Python for Data Analysis

Tour de Python Level 2 ●○○○

- Python `stdlib`
- `import` syntax
- Python packages

built-in Python

- Everything we've talked about so far is referred to as part of the Python "built-in"s
- Every Python session has access to everything we've learned no matter what
- The built-ins are general purpose building blocks: "primitive" data types (like strings, integers, dictionaries), control flow statements, basic operators, etc

moving on (briefly) to the Python `stdlib`

- every Python installation also comes with special data types, operators, functions, and methods to address specific types of problems
  - ex. `datetime` for storing time data that are cognizant of year/month/day/timezone
- by default these are not loaded into each Python session, but instead have to be **imported**
- `stdlib` = "standard library"

Python modules

- any `.py` file can also be referred to as a "Python module"
- modules can be imported using one of four styles of import syntax. here's one of them:

In [1]:
```
import math
```

- the list of modules already accessible to any vanilla Python installation because they are in the `stdlib` are listed online at [https://docs.python.org/3/library/ (https://docs.python.org/3/library/)](https://docs.python.org/3/library/)
- importing a module makes its code definitions accessible in whatever environment they are being imported to

Variants of import syntax and namespaces

- Python provides 3 styles of import syntax that affect the namespacing of the imported module and its members

In [2]:
```
import math
math.ceil(5)
```
Out[2]:
```
5
```

Anatomy of import syntax 1

if import syntax is:

import module_name

then call syntax is:

module_name.member_name

In [3]:
```python
import math as m
m.ceil(5)
```
Out[3]: 5

Anatomy of import syntax 2

if import syntax is:

import module_name as alias

then call syntax is:

alias.member_name

In [4]:
```
from math import ceil
ceil(5)
```
Out[4]:
```
5
```

Anatomy of import syntax 3

if import syntax is:

from module_name import member_name, ...

then call syntax is:

member_name

`stdlib` greatest hits

- `datetime`
- `random.seed`, `random.random`
- `os.path.exists`, `os.path.join`, `os.path.abspath`
- `csv.reader`, `csv.DictReader`
- `csv.writer`
- `json.loads`, `json.dumps`

Get your feet wet

In the Python interpreter, try using the 3 different styles of import syntax to import the following **functions**, and call them properly based on the type of import syntax you used. You will need to exit and re-enter your python session to clear your prior import syntax each time.

- `random.random`
- `os.getcwd`

Going past the `stdlib`

- remember: the `stdlib` is maintained by the Python Software Foundation and comes with every installation of Python
- other members of the Python community write their own extensions to the Python built-ins called **packages**
    - usually they are even more specialized than modules in the `stdlib`

Introducing our data analysis packages

- **Pandas**
    - used for processing tabular data
    - core data type is the `DataFrame`
    - port of R's DataFrame paradigm
- **Matplotlib**
    - used to generate charts such as histograms or box plots from Python data structures
    - port of MATLAB's charting functionalilty

~~Installing~~ python packages

- lucky you - you don't have to! For this class, since we used the Anaconda distribution of Python, the python packages we want to use are already installed!
  - the full list for your installation can be found at [https://docs.anaconda.com/anaconda/packages/pkg-docs (https://docs.anaconda.com/anaconda/packages/pkg-docs)](https://docs.anaconda.com/anaconda/packages/pkg-docs) ⇢
- more generally: there are many ways to find and download community-supported Python extensions, but the most popular way is via a *package manager* that downloads from PyPI at [https://pypi.python.org/pypi (https://pypi.python.org/pypi)](https://pypi.python.org/pypi) ⇢
  - popular *package managers* include `pip`, `pipenv`, and `conda`

pandas

Tour de Python Level 2 ○●○○

- `DataFrame`
- `Series`
- Python attributes
- `DataFrame` indexing
- Querying `DataFrames` with boolean series

In [5]:
```python
import pandas as pd
```

In [6]:
```python
df = pd.read_csv("iris.csv")
```

In [7]:
```python
type(df)
```

Out[7]:
```
pandas.core.frame.DataFrame
```

In [8]:
```python
df.head()
```

Out[8]:

	Sepal Length	Sepal Width	Petal Length	Petal Width	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [9]:
```
df.head(2)
```

Out[9]:

	Sepal Length	Sepal Width	Petal Length	Petal Width	Species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa

The `pandas` dataframe

- a two dimensional data structure representing tabular data
- has *columns* and *rows*
- each column's data is of the same *data type*

Creating a pandas dataframe

- use a convenience function against a file on disk
    - `pd.read_csv` (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html), for CSV data
    - `pd.read_table` (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_table.html), for general reading of tabular data, including `.tsv` files
    - `pd.read_json` (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_json.html) for JSON data
    - `pd.read_excel` (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_excel.html) for Excel files, particularly useful for excel files with many sheets
    - `pd.read_html` (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_html.html) for reading HTML `<table>s`

In [10]:
```
df = pd.read_csv("iris.csv")
```

Anatomy of a pandas dataframe convenience function

variable_name = pd.convenience_method(path_as_a_string)

- read_csv
- read_table
- read_json
- read_excel
- read_html

PS: Of course, remember that the path can be an absolute or relative path!

