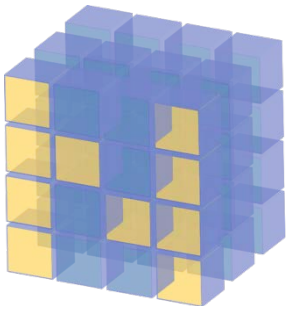
Programming in Python II

NumPy and Pandas



NumPy

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

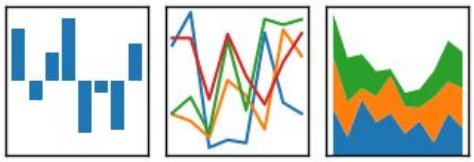


Table of Contents

Lab 1. Using NumPy Arrays

Description	*In this lab, you will learn the fundamentals of NumPy.*
What You Will Learn	• What is NumPy? • How to create arrays from lists • How to get elements from an array • How to create arrays in NumPy • How to perform slicing in NumPy • How to use Boolean array indexing • Exploring the various data types in NumPy • How to perform array math • How to perform cumulative sum in NumPy • How to perform NumPy Sorting • How to sort in reverse order • How to perform assignment of arrays • How to append to an array • How to create a view of an array • How to make a copy of an array • How to perform array comparison • How to expand and reduce the shape of an array • How to compress and flattem an array • How to implement the softmax function using NumPy
Duration	120 minutes

NumPy is the fundamental package for scientific computing with Python.

NumPy is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.

Creating Arrays from Lists

```
import numpy as np

l1 = [1,2,3,4,5]            # l1 is a list in Python
r1 = np.array(l1)           # rank 1 array

print(r1)                   # [1 2 3 4 5]
print(r1.shape)             # (5,); think of this as a 5x1 matrix
print(r1[0])                # 1
print(r1[1])                # 2
```

```
l2 = [6,7,8,9,0]
r2 = np.array([l1,l2])      # rank 2 array
print(r2)
```

```
'''
[[1 2 3 4 5]
 [6 7 8 9 0]]
'''
```

```
print(r2.shape)             # (2,5) 2 rows and 5 columns
print(r2[0,0])              # 1
print(r2[0,1])              # 2
print(r2[1,0])              # 6
```

In Python, a list can contain data of *different* types, and it can grow dynamically

A NumPy array is a matrix of values, all of the *same* type; size of a NumPy array is fixed and changes to it during runtime will create a new array and delete the original

The *rank* of an array is the dimension of the array

The shape of an array is a *tuple* of integers giving the size of the array along each dimension

NumPy arrays are much more efficient compared to Python's list

Row and Column indices are zero-based (0)

Accessing the Elements of an Array using a List

```
n = [4,3,2,1,0]      # list of indices
print(r1[n])         # print the numbers in the array based on their indices
'''
[5 4 3 2 1]
'''
```

```
print(r1[[4,3,2]])
'''
[5 4 3]
'''
```

You can get the elements in an array by passing a list containing the indices of the elements you want

Creating Arrays using NumPy

```
import numpy as np

a1 = np.zeros(2)      # array of rank 1 with all 0s
print(a1.shape)       # (2,)
print(a1[0])          # 0.0
print(a1[1])          # 0.0
```

```
a2 = np.zeros((2,3))  # array of rank 2 with all 0s; 2 rows and 3 columns
print(a2.shape)       # (2,3)
print(a2)
'''
 [[ 0.  0.  0.]
  [ 0.  0.  0.]]
'''
```

```
a3 = np.full((2,3), 8)       # array of rank 2 with all 8s
print(a3)
'''
[[ 8.  8.  8.]
 [ 8.  8.  8.]]
'''
```

```
a4 = np.eye(4)               # 4x4 identity matrix
print(a4)
'''
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
'''

print(a4[0,0])               # [row, column]
'''
1.0
'''
```

```
a5 = np.random.random((2,4)) # rank 2 array (2 rows 4 columns) with random values
                             # in the half-open interval [0.0, 1.0)
print(a5)
'''
[[ 0.21135397  0.39570425  0.25548923  0.05606163]
 [ 0.14495175  0.19093966  0.29366716  0.61189549]]
'''
```

```
a6 = np.arange(10)      # creates a range from 0 to 9
print(a6)               # [0 1 2 3 4 5 6 7 8 9]

a6 = np.arange(0,10,2)  # creates a range from 0 to 9, step 2
print(a6)               # [0 2 4 6 8]
```

