Introduction to Machine Learning and Toolkit

Legal Notices and Disclaimers

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

Audience includes:

- University level professors who may wish to use this content in their courses
- University level students or others who want to prepare for using machine learning and applying machine learning principles to data

Overview of Course

Prerequisites:

- Python* programming
- Calculus
- Linear algebra
- Statistics

Overview of Course

Each week:

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

Total Time: 12 weeks of lectures and exercises. Each week requires three hours to complete.

Our Toolset: Intel® oneAPI AI Analytics Toolkit (AI Kit)

- Intel® Extension for Scikit-learn*

Learning Objectives

- Demonstrate supervised learning algorithms
- Explain key concepts like under- and over-fitting, regularization, and cross-validation
- Classify the type of problem to be solved, choose the right algorithm, tune parameters, and validate a model
- Apply Intel® Extension for Scikit-learn* to leverage underlying compute capabilities of hardware

Our Toolset: Intel® oneAPI AI Analytics Toolkit (AI Kit)

Installation options

https://software.intel.com/ai

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

Our Toolset: Intel® Distribution for Python

Installation options

https://software.intel.com/ai

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® oneAPI AI Analytics Toolkit (AI Kit)

- **Jupyter notebooks:** interactive coding and visualization of output
- **NumPy, SciPy, Pandas:** numerical computation
- **Matplotlib, Seaborn:** data visualization
- **Scikit-learn:** machine learning

Our Toolset: Intel® oneAPI AI Analytics Toolkit (AI Kit)

- **Jupyter notebooks:** interactive coding and visualization of output

Week 1

- **NumPy, SciPy, Pandas:** numerical computation
- **Matplotlib, Seaborn:** data visualization
- **Scikit-learn:** machine learning

Our Toolset: Intel® oneAPI AI Analytics Toolkit (AI Kit)

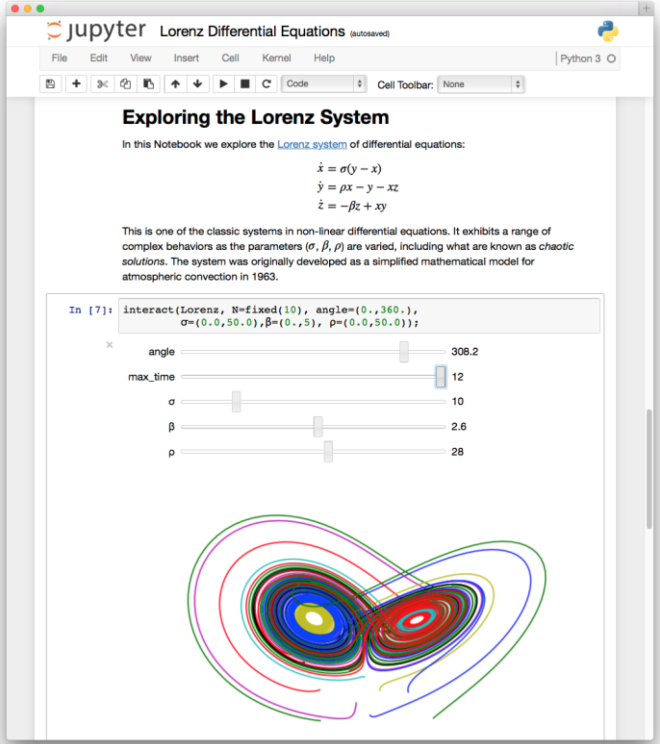
- **Jupyter notebooks:** interactive coding and visualization of output
- **NumPy, SciPy, Pandas:** numerical computation
- **Matplotlib, Seaborn:** data visualization
- **Scikit-learn:** machine learning

Weeks 2 – 12

Introduction to Jupyter Notebook

- Polyglot analysis environment—blends multiple languages

S:



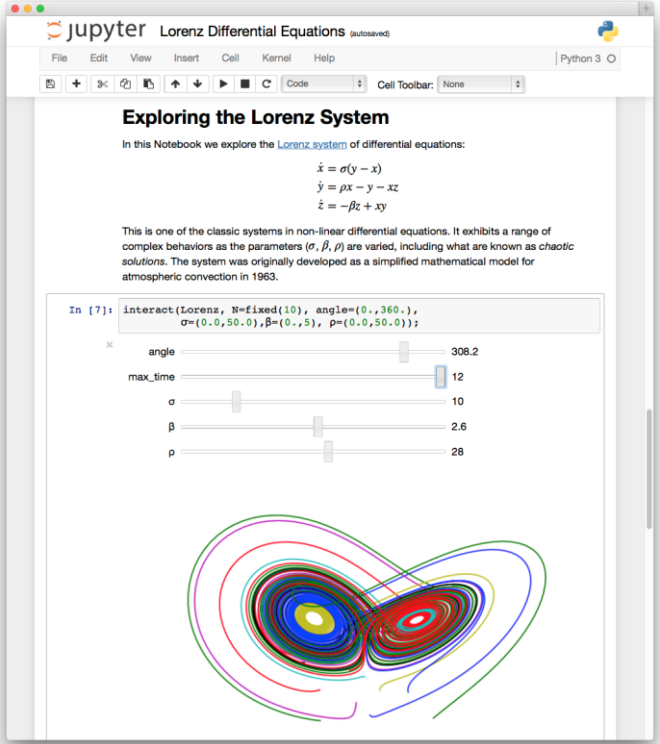
Source: http://jupyter.org/

Introduction to Jupyter Notebook

- Polyglot analysis environment—blends multiple languages

- Jupyter is an anagram of: Julia, Python, and R

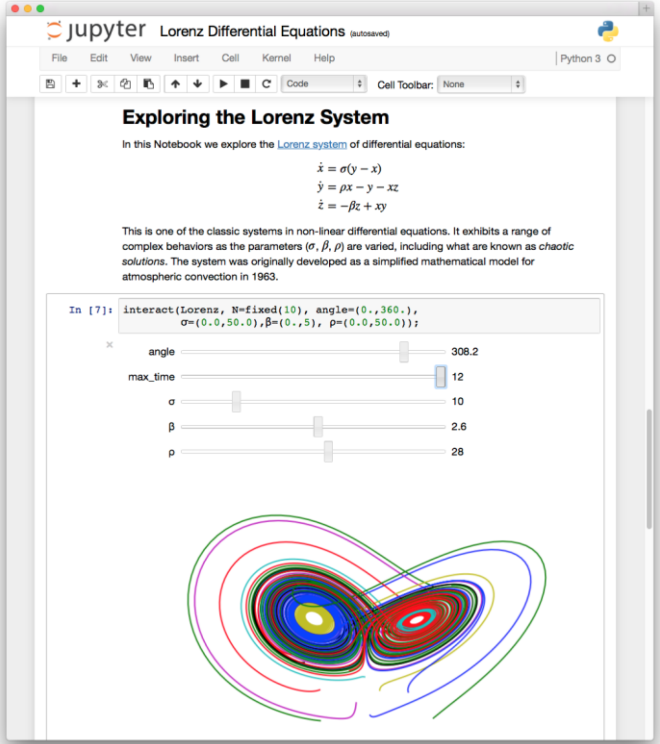
S:



Source: http://jupyter.org/

Introduction to Jupyter Notebook

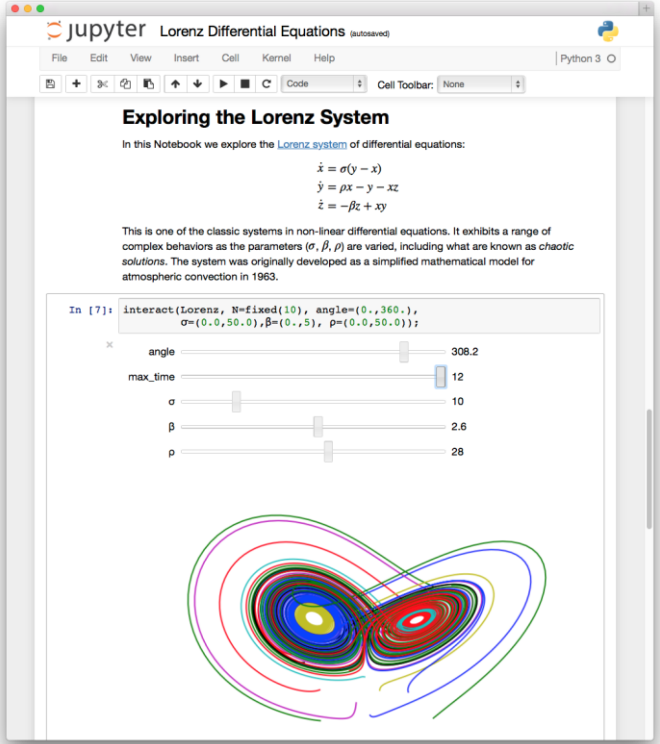
- Polyglot analysis environment—blends multiple languages
- Jupyter is an anagram of: Julia, Python, and R
- Supports multiple content types: code, narrative text, images, movies, etc.



