DATA ANALYSIS LIBRARY'S

Here's a list of some popular data analysis libraries in Python:

* Pandas
* NumPy
* Matplotlib
* Seaborn
* SciPy

We will be only covering the Numpy and Pandas in this week

1.Numpy

NumPy, short for Numerical Python, is one of the most important foundational pack- ages for numerical computing in Python.

To give you an idea of the performance difference, consider a NumPy array of one million integers, and the equivalent Python list

```
import numpy as np
```

```
my_arr = np.arange(1000000)
```

Generate some random data

```
data = np.random.randn(2, 3)
```

```
data
```

```
array([[ 0.56480701, -0.74533652,  0.91543219],
       [ 0.69019972,  1.73683124, -0.73877481]])
```

I then write mathematical operations with data

```
data*10
```

```
array([[ 5.64807006, -7.45336525,  9.15432188],
       [ 6.90199721, 17.36831241, -7.38774808]])
```

```
data + data
```

```
array([[ 1.12961401, -1.49067305,  1.83086438],
       [ 1.38039944,  3.47366248, -1.47754962]])
```

In the first example, all of the elements have been multiplied by 10. In the second, the corresponding values in each “cell” in the array have been added to each other

Every array has a shape, a tuple indicating the size of each dimension, and a dtype, an object describing the data type of the array

```
data.shape
```

```
(2, 3)
```

```
data.dtype
```

```
dtype('float64')
```

Creating ndarrays

The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array con- taining the passed data.

```
data1 = [6, 7.5, 8, 0, 1]
```

```
arr1 = np.array(data1)
```

```
arr1
```

```
array([6. , 7.5, 8. , 0. , 1. ])
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimen- sional array

```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
arr2 = np.array(data2)
```

```
arr2
```

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Since data2 was a list of lists, the NumPy array arr2 has two dimensions with shape inferred from the data. We can confirm this by inspecting the ndim and shape attributes

```
arr2.ndim
```

```
2
```

```
arr2.shape
```

```
(2, 4)
```

The data type is stored in a special dtype metadata object

```
arr1.dtype
```

```
dtype('float64')
```

```
arr2.dtype
```

```
dtype('int64')
```

In addition to np.array, there are a number of other functions for creating new arrays.

```
np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
np.zeros((3, 6))
```

```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

```
np.empty((2, 3, 2))
```

```
array([[[4.e-323, 4.e-323],
        [4.e-323, 4.e-323],
        [4.e-323, 4.e-323]],

       [[4.e-323, 4.e-323],
        [4.e-323, 4.e-323],
        [4.e-323, 4.e-323]]])
```

It’s not safe to assume that np.empty will return an array of all zeros. In some cases, it may return uninitialized “garbage” values

Arithmetic with NumPy Arrays

Arrays are important because they enable you to express batch operations on data without writing any for loops. NumPy users call this vectorization. Any arithmetic operations between equal-size arrays applies the operation element-wise

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
arr
```

```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

```
arr * arr
```

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

```
arr - arr
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Arithmetic operations with scalars propagate the scalar argument to each element in the array

```
1 / arr
```

```
array([[1.        , 0.5       , 0.33333333],
       [0.25      , 0.2       , 0.16666667]])
```

```
arr **0.5
```

```
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])
```

Comparisons between arrays of the same size yield boolean arrays

```
arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
```

```
arr2
```

```
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])
```

```
arr2 > arr
```

```
array([[False,  True, False],
       [ True, False,  True]])
```

Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists

```
arr = np.arange(10)
```

```
arr
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
arr[5]
```

```
5
```

```
arr[5:8]
```

```
array([5, 6, 7])
```

```
arr[5:8] = 12
```

```
arr
```

```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in arr[5:8] = 12, the value is propagated to the entire selection

Array slice plays an very import role here

```
arr_slice = arr[5:8]
```

```
arr_slice
```

```
array([12, 12, 12])
```

Now, when I change values in arr_slice, the mutations are reflected in the original array arr

```
arr_slice[1] = 12345
```

```
arr
```

```
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
           9])
```

The slice [:] will assign to all values in an array

```
arr_slice[:] = 64
```

```
arr
```

```
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

2D Array

In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
arr2d[2]
```

```
array([7, 8, 9])
```

individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements

```
arr2d[0][2]
```

```
3
```

```
arr2d[0,2]
```

```
3
```

Boolean Indexing

consider an example where we have some data in an array and an array of names with duplicates. I'm going to use here the randn function in numpy.random to generate some random normally distributed data

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
data = np.random.randn(7, 4)
```