Slicing

```
a7 = np.array([[1,2,3,4,5],
               [4,5,6,7,8],
               [9,8,7,6,5]])    # rank 2 array
print(a7)
'''
[[1 2 3 4 5]
 [4 5 6 7 8]
 [9 8 7 6 5]]
'''
```

```
b1 = a7[1:3, :3]     # row 1 to 3 (not inclusive) and first 3 columns
print(b1)
'''
[[4 5 6]
 [9 8 7]]
```

```
'''
```

```
b2 = a7[1:, 2:]       # row 1 onwards and column 2 onwards
                      # b2 is now pointing to a subset of a7
print(b2)
'''
[[6 7 8]
 [7 6 5]]
'''
```

When performing slicing, NumPy returns a reference to the original array; hence b2 is a reference to the original a7 array

```
b2[0,2] = 88          # b2[0,2] is pointing to a7[1,4]; modifying it will modify
                      # the original array
print(a7)
'''
[[ 1  2  3  4  5]
 [ 4  5  6  7 88]
 [ 9  8  7  6  5]]
'''
```

```
b3 = a7[1:, :]        # row 1 onwards and all columns
print(b3)
'''
[[ 4  5  6  7 88]
 [ 9  8  7  6  5]]
'''
```

```
b4 = a7[1, :]          # row 1 and all columns
print(b4)              # b4 is rank 1
                       # [ 4  5  6  7 88]
print(b4.shape)        # (5,)
```

```
b5 = a7[1:2, :]        # row 1 and all columns
print(b5)              # b5 is rank 2
                       # [[ 4  5  6  7 88]]
print(b5.shape)        # (1,5)
```

Note that b4 and b5 have different ranks

Mixing integer indexing with slices yields an array of lower rank; using only slices yields an array of the same rank as the original array

Boolean Array Indexing

```
a8 = np.array([[1,2,3],[4,5,6],[7,8,9]])
even_a8 = (a8 % 2 == 0)      # returns an array of bool representing even numbers
print(even_a8)
'''
[[False  True False]
 [ True False  True]
 [False  True False]]
'''

print(a8[even_a8])           # returns an array of even numbers
# [2 4 6 8]
```

```
print(a8[a8>5])              # returns an array of elements greater than 5
# [6 7 8 9]
```

Boolean Array Indexing is a technique in which you retrieve items from an array based on another array of objects of Boolean type (typically returned from a comparison operation)

Data Types

```
a9 = np.array([1,2,3,4,5])
print(a9.dtype)              # int64
```

```
a10 = np.array([1.0,2.0,3.0,4.0,5.0])
print(a10.dtype)             # float64
```

```
a11 = np.array([1,2], dtype=np.float64)
print(a11.dtype)             # float64
print(a11)                   # [ 1.  2.]
```

Array Math

```
x1 = np.array([[1,2,3],[4,5,6]])
y1 = np.array([[7,8,9],[2,3,4]])

print(x1 + y1)     # same as np.add(x1,y1)
'''
[[ 8 10 12]
 [ 6  8 10]]
'''
```

```
print(x1 - y1)     # same as np.subtract(x1,y1)
'''
[[-6 -6 -6]
 [ 2  2  2]]
'''
```

```
print(x1 * y1)     # same as np.multiply(x1,y1)
'''
```

```
[[ 7 16 27]
 [ 8 15 24]]
'''
```

```
print(x1 / y1)     # same as np.divide(x1,y1)
'''
[[ 0.14285714  0.25        0.33333333]
 [ 2.          1.66666667  1.5       ]]
'''
```

```
x2 = np.array([[1,2,3],[4,5,6]])
y2 = np.array([[7,8],[9,10], [11,12]])
print(np.dot(x2,y2))                    # matrix multiplication
'''
[[ 58  64]
 [139 154]]
'''
```

Cumulative Sum

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

print(a.cumsum())   # prints the cumulative sum of all the elements in the array
                    # [ 1  3  6 10 15 21 28 36 45]
```

```
print(a.cumsum(axis=0))  # sum over rows for each of the 3 columns
'''
[[ 1  2  3]
 [ 5  7  9]
 [12 15 18]]
'''
```

```
print(a.cumsum(axis=1))  # sum over columns for each of the 3 rows
'''
[[ 1  3  6]
 [ 4  9 15]
 [ 7 15 24]]
'''
```

NumPy Sorting