Creating a pandas dataframe inline

- instantiate a `dataframe` instance directly, passing it a `data` parameter with something that can be cast into a dataframe shape
- the general format for something that can be cast to a dataframe shape takes a form like: `[[row],[row],[row]]`

In [11]:
```
df_direct = pd.DataFrame(data=[["a", 1, 5.0], ["b", 2, 10.0]])
```

In [12]:
```
df_direct
```

Out[12]:

	0	1	2
0	a	1	5.0
1	b	2	10.0

Anatomy of instantiating a DataFrame directly

variable_name = pd.DataFrame(data=data_castable_to_dataframe)

In [13]:
```
df_direct_with_columns = pd.DataFrame(data=[["a", 1, 5.0], ["b", 2, 10.0]],
                                      columns=["letter", "integer", "float"])
```

In [14]:
```
df_direct_with_columns
```

Out[14]:

	letter	integer	float
0	a	1	5.0
1	b	2	10.0

Anatomy of instantiating a DataFrame directly

variable_name = pd.DataFrame(data=data_castable_to_dataframe, columns=list_of_column_names)

Python attributes

- instances of more complex data types have **attributes** associated with them
- they are accessible using the dot notation like `variable_name.attribute_name`
- these are not callable - in practical terms to us at this point, this means they don't need the parentheses `()` after them - and simply `return` the static data that attribute refers to

DataFrame attributes

- `DataFrames` are one case of a data type that has attributes associated with them
- three interesting ones for us are
    - `DataFrame.columns`
    - `DataFrame.shape`
    - `DataFrame.values`

In [15]:

```
df.columns
```

Out[15]:

```
Index(['Sepal Length', 'Sepal Width', 'Petal Length', 'Petal Width',
       'Species'],
      dtype='object')
```