```
names
```

```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

```
data
```

```
array([[ 0.54945915, -0.73327062,  0.02544553, -1.44177002],
       [ 0.54334125, -0.19410541,  1.67689246,  0.01489296],
       [ 0.18736736, -1.38409077, -0.051823  , -1.04222121],
       [ 0.09280284, -1.41103905,  0.33126233,  1.59784615],
       [-0.2215394 ,  1.09322488,  0.12326757, -0.81072075],
       [ 1.19161863,  0.9346388 , -0.14098926,  0.74414568],
       [-0.19598436,  0.74282769,  2.14212327,  0.03061705]])
```

If we wanted to select all the rows with corresponding name 'Bob'

We need to basically compare the entity

```
names == 'Bob'
```

```
array([ True, False, False,  True, False, False, False])
```

This boolean array can be passed when indexing the array

```
data[names == 'Bob']
```

```
array([[ 0.54945915, -0.73327062,  0.02544553, -1.44177002],
       [ 0.09280284, -1.41103905,  0.33126233,  1.59784615]])
```

Fancy Indexing

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays.

```
arr = np.empty((8, 4))
```

```
for i in range(8):
  arr[i]=i
```

```
arr
```

```
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
```

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order

```
arr[[4, 3, 0, 6]]
```

```
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping that similarly returns a view on the under- lying data without copying anything.

Arrays have the transpose method

```
arr = np.arange(15).reshape((3, 5))
```

```
arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
arr.T
```

```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

When doing matrix computations, you may do this very often

```
arr = np.random.randn(6, 3)
```

```
arr
```

```
array([[ 0.47609007, -1.11062779, -0.51613794],
       [ 1.57978036,  0.27520094, -2.69021687],
       [-0.07474682,  0.30732697, -0.28951754],
```

```
[ 0.00488057, -0.23164593, -2.15502824],
[-0.65110055,  0.20868373,  0.28230177],
[-0.34603105,  0.58425637,  0.33859501]])
```

```
np.dot(arr.T, arr)
```

```
array([[ 3.27164808, -0.45614903, -4.78552841],
       [-0.45614903,  1.84224375,  0.49985195],
       [-4.78552841,  0.49985195, 12.42597319]])
```

Mathematical and Statistical Methods

A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggre- gations (often called reductions) like sum, mean, and std (standard deviation) either by calling the array instance method or using the top-level NumPy function.

```
arr = np.random.randn(5, 4)
```

```
arr
```

```
array([[-0.92614829, -0.07778001,  1.14335012, -1.98623903],
       [ 0.62339756, -1.22851687,  0.14338313, -0.3229734 ],
       [-0.29682325, -0.2167675 ,  1.31881207, -1.44313694],
       [ 1.03622144, -1.12331283, -1.32203616,  0.70325805],
       [ 0.57901028, -0.8728018 ,  0.71604629, -0.79330305]])
```

```
arr.mean()
```

```
-0.21731801015955657
```

```
np.mean(arr)
```

```
-0.21731801015955657
```

```
arr.sum()
```

```
-4.346360203191131
```

Functions like mean and sum take an optional axis argument that computes the statis- tic over the given axis, resulting in an array with one fewer dimension

```
arr.mean(axis=1)
```

```
array([-0.4617043 , -0.1961774 , -0.1594789 , -0.17646738, -0.09276207])
```

```
arr.sum(axis=0)
```

```
array([ 1.01565774, -3.51917901,  1.99955544, -3.84239437])
```

Here, arr.mean(1) means “compute mean across the columns” where arr.sum(0) means “compute sum down the rows.”

Sorting

NumPy arrays can be sorted in-place with the sort method

```
arr = np.random.randn(6)
```

```
arr
```

```
array([ 2.77856954, -2.59573459,  0.01623111,  1.82516396, -0.03440429,
        0.49547627])
```

```
arr.sort()
```

```
arr
```

```
array([-2.59573459, -0.03440429,  0.01623111,  0.49547627,  1.82516396,
        2.77856954])
```

You can sort each one-dimensional section of values in a multidimensional array in-place along an axis by passing the axis number to sort.

```
arr = np.random.randn(5, 3)
```

```
arr
```

```
array([[-1.64179916,  0.67912643, -0.56570957],
       [ 0.65451301,  0.27910817,  2.83441566],
```

```
            [ 0.66170609, -1.0535962 , -1.18849023],
            [-0.01968455, -0.30647688, -1.2933455 ],
            [ 0.54623269,  1.30840424, -0.93414502]])
```

```
arr.sort(1)
```

```
arr
```

```
array([[-1.64179916, -0.56570957,  0.67912643],
       [ 0.27910817,  0.65451301,  2.83441566],
       [-1.18849023, -1.0535962 ,  0.66170609],
       [-1.2933455 , -0.30647688, -0.01968455],
       [-0.93414502,  0.54623269,  1.30840424]])
```

Unique

A commonly used one is np.unique, which returns the sorted unique values in an array

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
np.unique(names)
```

```
array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

```
ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
```

```
np.unique(ints)
```

```
array([1, 2, 3, 4])
```

Linear Algebra

Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library.