```
import numpy as np

persons = np.array(['Johnny','Mary','Peter','Will','Joe'])
ages    = np.array([34,12,37,5,13])
heights = np.array([1.76,1.2,1.68,0.5,1.25])

print('---Before sorting---')
print(persons)      # ['Johnny' 'Mary' 'Peter' 'Will' 'Joe']
print(ages)         # [34 12 37  5 13]
print(heights)      # [ 1.76  1.2   1.68  0.5   1.25]
```

```
sort_indices = np.argsort(ages)  # performs a sort based on ages
                                 # and returns an array of indices
                                 # indicating the sort order

print('---Sort indices---')
print(sort_indices)              # [3 1 4 0 2]
```

```
print('---After sorting---')
print(persons[sort_indices])      # ['Will' 'Mary' 'Joe' 'Johnny' 'Peter']
print(ages[sort_indices])         # [ 5 12 13 34 37]
print(heights[sort_indices])      # [ 0.5   1.2   1.25  1.76  1.68]
```

```
sort_indices = np.argsort(persons)   # sort based on names
print(persons[sort_indices])         # ['Joe' 'Johnny' 'Mary' 'Peter' 'Will']
print(ages[sort_indices])            # [13 34 12 37  5]
print(heights[sort_indices])         # [ 1.25  1.76  1.2   1.68  0.5 ]
```

Thr argsort() function returns the indices that would sort an array

Sorting in Reverse Order

```
reverse_sort_indices = np.argsort(persons)[::-1] # reverse the order of a list
print(persons[reverse_sort_indices])     # ['Will' 'Peter' 'Mary' 'Johnny' 'Joe']
print(ages[reverse_sort_indices])        # [ 5 37 12 34 13]
print(heights[reverse_sort_indices])     # [ 0.5   1.68  1.2   1.76  1.25]
```

In Python, [::-1] reverses the order of a list

Assigning and Appending Arrays

```
import numpy as np

x = np.arange(10)
y = x            # y is now pointing to x
print(x is y)    # True
print(x)         # [0 1 2 3 4 5 6 7 8 9]
print(y)         # [0 1 2 3 4 5 6 7 8 9]
```

```
y[0] = 88
print(x)         # [88  1  2  3  4  5  6  7  8  9]
print(y)         # [88  1  2  3  4  5  6  7  8  9]
```

```
x = np.append(x, [10,11])
print(x)         # [88  1  2  3  4  5  6  7  8  9 10 11]
```

```
print(y)          # [88  1  2  3  4  5  6  7  8  9]
```

```
y[0] = 0
print(x)          # [88  1  2  3  4  5  6  7  8  9 10 11]
print(y)          # [0 1 2 3 4 5 6 7 8 9]
```

```
z = y.copy()      # creates a copy of y and assign to z
print(y)          # [0 1 2 3 4 5 6 7 8 9]
print(z)          # [0 1 2 3 4 5 6 7 8 9]
```

```
y[0] = 88
print(y)          # [88  1 2 3 4 5 6 7 8 9]
print(z)          # [0 1 2 3 4 5 6 7 8 9]
```

When an NumPy array is assigned to another, both would be pointing to the same instance

The is keyword in Python test two variables to see if they are pointing to the same object in memory

When you append to an array, a new array is created

To create a copy of an array, use the copy() function

Array Subsetting and Comparison

```
names   = np.array(['Ann','Joe','Mark'])
heights = np.array([1.5, 1.78, 1.6])
weights = np.array([65, 46, 59])

bmi = weights/heights **2           # calculate the BMI
print(bmi)                          # [ 28.88888889  14.51836889  23.046875  ]

print("Overweight: " , names[bmi>25])                # Overweight:  ['Ann']
print("Underweight: " , names[bmi<18.5])             # Underweight:  ['Joe']
print("Healthy: " , names[(bmi>=18.5) & (bmi<=24.9)]) # Healthy:  ['Mark']
```

The formula for calculating BMI is:

$$BMI = weight / (height)^2$$

Concatenating Arrays Along the Axes

```
a = np.array([[1, 2, 3], [4, 5,6]])
b = np.array([[7, 8, 9], [10, 11, 12]])
```

```
print(a)
'''
[[1 2 3]
 [4 5 6]]
'''

print(b)
'''
[[ 7  8  9]
 [10 11 12]]
'''

# concatenate along the second axis (column)
print(np.c_[a,b])
'''
shape of a is 2,3
shape of b is 2,3
resultant shape is (2,3+3)
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
'''