In [16]:
```
df.shape
```
Out[16]:
```
(150, 5)
```

In [17]:
```
df.values
```

Out[17]:
```
array([[5.1, 3.5, 1.4, 0.2, 'setosa'],
       [4.9, 3.0, 1.4, 0.2, 'setosa'],
       [4.7, 3.2, 1.3, 0.2, 'setosa'],
       [4.6, 3.1, 1.5, 0.2, 'setosa'],
       [5.0, 3.6, 1.4, 0.2, 'setosa'],
       [5.4, 3.9, 1.7, 0.4, 'setosa'],
       [4.6, 3.4, 1.4, 0.3, 'setosa'],
       [5.0, 3.4, 1.5, 0.2, 'setosa'],
       [4.4, 2.9, 1.4, 0.2, 'setosa'],
       [4.9, 3.1, 1.5, 0.1, 'setosa'],
       [5.4, 3.7, 1.5, 0.2, 'setosa'],
       [4.8, 3.4, 1.6, 0.2, 'setosa'],
       [4.8, 3.0, 1.4, 0.1, 'setosa'],
       [4.3, 3.0, 1.1, 0.1, 'setosa'],
       [5.8, 4.0, 1.2, 0.2, 'setosa'],
       [5.7, 4.4, 1.5, 0.4, 'setosa'],
       [5.4, 3.9, 1.3, 0.4, 'setosa'],
       [5.1, 3.5, 1.4, 0.3, 'setosa'],
       [5.7, 3.8, 1.7, 0.3, 'setosa'],
       [5.1, 3.8, 1.5, 0.3, 'setosa'],
       [5.4, 3.4, 1.7, 0.2, 'setosa'],
       [5.1, 3.7, 1.5, 0.4, 'setosa'],
       [4.6, 3.6, 1.0, 0.2, 'setosa'],
       [5.1, 3.3, 1.7, 0.5, 'setosa'],
       [4.8, 3.4, 1.9, 0.2, 'setosa'],
       [5.0, 3.0, 1.6, 0.2, 'setosa'],
       [5.0, 3.4, 1.6, 0.4, 'setosa'],
       [5.2, 3.5, 1.5, 0.2, 'setosa'],
       [5.2, 3.4, 1.4, 0.2, 'setosa'],
       [4.7, 3.2, 1.6, 0.2, 'setosa'],
       [4.8, 3.1, 1.6, 0.2, 'setosa'],
       [5.4, 3.4, 1.5, 0.4, 'setosa'],
       [5.2, 4.1, 1.5, 0.1, 'setosa'],
       [5.5, 4.2, 1.4, 0.2, 'setosa'],
```

```
[4.9, 3.1, 1.5, 0.1, 'setosa'],
[5.0, 3.2, 1.2, 0.2, 'setosa'],
[5.5, 3.5, 1.3, 0.2, 'setosa'],
[4.9, 3.1, 1.5, 0.1, 'setosa'],
[4.4, 3.0, 1.3, 0.2, 'setosa'],
[5.1, 3.4, 1.5, 0.2, 'setosa'],
[5.0, 3.5, 1.3, 0.3, 'setosa'],
[4.5, 2.3, 1.3, 0.3, 'setosa'],
[4.4, 3.2, 1.3, 0.2, 'setosa'],
[5.0, 3.5, 1.6, 0.6, 'setosa'],
[5.1, 3.8, 1.9, 0.4, 'setosa'],
[4.8, 3.0, 1.4, 0.3, 'setosa'],
[5.1, 3.8, 1.6, 0.2, 'setosa'],
[4.6, 3.2, 1.4, 0.2, 'setosa'],
[5.3, 3.7, 1.5, 0.2, 'setosa'],
[5.0, 3.3, 1.4, 0.2, 'setosa'],
[7.0, 3.2, 4.7, 1.4, 'versicolor'],
[6.4, 3.2, 4.5, 1.5, 'versicolor'],
[6.9, 3.1, 4.9, 1.5, 'versicolor'],
[5.5, 2.3, 4.0, 1.3, 'versicolor'],
[6.5, 2.8, 4.6, 1.5, 'versicolor'],
[5.7, 2.8, 4.5, 1.3, 'versicolor'],
[6.3, 3.3, 4.7, 1.6, 'versicolor'],
[4.9, 2.4, 3.3, 1.0, 'versicolor'],
[6.6, 2.9, 4.6, 1.3, 'versicolor'],
[5.2, 2.7, 3.9, 1.4, 'versicolor'],
[5.0, 2.0, 3.5, 1.0, 'versicolor'],
[5.9, 3.0, 4.2, 1.5, 'versicolor'],
[6.0, 2.2, 4.0, 1.0, 'versicolor'],
[6.1, 2.9, 4.7, 1.4, 'versicolor'],
[5.6, 2.9, 3.6, 1.3, 'versicolor'],
[6.7, 3.1, 4.4, 1.4, 'versicolor'],
[5.6, 3.0, 4.5, 1.5, 'versicolor'],
[5.8, 2.7, 4.1, 1.0, 'versicolor'],
[6.2, 2.2, 4.5, 1.5, 'versicolor'],
[5.6, 2.5, 3.9, 1.1, 'versicolor'],
[5.9, 3.2, 4.8, 1.8, 'versicolor'],
[6.1, 2.8, 4.0, 1.3, 'versicolor'],
```

```
[6.3, 2.5, 4.9, 1.5, 'versicolor'],
[6.1, 2.8, 4.7, 1.2, 'versicolor'],
[6.4, 2.9, 4.3, 1.3, 'versicolor'],
[6.6, 3.0, 4.4, 1.4, 'versicolor'],
[6.8, 2.8, 4.8, 1.4, 'versicolor'],
[6.7, 3.0, 5.0, 1.7, 'versicolor'],
[6.0, 2.9, 4.5, 1.5, 'versicolor'],
[5.7, 2.6, 3.5, 1.0, 'versicolor'],
[5.5, 2.4, 3.8, 1.1, 'versicolor'],
[5.5, 2.4, 3.7, 1.0, 'versicolor'],
[5.8, 2.7, 3.9, 1.2, 'versicolor'],
[6.0, 2.7, 5.1, 1.6, 'versicolor'],
[5.4, 3.0, 4.5, 1.5, 'versicolor'],
[6.0, 3.4, 4.5, 1.6, 'versicolor'],
[6.7, 3.1, 4.7, 1.5, 'versicolor'],
[6.3, 2.3, 4.4, 1.3, 'versicolor'],
[5.6, 3.0, 4.1, 1.3, 'versicolor'],
[5.5, 2.5, 4.0, 1.3, 'versicolor'],
[5.5, 2.6, 4.4, 1.2, 'versicolor'],
[6.1, 3.0, 4.6, 1.4, 'versicolor'],
[5.8, 2.6, 4.0, 1.2, 'versicolor'],
[5.0, 2.3, 3.3, 1.0, 'versicolor'],
[5.6, 2.7, 4.2, 1.3, 'versicolor'],
[5.7, 3.0, 4.2, 1.2, 'versicolor'],
[5.7, 2.9, 4.2, 1.3, 'versicolor'],
[6.2, 2.9, 4.3, 1.3, 'versicolor'],
[5.1, 2.5, 3.0, 1.1, 'versicolor'],
[5.7, 2.8, 4.1, 1.3, 'versicolor'],
[6.3, 3.3, 6.0, 2.5, 'virginica'],
[5.8, 2.7, 5.1, 1.9, 'virginica'],
[7.1, 3.0, 5.9, 2.1, 'virginica'],
[6.3, 2.9, 5.6, 1.8, 'virginica'],
[6.5, 3.0, 5.8, 2.2, 'virginica'],
[7.6, 3.0, 6.6, 2.1, 'virginica'],
[4.9, 2.5, 4.5, 1.7, 'virginica'],
[7.3, 2.9, 6.3, 1.8, 'virginica'],
[6.7, 2.5, 5.8, 1.8, 'virginica'],
[7.2, 3.6, 6.1, 2.5, 'virginica'],
```

```
[6.5, 3.2, 5.1, 2.0, 'virginica'],
[6.4, 2.7, 5.3, 1.9, 'virginica'],
[6.8, 3.0, 5.5, 2.1, 'virginica'],
[5.7, 2.5, 5.0, 2.0, 'virginica'],
[5.8, 2.8, 5.1, 2.4, 'virginica'],
[6.4, 3.2, 5.3, 2.3, 'virginica'],
[6.5, 3.0, 5.5, 1.8, 'virginica'],
[7.7, 3.8, 6.7, 2.2, 'virginica'],
[7.7, 2.6, 6.9, 2.3, 'virginica'],
[6.0, 2.2, 5.0, 1.5, 'virginica'],
[6.9, 3.2, 5.7, 2.3, 'virginica'],
[5.6, 2.8, 4.9, 2.0, 'virginica'],
[7.7, 2.8, 6.7, 2.0, 'virginica'],
[6.3, 2.7, 4.9, 1.8, 'virginica'],
[6.7, 3.3, 5.7, 2.1, 'virginica'],
[7.2, 3.2, 6.0, 1.8, 'virginica'],
[6.2, 2.8, 4.8, 1.8, 'virginica'],
[6.1, 3.0, 4.9, 1.8, 'virginica'],
[6.4, 2.8, 5.6, 2.1, 'virginica'],
[7.2, 3.0, 5.8, 1.6, 'virginica'],
[7.4, 2.8, 6.1, 1.9, 'virginica'],
[7.9, 3.8, 6.4, 2.0, 'virginica'],
[6.4, 2.8, 5.6, 2.2, 'virginica'],
[6.3, 2.8, 5.1, 1.5, 'virginica'],
[6.1, 2.6, 5.6, 1.4, 'virginica'],
[7.7, 3.0, 6.1, 2.3, 'virginica'],
[6.3, 3.4, 5.6, 2.4, 'virginica'],
[6.4, 3.1, 5.5, 1.8, 'virginica'],
[6.0, 3.0, 4.8, 1.8, 'virginica'],
[6.9, 3.1, 5.4, 2.1, 'virginica'],
[6.7, 3.1, 5.6, 2.4, 'virginica'],
[6.9, 3.1, 5.1, 2.3, 'virginica'],
[5.8, 2.7, 5.1, 1.9, 'virginica'],
[6.8, 3.2, 5.9, 2.3, 'virginica'],
[6.7, 3.3, 5.7, 2.5, 'virginica'],
[6.7, 3.0, 5.2, 2.3, 'virginica'],
[6.3, 2.5, 5.0, 1.9, 'virginica'],
[6.5, 3.0, 5.2, 2.0, 'virginica'],
```

```
       [6.2, 3.4, 5.4, 2.3, 'virginica'],
       [5.9, 3.0, 5.1, 1.8, 'virginica']], dtype=object)
```