Source: http://jupyter.org/

Introduction to Jupyter Notebook

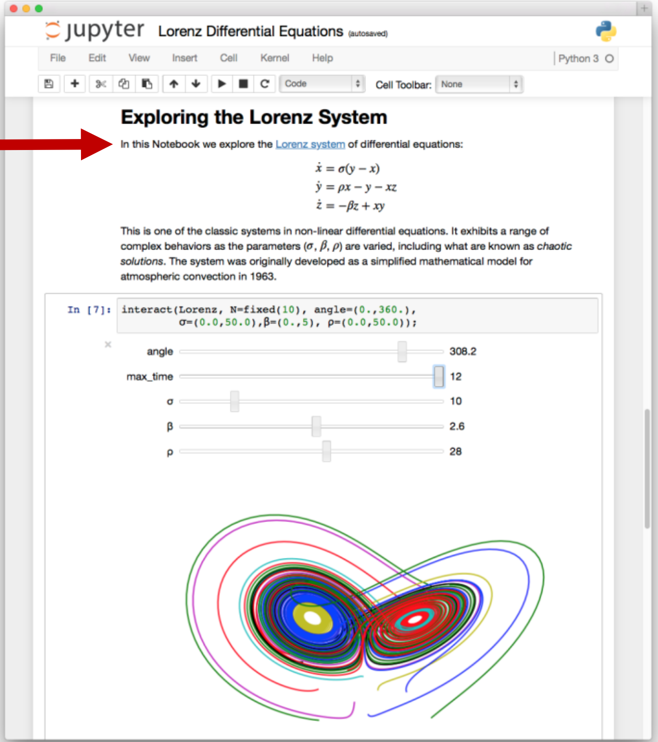
- HTML & Markdown
- LaTeX (equations)
- Code



Source: http://jupyter.org/

Introduction to Jupyter Notebook

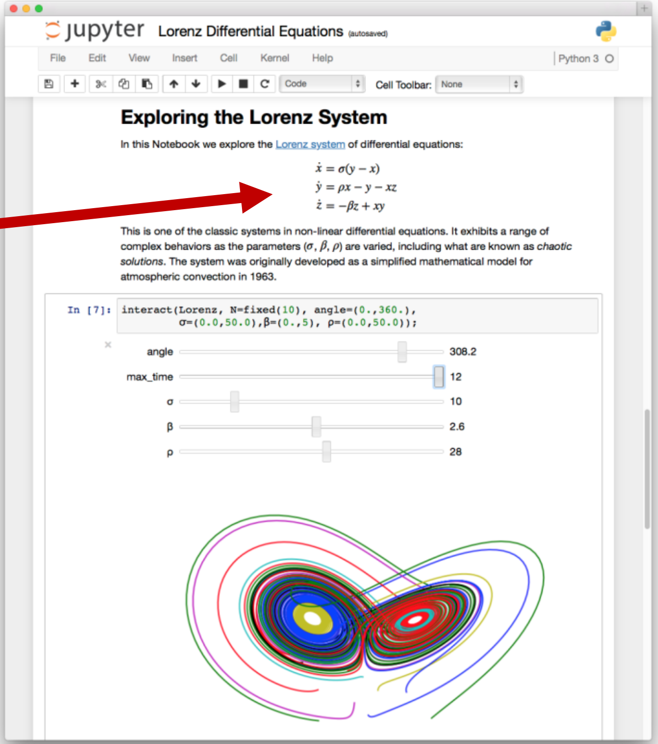
- HTML & Markdown
- LaTeX (equations)
- Code



Source: http://jupyter.org/

Introduction to Jupyter Notebook

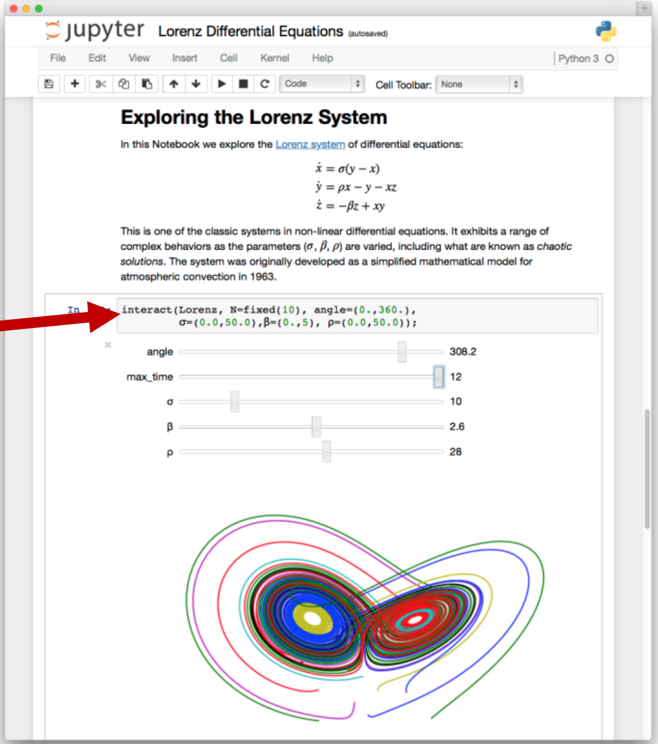
- HTML & Markdown
- LaTeX (equations)
- Code