# concatenate along the first axis (row)
print(np.r_[a,b])
'''
shape of a is 2,3
shape of b is 2,3
resultant shape is (2+2,3)
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
'''
```

The c_ object allows you to concatenate two arrays along the second axis
The r_ object allows you to concatenate two arrays along the first axis

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

print(a)
'''
[1 2 3]
'''

print(b)
'''
[4 5 6]
'''

print(np.c_[a,b])  # concatenate along the second axis (column)
'''
shape of a is 3,1
shape of b is 3,1
resultant shape is 3,1+1
[[1 4]
 [2 5]
 [3 6]]
'''
```

```
print(np.r_[a,b])  # concatenate along the first axis (row)
'''
shape of a is 3,1
shape of b is 3,1
resultant shape is 3+3,1
[1 2 3 4 5 6]
'''
```

Reshaping Arrays

```
b = np.array([1,2,3,4,5])
print(b)
print(b.shape)
'''
[1 2 3 4 5]
(5,)
'''

#---convert into n rows and 1 column---
print(b.reshape(-1,1))
print(b.reshape(-1,1).shape)
'''
[[1]
 [2]
 [3]
 [4]
 [5]]
(5, 1)
'''

#---convert into 1 row and n columns---
print(b.reshape(1,-1))
print(b.reshape(1,-1).shape)
'''
[[1 2 3 4 5]]
(1, 5)
'''
```

The -1 means we don't know how many rows/columns it will take; let NumPy figure it out

Matrices

```
a = np.array([[1,2],[3,4]])
print(a)
'''
[[1 2]
 [3 4]]
'''

m1 = np.matrix('1 2; 3 4')
print(m1)
'''
[[1 2]
 [3 4]]
'''

m2 = np.matrix([[1, 2], [3, 4]])
```

```
print(m2)
'''
[[1 2]
 [3 4]]
'''

if (np.array_equal(a,m1)):
    print("Equal")
'''
Equal
'''
```

NumPy matrices are strictly 2-dimensional; while NumPy arrays are n-dimensional

A Matrix object is a subclass of ndarray, so it inherits all the attributes and methods of ndarrays

Expanding the Shape

```
import numpy as np
a = np.array([2, 4])
print(a.shape)
'''
(2,)
'''

b = np.expand_dims(a, axis = 0)
print(b.shape)
'''
(1, 2)
'''

print(b)
'''
[[2 4]]
'''
```

The expand_dims() function expands the shape of an array; it insert a new axis that will appear at the axis position in the expanded array shape

```
c = np.expand_dims(a, axis = 1)
print(c.shape)
'''
(2, 1)
'''

print(c)
'''
[[2]
 [4]]
```

```
'''

d = np.array([[2,3,4],[6,7,8]])
print(d)
'''
[[2 3 4]
 [6 7 8]]
'''

print(d.shape)
'''
(2, 3)
'''

e = np.expand_dims(d, axis = 0)          # same as e = d[None, :, :,]
print(e.shape)                           # same as e = d[np.newaxis, :, :,]
'''
(1, 2, 3)
'''

print(e)
'''
[[[2 3 4]
  [6 7 8]]]
'''

e = np.expand_dims(d, axis = 1)          # same as e = d[:, None, :,]
print(e.shape)                           # same as e = d[:, np.newaxis, :,]
'''
(2, 1, 3)
'''

print(e)
'''
[[[2 3 4]]

 [[6 7 8]]]
'''

e = np.expand_dims(d, axis = 2)          # same as e = d[:, :, None]
print(e.shape)                           # same as e = d[:, :, np.newaxis]
'''
(2, 3, 1)
'''

print(e)
'''
[[[2]
  [3]
  [4]]

 [[6]
  [7]
  [8]]]
'''
```

You can expand your array's dimension to a new dimension by using the `np.newaxis` object

The `np.newaxis` object is equivalent to `None`

The `expand_dims()` expands the dimension one at a time, but you can expand multiple dimensions at once using the `np.newaxis` object

Broadcasting

```
import numpy as np

a = np.array([[2,3,4]])
a = a[:,:,None]
print(a)
'''
[[[2]
  [3]
  [4]]]
'''
print(a.shape)       # (1,3,1)

b = np.broadcast_to(a, (1,3,2))
print(b)
'''
[[[2 2]
  [3 3]
  [4 4]]]
'''

c = np.broadcast_to(a, (2,3,3))
print(c)
'''
[[[2 2 2]
  [3 3 3]
  [4 4 4]]

 [[2 2 2]
  [3 3 3]
  [4 4 4]]]
'''

# Error
d = np.broadcast_to(a, (1,4,1))
```