Thus, there is a function dot, both an array method and a function in the numpy namespace, for matrix multiplication

```
x = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
y = np.array([[6., 23.], [-1, 7], [8, 9]])
```

```
x
```

```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

```
y
```

```
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])
```

```
x.dot(y)
```

```
array([[ 28.,  64.],
       [ 67., 181.]])
```

```
np.dot(x, y)
```

```
array([[ 28.,  64.],
       [ 67., 181.]])
```

x.dot(y) is equivalent to np.dot(x, y)

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

2.Pandas

Pandas is an effective library for working with and analysing data. It provides operations for reading and writing data, data cleaning, filtering, aggregation, and more. It also includes data structures like DataFrame and Series.

Installling pandas in pythonIDE

```
!pip install pandas
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (1.5.3)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from
```

Importing some modules from the library or importing the overall library is possible in python

```
import pandas as pd
```

whenever you see pd. in code, it's referring to pandas

You may also find it eas- ier to import Series and DataFrame into the local namespace since they are so fre- quently used

```
from pandas import Series, DataFrame
```

To start with the Pandas library, we need to get comfortable with two data structures: series and dataframe.

Series

A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index.

The simplest series is formed from only an array of data.

```
obj = pd.Series([4, 7, -5, 3])
```

```
obj
```

```
0    4
1    7
2   -5
```

```
3    3
dtype: int64
```

You can get the array representation and index object of the Series via its values and index attributes, respectively

```
obj.values
```

```
array([ 4,  7, -5,  3])
```

```
obj.index # like range(4)
```

```
RangeIndex(start=0, stop=4, step=1)
```

Frequently, it will be preferable to construct a series with an index that labels each data point.

```
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

```
obj2
```

```
d    4
b    7
a   -5
c    3
dtype: int64
```

```
obj2.index
```

```
Index(['d', 'b', 'a', 'c'], dtype='object')
```

you can use labels in the index when selecting single values or a set of values

```
obj2['a']
```

```
-5
```

```
obj2['d'] = 6
```

```
obj2[['c', 'a', 'd']]
```

```
c    3
a   -5
```

```
d    6
dtype: int64
```

Using NumPy functions or NumPy-like operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link

```
obj2[obj2 > 0]
```

```
d    6
b    7
c    3
dtype: int64
```

```
obj2 * 2
```

```
d    12
b    14
a   -10
c     6
dtype: int64
```

```
np.exp(obj2)
```

```
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a map- ping of index values to data values. It can be used in many contexts where you might use a dict

```
'b' in obj2
```

```
True
```

```
'e' in obj2
```

```
False
```

If your data is in a Python dict, you may use it to build a Series by supplying the dict:

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

```
obj3 = pd.Series(sdata)
```

```
obj3
```

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

If you are passing merely a dict, the keys of the dict will be sorted in order in the index of the resultant Series.

You can override this by passing the dict keys in the order you want them to appear in the resulting Series

```
states = ['California', 'Ohio', 'Oregon', 'Texas']
```

```
obj4 = pd.Series(sdata, index=states)
```

```
obj4
```

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Three values from sdata were inserted into the correct places in this case, but since 'California' was not discovered, it displays as NaN (not a number), which pandas uses to indicate missing or NA values. 'Utah' is not present in the final object since it was not included in the states.

To find missing data, utilise Pandas' isnull and notnull functions.

```
pd.isnull(obj4)
```

```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

```
pd.notnull(obj4)
```

```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series also has these as instance methods

```
obj4.isnull()
```

```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

The Series is appropriate for many applications since it automatically aligns by index label during arithmetic operations. To find missing data, utilise the null methods in pandas.

```
obj3
```

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

```
obj4
```

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
obj3 + obj4
```

```
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

Data alignment features will be addressed in more detail later. If you have experience with databases, you can think about this as being similar to a join operation.

DataFrame

A DataFrame represents a rectangular table of data and contains an ordered collec- tion of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index.

