Pandas Introduction

What is Pandas?

- Library for manupulating tables of data
- Primarily used for cleaning and restructuring data in preperation for plotting and modeling
- 2 primary data structures
    - Series - 1D, columns of data
    - DataFrames - 2D, tables of data
- Columnar
    - Most operations are designed to operate on columns of data, not individual elements or rows

In [1]:
```
import matplotlib.pyplot as plt
import sklearn.ensemble as mdl
import pandas as pd
import numpy as np
datapath = 'IRIS-1.csv'
```

Caveats

- Pandas offers multiple ways to do things. Some ways are newer and have learned from the mistakes of the old ways. This can be confusing and frustrating
- Pandas documentation is complex and not well organized
- It can be difficult to predict when a copy is made versus a view is created - this makes optimization challenging

Creating DataFrames

- Read from a csv file

In [2]:
```
df1 = pd.read_csv(datapath)
```

- Show the first 5 lines of the file

In [3]:
```
df1.head()
```

Out[3]:

	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

- From existing lists, Numpy arrays, or series

In [4]:
```
df2 = pd.DataFrame( {"column1" : [0.0, 1.0, 2.0],
                     "column2" : np.random.randint(10,size = (3)),
                     "column3" : df1["species"][0:3] } )
df2.head()
```

Out[4]:

	column1	column2	column3
0	0.0	3	Iris-setosa
1	1.0	7	Iris-setosa
2	2.0	4	Iris-setosa

Investigating DataFrames

- There are multiple functions to investigate existing DataFrames

In [5]:
```
df1.head(10)
```

Out[5]:

	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
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

In [6]:
```
df1.dtypes
```

Out[6]:
```
sepal_length    float64
sepal_width     float64
petal_length    float64
petal_width     float64
species          object
dtype: object
```

In [7]:
```
df1.shape
```

Out[7]:
```
(150, 5)
```

In [8]:
```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    object 
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

In [9]:
```
help(df1.info)
```

```
Help on method info in module pandas.core.frame:

info(verbose: 'bool | None' = None, buf: 'WriteBuffer[str] | None' = None, max_cols: 'int | None' = None, memory
_usage: 'bool | str | None' = None, show_counts: 'bool | None' = None) -> 'None' method of pandas.core.frame.Dat
aFrame instance
    Print a concise summary of a DataFrame.

    This method prints information about a DataFrame including
    the index dtype and columns, non-null values and memory usage.

    Parameters
    ----------
    verbose : bool, optional
        Whether to print the full summary. By default, the setting in
        ``pandas.options.display.max_info_columns`` is followed.
    buf : writable buffer, defaults to sys.stdout
        Where to send the output. By default, the output is printed to
        sys.stdout. Pass a writable buffer if you need to further process
        the output.
    max_cols : int, optional
        When to switch from the verbose to the truncated output. If the
        DataFrame has more than `max_cols` columns, the truncated output
        is used. By default, the setting in
        ``pandas.options.display.max_info_columns`` is used.
    memory_usage : bool, str, optional
        Specifies whether total memory usage of the DataFrame
        elements (including the index) should be displayed. By default,
        this follows the ``pandas.options.display.memory_usage`` setting.

        True always show memory usage. False never shows memory usage.
        A value of 'deep' is equivalent to "True with deep introspection".
        Memory usage is shown in human-readable units (base-2
        representation). Without deep introspection a memory estimation is
        made based in column dtype and number of rows assuming values
```

```
    consume the same memory amount for corresponding dtypes. With deep
    memory introspection, a real memory usage calculation is performed
    at the cost of computational resources. See the
    :ref:`Frequently Asked Questions <df-memory-usage>` for more
    details.
show_counts : bool, optional
    Whether to show the non-null counts. By default, this is shown
    only if the DataFrame is smaller than
    ``pandas.options.display.max_info_rows`` and
    ``pandas.options.display.max_info_columns``. A value of True always
    shows the counts, and False never shows the counts.

Returns
-------
None
    This method prints a summary of a DataFrame and returns None.