The `broadcast_to()` function broadcasts an array to a new shape

Getting the Index of Items using where

```
import numpy as np

a = np.array([0,1,2,3,4,5,2])
b = np.array([[0,1,2,3,4,5,2],[0,1,2,3,4,5,2]])

print(a)
'''
[0 1 2 3 4 5 2]
'''

print(b)      # b is a 2-d array
'''
[[0 1 2 3 4 5 2]
 [0 1 2 3 4 5 2]]
'''

print(np.where(a == 2))
'''
(array([2, 6]),)
'''

print(np.where(b == 2))
'''
(array([0, 0, 1, 1]), array([2, 6, 2, 6]))
'''

print(np.where((b<2) | (b>5)))
'''
(array([0, 0, 1, 1]), array([0, 1, 0, 1]))
'''

print(np.where((b>2) & (b<5)))
'''
(array([0, 0, 1, 1]), array([3, 4, 3, 4]))
'''

print(b[np.where((b>2) & (b<5))])
print(b[(b>2) & (b<5)])            # same as above
'''
[3 4 3 4]
'''
```

The where() function returns the indices of the items satisfying the specified condition

Argmax and Argmin

```
a = np.array([0,1,2,30,4,5])
print(max(a))
'''
30
'''

print(sum(a))
'''
42
'''
```

```
print(np.argmax(a))
'''
3
'''

print(np.argmin(a))
'''
0
'''
```

Lab 2. Using Pandas - Series

Description	*In this lab, you will learn how to get started with Pandas.*
What You Will Learn	• How to create a series in Pandas • How to create a series using a specified index • How to get the index of a series • How to specify a datetime range
Duration	30 minutes

Pandas stands for Panel Data Analysis
Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

Creating a Series

```
import pandas as pd

series = pd.Series([1,2,3,4,5])
print(series)
'''
0    1
1    2
2    3
3    4
4    5
dtype: int64
'''
```

A Series is a one-dimensional NumPy-like array, with each element having an index (0,1,2,... by default); a Series behaves like a dictionary, with an index:

SERIES

index	element
0	**1**
1	**2**
2	**3**
3	**4**
4	**5**

Creating a Series with a Specified Index

```
series = pd.Series([1,2,3,4,5], index=['a','b','c','d','e'])
print(series)
'''
a    1
b    2
c    3
d    4
e    5
dtype: int64
'''
```

You can specify an optional index for a series using the `index` parameter

SERIES

index	element
a	1
b	2
c	3
d	4
e	5

The index in a series need *not* be unique

Getting the Rows in a Series

```
import pandas as pd

series = pd.Series([1,2,3,4,5], index=[2,'b','c','c','e'])
print(series)
'''
2    1
b    2
c    3
c    4
e    5
'''
```

```
print(series[2])            # 1 - based on the index value
print(series['c'])          # returns a series based on the index value
'''
c    3
c    4
dtype: int64
'''

print(series['c':'e'])      # returns a series based on the index slice
'''
```

```
c    3
c    4
e    5
dtype: int64
'''
```

```
print(series.loc['e'])      # 5  - based on the index value
print(series.loc['c':'e'])  # returns a series based on the index slice
'''
c    3
c    4
e    5
dtype: int64
'''
```

```
print(series.iloc[2])     # 3  - based on the position of the row
print(series.iloc[2:])    # returns a Series based on the positions
'''
c    3
c    4
e    5
dtype: int64
'''
```

The **loc[]** indexer allows you to select rows by *index value*

The **iloc[]** indexer allows you to select rows by *position*

Getting the Index of a Series

```
print(series.index)            # Index(['a', 'b', 'c', 'c', 'e'], dtype='object')
```

You can get the index for a series using the index property

Creating a Date Time Range

```
dates1 = pd.date_range('20160525', periods=12)
print(dates1)
'''
DatetimeIndex(['2016-05-25', '2016-05-26', '2016-05-27', '2016-05-28',
               '2016-05-29', '2016-05-30', '2016-05-31', '2016-06-01',
               '2016-06-02', '2016-06-03', '2016-06-04', '2016-06-05'],
              dtype='datetime64[ns]', freq='D')
'''
```

