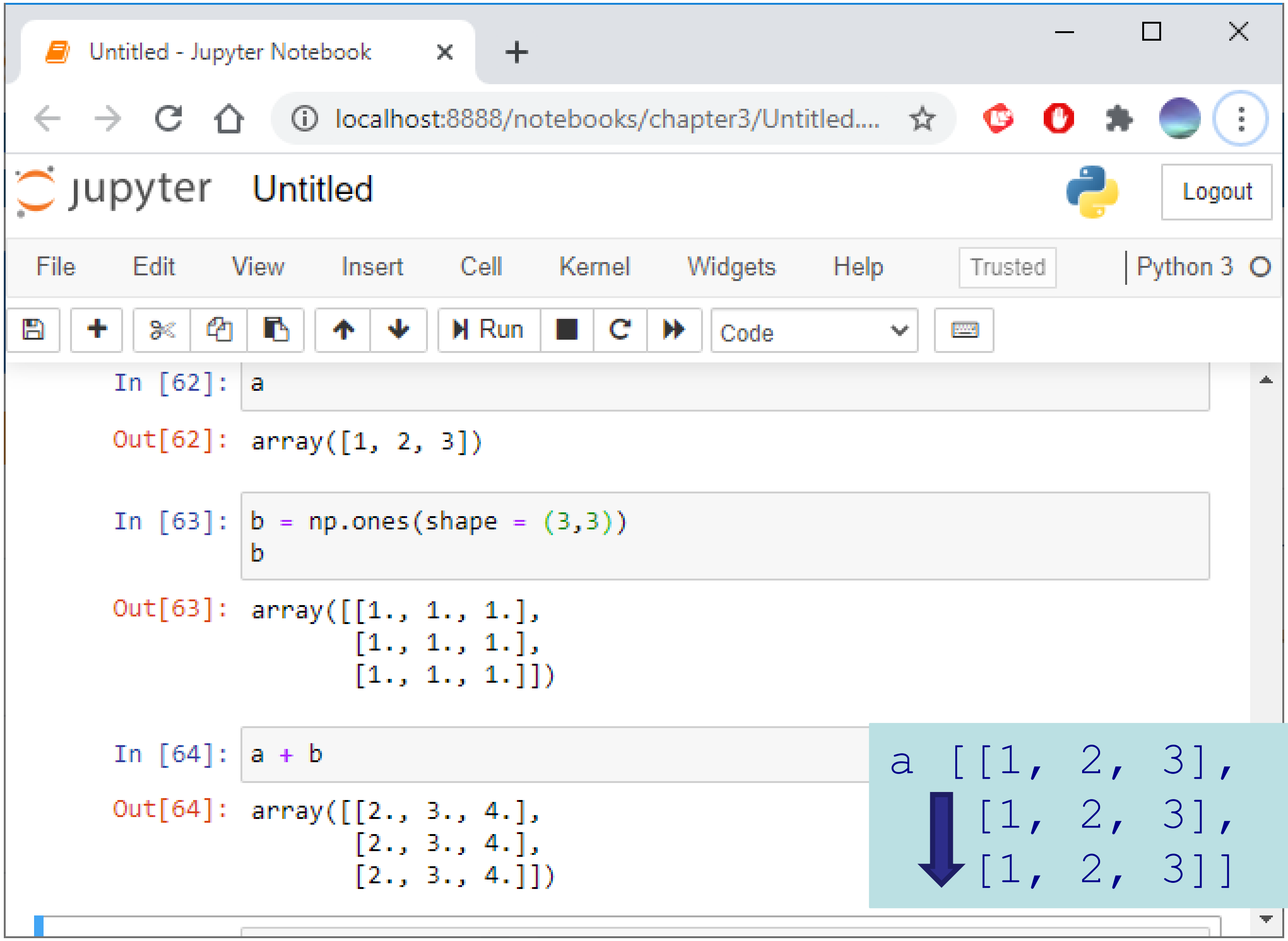
```
In [1]: a = np.array([1, 2, 3])
        b = np.array([4, 5, 6])
        a + b
Out[1]: array([ 5, 7 , 9])
```

- Broadcasting allows binary operations to be performed on arrays of different sizes

```
In [2]: a + 2
Out[2]: array([ 3, 4 , 5])
```

```
a + 2 ≡
[1, 2, 3] + [2, 2, 2]
```

Broadcasting (cont.)