There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
 'year': [2000, 2001, 2002, 2001, 2002, 2003],
 'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order

```
frame
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Next steps: Generate code with `frame` View recommended plots

For large DataFrames, the head method selects only the first five rows

```
frame.head()
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

Next steps: Generate code with `frame` View recommended plots

If you specify a sequence of columns, the DataFrame's columns will be arranged in that order

```
pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

If you pass a column that isn't contained in the dict, it will appear with missing values in the result

```
frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
 index=['one', 'two', 'three', 'four','five', 'six'])

frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN
six	2003	Nevada	3.2	NaN

```
frame2.columns
```

```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute

```
frame2['state']
```

```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

```
frame2.year
```

```
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

Rows can also be retrieved by position or name with the special loc attribute

```
frame2.loc['three']
```

```
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

We can also initialise some values as contant for every row

```
frame2['debt'] = 16.5
```

```
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

Next steps: Generate code with `frame2` View recommended plots

```
frame2['debt'] = np.arange(6.)
```

```
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0
six	2003	Nevada	3.2	5.0

Next steps: Generate code with `frame2` View recommended plots

You can transpose the DataFrame (swap rows and columns)

```
frame2.T
```

	one	two	three	four	five	six
year	2000	2001	2002	2001	2002	2003
state	Ohio	Ohio	Ohio	Nevada	Nevada	Nevada
pop	1.5	1.7	3.6	2.4	2.9	3.2
debt	0.0	1.0	2.0	3.0	4.0	5.0

If the DataFrame's columns are different dtypes, the dtype of the values array will be chosen to accommodate all of the columns

```
frame2.values
```

```
array([[2000, 'Ohio', 1.5, 0.0],
       [2001, 'Ohio', 1.7, 1.0],
       [2002, 'Ohio', 3.6, 2.0],
       [2001, 'Nevada', 2.4, 3.0],
       [2002, 'Nevada', 2.9, 4.0],
       [2003, 'Nevada', 3.2, 5.0]], dtype=object)
```

Reindexing

An important method on pandas objects is reindex, which means to create a new object with the data conformed to a new index.

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
```

```
obj
```

```
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

Calling reindex on this Series rearranges the data according to the new index, intro- ducing missing values if any index values were not already present

```
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
```

```
obj2
```

```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

Dropping Entries from an Axis

Dropping one or more entries from an axis is easy if you already have an index array or list without those entries.

```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
obj
```

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

```
new_obj = obj.drop('c')
```

```
new_obj
```

```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

```
obj.drop(['d', 'c'])
```

```
a    0.0
b    1.0
e    4.0
dtype: float64
```

With DataFrame, index values can be deleted from either axis.

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
index=['Ohio', 'Colorado', 'Utah', 'New York'],
columns=['one', 'two', 'three', 'four'])
```

```
data
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Next steps: Generate code with `data` View recommended plots

Calling drop with a sequence of labels will drop values from the row labels (axis 0)

```
data.drop(['Colorado', 'Ohio'])
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

You can drop values from the columns by passing axis=1 or axis='columns'

```
data.drop('two', axis=1)
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

```
data.drop(['two', 'four'], axis='columns')
```

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

Sorting

Sorting a dataset by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the sort_index method, which returns a new, sorted object

```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
```

```
obj.sort_index()
```

```
a    1
b    2
c    3
d    0
dtype: int64
```

The data is sorted in ascending order by default, but can be sorted in descending order

```
obj.sort_index(ascending=False)
```

```
d    0
c    3
b    2
a    1
dtype: int64
```

To sort a Series by its values, use its sort_values method

```
obj = pd.Series([4, 7, -3, 2])
```

```
obj.sort_values()
```

```
2   -3
3    2
0    4
```

```
1    7
dtype: int64
```

When sorting a DataFrame, you can use the data in one or more columns as the sort keys.

```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
```

```
frame
```

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

Next steps: Generate code with `frame` View recommended plots

```
frame.sort_values(by='b')
```

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

To sort by multiple columns, pass a list of names

```
frame.sort_values(by=['a', 'b'])
```

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

Reading Data

CSV Data

we can use read_csv to read it into a DataFrame

```
df = pd.read_csv("/content/Position_Salaries.csv")
```

```
df
```

	Position	Level	Salary
0	Business Analyst	1	45000
1	Junior Consultant	2	50000
2	Senior Consultant	3	60000
3	Manager	4	80000
4	Country Manager	5	110000
5	Region Manager	6	150000
6	Partner	7	200000
7	Senior Partner	8	300000
8	C-level	9	500000
9	CEO	10	1000000

Next steps: Generate code with df View recommended plots

A file will not always have a header row

```
!cat /content/Position_Salaries.csv
```

```
Position,Level,Salary
Business Analyst,1,45000
Junior Consultant,2,50000
Senior Consultant,3,60000
Manager,4,80000
Country Manager,5,110000
Region Manager,6,150000
Partner,7,200000
Senior Partner,8,300000
```

```
C-level,9,500000
CEO,10,1000000
```