The `date_range()` function returns a fixed frequency datetime index, with `day` (calendar) as the default frequency

The date can be specified in the `YYYYMMDD` format

The `periods` parameter specifies the range of the dates to generate

```
dates2 = pd.date_range('2016-05-01', periods=12, freq='M')
print(dates2)
'''
DatetimeIndex(['2016-05-31', '2016-06-30', '2016-07-31', '2016-08-31',
               '2016-09-30', '2016-10-31', '2016-11-30', '2016-12-31',
               '2017-01-31', '2017-02-28', '2017-03-31', '2017-04-30'],
              dtype='datetime64[ns]', freq='M')
'''
```

The date can also be specified in the `YYYY-MM-DD` format

The `freq='M'` argument specifies the frequency in months

Note that the day of each month is the last day of the month; if you want the day to start from the first day of the month, use `freq='MS'`

```
dates3 = pd.date_range('2016/05/17 09:00:00', periods=8, freq='H')
print(dates3)
'''
DatetimeIndex(['2016-05-17 09:00:00', '2016-05-17 10:00:00',
               '2016-05-17 11:00:00', '2016-05-17 12:00:00',
               '2016-05-17 13:00:00', '2016-05-17 14:00:00',
               '2016-05-17 15:00:00', '2016-05-17 16:00:00'],
              dtype='datetime64[ns]', freq='H')
'''
```

The date and time can be specified in the `YYYY/MM/DD HH:MM:SS` format

The `freq='H'` argument specifies the frequency in hours

```
# Displaying a specific day for each month
dates4 = pd.date_range('2016-05-01', periods=12, freq='MS') +
         pd.DateOffset(days=24)                              # add another 24 days
print(dates4)
'''
DatetimeIndex(['2016-05-25', '2016-06-25', '2016-07-25', '2016-08-25',
               '2016-09-25', '2016-10-25', '2016-11-25', '2016-12-25',
               '2017-01-25', '2017-02-25', '2017-03-25', '2017-04-25'],
              dtype='datetime64[ns]', freq=None)
'''
```

Lab 3. Using Pandas - DataFrame

Description	*In this lab, you will learn how to use another important data structure in Pandas – DataFrame.*
What You Will Learn	• How to create a dataframe • How to specify the index in a DataFrame • How to get the index of a DataFrame • How to create a DataFrame from a dictionary • How to transpose a DataFrame
Duration	30 minutes

Creating a DataFrame

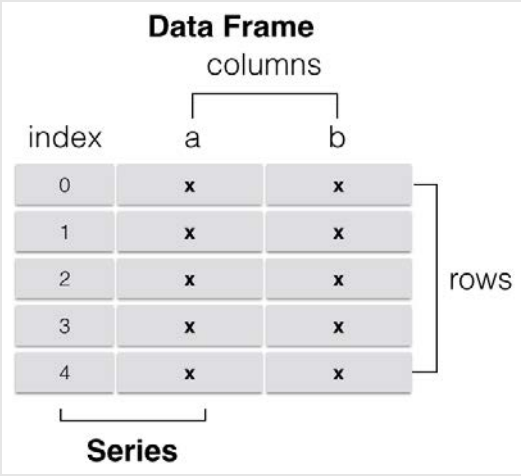
A DataFrame is a two-dimensional NumPy-like array; think of it as a table

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10,4),
                  columns = list('ABCD'))
print(df)
'''
          A         B         C         D
0 -0.034604  0.328973  0.588920  0.507369
1  0.352277  1.454767 -0.628235 -0.056479
2 -2.166704  1.847043 -0.484983  0.258050
3 -0.426012 -1.632111  2.296844 -0.614249
4 -0.156910  0.397193  0.972543  0.928731
5  0.131071  0.470446  0.551828 -0.304608
6  0.113463  0.937863 -0.278803  0.367282
7 -0.444442 -0.018129 -0.365461  1.597252
8  0.168080  0.477345 -0.933668 -0.288315
9 -0.978167  0.346935 -0.660673 -0.497903
'''
```

Like a Series, a DataFrame also has default index of 0,1,2,3, etc

The `random.randn()` function generates an array of the specified shape (10 rows and 4 columns; in this example) filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1

The `columns` parameter allows you to specify a list containing the column headers



Specifying the Column Headers

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10,4),
                  columns = ["winter", "spring", "summer", "autumn"])
print(df)
'''
     winter    spring    summer    autumn
0 -0.588079 -0.595181 -1.809471  1.060956
1  1.956781 -0.817834 -1.066382  0.000447
2  0.539413  0.915071 -1.142391 -0.360765
3 -0.909581  0.733566  1.511741 -0.545792
4  0.161611  0.829067  0.270995  0.439938
5 -1.245777  0.938487 -0.327234 -0.759570
6  0.669592  1.382058  2.200700 -0.137044
7 -0.136089  1.808377  2.683876 -1.738247
8 -0.013207  2.105848 -0.324280  0.914798
9 -0.455316  0.685260  0.282637 -0.705524
'''

# setting the columns after creation
df.columns = ["winter", "spring", "summer", "autumn"]
```

