intel Nervana AI Academy

INTRODUCTION TO MACHINE LEARNING AND TOOLKIT

OVERVIEW OF COURSE

Topics include:

- Introduction and exploratory analysis (Week 1)
- Supervised machine learning (Weeks 2–10)
- Unsupervised machine learning (Weeks 11–12)

OVERVIEW OF COURSE

Topics include:

- Introduction and exploratory analysis (Week 1)
- Supervised machine learning (Weeks 2–10)
- Unsupervised machine learning (Weeks 11–12)

Each week:

- Lecture
- Exercises with solutions
- Time commitment: ~3 hours per week

OUR TOOLSET: INTEL® DISTRIBUTION FOR PYTHON

- Accelerated performance from Intel's Math Kernel Library (MKL)

OUR TOOLSET: INTEL® DISTRIBUTION FOR PYTHON

- Accelerated performance from Intel's Math Kernel Library (MKL)
- Also contains Data Analytics Acceleration Library (DAAL), Message Passing Interface (MPI), and Threading Building Blocks (TBB)

OUR TOOLSET: INTEL® DISTRIBUTION FOR PYTHON

- Accelerated performance from Intel's Math Kernel Library (MKL)
- Also contains Data Analytics Acceleration Library (DAAL), Message Passing Interface (MPI), and Threading Building Blocks (TBB)

INSTALLATION OPTIONS

	software.intel.com/
Monolithic Distribution	intel-distribution-for-python
Anaconda Package Manager	articles/using-intel-distribution-for-python-with-anaconda

OUR TOOLSET: INTEL® DISTRIBUTION FOR PYTHON

- Accelerated performance from Intel's Math Kernel Library (MKL)
- Also contains Data Analytics Acceleration Library (DAAL), Message Passing Interface (MPI), and Threading Building Blocks (TBB)

INSTALLATION OPTIONS

	software.intel.com/
Monolithic Distribution	intel-distribution-for-python
Anaconda Package Manager	articles/using-intel-distribution-for-python-with-anaconda

Seaborn is also required: `conda install seaborn`

OUR TOOLSET: INTEL® DISTRIBUTION FOR PYTHON

Jupyter notebooks:

- Interactive Coding and Visualization of Output

NumPy, SciPy, Pandas:

- Numerical Computation

Matplotlib, Seaborn:

- Data Visualization

Scikit-learn:

- Machine Learning

OUR TOOLSET: INTEL® DISTRIBUTION FOR PYTHON

Jupyter notebooks:

- Interactive Coding and Visualization of Output

NumPy, SciPy, Pandas:

- Numerical Computation

Matplotlib, Seaborn:

- Data Visualization

WEEK 1

Scikit-learn:

- Machine Learning

OUR TOOLSET: INTEL® DISTRIBUTION FOR PYTHON

Jupyter notebooks:

- Interactive Coding and Visualization of Output

NumPy, SciPy, Pandas:

- Numerical Computation

Matplotlib, Seaborn:

- Data Visualization

Scikit-learn:

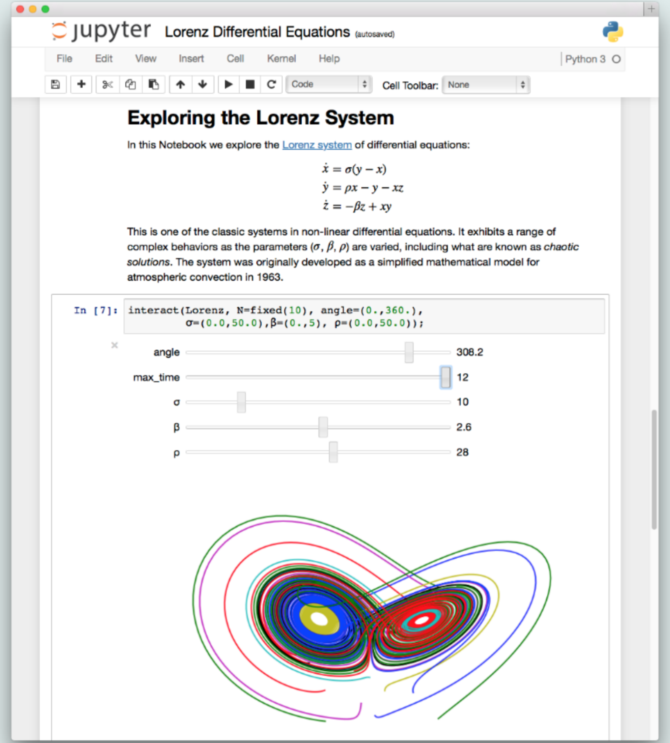
- Machine Learning

WEEKS 2-12

JUPYTER NOTEBOOK

INTRODUCTION TO JUPYTER NOTEBOOK

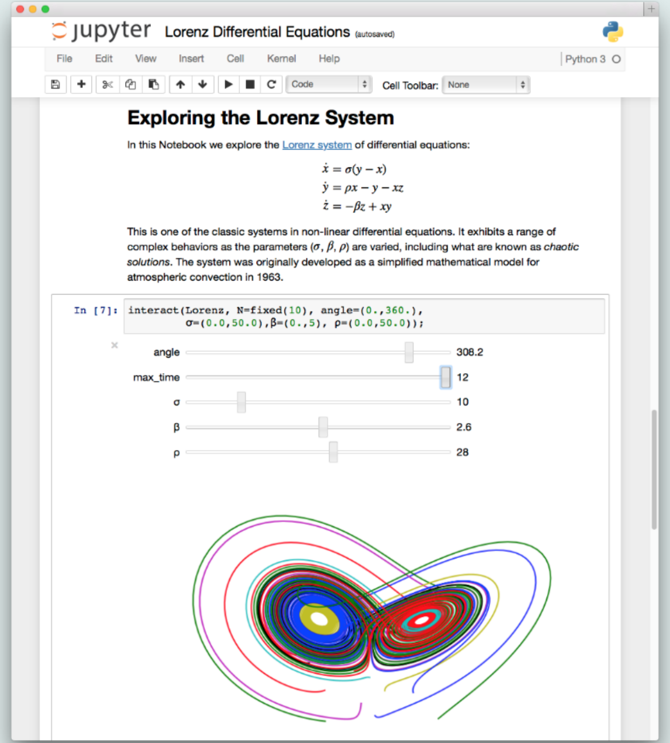
- Polyglot analysis environment—blends multiple languages



Source: http://jupyter.org/

INTRODUCTION TO JUPYTER NOTEBOOK

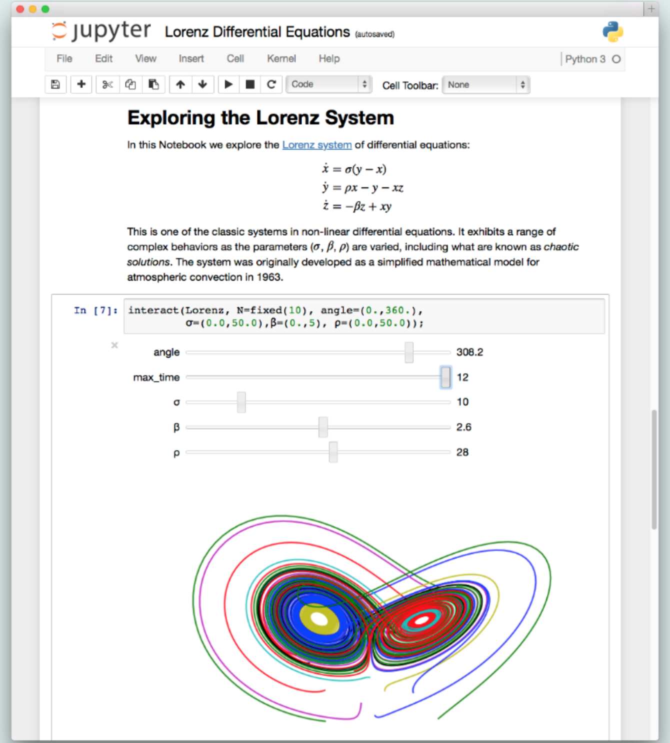
- Polyglot analysis environment—blends multiple languages
- Jupyter is an anagram of: Julia, Python, and R