Series

The other important data type in the `pandas` package is that of a [`<class 'pandas.core.series.Series>`](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html) (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html), which is effectively the one-dimensional representation of a DataFrame axis - for example one row, or one column.

In [18]:
```
sepal_length = df['Sepal Length']
```

In [19]:
```
type(sepal_length)
```

Out[19]:
```
pandas.core.series.Series
```

In [20]:
```
sepal_length
```

Out[20]:
```
0      5.1
1      4.9
2      4.7
3      4.6
4      5.0
5      5.4
6      4.6
7      5.0
8      4.4
9      4.9
10     5.4
11     4.8
12     4.8
13     4.3
14     5.8
15     5.7
16     5.4
17     5.1
18     5.7
19     5.1
20     5.4
21     5.1
22     4.6
23     5.1
24     4.8
25     5.0
26     5.0
27     5.2
28     5.2
29     4.7
      ...
120    6.9
121    5.6
122    7.7
```

```
123    6.3
124    6.7
125    7.2
126    6.2
127    6.1
128    6.4
129    7.2
130    7.4
131    7.9
132    6.4
133    6.3
134    6.1
135    7.7
136    6.3
137    6.4
138    6.0
139    6.9
140    6.7
141    6.9
142    5.8
143    6.8
144    6.7
145    6.7
146    6.3
147    6.5
148    6.2
149    5.9
Name: Sepal Length, Length: 150, dtype: float64
```

Series attributes

```
In [21]: sepal_length.name
```

```
Out[21]: 'Sepal Length'
```

```
In [22]: sepal_length.dtype
```

```
Out[22]: dtype('float64')
```

```
In [23]: sepal_length.shape
```

```
Out[23]: (150,)
```

Indexing a DataFrame

Index notation like we are used to with one-dimensions data structures like lists and dictionaries is modified a bit for two-dimensional DataFrames.

To illuminate series, we already saw the following:

In [24]:
```
sepal_length = df['Sepal Length']
```

Anatomy of basic indexing for columns

variable_name[column_label]

In [25]:
```
df.iloc[1]
```

Out[25]:
```
Sepal Length       4.9
Sepal Width          3
Petal Length       1.4
Petal Width        0.2
Species         setosa
Name: 1, dtype: object
```

Anatomy of one-dimensional iloc indexing for rows

variable_name.iloc[row_index]

In [26]:
```
df.iloc[1,1]
```
Out[26]:
```
3.0
```

Anatomy of two-dimensional iloc indexing for cells

variable_name.iloc[row_index,column_index]

In [27]:
```
df.loc[1]
```

Out[27]:
```
Sepal Length         4.9
Sepal Width            3
Petal Length         1.4
Petal Width          0.2
Species           setosa
Name: 1, dtype: object
```

In [28]:
```
df.loc[1,'Sepal Width']
```

Out[28]:
```
3.0
```

Anatomy of one- and two-dimensional `loc` indexing

variable_name.loc[row_label]

variable_name.loc[row_label,column_label]

Basic querying with a dataframe

In [29]:

```
# you can use expressions to slice and dice using logic
print(df[df['Sepal Length'] == 6.9])
```

```
     Sepal Length  Sepal Width  Petal Length  Petal Width     Species
52            6.9          3.1           4.9          1.5  versicolor
120           6.9          3.2           5.7          2.3   virginica
139           6.9          3.1           5.4          2.1   virginica
141           6.9          3.1           5.1          2.3   virginica
```

In [30]:
```
# how does this work? by supplying to the index a boolean array
boolean_series = df['Sepal Length'] == 6.9
```

In [31]:
```
boolean_series.head()
```

Out[31]:
```
0    False
1    False
2    False
3    False
4    False
Name: Sepal Length, dtype: bool
```

In [32]:
```
# this can get quite complex
df[(df['Sepal Length']==6.9) & (df['Species']=='versicolor')]
```