Source: http://jupyter.org/

Introduction to Jupyter Notebook

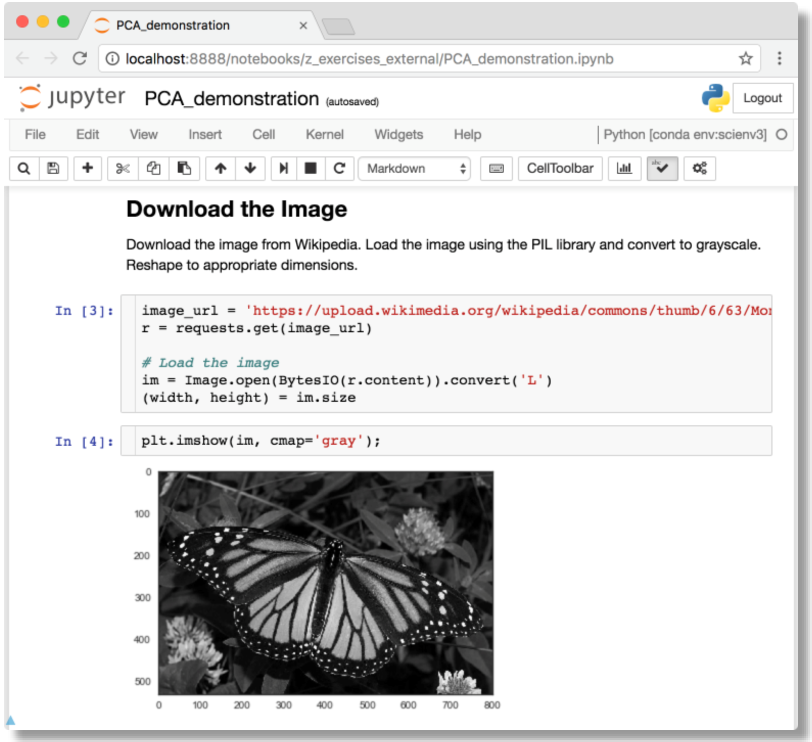
- HTML & Markdown
- LaTeX (equations)
- Code