Source: http://jupyter.org/

INTRODUCTION TO JUPYTER NOTEBOOK

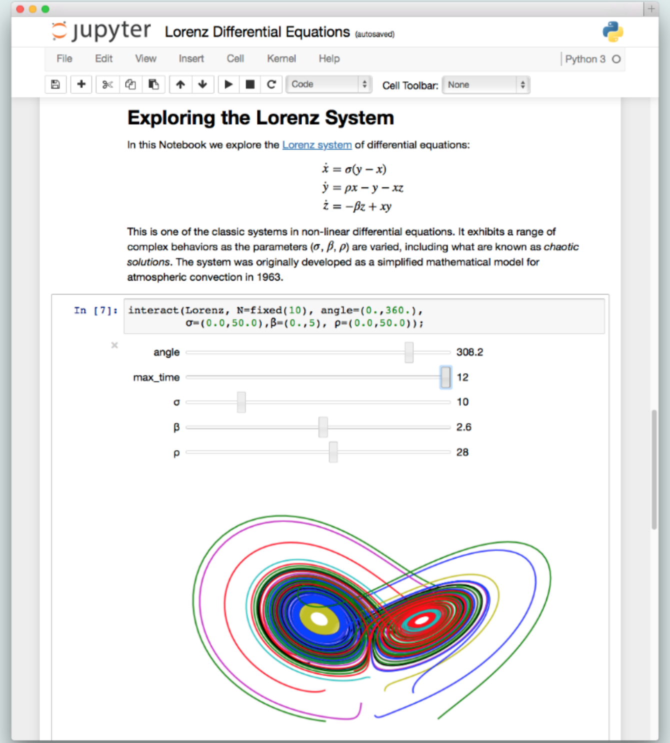
- Polyglot analysis environment—blends multiple languages
- Jupyter is an anagram of: Julia, Python, and R
- Supports multiple content types: code, narrative text, images, movies, etc.



Source: http://jupyter.org/

INTRODUCTION TO JUPYTER NOTEBOOK

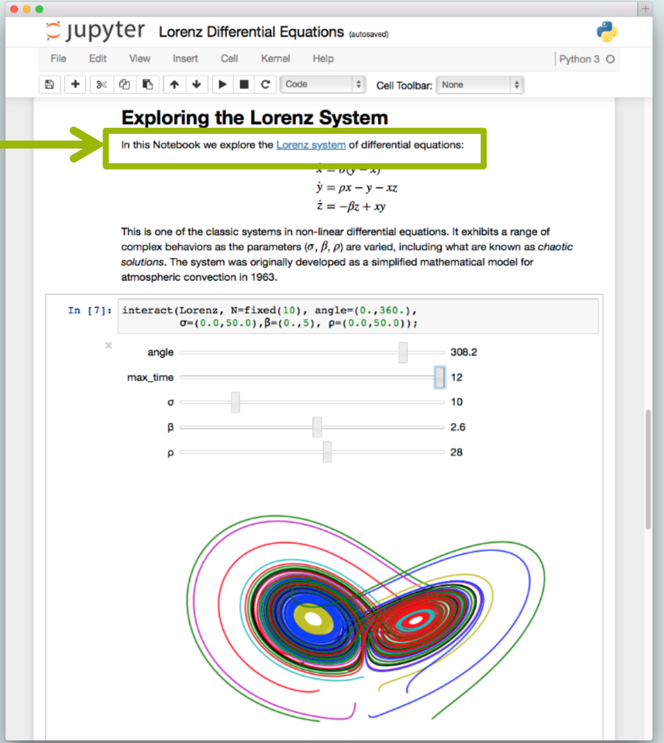
- HTML & Markdown
- LaTeX (equations)
- Code



Source: http://jupyter.org/

INTRODUCTION TO JUPYTER NOTEBOOK

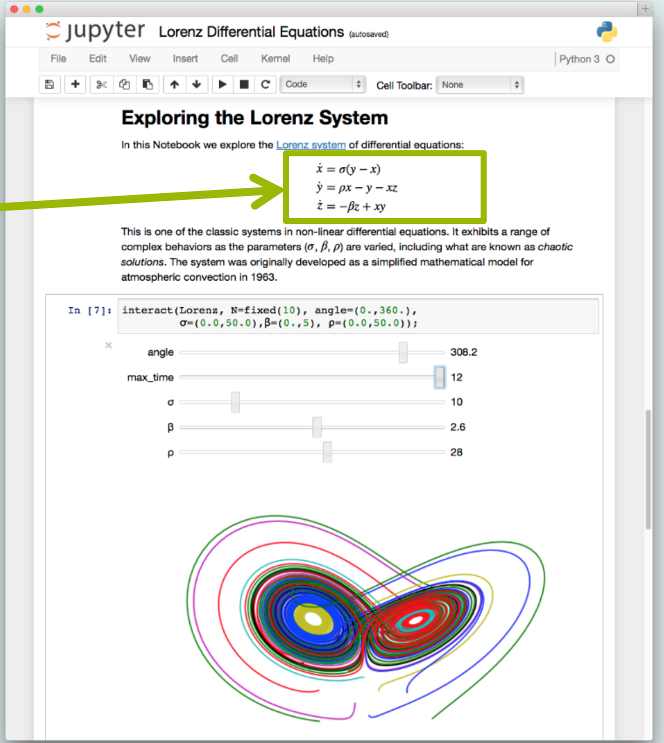
- HTML & Markdown
- LaTeX (equations)
- Code



Source: http://jupyter.org/

INTRODUCTION TO JUPYTER NOTEBOOK

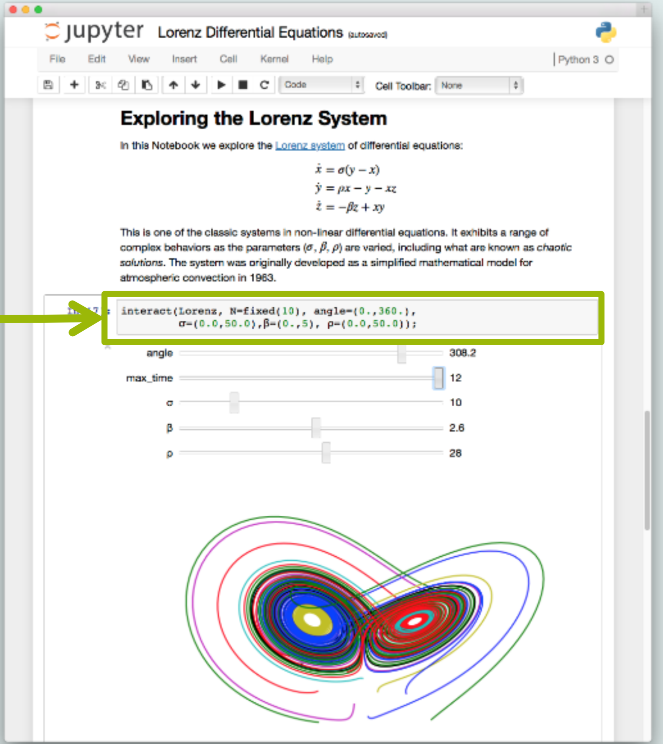
- HTML & Markdown
- LaTeX (equations)
- Code