Rules of Broadcasting

- A set of rules that control what happens when a binary operation is applied to two arrays of different shapes

Rules of Broadcasting - Rule 1

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the array with fewer dimensions is padded with ones on its leading (left) side

```
a = np.arange(12).reshape(3, 4)
b = np.array([1 , 2 , 3, 4])
a + b

a.shape = (3, 4)
b.shape = (4,)
```

- Rule 1 applies:

```
b.shape ->  (1, 4)
```

Rules of Broadcasting – Rule 2

- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape

```
a = np.arange(12).reshape(3, 4)
b = np.array([1 , 2 , 3, 4]).reshape(1,4)
a + b
```

```
a.shape = (3, 4)
b.shape = (1, 4)
```

```
b [[1, 2, 3, 4]]
```

- Rule 2 applies to the array b

```
b.shape -> (3, 4)
```

```
b [[1, 2, 3, 4],
   [1, 2, 3, 4],
   [1, 2, 3, 4]]
```

Rules of Broadcasting (cont.)

- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised

```
a = np.arange(12).reshape(3, 4)
b = np.array([1 , 2 , 3, 4]).reshape(2,2)
a + b
```

```
a.shape = (3, 4)
b.shape = (2, 2)
```

- By rule 3, the arrays a and b are not compatible

Broadcasting Example 1

- Consider adding the following two arrays

```
In [3]: M = np.ones((2, 3))
        a = np.arange(3)
```

```
M is [[1, 1, 1],
      [1, 1, 1]]
a is [0, 1, 2]
```

- The shapes of the arrays are

```
M.shape = (2, 3)
a.shape = (3,)
```

- By rule 1, array a has fewer dimensions, so we pad its shape on the left by ones:

```
M.shape -> (2, 3)
a.shape -> (1, 3)
```

```
M is [[1, 1, 1],
      [1, 1, 1]]
a is [[0, 1, 2]]
```

- By rule 2, the shape of the two arrays do not match on the first dimension, so we stretch this dimension in a to match:

```
M.shape -> (2, 3)
a.shape -> (2, 3)
```

```
M is [[1, 1, 1],
      [1, 1, 1]]
a is [[0, 1, 2],
      [0, 1, 2]]
```

```
M + a is
  [[1, 2, 3],
   [1, 2, 3]]
```

Broadcasting Example 2

- Consider adding the following two arrays

```
a = np.arange(3)
b = np.arange(3).reshape(3, 1)
```

```
a is [0, 1, 2]
b is [[0],
      [1],
      [2]]
```

- The shapes of the arrays are

```
a.shape = (3,)
b.shape = (3,1)
```

- By rule 1, we pad the shape of a on the left by 1s:

```
a.shape -> (1, 3)
b.shape -> (3, 1)
```

```
a [[0, 1, 2]]
```

```
b [[0],
   [1],
   [2]]
```

- By rule 2, we upgrade each of these ones to match the corresponding size of the other array:

```
a.shape -> (3, 3)
b.shape -> (3, 3)
```

```
a is [[0, 1, 2],
      [0, 1, 2],
      [0, 1, 2]]
```

```
b is [[0, 0, 0],
      [1, 1, 1],
      [2, 2, 2]]
```

```
Now the two arrays have
the same shape and can
be added
```

```
a + b is [[0, 1, 2],
          [1, 2, 3],
          [2, 3, 4]]
```

Broadcasting Example 3

- Consider adding the following two arrays

```
In [3]: M = np.ones((3, 2))
        a = np.arange(3)
```

```
M is [[1, 1],
      [1, 1],
      [1, 1]]
a is [0, 1, 2]
```

- The shapes of the arrays are

```
M.shape = (3, 2)
a.shape = (3,)
```

- By rule 1, we pad array a by 1s on the left:

```
M.shape -> (3, 2)
a.shape -> (1, 3)
```

```
a is [[0, 1, 2]]
```

```
M [[1, 1],
   [1, 1],
   [1, 1]]
```

- By rule 2, the first dimension of a is stretched to match that of M:

```
M.shape -> (3, 2)
a.shape -> (3, 3)
```

```
a [[0, 1, 2],
   [0, 1, 2],
   [0, 1, 2]]
```

```
M [[1, 1],
   [1, 1],
   [1, 1]]
```

- Now rule 3 applies, the two arrays do not match on the second dimension and neither is equal to 1, so the two arrays are not compatible