Out[32]:

	Sepal Length	Sepal Width	Petal Length	Petal Width	Species
52	6.9	3.1	4.9	1.5	versicolor

Anatomy of boolean array indexing

variable_name[series_wise_boolean_expression]

Use `&` and `|` to represent `and` and `or`, respectively

Grouping data

In [33]:

```
groups = df.groupby("Species")
```

In [34]:
```
for key, group in groups:
    print(key)
    print(group.head())
```

```
setosa
   Sepal Length  Sepal Width  Petal Length  Petal Width Species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
versicolor
    Sepal Length  Sepal Width  Petal Length  Petal Width     Species
50           7.0          3.2           4.7          1.4  versicolor
51           6.4          3.2           4.5          1.5  versicolor
52           6.9          3.1           4.9          1.5  versicolor
53           5.5          2.3           4.0          1.3  versicolor
54           6.5          2.8           4.6          1.5  versicolor
virginica
     Sepal Length  Sepal Width  Petal Length  Petal Width    Species
100           6.3          3.3           6.0          2.5  virginica
101           5.8          2.7           5.1          1.9  virginica
102           7.1          3.0           5.9          2.1  virginica
103           6.3          2.9           5.6          1.8  virginica
104           6.5          3.0           5.8          2.2  virginica
```

In [35]:
```python
# This gives you a convenient way to apply logic based on a group filter
# For example, use the DataFrame.describe method to easily get summary statistics
 on each species group
for key, group in groups:
    print(key)
    print(group.describe())
```

```
setosa
       Sepal Length  Sepal Width  Petal Length  Petal Width
count      50.00000    50.000000     50.000000     50.00000
mean        5.00600     3.418000      1.464000      0.24400
std         0.35249     0.381024      0.173511      0.10721
min         4.30000     2.300000      1.000000      0.10000
25%         4.80000     3.125000      1.400000      0.20000
50%         5.00000     3.400000      1.500000      0.20000
75%         5.20000     3.675000      1.575000      0.30000
max         5.80000     4.400000      1.900000      0.60000
versicolor
       Sepal Length  Sepal Width  Petal Length  Petal Width
count     50.000000    50.000000     50.000000    50.000000
mean       5.936000     2.770000      4.260000     1.326000
std        0.516171     0.313798      0.469911     0.197753
min        4.900000     2.000000      3.000000     1.000000
25%        5.600000     2.525000      4.000000     1.200000
50%        5.900000     2.800000      4.350000     1.300000
75%        6.300000     3.000000      4.600000     1.500000
max        7.000000     3.400000      5.100000     1.800000
virginica
       Sepal Length  Sepal Width  Petal Length  Petal Width
count      50.00000    50.000000     50.000000     50.00000
mean        6.58800     2.974000      5.552000      2.02600
std         0.63588     0.322497      0.551895      0.27465
min         4.90000     2.200000      4.500000      1.40000
25%         6.22500     2.800000      5.100000      1.80000
50%         6.50000     3.000000      5.550000      2.00000
```

75%	6.90000	3.175000	5.875000	2.30000
max	7.90000	3.800000	6.900000	2.50000

In [36]:
```
# You can chain an aggregation onto a groupby to get groupwise stats outside of wh
at is in `describe`
print(df.groupby("Species").sum())
```

```
            Sepal Length  Sepal Width  Petal Length  Petal Width
Species
setosa             250.3        170.9          73.2         12.2
versicolor         296.8        138.5         213.0         66.3
virginica          329.4        148.7         277.6        101.3
```

In [37]:
```
print(df.groupby("Species").max())
```

```
            Sepal Length  Sepal Width  Petal Length  Petal Width
Species
setosa               5.8          4.4           1.9          0.6
versicolor           7.0          3.4           5.1          1.8
virginica            7.9          3.8           6.9          2.5
```

In [38]:
```
print(df.groupby("Species").min())
```

```
            Sepal Length  Sepal Width  Petal Length  Petal Width
Species
setosa               4.3          2.3           1.0          0.1
versicolor           4.9          2.0           3.0          1.0
virginica            4.9          2.2           4.5          1.4
```

Get your feet wet

Choose any of the data sets I've provided in Canvas to begin practicing with these first 5 pandas tasks.

Try to:

1. Load the data as a pandas DataFrame.
    - HINT: Use a convenience method to pull the data into a DataFrame from a file path!
2. Describe the data in the DataFrame using the describe() method.
3. Select just row 5 from the DataFrame. Now how about the value from row 5, column 2. How about selecting a whole column by its label?
4. Use the groupby() method against a categorial column in your data.

Tour de Python Level 2 ○○●○

- `pandas` based processing techniques for
  - dealing with duplicates
  - dealing with sparse data
  - applying custom logic
  - quick vis with just pandas

Dealing with duplicates

In [39]:
```
df[df.duplicated()]
```

Out[39]:

	Sepal Length	Sepal Width	Petal Length	Petal Width	Species
34	4.9	3.1	1.5	0.1	setosa
37	4.9	3.1	1.5	0.1	setosa
142	5.8	2.7	5.1	1.9	virginica

In [40]:
```
df[df.duplicated(keep=False)]
```

Out[40]:

	Sepal Length	Sepal Width	Petal Length	Petal Width	Species
9	4.9	3.1	1.5	0.1	setosa
34	4.9	3.1	1.5	0.1	setosa
37	4.9	3.1	1.5	0.1	setosa
101	5.8	2.7	5.1	1.9	virginica
142	5.8	2.7	5.1	1.9	virginica