Source: http://jupyter.org/

INTRODUCTION TO JUPYTER NOTEBOOK

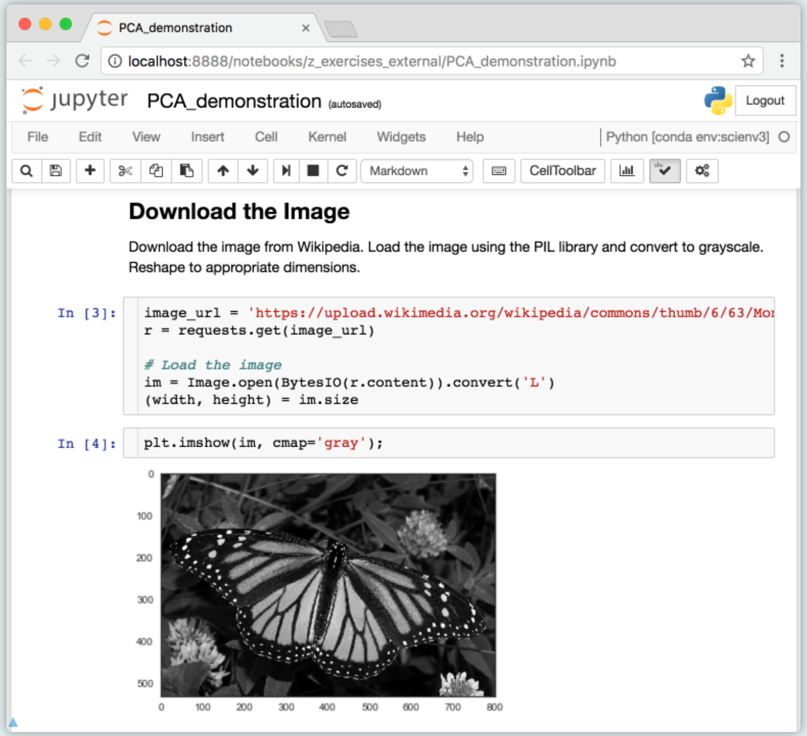
- HTML & Markdown
- LaTeX (equations)
- Code



Source: http://jupyter.org/

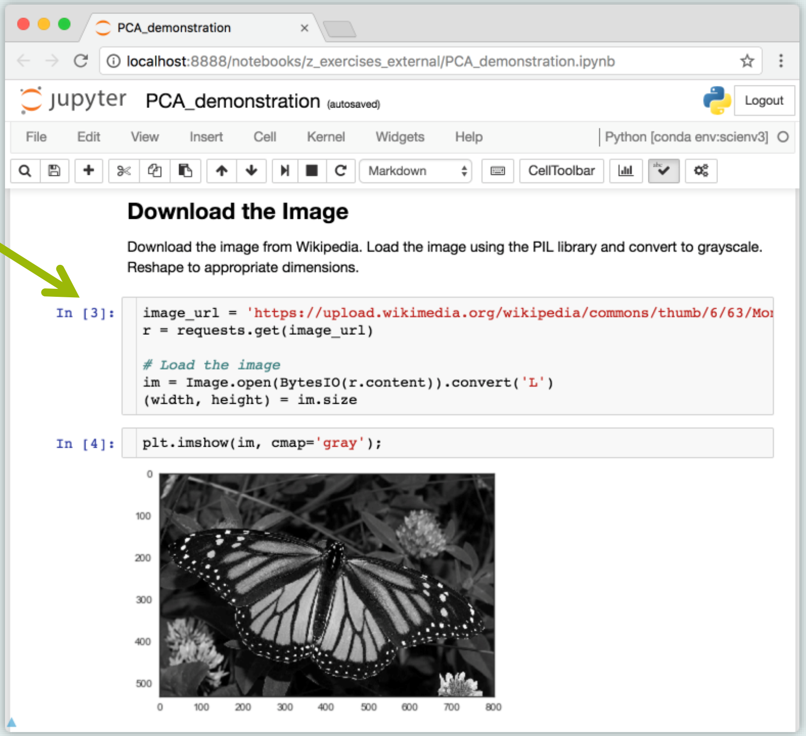
INTRODUCTION TO JUPYTER NOTEBOOK

- Code is divided into cells to control execution
- Enables interactive development
- Ideal for exploratory analysis and model building



INTRODUCTION TO JUPYTER NOTEBOOK

- Code is divided into cells to control execution
- Enables interactive development
- Ideal for exploratory analysis and model building