Source: http://jupyter.org/

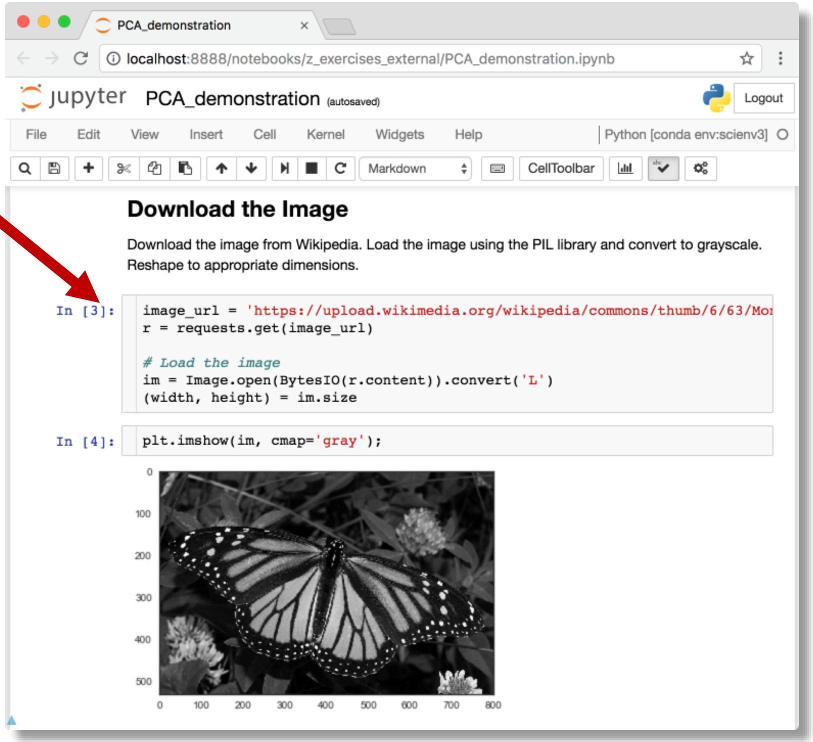
Introduction to Jupyter Notebook

- 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

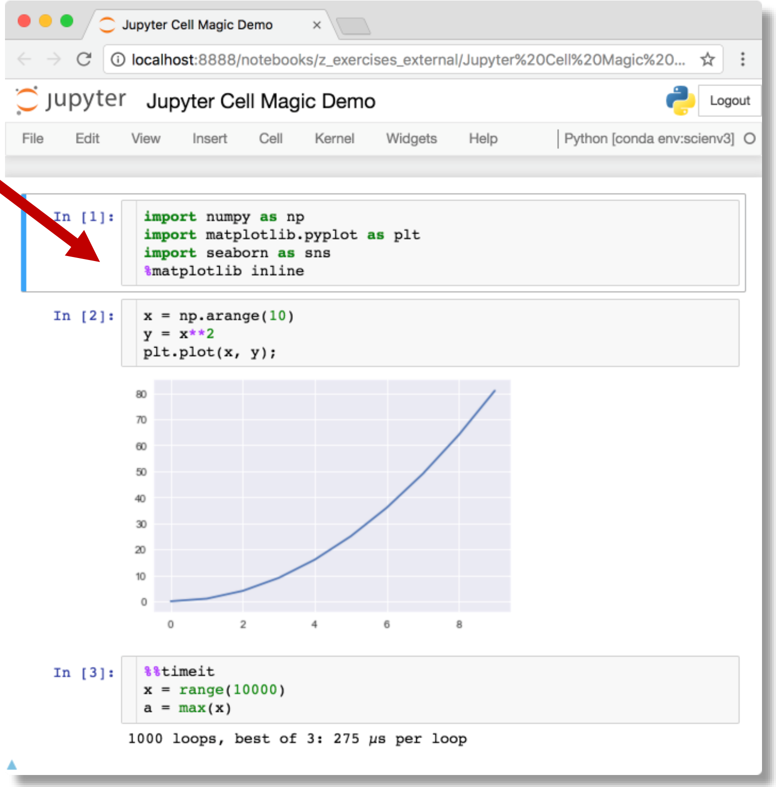


Jupyter Cell Magics

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

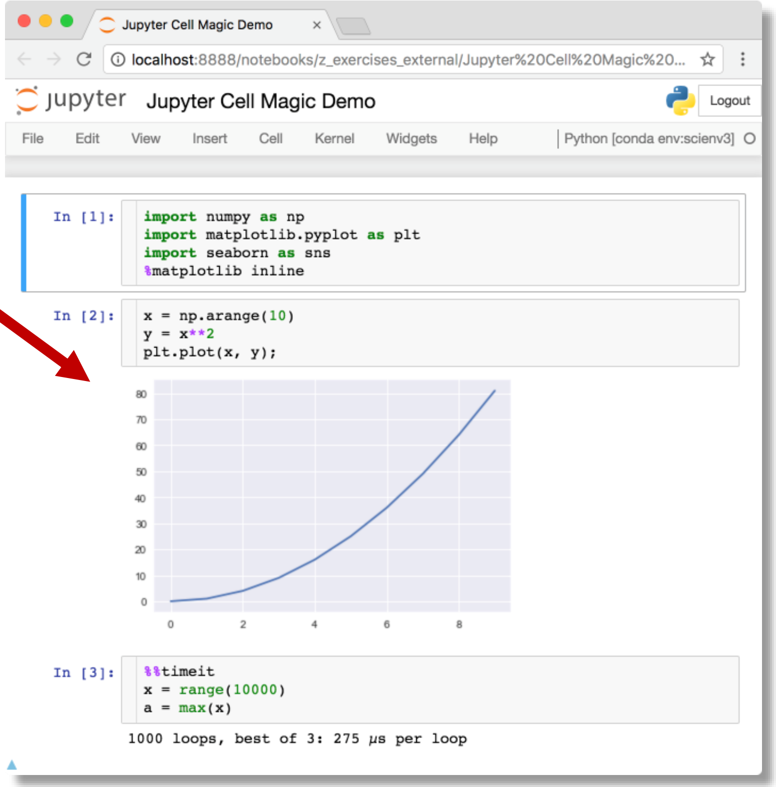
Jupyter Cell Magics

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



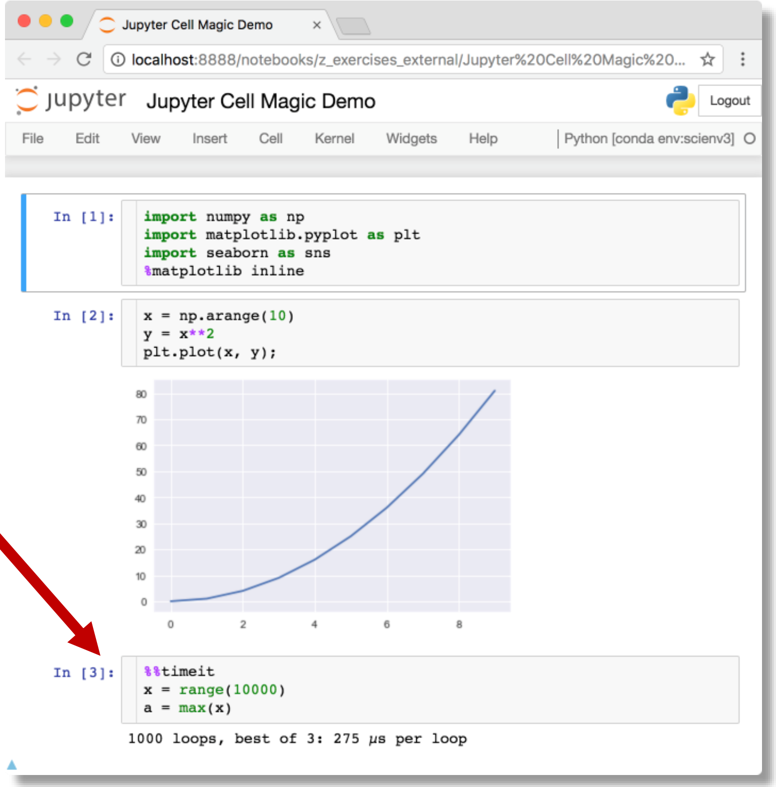
Jupyter Cell Magics

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



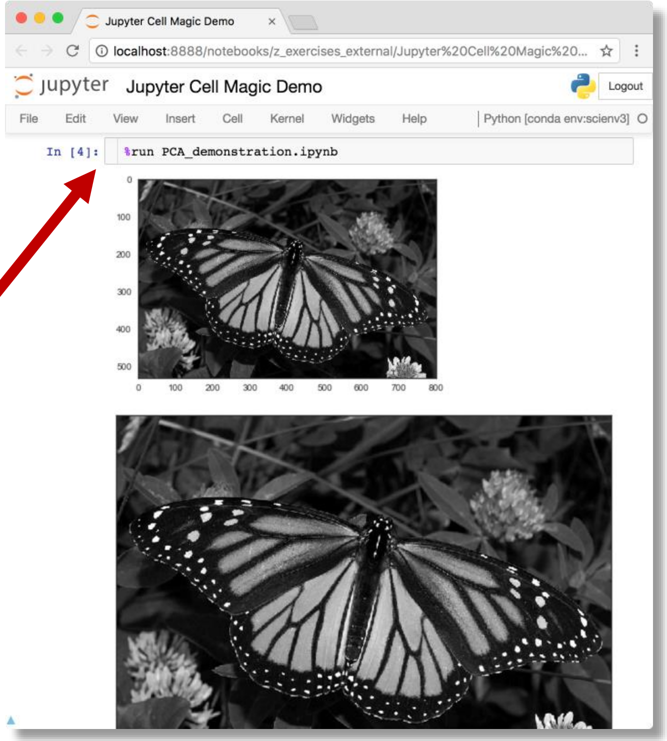
Jupyter Cell Magics

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



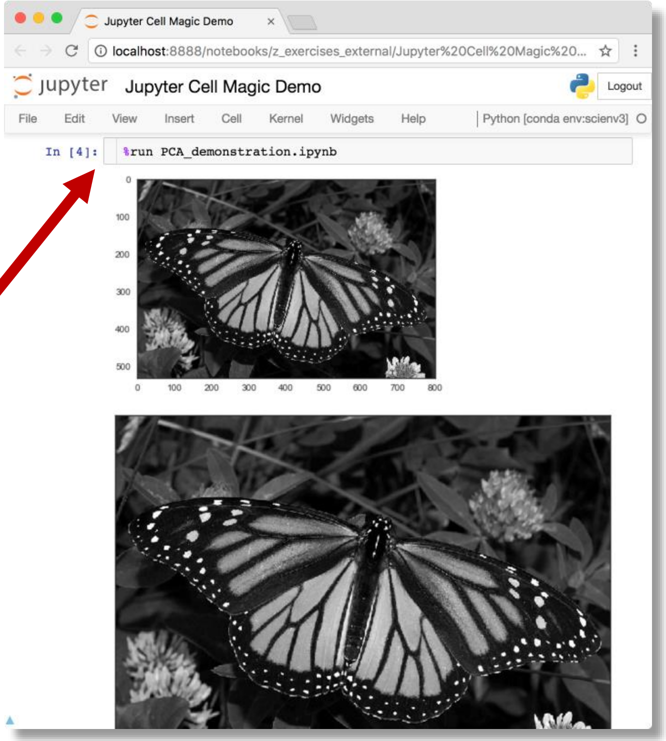
Jupyter Cell Magics

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

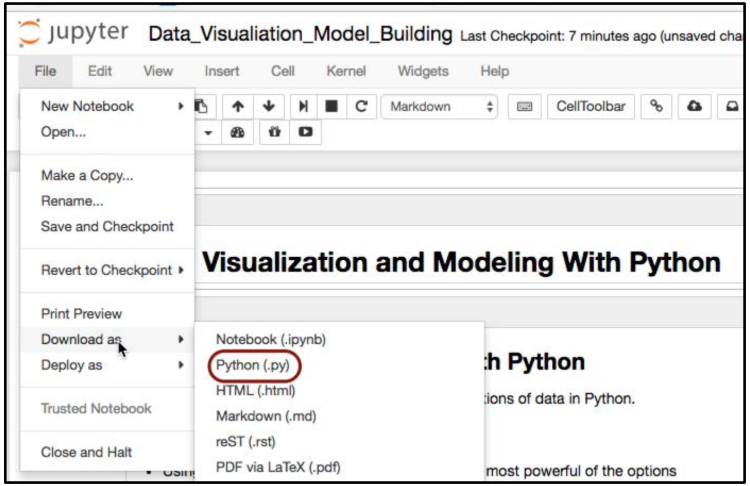
Making Jupyter Notebooks Reusable

To extract Python code from a Jupyter notebook:

Convert from Command Line

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

Export from Notebook

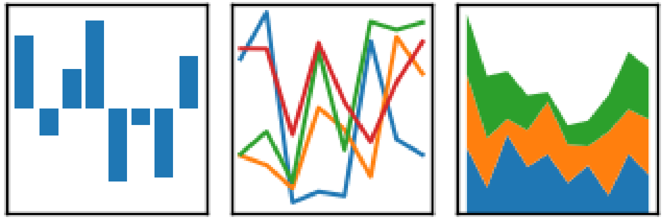


Introduction to Pandas

- 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

Output

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


print(step_counts)
```

Software

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

Output