In [41]:
```
dropped_df = df.drop_duplicates()
```

In [42]:
```
dropped_df.shape
```

Out[42]:
```
(147, 5)
```

Dealing with sparse data

In [43]:
```
sparse_df = pd.read_csv("hepatitis.csv", na_values="?", header=None)
```

In [44]:
```
sparse_df.head()
```

Out[44]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	2	30	2	1.0	2	2.0	2.0	2.0	1.0	2.0	2.0	2.0	2.0	2.0	1.0	85.0	18.0	4.0	NaN	1
1	2	50	1	1.0	2	1.0	2.0	2.0	1.0	2.0	2.0	2.0	2.0	2.0	0.9	135.0	42.0	3.5	NaN	1
2	2	78	1	2.0	2	1.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.7	96.0	32.0	4.0	NaN	1
3	2	31	1	NaN	1	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.7	46.0	52.0	4.0	80.0	1
4	2	34	1	2.0	2	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	1.0	NaN	200.0	4.0	NaN	1

In [45]:
```
sparse_df.shape
```
Out[45]:
```
(155, 20)
```
In [46]:
```
sparse_df.dropna().shape
```
Out[46]:
```
(80, 20)
```

In [47]:
```
sparse_df.fillna(1000).head()
```

Out[47]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	2	30	2	1.0	2	2.0	2.0	2.0	1.0	2.0	2.0	2.0	2.0	2.0	1.0	85.0	18.0	4.0	1000.0	1
1	2	50	1	1.0	2	1.0	2.0	2.0	1.0	2.0	2.0	2.0	2.0	2.0	0.9	135.0	42.0	3.5	1000.0	1
2	2	78	1	2.0	2	1.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.7	96.0	32.0	4.0	1000.0	1
3	2	31	1	1000.0	1	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.7	46.0	52.0	4.0	80.0	1
4	2	34	1	2.0	2	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	1.0	1000.0	200.0	4.0	1000.0	1

In [48]:
```
sparse_df.interpolate().head()
```

Out[48]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	2	30	2	1.0	2	2.0	2.0	2.0	1.0	2.0	2.0	2.0	2.0	2.0	1.0	85.0	18.0	4.0	NaN	1
1	2	50	1	1.0	2	1.0	2.0	2.0	1.0	2.0	2.0	2.0	2.0	2.0	0.9	135.0	42.0	3.5	NaN	1
2	2	78	1	2.0	2	1.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.7	96.0	32.0	4.0	NaN	1
3	2	31	1	2.0	1	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	0.7	46.0	52.0	4.0	80.0	1
4	2	34	1	2.0	2	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	1.0	70.5	200.0	4.0	77.5	1

Applying custom logic cellwise

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz"

In [49]:

```
import numpy as np
num_df = pd.DataFrame(np.random.randint(0,100,size=(100, 4)), columns=['A','B','C'
,'D'])
```

In [50]:

```
num_df.head()
```

Out[50]:

	A	B	C	D
0	34	5	68	78
1	28	23	30	65
2	82	88	82	48
3	89	75	9	53
4	93	77	79	52

In [51]:

```python
def fizz_buzz_ify(cell):
    cell = float(cell)
    if (cell % 3.0 == 0) & (cell % 5.0 == 0):
        return "FizzBuzz"
    elif cell % 3.0 == 0:
        return "Fizz"
    elif cell % 5.0 == 0:
        return "Buzz"
    else:
        return cell
```

In [52]:

```
num_df.applymap(fizz_buzz_ify).head()
```

Out[52]:

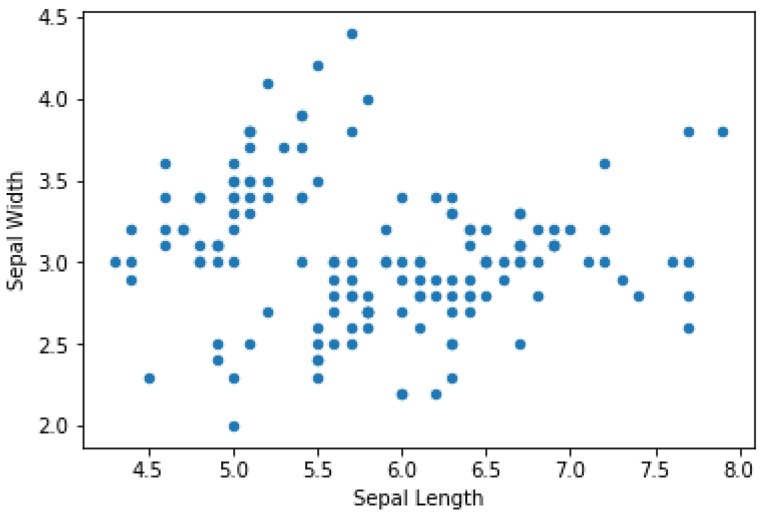
	A	B	C	D
0	34	Buzz	68	Fizz
1	28	23	FizzBuzz	Buzz
2	82	88	82	Fizz
3	89	FizzBuzz	Fizz	53
4	Fizz	77	79	52

Quick vis with just pandas

Pandas also includes some built-in visualization methods against dataframes for common plots. It is as simple as calling the `hist()` or `plot()` method on a dataframe to get a visualization.