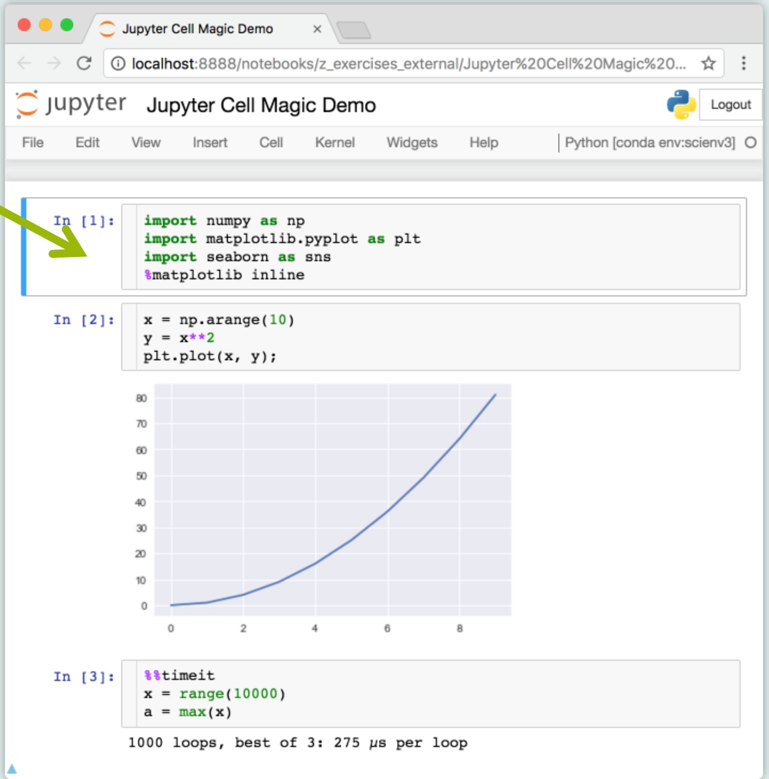


JUPYTER CELL MAGICS

- %matplotlib inline: display plots inline in Jupyter notebook

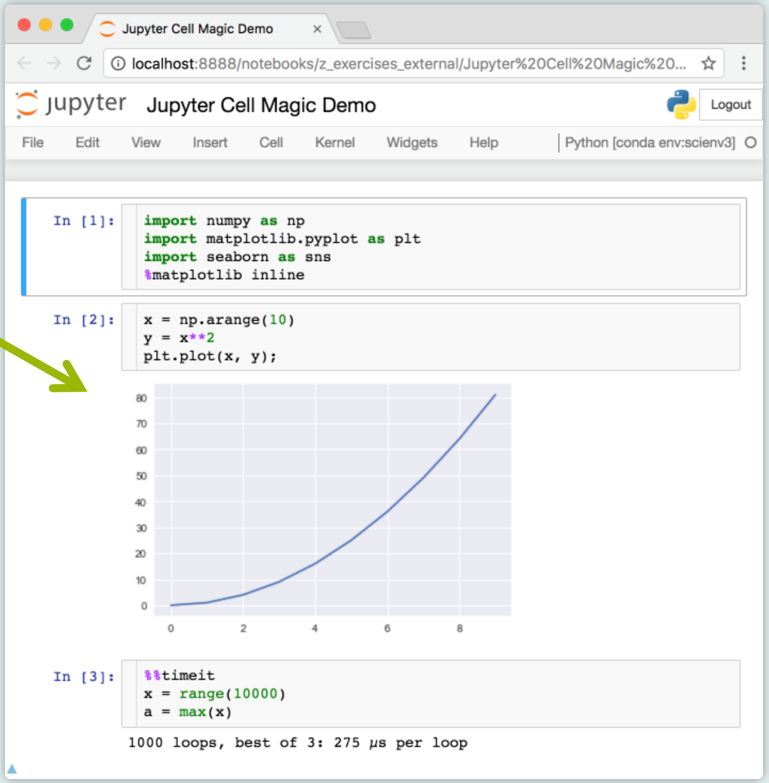
JUPYTER CELL MAGICS

- %matplotlib inline: display plots inline in Jupyter notebook



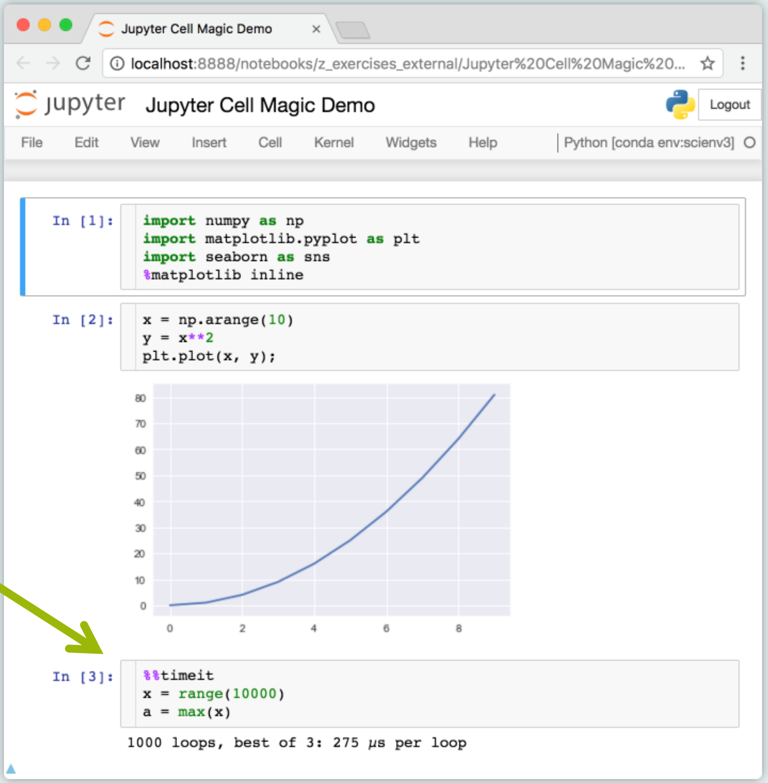
JUPYTER CELL MAGICS

- %matplotlib inline: display plots inline in Jupyter notebook



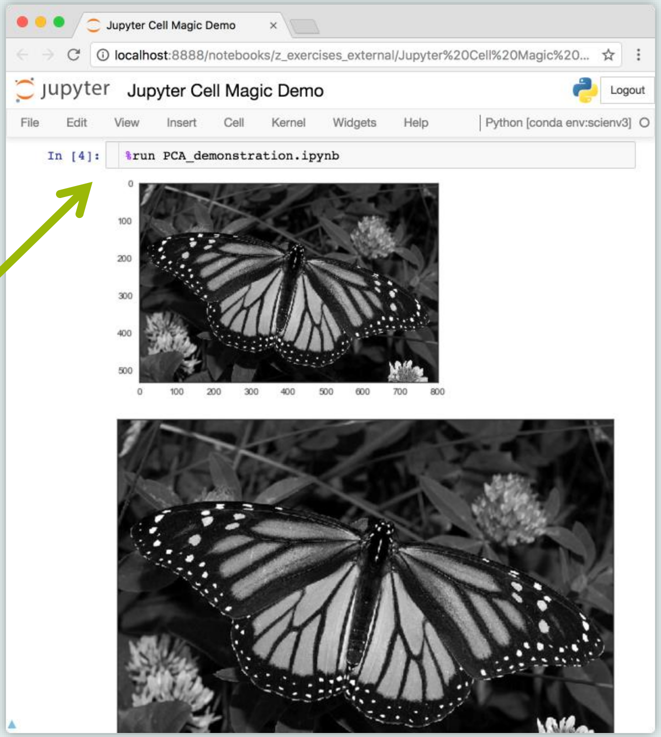
JUPYTER CELL MAGICS

- %matplotlib inline: display plots inline in Jupyter notebook
- %%timeit: time how long a cell takes to execute



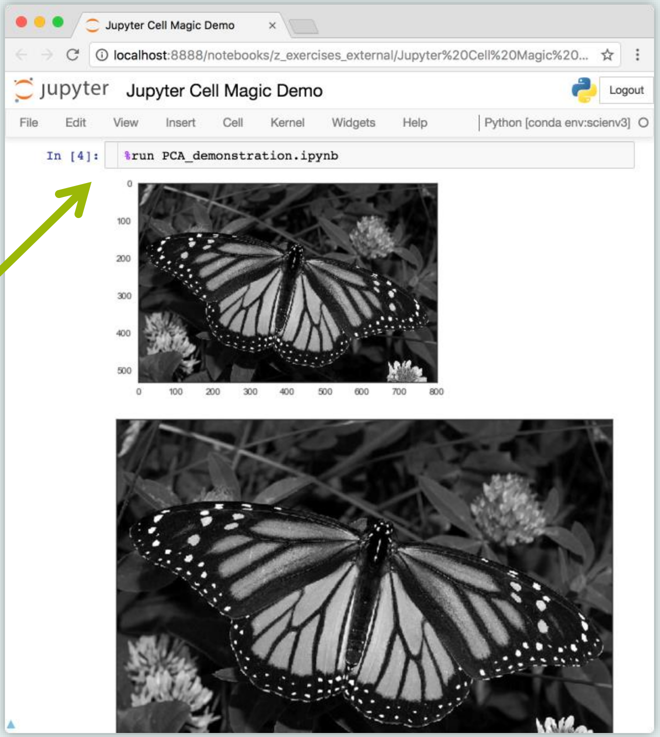
JUPYTER CELL MAGICS

- %matplotlib inline: display plots inline in Jupyter notebook
- %%timeit: time how long a cell takes to execute
- %run filename.ipynb: execute code from another notebook or python file



JUPYTER CELL MAGICS

- %matplotlib inline: display plots inline in Jupyter notebook
- %%timeit: time how long a cell takes to execute
- %run filename.ipynb: execute code from another notebook or python file
- %load filename.py: copy contents of the file and paste into the cell



JUPYTER KEYBOARD SHORTCUTS

Keyboard shortcuts

The Jupyter Notebook has two different keyboard input modes. **Edit mode** allows you to type code/text into a cell and is indicated by a green cell border. **Command mode** binds the keyboard to notebook level actions and is indicated by a grey cell border with a blue left margin.

Command Mode (press `Esc` to enable)

`F`: find and replace	`Shift-J`: extend selected cells below
`Ctrl-Shift-P`: open the command palette	`A`: insert cell above
`Enter`: enter edit mode	`B`: insert cell below
`Shift-Enter`: run cell, select below	`X`: cut selected cells
`Ctrl-Enter`: run selected cells	`C`: copy selected cells
`Alt-Enter`: run cell, insert below	`Shift-V`: paste cells above

Keyboard shortcuts can be viewed from Help → Keyboard Shortcuts

MAKING JUPYTER NOTEBOOKS REUSABLE

To extract Python code from a Jupyter notebook:

Convert from Command Line

```
>>> jupyter nbconvert --to python
    notebook.ipynb
```

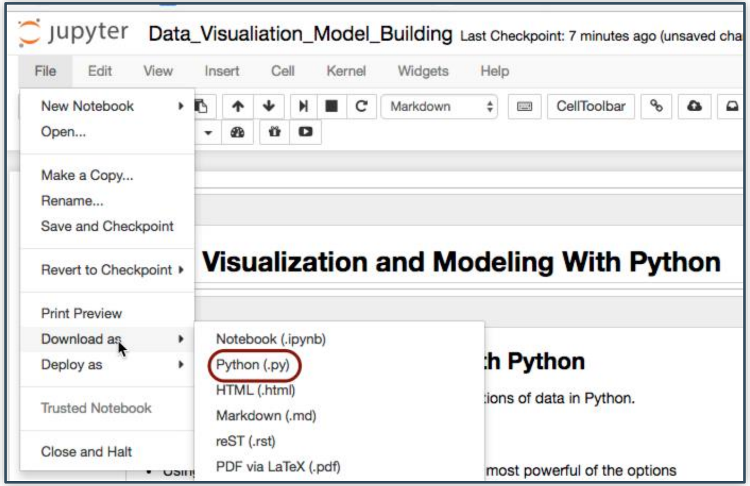
MAKING JUPYTER NOTEBOOKS REUSABLE

To extract Python code from a Jupyter notebook:

Convert from Command Line

```
>>> jupyter nbconvert --to python
    notebook.ipynb
```

Export from Notebook



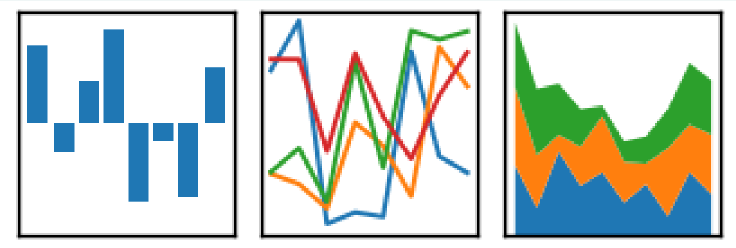
PANDAS

INTRODUCTION TO PANDAS

- Library for computation with tabular data
- Mixed types of data allowed in a single table
- Columns and rows of data can be named
- Advanced data aggregation and statistical functions

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Source: http://pandas.pydata.org/

INTRODUCTION TO PANDAS

Basic data structures

Type		Pandas Name
Vector (1 Dimension)	→	Series

INTRODUCTION TO PANDAS

Basic data structures

Type		Pandas Name
Vector (1 Dimension)	→	Series
Array (2 Dimensions)	→	DataFrame

PANDAS SERIES CREATION AND INDEXING

Use data from step tracking application to create a Pandas Series

CODE

```
import pandas as pd

step_data = [3620, 7891, 9761,
             3907, 4338, 5373]

step_counts = pd.Series(step_data,
                name='steps')

print(step_counts)
```

OUTPUT

PANDAS SERIES CREATION AND INDEXING

Use data from step tracking application to create a Pandas Series

CODE

```
import pandas as pd

step_data = [3620, 7891, 9761,
             3907, 4338, 5373]

step_counts = pd.Series(step_data,
                        name='steps')

print(step_counts)
```

OUTPUT

```
>>> 0  3620
    1  7891
    2  9761
    3  3907
    4  4338
    5  5373
    Name: steps, dtype: int64
```

PANDAS SERIES CREATION AND INDEXING

Add a date range to the Series

CODE

```
step_counts.index = pd.date_range('20150329',
                                  periods=6)

print(step_counts)
```

OUTPUT

PANDAS SERIES CREATION AND INDEXING

Add a date range to the Series

CODE

```
step_counts.index = pd.date_range('20150329',
                                   periods=6)

print(step_counts)
```

OUTPUT

```
>>>  2015-03-29 3620
     2015-03-30 7891
     2015-03-31 9761
     2015-04-01 3907
     2015-04-02 4338
     2015-04-03 5373
     Freq: D, Name: steps,
     dtype: int64
```

PANDAS SERIES CREATION AND INDEXING

Select data by the index values

CODE

```
# Just like a dictionary
print(step_counts['2015-04-01'])
```

OUTPUT

PANDAS SERIES CREATION AND INDEXING

Select data by the index values

CODE

```
# Just like a dictionary
print(step_counts['2015-04-01'])
```

OUTPUT

```
>>> 3907
```

PANDAS SERIES CREATION AND INDEXING

Select data by the index values

CODE

```
# Just like a dictionary
print(step_counts['2015-04-01'])

# Or by index position—like an array
print(step_counts[3])
```

OUTPUT

```
>>> 3907
```

PANDAS SERIES CREATION AND INDEXING

Select data by the index values

CODE

```
# Just like a dictionary
print(step_counts['2015-04-01'])

# Or by index position—like an array
print(step_counts[3])
```

OUTPUT

```
>>> 3907

>>> 3907
```

PANDAS SERIES CREATION AND INDEXING

Select data by the index values

CODE

```
# Just like a dictionary
print(step_counts['2015-04-01'])

# Or by index position—like an array
print(step_counts[3])

# Select all of April
print(step_counts['2015-04'])
```

OUTPUT

```
>>> 3907


>>> 3907
```

PANDAS SERIES CREATION AND INDEXING

Select data by the index values

CODE	OUTPUT
`# Just like a dictionary` `print(step_counts['2015-04-01'])`	`>>> 3907`
`# Or by index position—like an array` `print(step_counts[3])`	`>>> 3907`

`# Select all of April`
`print(step_counts['2015-04'])`
	`>>> 2015-04-01 3907`
	`2015-04-02 4338`
	`2015-04-03 5373`
	`Freq: D, Name: steps,`
	`dtype: int64`

PANDAS DATA TYPES AND IMPUTATION

Data types can be viewed and converted

CODE

```
# View the data type
print(step_counts.dtypes)
```

OUTPUT

PANDAS DATA TYPES AND IMPUTATION

Data types can be viewed and converted

CODE

```
# View the data type
print(step_counts.dtypes)
```

OUTPUT

```
>>>  int64
```

PANDAS DATA TYPES AND IMPUTATION

Data types can be viewed and converted

CODE

```
# View the data type
print(step_counts.dtypes)

# Convert to a float
step_counts = step_counts.astype(np.float)

# View the data type
print(step_counts.dtypes)
```

OUTPUT

```
>>> int64
```

PANDAS DATA TYPES AND IMPUTATION

Data types can be viewed and converted

CODE

```
# View the data type
print(step_counts.dtypes)

# Convert to a float
step_counts = step_counts.astype(np.float)

# View the data type
print(step_counts.dtypes)
```

OUTPUT

```
>>> int64

>>> float64
```

PANDAS DATA TYPES AND IMPUTATION

Invalid data points can be easily filled with values

CODE

```
# Create invalid data
step_counts[1:3] = np.NaN

# Now fill it in with zeros
step_counts = step_counts.fillna(0.)
# equivalently,
# step_counts.fillna(0., inplace=True)

print(step_counts[1:3])
```

OUTPUT

PANDAS DATA TYPES AND IMPUTATION

Invalid data points can be easily filled with values

CODE

```
# Create invalid data
step_counts[1:3] = np.NaN

# Now fill it in with zeros
step_counts = step_counts.fillna(0.)
# equivalently,
# step_counts.fillna(0., inplace=True)

print(step_counts[1:3])
```

OUTPUT

```
>>> 2015-03-30 0.0
    2015-03-31 0.0
    Freq: D, Name: steps,
    dtype: float64
```

PANDAS DATAFRAME CREATION AND METHODS

DataFrames can be created from lists, dictionaries, and Pandas Series

CODE	OUTPUT

```
# Cycling distance
cycling_data = [10.7, 0, None, 2.4, 15.3,
               10.9, 0, None]

# Create a tuple of data
joined_data = list(zip(step_data,
                       cycling_data))

# The dataframe
activity_df = pd.DataFrame(joined_data)

print(activity_df)
```

PANDAS DATAFRAME CREATION AND METHODS

DataFrames can be created from lists, dictionaries, and Pandas Series

CODE

```
# Cycling distance
cycling_data = [10.7, 0, None, 2.4, 15.3,
                10.9, 0, None]

# Create a tuple of data
joined_data = list(zip(step_data,
                       cycling_data))

# The dataframe
activity_df = pd.DataFrame(joined_data)

print(activity_df)
```

OUTPUT

```
>>>
```

	0	1
0	3620	10.7
1	7891	0.0
2	9761	NaN
3	3907	2.4
4	4338	15.3
5	5373	10.9

PANDAS DATAFRAME CREATION AND METHODS

Labeled columns and an index can be added

CODE

```
# Add column names to dataframe
activity_df = pd.DataFrame(joined_data,
                index=pd.date_range('20150329',
                periods=6),
                columns=['Walking','Cycling'])

print(activity_df)
```