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

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

```
# 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

>>>  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	Output
`# View the data type` `print(step_counts.dtypes)`	`>>> int64`
`# Convert to a float` `step_counts = step_counts.astype(np.float)`	
`# View the data type` `print(step_counts.dtypes)`	`>>> 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

```python
# 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

Output

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

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

Output

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

Indexing DataFrame Columns

DataFrame columns can be indexed by name

Code

```python
# 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

Output

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

Indexing DataFrame Columns

DataFrame columns can also be indexed as properties

Code

```
# 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

Output

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

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

Output

```
# 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])
```

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

>>>

	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

Output

```
# 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:])
```

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

Output

```
# 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:])
```

intel Software

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

Output

```
# 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)
```

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	Output

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

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

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

```python
# 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

Output

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

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

Output

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

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

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

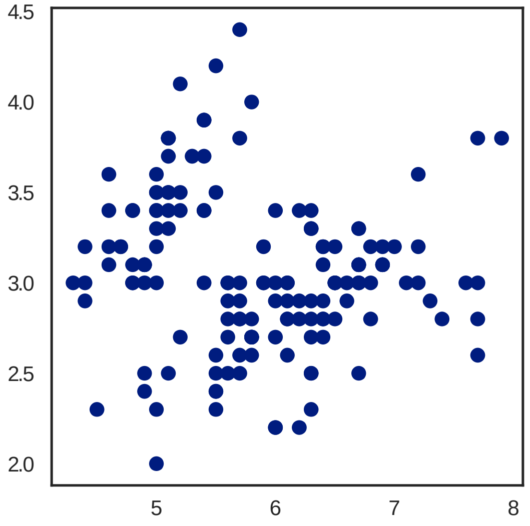
Scatter plots can be created from Pandas Series

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

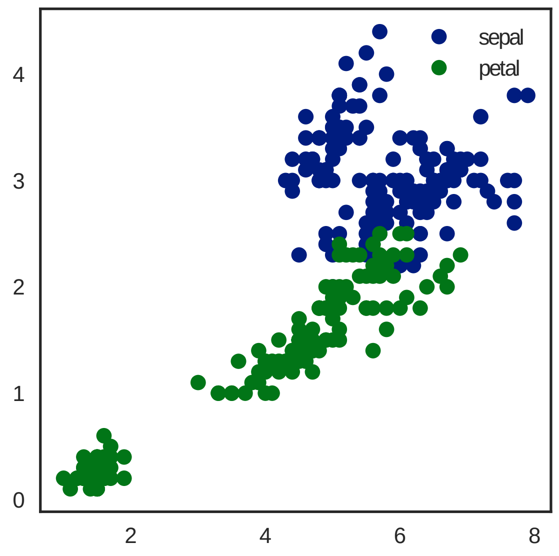
Multiple layers of data can also be added