Specifying the Index in a DataFrame

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(4,4),
                  columns = list('ABCD'),
                  index = [1,2,3,4])
print(df)
'''
          A         B         C         D
1  1.958580 -1.381735  0.707835  0.739938
2 -1.181836  0.101458 -0.689767  0.838065
3  1.892268  0.918748 -0.278393 -0.447635
4  0.376219  1.228171  1.270746 -1.184701
'''
```

The index parameter allows you to specify the index to use for the DataFrame

The index in a dataframe need not be unique

```
# setting the index
df.index = [1,2,3,4]
```

The index property allows you to get and set the index of a DataFrame

Assigning Dates to the Index of a DataFrame

```
import pandas as pd
import numpy as np

days = pd.date_range('20150525', periods=10)
print(days)
'''
DatetimeIndex(['2015-05-25', '2015-05-26', '2015-05-27', '2015-05-28',
               '2015-05-29', '2015-05-30', '2015-05-31', '2015-06-01',
               '2015-06-02', '2015-06-03'],
              dtype='datetime64[ns]', freq='D')
'''

df = pd.DataFrame(np.random.randn(10,4),
                  index = days,
                  columns=list('ABCD'))
print(df)
'''
                   A         B         C         D
2015-05-25 -0.400903 -0.189320 -0.533159 -0.275094
2015-05-26  0.088647 -0.392920  0.294948 -0.546524
2015-05-27  0.487690  0.791529  1.041841  1.432100
```

```
2015-05-28  0.261066 -0.989804 -0.641740  0.461687
2015-05-29 -0.621139 -0.057862  0.006402 -0.600170
2015-05-30  1.576299  0.574229  0.431670 -0.143798
2015-05-31  0.261115  0.274218 -0.848949  0.114563
2015-06-01  0.577562  0.861274  0.367894 -1.296077
2015-06-02  1.334573 -0.885431  1.142181 -0.660171
2015-06-03 -0.450458  0.467886  1.276080 -0.338576
'''
```

Creating a DataFrame From a Dictionary

```
import pandas as pd
import numpy as np

dict = {
  'w': 2.5,
  'x': [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0],
  'y': pd.Timestamp('20160517'),
  'z': np.array([5] * 20, dtype='int32'),
}

df2 = pd.DataFrame(dict)     # convert dict to dataframe
print(df2)
```

```
'''
      w  x          y  z
0   2.5  1 2016-05-17  5
1   2.5  2 2016-05-17  5
2   2.5  3 2016-05-17  5
3   2.5  4 2016-05-17  5
4   2.5  5 2016-05-17  5
5   2.5  6 2016-05-17  5
6   2.5  7 2016-05-17  5
7   2.5  8 2016-05-17  5
8   2.5  9 2016-05-17  5
9   2.5  0 2016-05-17  5
10  2.5  1 2016-05-17  5
11  2.5  2 2016-05-17  5
12  2.5  3 2016-05-17  5
13  2.5  4 2016-05-17  5
14  2.5  5 2016-05-17  5
15  2.5  6 2016-05-17  5
16  2.5  7 2016-05-17  5
17  2.5  8 2016-05-17  5
18  2.5  9 2016-05-17  5
19  2.5  0 2016-05-17  5
'''
```

The *keys* in the dictionary will be used as the column names

The `dtypes` property shows the data types of each column

The `Timestamp()` function returns a `datetime64[ns]` object from a string

```
print(df2.dtypes)            # get the data types for each column
'''
w           float64
x             int64
```