OUTPUT

PANDAS DATAFRAME CREATION AND METHODS

Labeled columns and an index can be added

CODE

```
# Add column names to dataframe
activity_df = pd.DataFrame(joined_data,
              index=pd.date_range('20150329',
              periods=6),
              columns=['Walking','Cycling'])

print(activity_df)
```

OUTPUT

>>>

	Walking	Cycling
2015-03-29	3620	10.7
2015-03-30	7891	0.0
2015-03-31	9761	NaN
2015-04-01	3907	2.4
2015-04-02	4338	15.3
2015-04-03	5373	10.9

INDEXING DATAFRAME ROWS

DataFrame rows can be indexed by row using the 'loc' and 'iloc' methods

CODE

```
# Select row of data by index name
print(activity_df.loc['2015-04-01'])
```

OUTPUT

INDEXING DATAFRAME ROWS

DataFrame rows can be indexed by row using the 'loc' and 'iloc' methods

CODE

```
# Select row of data by index name
print(activity_df.loc['2015-04-01'])
```

OUTPUT

```
>>> Walking 3907.0
    Cycling 2.4
    Name: 2015-04-01,
    dtype: float64
```

INDEXING DATAFRAME ROWS

DataFrame rows can be indexed by row using the 'loc' and 'iloc' methods

CODE

```
# Select row of data by integer position
print(activity_df.iloc[-3])
```

OUTPUT

INDEXING DATAFRAME ROWS

DataFrame rows can be indexed by row using the 'loc' and 'iloc' methods

CODE

```
# Select row of data by integer position
print(activity_df.iloc[-3])
```

OUTPUT

```
>>> Walking 3907.0
    Cycling 2.4
    Name: 2015-04-01,
    dtype: float64
```

INDEXING DATAFRAME COLUMNS

DataFrame columns can be indexed by name

CODE

```
# Name of column
print(activity_df['Walking'])
```

OUTPUT

INDEXING DATAFRAME COLUMNS

DataFrame columns can be indexed by name

CODE

```
# Name of column
print(activity_df['Walking'])
```

OUTPUT

```
>>> 2015-03-29 3620
    2015-03-30 7891
    2015-03-31 9761
    2015-04-01 3907
    2015-04-02 4338
    2015-04-03 5373
    Freq: D, Name: Walking,
    dtype: int64
```

INDEXING DATAFRAME COLUMNS

DataFrame columns can also be indexed as properties

CODE

```
# Object-oriented approach
print(activity_df.Walking)
```

OUTPUT

INDEXING DATAFRAME COLUMNS

DataFrame columns can also be indexed as properties

CODE

```python
# Object-oriented approach
print(activity_df.Walking)
```

OUTPUT

```
>>>  2015-03-29 3620
     2015-03-30 7891
     2015-03-31 9761
     2015-04-01 3907
     2015-04-02 4338
     2015-04-03 5373
     Freq: D, Name: Walking,
     dtype: int64
```

INDEXING DATAFRAME COLUMNS

DataFrame columns can be indexed by integer

CODE

```
# First column
print(activity_df.iloc[:,0])
```

OUTPUT

INDEXING DATAFRAME COLUMNS

DataFrame columns can be indexed by integer

CODE

```
# First column
print(activity_df.iloc[:,0])
```

OUTPUT

```
>>>  2015-03-29 3620
     2015-03-30 7891
     2015-03-31 9761
     2015-04-01 3907
     2015-04-02 4338
     2015-04-03 5373
     Freq: D, Name: Walking,
     dtype: int64
```

READING DATA WITH PANDAS

CSV and other common filetypes can be read with a single command

CODE

```
# The location of the data file
filepath = 'data/Iris_Data/Iris_Data.csv'

# Import the data
data = pd.read_csv(filepath)

# Print a few rows
print(data.iloc[:5])
```

OUTPUT

READING DATA WITH PANDAS

CSV and other common filetypes can be read with a single command

CODE

```python
# The location of the data file
filepath = 'data/Iris_Data/Iris_Data.csv'

# Import the data
data = pd.read_csv(filepath)

# Print a few rows
print(data.iloc[:5])
```

OUTPUT

>>>

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

ASSIGNING NEW DATA TO A DATAFRAME

Data can be (re)assigned to a DataFrame column

CODE

```
# Create a new column that is a product
# of both measurements
data['sepal_area'] = data.sepal_length
                     *data.sepal_width

# Print a few rows and columns
print(data.iloc[:5, -3:])
```

OUTPUT

ASSIGNING NEW DATA TO A DATAFRAME

Data can be (re)assigned to a DataFrame column

CODE

```
# Create a new column that is a product
# of both measurements
data['sepal_area'] = data.sepal_length
                     *data.sepal_width

# Print a few rows and columns
print(data.iloc[:5, -3:])
```

OUTPUT

>>>

	petal_width	species	sepal_area
0	0.2	Iris-setosa	17.85
1	0.2	Iris-setosa	14.70
2	0.2	Iris-setosa	15.04
3	0.2	Iris-setosa	14.26
4	0.2	Iris-setosa	18.00

APPLYING A FUNCTION TO A DATAFRAME COLUMN

Functions can be applied to columns or rows of a DataFrame or Series

CODE

```
# The lambda function applies what
# follows it to each row of data
data['abbrev'] = (data
                  .species
                  .apply(lambda x:
                  x.replace('Iris-','')))

# Note that there are other ways to
# accomplish the above

print(data.iloc[:5, -3:])
```

OUTPUT

APPLYING A FUNCTION TO A DATAFRAME COLUMN

Functions can be applied to columns or rows of a DataFrame or Series

CODE

```
# The lambda function applies what
# follows it to each row of data
data['abbrev'] = (data
                  .species
                  .apply(lambda x:
                  x.replace('Iris-','')))

# Note that there are other ways to
# accomplish the above

print(data.iloc[:5, -3:])
```

OUTPUT

```
>>>
```

	petal_width	species	abbrev
0	0.2	Iris-setosa	setosa
1	0.2	Iris-setosa	setosa
2	0.2	Iris-setosa	setosa
3	0.2	Iris-setosa	setosa
4	0.2	Iris-setosa	setosa

CONCATENATING TWO DATAFRAMES

Two DataFrames can be concatenated along either dimension

CODE

```
# Concatenate the first two and
# last two rows
small_data = pd.concat([data.iloc[:2],
                        data.iloc[-2:]])



print(small_data.iloc[:,-3:])

# See the 'join' method for
# SQL style joining of dataframes
```

OUTPUT

CONCATENATING TWO DATAFRAMES

Two DataFrames can be concatenated along either dimension

CODE

```
# Concatenate the first two and
# last two rows
small_data = pd.concat([data.iloc[:2],
                        data.iloc[-2:]])



print(small_data.iloc[:,-3:])

# See the 'join' method for
# SQL style joining of dataframes
```

OUTPUT

>>>

	petal_length	petal_width	species
0	1.4	0.2	Iris-setosa
1	1.4	0.2	Iris-setosa
148	5.4	2.3	Iris-virginica
149	5.1	1.8	Iris-virginica

AGGREGATED STATISTICS WITH GROUPBY

Using the groupby method calculated aggregated DataFrame statistics

CODE

```
# Use the size method with a
# DataFrame to get count
# For a Series, use the .value_counts
# method
group_sizes = (data
               .groupby('species')
               .size())

print(group_sizes)
```

OUTPUT

AGGREGATED STATISTICS WITH GROUPBY

Using the groupby method calculated aggregated DataFrame statistics

CODE

```
# Use the size method with a
# DataFrame to get count
# For a Series, use the .value_counts
# method
group_sizes = (data
               .groupby('species')
               .size())

print(group_sizes)
```