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

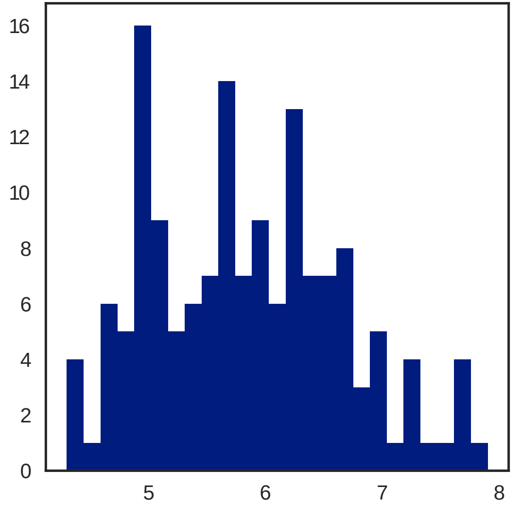
Histograms with Matplotlib

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

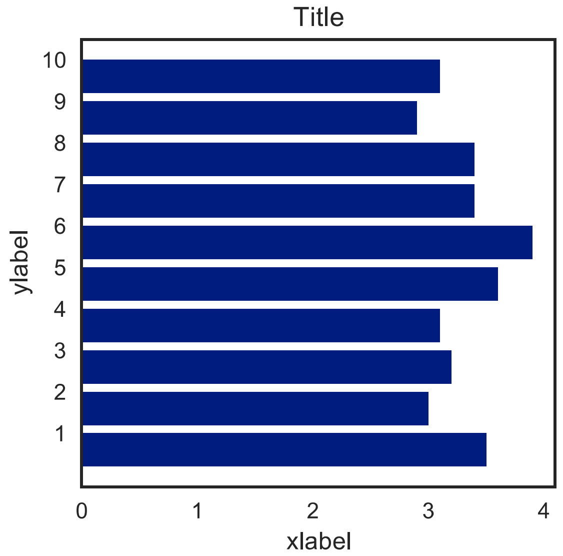
Code

```
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

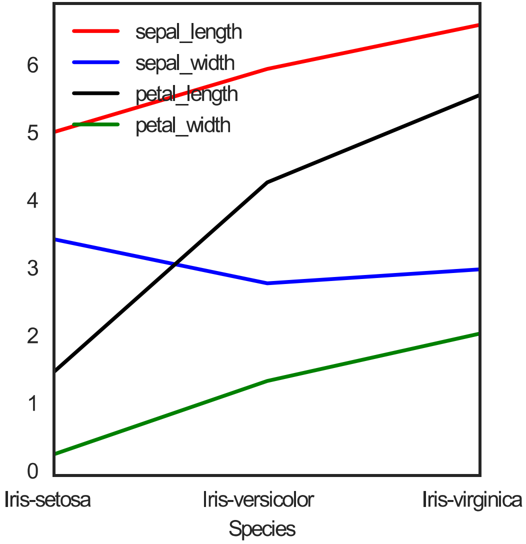
Incorporating Statistical Calculations

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

Output

```
import seaborn as sns

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

Statistical Plotting with Seaborn

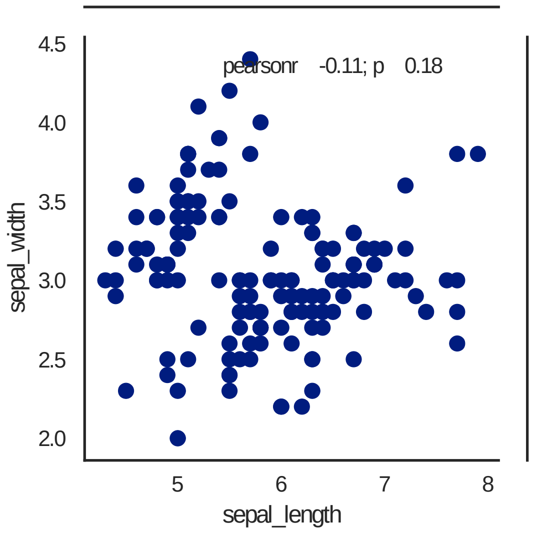
Joint distribution and scatter plots can be created

Code

```python
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

Output

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

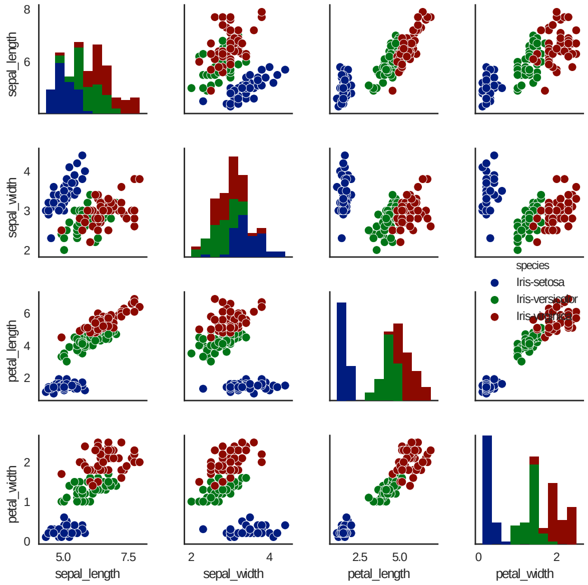
Statistical Plotting with Seaborn

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

Code

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

Output



intel®

Software