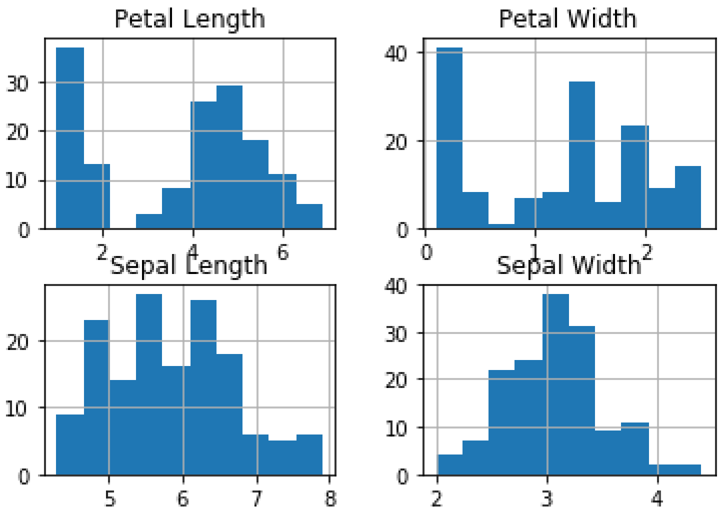
In [54]:
```
df.plot('Sepal Length', 'Sepal Width', kind="scatter")
```

Out[54]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x110841358>
```



```
In [55]: df.hist()
```

```
Out[55]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x110837e80>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1146e75f8>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x114777b70>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1147a8128>]],
               dtype=object)
```

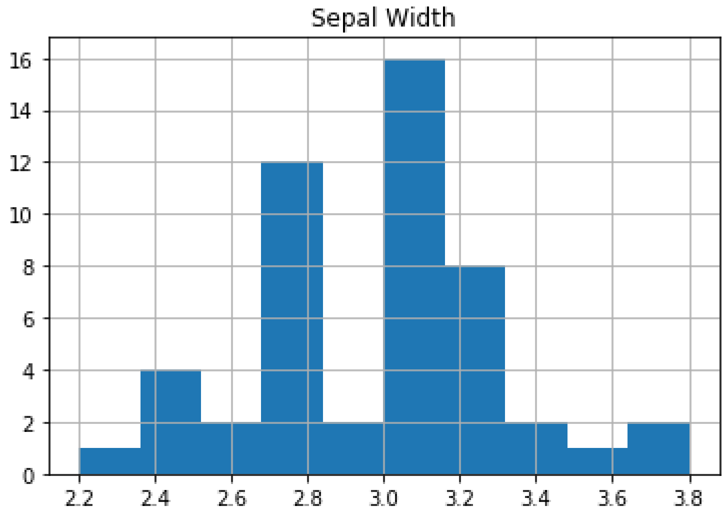


In [56]:

```
df[df['Species'] == 'virginica'].hist(column=['Sepal Width'])
```

Out[56]:

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x114833320>]],
      dtype=object)
```



Exercises

Using the `chipotle.tsv` file from Canvas, answer the following questions. (HINT: What convenience method works on `.tsv` s?)

1. What is the number of observations in this dataset?
    - HINT: (1) and (2) can be answered with the same DataFrame attribute!
2. What is the number of columns in the dataset?
3. What are the names of all the columns of this dataset?
4. What was the most ordered item?
    - HINT: Consider a groupby with an aggregation!
    - HINT: You will need to add up the `quantity` field across items of the same `item_name` and look at the results. There is an aggregation method called `sum()`.
5. How many times was a Veggie Salad Bowl ordered?

Matplotlib

Tour de Python Level 2 ○○○●

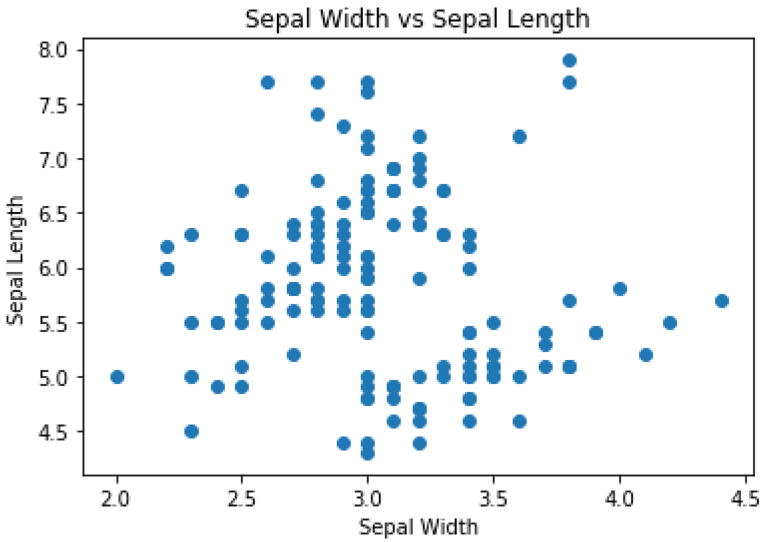
- basic `Matplotlib`
- a realistic example

In [57]:

```
import matplotlib.pyplot as plt
```

In [58]:

```
plt.scatter(df['Sepal Width'], df['Sepal Length'])
plt.xlabel('Sepal Width')
plt.ylabel('Sepal Length')
plt.title('Sepal Width vs Sepal Length')
plt.show()
```



A more realistic example

Take a look at the file `"gdp_time_series"` in your terminal with `cat`. You'll notice it's not so well formatted...