OUTPUT

```
>>>  species
     Iris-setosa         50
     Iris-versicolor     50
     Iris-virginica      50
     dtype: int64
```

PERFORMING STATISTICAL CALCULATIONS

Pandas contains a variety of statistical methods—mean, median, and mode

CODE

```
# Mean calculated on a DataFrame
print(data.mean())
```

OUTPUT

PERFORMING STATISTICAL CALCULATIONS

Pandas contains a variety of statistical methods—mean, median, and mode

CODE

```
# Mean calculated on a DataFrame
print(data.mean())
```

OUTPUT

```
>>> sepal_length 5.843333
    sepal_width 3.054000
    petal_length 3.758667
    petal_width 1.198667
    dtype: float64
```

PERFORMING STATISTICAL CALCULATIONS

Pandas contains a variety of statistical methods—mean, median, and mode

CODE

```
# Mean calculated on a DataFrame
print(data.mean())




# Median calculated on a Series
print(data.petal_length.median())
```

OUTPUT

```
>>> sepal_length 5.843333
    sepal_width 3.054000
    petal_length 3.758667
    petal_width 1.198667
    dtype: float64

>>> 4.35
```

PERFORMING STATISTICAL CALCULATIONS

Pandas contains a variety of statistical methods—mean, median, and mode

CODE	OUTPUT

`# Mean calculated on a DataFrame`
`print(data.mean())`
	`>>> sepal_length 5.843333`
	`sepal_width 3.054000`
	`petal_length 3.758667`
	`petal_width 1.198667`
	`dtype: float64`

`# Median calculated on a Series` `print(data.petal_length.median())`	`>>> 4.35`
`# Mode calculated on a Series` `print(data.petal_length.mode())`	`>>> 0 1.5` `dtype: float64`

PERFORMING STATISTICAL CALCULATIONS

Standard deviation, variance, SEM, and quantiles can also be calculated

CODE

```
# Standard dev, variance, and SEM
print(data.petal_length.std(),
      data.petal_length.var(),
      data.petal_length.sem())
```

OUTPUT

PERFORMING STATISTICAL CALCULATIONS

Standard deviation, variance, SEM, and quantiles can also be calculated

CODE

```
# Standard dev, variance, and SEM
print(data.petal_length.std(),
      data.petal_length.var(),
      data.petal_length.sem())
```

OUTPUT

```
>>> 1.76442041995
    3.11317941834
    0.144064324021
```

PERFORMING STATISTICAL CALCULATIONS

Standard deviation, variance, SEM, and quantiles can also be calculated

CODE

```
# Standard dev, variance, and SEM
print(data.petal_length.std(),
      data.petal_length.var(),
      data.petal_length.sem())

# As well as quantiles
print(data.quantile(0))
```

OUTPUT

```
>>> 1.76442041995
    3.11317941834
    0.144064324021


>>> sepal_length 4.3
    sepal_width 2.0
    petal_length 1.0
    petal_width 0.1
    Name: 0, dtype: float64
```

PERFORMING STATISTICAL CALCULATIONS

Multiple calculations can be presented in a DataFrame

CODE

```
print(data.describe())
```

OUTPUT

PERFORMING STATISTICAL CALCULATIONS

Multiple calculations can be presented in a DataFrame

CODE

```
print(data.describe())
```

OUTPUT

>>>

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

SAMPLING FROM DATAFRAMES

DataFrames can be randomly sampled from

CODE

```
# Sample 5 rows without replacement
sample = (data
         .sample(n=5,
                 replace=False,
                 random_state=42))



print(sample.iloc[:,-3:])
```

OUTPUT

SAMPLING FROM DATAFRAMES

DataFrames can be randomly sampled from

CODE

```
# Sample 5 rows without replacement
sample = (data
          .sample(n=5,
                  replace=False,
                  random_state=42))


print(sample.iloc[:,-3:])
```

OUTPUT

>>>

	petal_length	petal_width	species
73	4.7	1.2	Iris-versicolor
18	1.7	0.3	Iris-setosa
118	6.9	2.3	Iris-virginica
78	4.5	1.5	Iris-versicolor
76	4.8	1.4	Iris-versicolor

SAMPLING FROM DATAFRAMES

DataFrames can be randomly sampled from

CODE

```
# Sample 5 rows without replacement
sample = (data
          .sample(n=5,
                  replace=False,
                  random_state=42))


print(sample.iloc[:,-3:])
```

OUTPUT

>>>

	petal_length	petal_width	species
73	4.7	1.2	Iris-versicolor
18	1.7	0.3	Iris-setosa
118	6.9	2.3	Iris-virginica
78	4.5	1.5	Iris-versicolor
76	4.8	1.4	Iris-versicolor

SciPy and NumPy also contain a variety of statistical functions.

VISUALIZATION LIBRARIES

VISUALIZATION LIBRARIES

Visualizations can be created in multiple ways:

- Matplotlib
- Pandas (via Matplotlib)
- Seaborn
  - Statistically-focused plotting methods
  - Global preferences incorporated by Matplotlib

BASIC SCATTER PLOTS WITH MATPLOTLIB

Scatter plots can be created from Pandas Series

CODE

```
Import matplotlib.pyplot as plt


plt.plot(data.sepal_length,
         data.sepal_width,
         ls ='', marker='o')
```

OUTPUT

BASIC SCATTER PLOTS WITH MATPLOTLIB

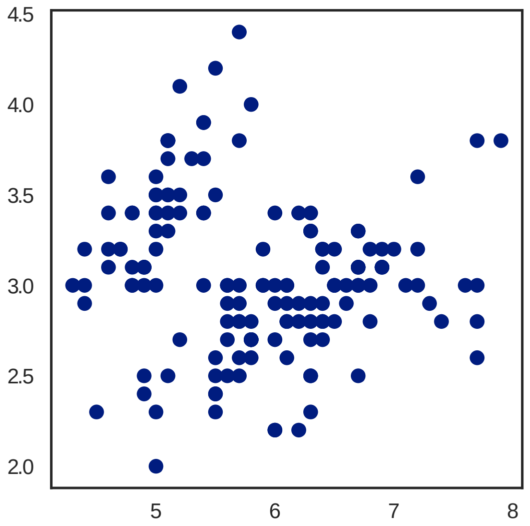
Scatter plots can be created from Pandas Series

CODE

```
Import matplotlib.pyplot as plt


plt.plot(data.sepal_length,
         data.sepal_width,
         ls ='', marker='o')
```

OUTPUT



BASIC SCATTER PLOTS WITH MATPLOTLIB

Multiple layers of data can also be added

CODE

```
plt.plot(data.sepal_length,
         data.sepal_width,
         ls ='', marker='o',
         label='sepal')

plt.plot(data.petal_length,
         data.petal_width,
         ls ='', marker='o',
         label='petal')
```

OUTPUT

BASIC SCATTER PLOTS WITH MATPLOTLIB

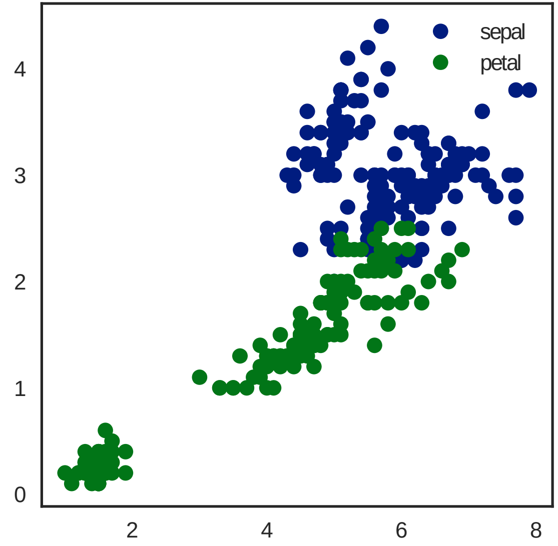
Multiple layers of data can also be added