See Also
--------
DataFrame.describe: Generate descriptive statistics of DataFrame
    columns.
DataFrame.memory_usage: Memory usage of DataFrame columns.

Examples
--------
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                   "float_col": float_values})
>>> df
    int_col text_col  float_col
0        1    alpha       0.00
1        2     beta       0.25
2        3    gamma       0.50
3        4    delta       0.75
4        5  epsilon       1.00

Prints information of all columns:

>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   int_col    5 non-null      int64
 1   text_col   5 non-null      object
 2   float_col  5 non-null      float64
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes

Prints a summary of columns count and its dtypes but not per column
information:

>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes

Pipe output of DataFrame.info to buffer instead of sys.stdout, get
buffer content and writes to a text file:

>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w",
...           encoding="utf-8") as f:  # doctest: +SKIP
...     f.write(s)
260

The `memory_usage` parameter allows deep introspection mode, specially
useful for big DataFrames and fine-tune memory optimization:

>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
```

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   column_1  1000000 non-null  object
 1   column_2  1000000 non-null  object
 2   column_3  1000000 non-null  object
dtypes: object(3)
memory usage: 22.9+ MB

>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column    Non-Null Count    Dtype
---  ------    --------------    -----
 0   column_1  1000000 non-null  object
 1   column_2  1000000 non-null  object
 2   column_3  1000000 non-null  object
dtypes: object(3)
memory usage: 165.9 MB
```

Indexing / Selecting / Slicing Columns

- Pandas has multiple ways to index. The slice operator works on columns

In [10]:
```
df1.head(1)
```

Out[10]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa

In [11]:
```
df1["sepal_length"][0:2]
```

Out[11]:
```
0    5.1
1    4.9
Name: sepal_length, dtype: float64
```

Or another way...

In [12]:
```
df1.sepal_length[0:2]
```

Out[12]:
```
0    5.1
1    4.9
Name: sepal_length, dtype: float64
```

In [13]:
```
df1[["sepal_length","species"]][0:5]
```

Out[13]:

	sepal_length	species
0	5.1	Iris-setosa
1	4.9	Iris-setosa
2	4.7	Iris-setosa
3	4.6	Iris-setosa
4	5.0	Iris-setosa

Indexing

- You can index by position (numerical index). This follows the Numpy pattern of row, then column:

In [14]:
```
df1.iloc[5]
```

Out[14]:
```
sepal_length            5.4
sepal_width             3.9
petal_length            1.7
petal_width             0.4
species         Iris-setosa
Name: 5, dtype: object
```

Creating a New Column

- The simplest way to create a new column:

In [15]:
```python
extra_col = np.random.randint(2,size=(150))
```

In [16]:
```python
df1["Is_pretty"] = extra_col==1
df1.head()
```

Out[16]:

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

- The assign method is used too, since it returns a new DataFrame and can be used with method chaining:

In [17]:
```python
new_df = df1.assign(Smells_bad = np.ones(150)==1)
new_df.head()
```

Out[17]:

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

Modifying a column

- Convert data types - may need to specify function for parsing /conversion
- Cleaning data
- Extracting fields from complex types
    - e.g., hour, month, etc... from date times

1. Get the Series for the column of interest

In [18]:
```python
column = new_df["Smells_bad"]
```

2. Use the map() method to apply a function to each element in the Series and return a new Series

In [19]:
```python
converted = column.map(lambda s: (not s))
converted.head()
```

Out[19]:
```
0    False
1    False
2    False
3    False
4    False
Name: Smells_bad, dtype: bool
```

3. Then update the df, either by adding a new column or overwritng the orignal column

In [20]:
```python
df1["Smells_bad"] = converted
df1.head()
```

Out[20]:

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

Dropping a Column

- I prefer to use the drop() method becuase it returns a DataFrame object, so it work with chaining:

In [21]:
```
new_df = df1.drop(columns=["Smells_bad"])
```

- You might also see this format

In [22]:
```
df1.head()
```

Out[22]:

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

In [23]:
```
del df1["Smells_bad"]
df1.head()
```

Out[23]:

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

Filtering

We can apply boolean indexing to filter our dataframe

In [24]:
```
df1_filtered = df1[df1['sepal_length'] > 5]
df1_filtered.head()
```

Out[24]:

	sepal_length	sepal_width	petal_length	petal_width	species	Is_pretty
0	5.1	3.5	1.4	0.2	Iris-setosa	False
5	5.4	3.9	1.7	0.4	Iris-setosa	True
10	5.4	3.7	1.5	0.2	Iris-setosa	False
14	5.8	4.0	1.2	0.2	Iris-setosa	True
15	5.7	4.4	1.5	0.4	Iris-setosa	True

We can also use string operations to slice based on string properties. We can also find out how many unique values there are in a column using the following code.

In [25]:
```
df1_filtered2 = df1[df1['species'].str.len() > 11]
print(df1_filtered2.species.unique())
```

```
['Iris-versicolor' 'Iris-virginica']
```

Notice in the preceeding cell that the second line with the unique call uses a different filtering syntax that allows you to refer to a column (if it doesn't have spaces) directly after the dataframe name. This is a strong reason to avoid using spaces in your column names.

You can slice multiple columns using double brackets or a single column with a single bracket. If you are slicing a single column with a single bracket, the return type will be a Series (not a DataFrame)

In [26]:
```
df1[['sepal_length', 'sepal_width']]
```

Out[26]:

	sepal_length	sepal_width
0	5.1	3.5
1	4.9	3.0
2	4.7	3.2
3	4.6	3.1
4	5.0	3.6
...	...	...
145	6.7	3.0
146	6.3	2.5
147	6.5	3.0
148	6.2	3.4
149	5.9	3.0

150 rows × 2 columns

We can sort a DataFrame with a simple method call. You should add a more complex sort with multiple columns where some are ascending and some are descending.

In [27]:

```
df1_sorted = df1.sort_values(by = 'sepal_length')
df1_sorted.head(20)
```

Out[27]:

	sepal_length	sepal_width	petal_length	petal_width	species	Is_pretty
13	4.3	3.0	1.1	0.1	Iris-setosa	False
42	4.4	3.2	1.3	0.2	Iris-setosa	True
38	4.4	3.0	1.3	0.2	Iris-setosa	True
8	4.4	2.9	1.4	0.2	Iris-setosa	False
41	4.5	2.3	1.3	0.3	Iris-setosa	False
22	4.6	3.6	1.0	0.2	Iris-setosa	True
3	4.6	3.1	1.5	0.2	Iris-setosa	False
6	4.6	3.4	1.4	0.3	Iris-setosa	False
47	4.6	3.2	1.4	0.2	Iris-setosa	False
2	4.7	3.2	1.3	0.2	Iris-setosa	True
29	4.7	3.2	1.6	0.2	Iris-setosa	True
12	4.8	3.0	1.4	0.1	Iris-setosa	True
45	4.8	3.0	1.4	0.3	Iris-setosa	True
24	4.8	3.4	1.9	0.2	Iris-setosa	False
11	4.8	3.4	1.6	0.2	Iris-setosa	True
30	4.8	3.1	1.6	0.2	Iris-setosa	False
57	4.9	2.4	3.3	1.0	Iris-versicolor	True
106	4.9	2.5	4.5	1.7	Iris-virginica	False
34	4.9	3.1	1.5	0.1	Iris-setosa	True
9	4.9	3.1	1.5	0.1	Iris-setosa	True

You can also call methods that will provide basic descriptive statistics on a dataframe using simple method calls. Add a few in the following cell.

In [28]:

```
skewness = df1.select_dtypes(include=['float64']).skew()
print(skewness)
```

```
sepal_length    0.314911
sepal_width     0.334053
petal_length   -0.274464
petal_width    -0.104997
dtype: float64
```

That's it! Except, there is still a lot to learn about DataFrames. There is a lot more to learn, and you can start by digging into the official documentation here: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html