

Python Libraries

Python, like other programming languages, has an abundance of additional modules or libraries that augment the base framework and functionality of the language.

Think of a library as a collection of functions that can be accessed to complete certain programming tasks without having to write your own algorithm.

For this course, we will focus primarily on the following libraries:

- **Numpy** is a library for working with arrays of data.
- **Pandas** provides high-performance, easy-to-use data structures and data analysis tools.
- **Scipy** is a library of techniques for numerical and scientific computing.
- **Matplotlib** is a library for making graphs.
- **Seaborn** is a higher-level interface to Matplotlib that can be used to simplify many graphing tasks.
- **Statsmodels** is a library that implements many statistical techniques.

Documentation

Reliable and accessible documentation is an absolute necessity when it comes to knowledge transfer of programming languages. Luckily, python provides a significant amount of detailed documentation that explains the ins and outs of the language syntax, libraries, and more.

Understanding how to read documentation is crucial for any programmer as it will serve as a fantastic resource when learning the intricacies of python.

Here is the link to the documentation of the python standard library: [Python Standard Library \(https://docs.python.org/3/library/index.html#library-index\)](https://docs.python.org/3/library/index.html#library-index)

Importing Libraries

When using Python, you must always begin your scripts by importing the libraries that you will be using.

The following statement imports the numpy and pandas library, and gives them abbreviated names:

```
In [6]: import numpy as np  
       import pandas as pd
```

Utilizing Library Functions

After importing a library, its functions can then be called from your code by prepending the library name to the function name. For example, to use the 'dot' function from the 'numpy' library, you would enter 'numpy.dot'. To avoid repeatedly having to type the library name in your scripts, it is conventional to define a two or three letter abbreviation for each library, e.g. 'numpy' is usually abbreviated as 'np'. This allows us to use 'np.dot' instead of 'numpy.dot'. Similarly, the Pandas library is typically abbreviated as 'pd'.

The next cell shows how to call functions within an imported library:

```
In [3]: a = np.array([0,1,2,3,4,5,6,7,8,9,10])  
       np.mean(a)  
Out[3]: 5.0
```

As you can see, we used the mean() function within the numpy library to calculate the mean of the numpy 1-dimensional array.

Data Management

Data management is a crucial component to statistical analysis and data science work. The following code will show how to import data via the pandas library, view your data, and transform your data.

The main data structure that Pandas works with is called a **Data Frame**. This is a two-dimensional table of data in which the rows typically represent cases (e.g. Cartwheel Contest Participants), and the columns represent variables. Pandas also has a one-dimensional data structure called a **Series** that we will encounter when accessing a single column of a Data Frame.

Pandas has a variety of functions named 'read_xxx' for reading data in different formats. Right now we will focus on reading 'csv' files, which stands for comma-separated values. However the other file formats include excel, json, and sql just to name a few.

This is a link to the .csv that we will be exploring in this tutorial: [Cartwheel Data \(https://www.coursera.org/learn/understanding-visualization-data/resources/0rVxx\)](https://www.coursera.org/learn/understanding-visualization-data/resources/0rVxx) (Link goes to the dataset section of the Resources for this course)

There are many other options to 'read_csv' that are very useful. For example, you would use the option `sep='\\t'` instead of the default `sep=' '`, if the fields of your data file are delimited by tabs instead of commas. See [here \(https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html\)](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html) for the full documentation for 'read_csv'.

Importing Data

```
In [4]: # Store the url string that hosts our .csv file (note that this is a different url than in the video)
url = "Cartwheeldata.csv"

# Read the .csv file and store it as a pandas Data Frame
df = pd.read_csv(url)

# Output object type
type(df)
```

Out[4]: pandas.core.frame.DataFrame

Viewing Data

```
In [5]: # We can view our Data Frame by calling the head() function
df.head()
```

Out[5]:

	ID	Age	Gender	GenderGroup	Glasses	GlassesGroup	Height	Wingspan	CWDistance	Complete	CompleteGroup	Score
0	1	56	F	1	Y	1	62.0	61.0	79	Y	1	7
1	2	26	F	1	Y	1	62.0	60.0	70	Y	1	8
2	3	33	F	1	Y	1	66.0	64.0	85	Y	1	7
3	4	39	F	1	N	0	64.0	63.0	87	Y	1	10
4	5	27	M	2	N	0	73.0	75.0	72	N	0	4

The head() function simply shows the first 5 rows of our Data Frame. If we wanted to show the entire Data Frame we would simply write the following:

```
In [7]: # Output entire Data Frame
df
```

Out[7]:

	ID	Age	Gender	GenderGroup	Glasses	GlassesGroup	Height	Wingspan	CWDistance	Complete	CompleteGroup	Score
0	1	56	F	1	Y	1	62.00	61.0	79	Y	1	7
1	2	26	F	1	Y	1	62.00	60.0	70	Y	1	8
2	3	33	F	1	Y	1	66.00	64.0	85	Y	1	7
3	4	39	F	1	N	0	64.00	63.0	87	Y	1	10
4	5	27	M	2	N	0	73.00	75.0	72	N	0	4
5	6	24	M	2	N	0	75.00	71.0	81	N	0	3
6	7	28	M	2	N	0	75.00	76.0	107	Y	1	10
7	8	22	F	1	N	0	65.00	62.0	98	Y	1	9
8	9	29	M	2	Y	1	74.00	73.0	106	N	0	5
9	10	33	F	1	Y	1	63.00	60.0	65	Y	1	8
10	11	30	M	2	Y	1	69.50	66.0	96	Y	1	6
11	12	28	F	1	Y	1	62.75	58.0	79	Y	1	10
12	13	25	F	1	Y	1	65.00	64.5	92	Y	1	6
13	14	23	F	1	N	0	61.50	57.5	66	Y	1	4
14	15	31	M	2	Y	1	73.00	74.0	72	Y	1	9
15	16	26	M	2	Y	1	71.00	72.0	115	Y	1	6
16	17	26	F	1	N	0	61.50	59.5	90	N	0	10
17	18	27	M	2	N	0	66.00	66.0	74	Y	1	5
18	19	23	M	2	Y	1	70.00	69.0	64	Y	1	3
19	20	24	F	1	Y	1	68.00	66.0	85	Y	1	8
20	21	23	M	2	Y	1	69.00	67.0	66	N	0	2
21	22	29	M	2	N	0	71.00	70.0	101	Y	1	8
22	23	25	M	2	N	0	70.00	68.0	82	Y	1	4
23	24	26	M	2	N	0	69.00	71.0	63	Y	1	5
24	25	23	F	1	Y	1	65.00	63.0	67	N	0	3

As you can see, we have a 2-Dimensional object where each row is an independent observation of our cartwheel data.

To gather more information regarding the data, we can view the column names and data types of each column with the following functions:

```
In [10]: df.columns  
Out[10]: Index([u'ID', u'Age', u'Gender', u'GenderGroup', u'Glasses', u'GlassesGroup',  
               u'Height', u'Wingspan', u'CWDistance', u'Complete', u'CompleteGroup',  
               u'Score'],  
              dtype='object')
```

Lets say we would like to splice our data frame and select only specific portions of our data. There are three different ways of doing so.

1. .loc()
2. .iloc()
3. .ix()

We will cover the .loc() and .iloc() splicing functions.

.loc()

.loc() takes two single/list/range operator separated by '.'. The first one indicates the row and the second one indicates columns.

```
In [11]: # Return all observations of CWDistance  
df.loc[:, "CWDistance"]  
  
Out[11]: 0      79  
1      70  
2      85  
3      87  
4      72  
5      81  
6     107  
7      98  
8     106  
9      65  
10     96  
11     79  
12     92  
13     66  
14     72  
15    115  
16     90  
17     74  
18     64  
19     85  
20     66  
21    101  
22     82  
23     63  
24     67  
  
Name: CWDistance, dtype: int64
```

```
In [12]: # Select all rows for multiple columns, ["CWDistance", "Height", "Wingspan"]
df.loc[:,["CWDistance", "Height", "Wingspan"]]
```

Out[12]:

	CWDistance	Height	Wingspan
0	79	62.00	61.0
1	70	62.00	60.0
2	85	66.00	64.0
3	87	64.00	63.0
4	72	73.00	75.0
5	81	75.00	71.0
6	107	75.00	76.0
7	98	65.00	62.0
8	106	74.00	73.0
9	65	63.00	60.0
10	96	69.50	66.0
11	79	62.75	58.0
12	92	65.00	64.5
13	66	61.50	57.5
14	72	73.00	74.0
15	115	71.00	72.0
16	90	61.50	59.5
17	74	66.00	66.0
18	64	70.00	69.0
19	85	68.00	66.0
20	66	69.00	67.0
21	101	71.00	70.0
22	82	70.00	68.0
23	63	69.00	71.0
24	67	65.00	63.0

```
In [13]: # Select few rows for multiple columns, ["CWDistance", "Height", "Wingspan"]
df.loc[:9, ["CWDistance", "Height", "Wingspan"]]
```

Out[13]:

	CWDistance	Height	Wingspan
0	79	62.0	61.0
1	70	62.0	60.0
2	85	66.0	64.0
3	87	64.0	63.0
4	72	73.0	75.0
5	81	75.0	71.0
6	107	75.0	76.0
7	98	65.0	62.0
8	106	74.0	73.0
9	65	63.0	60.0

```
In [14]: # Select range of rows for all columns
df.loc[10:15]
```

Out[14]:

	ID	Age	Gender	GenderGroup	Glasses	GlassesGroup	Height	Wingspan	CWDistance	Complete	CompleteGroup	Score
10	11	30	M	2	Y	1	69.50	66.0	96	Y	1	6
11	12	28	F	1	Y	1	62.75	58.0	79	Y	1	10
12	13	25	F	1	Y	1	65.00	64.5	92	Y	1	6
13	14	23	F	1	N	0	61.50	57.5	66	Y	1	4
14	15	31	M	2	Y	1	73.00	74.0	72	Y	1	9
15	16	26	M	2	Y	1	71.00	72.0	115	Y	1	6

The .loc() function requires two arguments, the indices of the rows and the column names you wish to observe.