CODE

```
plt.plot(data.sepal_length,
         data.sepal_width,
         ls ='', marker='o',
         label='sepal')

plt.plot(data.petal_length,
         data.petal_width,
         ls ='', marker='o',
         label='petal')
```

OUTPUT



HISTOGRAMS WITH MATPLOTLIB

Histograms can be created from Pandas Series

CODE

```
plt.hist(data.sepal_length, bins=25)
```

OUTPUT

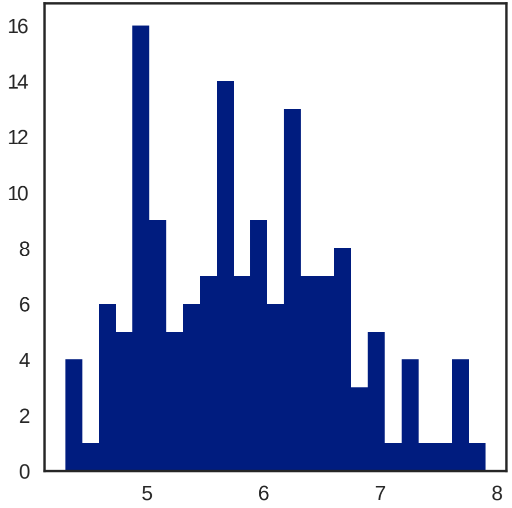
HISTOGRAMS WITH MATPLOTLIB

Histograms can be created from Pandas Series

CODE

```
plt.hist(data.sepal_length, bins=25)
```

OUTPUT



CUSTOMIZING MATPLOTLIB PLOTS

Every feature of Matplotlib plots can be customized

CODE

```
fig, ax = plt.subplots()

ax.barh(np.arange(10),
        data.sepal_width.iloc[:10])

# Set position of ticks and tick labels
ax.set_yticks(np.arange(0.4,10.4,1.0))
ax.set_yticklabels(np.arange(1,11))
ax.set(xlabel='xlabel', ylabel='ylabel',
       title='Title')
```

OUTPUT

CUSTOMIZING MATPLOTLIB PLOTS

Every feature of Matplotlib plots can be customized

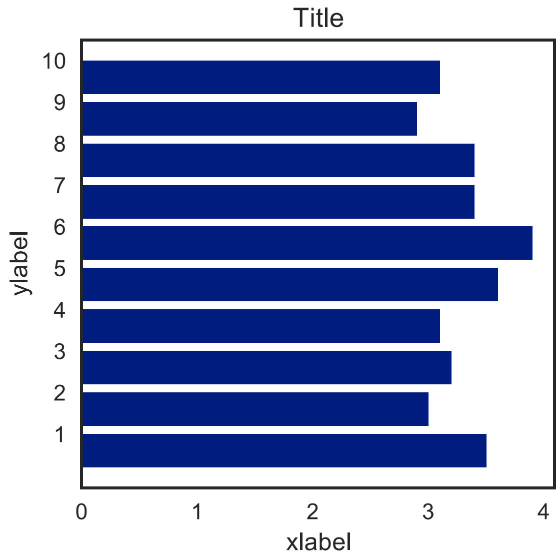
CODE

```
fig, ax = plt.subplots()

ax.barh(np.arange(10),
        data.sepal_width.iloc[:10])

# Set position of ticks and tick labels
ax.set_yticks(np.arange(0.4,10.4,1.0))
ax.set_yticklabels(np.arange(1,11))
ax.set(xlabel='xlabel', ylabel='ylabel',
       title='Title')
```

OUTPUT



INCORPORATING STATISTICAL CALCULATIONS

Statistical calculations can be included with Pandas methods

CODE

```
(data
 .groupby('species')
 .mean()
 .plot(color=['red','blue’,
                'black','green’],
       fontsize=10.0, figsize=(4,4)))
```

OUTPUT

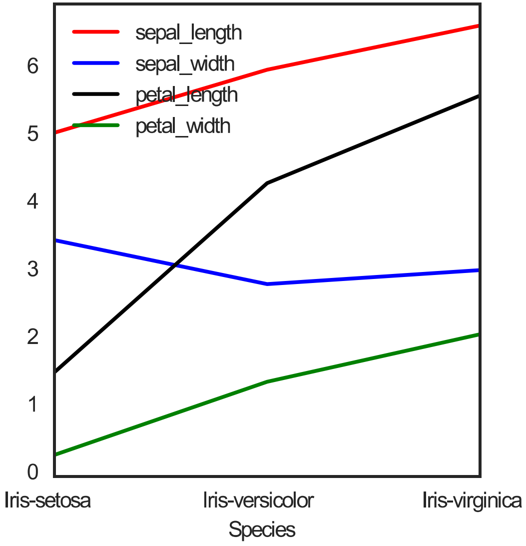
INCORPORATING STATISTICAL CALCULATIONS

Statistical calculations can be included with Pandas methods

CODE

```
(data
 .groupby('species')
 .mean()
 .plot(color=['red','blue',
                 'black','green'],
        fontsize=10.0, figsize=(4,4)))
```

OUTPUT



STATISTICAL PLOTTING WITH SEABORN

Joint distribution and scatter plots can be created

CODE

```
import seaborn as sns

sns.jointplot(x='sepal_length',
              y='sepal_width',
              data=data, size=4)
```

OUTPUT

STATISTICAL PLOTTING WITH SEABORN

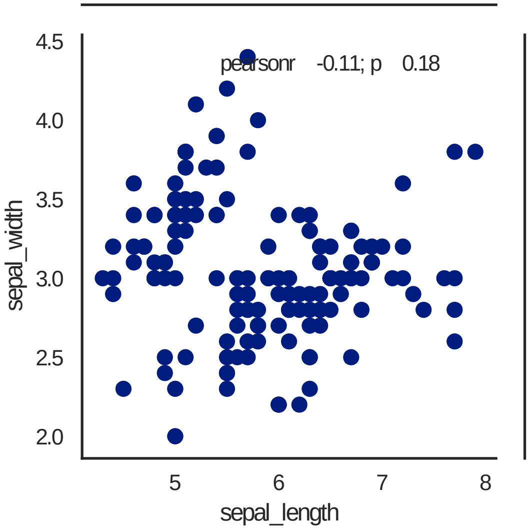
Joint distribution and scatter plots can be created

CODE

```
import seaborn as sns

sns.jointplot(x='sepal_length',
              y='sepal_width',
              data=data, size=4)
```

OUTPUT



STATISTICAL PLOTTING WITH SEABORN

Correlation plots of all variable pairs can also be made with Seaborn

CODE

```
sns.pairplot(data, hue='species', size=3)
```

OUTPUT

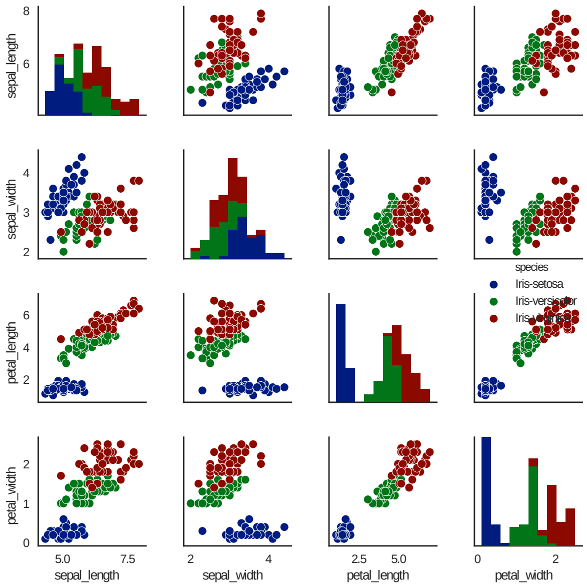
STATISTICAL PLOTTING WITH SEABORN

Correlation plots of all variable pairs can also be made with Seaborn

CODE

```
sns.pairplot(data, hue='species', size=3)
```

OUTPUT



intel Nervana™ AI Academy