To read this file, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself

```
pd.read_csv("/content/Position_Salaries.csv")
```

	Position	Level	Salary
0	Business Analyst	1	45000
1	Junior Consultant	2	50000
2	Senior Consultant	3	60000
3	Manager	4	80000
4	Country Manager	5	110000
5	Region Manager	6	150000
6	Partner	7	200000
7	Senior Partner	8	300000
8	C-level	9	500000
9	CEO	10	1000000

```
pd.read_csv('/content/Position_Salaries.csv', names=['Position', 'Level', 'Salary'])
```

	Position	Level	Salary
0	Position	Level	Salary
1	Business Analyst	1	45000
2	Junior Consultant	2	50000
3	Senior Consultant	3	60000
4	Manager	4	80000
5	Country Manager	5	110000
6	Region Manager	6	150000
7	Partner	7	200000
8	Senior Partner	8	300000
9	C-level	9	500000
10	CEO	10	1000000

```
data = pd.read_csv("/content/Position_Salaries.csv")
```

```
data
```

	Position	Level	Salary
0	Business Analyst	1	45000
1	Junior Consultant	2	50000
2	Senior Consultant	3	60000
3	Manager	4	80000
4	Country Manager	5	110000
5	Region Manager	6	150000
6	Partner	7	200000
7	Senior Partner	8	300000
8	C-level	9	500000
9	CEO	10	1000000

Next steps: Generate code with `data` View recommended plots

```
data.to_csv("/content/Salaries.csv")
```

```
!cat /content/Salaries.csv
```

```
,Position,Level,Salary
0,Business Analyst,1,45000
1,Junior Consultant,2,50000
2,Senior Consultant,3,60000
3,Manager,4,80000
4,Country Manager,5,110000
5,Region Manager,6,150000
6,Partner,7,200000
7,Senior Partner,8,300000
8,C-level,9,500000
9,CEO,10,1000000
```

JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is a much more free-form data format than a tabular text form like CSV

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
 {"name": "Katie", "age": 38,
 "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

. I'll use json here, as it is built into the Python standard library. To convert a JSON string to Python form, use json.loads

```
import json
```

```
result = json.loads(obj)
```

```
result
```

```
{'name': 'Wes',
 'places_lived': ['United States', 'Spain', 'Germany'],
 'pet': None,
 'siblings': [{'name': 'Scott', 'age': 30, 'pets': ['Zeus', 'Zuko']},
  {'name': 'Katie', 'age': 38, 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

json.dumps, on the other hand, converts a Python object back to JSON

```
asjson = json.dumps(result)
```

```
asjson
```

```
'{"name": "Wes", "places_lived": ["United States", "Spain", "Germany"], "pet": null, "s
iblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]}, {"name": "Katie", "a
ge": 38, "pets": ["Sixes", "Stache", "Cisco"]}]}'
```

convert a JSON object or list of objects to a DataFrame or some other data structure for analysis

```
siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])
```

```
siblings
```

	name	age
0	Scott	30
1	Katie	38

Next steps: Generate code with `siblings` View recommended plots

XML and HTML: Web Scraping

Python has many libraries for reading and writing data in the ubiquitous HTML and XML formats. Examples include lxml, Beautiful Soup, and html5lib. While lxml is comparatively much faster in general, the other libraries can better handle malformed HTML or XML files.

you must install some additional libraries used by read_html

```
pip install beautifulsoup4 html5lib
```

```
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: html5lib in /usr/local/lib/python3.10/dist-packages (1.1)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: six>=1.9 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-packages (
```

```
pip install lxml
```

```
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (4.9.4)
```

The pandas.read_html function has a number of options, but by default it searches for and attempts to parse all tabular data contained within

tags.

```
tables = pd.read_html('/content/sample.html')
```

```
len(tables)
```

```
1
```

```
failures = tables[0]
```

```
failures.head()
```

	SKILL 1	SKILL 2	SKILL 3
0	NaN	NaN	NaN
1	Cleaning kaktus in your backyard	Storing some fat for you	Taking you through the desert
2	NaN	NaN	NaN

Next steps: Generate code with `failures` View recommended plots

Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the ExcelFile class or pandas.read_excel function.

To use ExcelFile, create an instance by passing a path to an xls or xlsx file

```
xlsx = pd.ExcelFile('/content/data_dictionary.xlsx')
```

```
pd.read_excel(xlsx)
```

	Unnamed: 0	Unnamed: 1	Unnamed: 2
0	NaN	NaN	NaN
1	NaN	NaN	NaN