In the above case : specifies all rows, and our column is **CWDistance**. df.loc[:, "CWDistance"]

Now, let's say we only want to return the first 10 observations:

```
In [15]: df.loc[:9, "CWDistance"]  
Out[15]: 0    79  
1    70  
2    85  
3    87  
4    72  
5    81  
6   107  
7    98  
8   106  
9    65  
Name: CWDistance, dtype: int64
```

.iloc()

.iloc() is integer based slicing, whereas .loc() used labels/column names. Here are some examples:

```
In [16]: df.iloc[:4]  
Out[16]:
```

	ID	Age	Gender	GenderGroup	Glasses	GlassesGroup	Height	Wingspan	CWDistance	Complete	CompleteGroup	Score
0	1	56	F	1	Y	1	62.0	61.0	79	Y	1	7
1	2	26	F	1	Y	1	62.0	60.0	70	Y	1	8
2	3	33	F	1	Y	1	66.0	64.0	85	Y	1	7
3	4	39	F	1	N	0	64.0	63.0	87	Y	1	10

```
In [17]: df.iloc[1:5, 2:4]
```

```
Out[17]:
```

	Gender	GenderGroup
1	F	1
2	F	1
3	F	1
4	M	2

```
In [18]: df.iloc[1:5, ["Gender", "GenderGroup"]]

TypeErrorTraceback (most recent call last)
<ipython-input-18-38420b6cd49e> in <module>()
----> 1 df.iloc[1:5, ["Gender", "GenderGroup"]]

/opt/conda/envs/python2/lib/python2.7/site-packages/pandas/core/indexing.pyc in __getitem__(self, key)
    1470         except (KeyError, IndexError):
    1471             pass
-> 1472         return self._getitem_tuple(key)
    1473     else:
    1474         # we by definition only have the 0th axis

/opt/conda/envs/python2/lib/python2.7/site-packages/pandas/core/indexing.pyc in _getitem_tuple(self, tup)
    2011     def _getitem_tuple(self, tup):
    2012:
-> 2013         self._has_valid_tuple(tup)
    2014         try:
    2015             return self._getitem_lowerdim(tup)

/opt/conda/envs/python2/lib/python2.7/site-packages/pandas/core/indexing.pyc in _has_valid_tuple(self, key)
    220             raise IndexingError('Too many indexers')
    221         try:
-> 222             self._validate_key(k, i)
    223         except ValueError:
    224             raise ValueError("Location based indexing can only have "

/opt/conda/envs/python2/lib/python2.7/site-packages/pandas/core/indexing.pyc in _validate_key(self, key, axis)
    1965         l = len(self.obj_.get_axis(axis))
    1966:
-> 1967         if len(arr) and (arr.max() >= l or arr.min() < -l):
    1968             raise IndexError("positional indexers are out-of-bounds")
    1969         else:

/opt/conda/envs/python2/lib/python2.7/site-packages/numpy/core/_methods.pyc in _amax(a, axis, out, keepdims, initial)
    26 def _amax(a, axis=None, out=None, keepdims=False,
    27             initial=_NoValue):
-> 28     return umr_maximum(a, axis, None, out, keepdims, initial)
    29
    30 def _amin(a, axis=None, out=None, keepdims=False,
    31             initial=_NoValue):
    32     return umr_minimum(a, axis, None, out, keepdims, initial)

TypeError: cannot perform reduce with flexible type
```

We can view the data types of our data frame columns with by calling .dtypes on our data frame:

```
In [19]: df.dtypes

Out[19]: ID          int64
Age          int64
Gender       object
GenderGroup  int64
Glasses      object
GlassesGroup int64
Height        float64
Wingspan     float64
CWDistance   int64
Complete     object
CompleteGroup int64
Score         int64
dtype: object
```

The output indicates we have integers, floats, and objects with our Data Frame.

We may also want to observe the different unique values within a specific column, lets do this for Gender:

```
In [20]: # List unique values in the df['Gender'] column
df.Gender.unique()

Out[20]: array(['F', 'M'], dtype=object)

In [21]: # Lets explore df["GenderGroup] as well
df.GenderGroup.unique()

Out[21]: array([1, 2])
```

It seems that these fields may serve the same purpose, which is to specify male vs. female. Lets check this quickly by observing only these two columns:

```
In [ ]: # Use .loc() to specify a list of multiple column names
df.loc[:, ["Gender", "GenderGroup"]]
```

From eyeballing the output, it seems to check out. We can streamline this by utilizing the groupby() and size() functions.

```
In [ ]: df.groupby(['Gender','GenderGroup']).size()
```

This output indicates that we have two types of combinations.

- Case 1: Gender = F & Gender Group = 1
- Case 2: Gender = M & GenderGroup = 2.

This validates our initial assumption that these two fields essentially portray the same information.