In [59]:

```
# first, our container list
data_list_of_lists = []

# now open our file
f = open("gdp_time_series", "r")
# f that is my file "gdp_time_series" on disk
# iterate through each row after row 3; if you look at the data you'll see there's
# non-data in the first 3 lines
for row in f.readlines()[3:]: # use slice notation to skip the first 3 lines
    # split on arbitrary amount of whitespace
    current_row = row.split()
    # row.split is going to cause each row to turn into a list of strings
    # i.e. ['1950','0.02',...]

    # now add that list to our container list
    data_list_of_lists.append(current_row)
f.close()

# at the end of this for loop, in general, the data will look like:
# [['YEAR','AUSTRIA','CANADA'....],['1950','0.02'..]]
#
# now that we have a bunch of data in our list of lists, instantiate a DataFrame d
irectly
# the first list in our list of lists is the header column
# the rest are our actual data
# so we slice the list of lists when we specify the data and the columns
df = pd.DataFrame(data=data_list_of_lists[1:],columns=data_list_of_lists[0])
```

In [60]:
```
df.head()
```

Out[60]:

	YEAR	AUSTRIA	CANADA	FRANCE	GERMANY	GREECE	ITALY	SWEDEN	UK	USA
0	1950	0.027523	3.651109	10.652861	5.725433	18.423605	0.799001	17.072701	1.033571	4.470303
1	1951	0.029406	3.734242	11.186672	6.256754	19.86624	0.829484	17.445339	1.060015	4.734335
2	1952	0.029357	3.932222	11.480235	6.70308	19.750938	0.859817	17.011088	1.104598	4.826502
3	1953	0.030603	4.019939	11.688318	7.256435	22.217731	0.916962	18.063728	1.152221	4.981746
4	1954	0.033678	3.860731	12.092329	7.72644	22.690231	0.942153	19.031748	1.191948	4.79081

In [61]:
```
df.describe()
```

Out[61]:

	YEAR	AUSTRIA	CANADA	FRANCE	GERMANY	GREECE	ITALY	SWEDEN	UK	USA
count	34	34	34	34	34	34	34	34	34	34
unique	34	34	34	34	34	34	34	34	33	33
top	1983	0.06178	6.062678	13.194351	15.720841	77.985801	2.825328	34.391346	1.355527	5.160474
freq	1	1	1	1	1	1	1	1	2	2

In [62]:
```
df.dtypes
```

Out[62]:
```
YEAR       object
AUSTRIA    object
CANADA     object
FRANCE     object
GERMANY    object
GREECE     object
ITALY      object
SWEDEN     object
UK         object
USA        object
dtype: object
```

In [63]:
```
# we sent it all the data as strings, but we actually want to be able to do math o
n them
# so let's set the dtype of the entire dataframe as float
# here we overwrite 'df'
df = pd.DataFrame(data=data_list_of_lists[1:], columns=data_list_of_lists[0], dtyp
e=float)
```

In [64]:
```
df.dtypes
```

Out[64]:
```
YEAR       float64
AUSTRIA    float64
CANADA     float64
FRANCE     float64
GERMANY    float64
GREECE     float64
ITALY      float64
SWEDEN     float64
UK         float64
USA        float64
dtype: object
```

In [65]:

```
df.describe()
```

Out[65]:

	YEAR	AUSTRIA	CANADA	FRANCE	GERMANY	GREECE	ITALY	SWEDEN	UK	USA
count	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000
mean	1966.500000	0.065533	5.817088	20.957515	13.428460	50.932949	1.757668	28.073149	1.576265	6.241882
std	9.958246	0.025962	1.611434	7.369126	4.476840	24.196637	0.649375	7.221651	0.325664	1.227840
min	1950.000000	0.027523	3.651109	10.652861	5.725433	18.423605	0.799001	17.011088	1.033571	4.470303
25%	1958.250000	0.043910	4.369293	14.160714	9.814249	28.701115	1.141946	20.886017	1.284349	5.080982
50%	1966.500000	0.061104	5.578620	20.049311	12.990514	46.669707	1.720711	28.657428	1.558952	6.206709
75%	1974.750000	0.087410	7.371888	27.614323	16.959558	74.144169	2.340225	34.850870	1.884099	7.327845
max	1983.000000	0.107894	8.382785	32.095989	19.985983	85.949501	2.825328	38.665154	2.079010	8.164851

In [66]:

```
df = df.astype({"YEAR": object})
```

In [67]:

```
df.head()
```

Out[67]:

	YEAR	AUSTRIA	CANADA	FRANCE	GERMANY	GREECE	ITALY	SWEDEN	UK	USA
0	1950	0.027523	3.651109	10.652861	5.725433	18.423605	0.799001	17.072701	1.033571	4.470303
1	1951	0.029406	3.734242	11.186672	6.256754	19.866240	0.829484	17.445339	1.060015	4.734335
2	1952	0.029357	3.932222	11.480235	6.703080	19.750938	0.859817	17.011088	1.104598	4.826502
3	1953	0.030603	4.019939	11.688318	7.256435	22.217731	0.916962	18.063728	1.152221	4.981746
4	1954	0.033678	3.860731	12.092329	7.726440	22.690231	0.942153	19.031748	1.191948	4.790810

In [68]:

```
plt.plot(df['YEAR'],df['AUSTRIA'])
plt.ylabel('Per Capita Annual GDP')
plt.show()
```