```
y    datetime64[ns]
z             int32
dtype: object
'''
```

Viewing Data

```
print(df2.head())
'''
     w  x          y  z
0  2.5  1 2016-05-17  5
1  2.5  2 2016-05-17  5
2  2.5  3 2016-05-17  5
3  2.5  4 2016-05-17  5
4  2.5  5 2016-05-17  5
'''
```

The `head()` function returns the first 5 (default) rows in the DataFrame

You can specify the number of rows to return by passing a number into the function, e.g. `head(10)` to return the top 10 rows

```
print(df2.tail())
'''
      w  x          y  z
15  2.5  6 2016-05-17  5
16  2.5  7 2016-05-17  5
17  2.5  8 2016-05-17  5
18  2.5  9 2016-05-17  5
19  2.5  0 2016-05-17  5
'''
```

The `tail()` function returns the last 5 (default) rows in the DataFrame

You can specify the number of rows to return by passing a number into the function, e.g. `tail(10)` to return the last 10 rows

```
print(df2.values)
'''
[[2.5 1 Timestamp('2016-05-17 00:00:00') 5]
 [2.5 2 Timestamp('2016-05-17 00:00:00') 5]
 [2.5 3 Timestamp('2016-05-17 00:00:00') 5]
...
 [2.5 8 Timestamp('2016-05-17 00:00:00') 5]
 [2.5 9 Timestamp('2016-05-17 00:00:00') 5]
 [2.5 0 Timestamp('2016-05-17 00:00:00') 5]]
'''
```

The values property returns the values of a DataFrame

The result is a NumPy 2D array

```
print(df2.columns)
'''
Index(['w', 'x', 'y', 'z'], dtype='object')
'''
```

The columns property returns the columns of a DataFrame

```
print(df2.T)
'''
                    0                   1                   2  \
w                 2.5                 2.5                 2.5
x                   1                   2                   3
y  2016-05-17 00:00:00  2016-05-17 00:00:00  2016-05-17 00:00:00
z                   5                   5                   5

                    3                   4                   5  \
w                 2.5                 2.5                 2.5
x                   4                   5                   6
y  2016-05-17 00:00:00  2016-05-17 00:00:00  2016-05-17 00:00:00
z                   5                   5                   5

                    6                   7                   8  \
w                 2.5                 2.5                 2.5
x                   7                   8                   9
y  2016-05-17 00:00:00  2016-05-17 00:00:00  2016-05-17 00:00:00
z                   5                   5                   5

                    9                  10                  11  \
w                 2.5                 2.5                 2.5
x                   0                   1                   2
y  2016-05-17 00:00:00  2016-05-17 00:00:00  2016-05-17 00:00:00
z                   5                   5                   5

                   12                  13                  14  \
w                 2.5                 2.5                 2.5
x                   3                   4                   5
y  2016-05-17 00:00:00  2016-05-17 00:00:00  2016-05-17 00:00:00
z                   5                   5                   5

                   15                  16                  17  \
w                 2.5                 2.5                 2.5
x                   6                   7                   8
y  2016-05-17 00:00:00  2016-05-17 00:00:00  2016-05-17 00:00:00
z                   5                   5                   5
```

```
                    18                   19
w                  2.5                  2.5
x                    9                    0
y   2016-05-17 00:00:00  2016-05-17 00:00:00
z                    5                    5
'''
```

The T property transposes (interchanges) the index and columns of the DataFrame

The property T is an accessor to the method `transpose()`

If the dataframe contains a column with the name T, the T property will take precedence

Lab 4. Pandas DataFrame – Selecting Rows and Columns

Description	*In this lab, you will learn how to select rows and columns from DataFrames.*
What You Will Learn	• How to select specific column(s) • How to select row(s) and column(s)
Duration	45 minutes

Creating the DataFrame

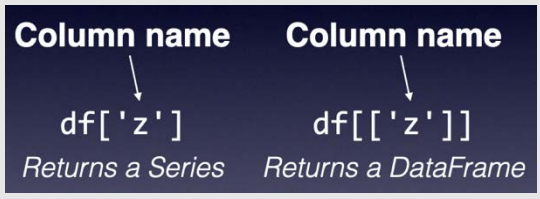
```
dict2 = {
  'w': 2.5,
  'x': pd.Series(3.5, index=list(range(1000)), dtype='float'),
  'y': np.random.randn(1000),
  'z': np.random.randn(1000),
}
df3 = pd.DataFrame(dict2)
print(df3)
'''
      w    x         y         z
0    2.5  3.5  0.360326 -0.463550
1    2.5  3.5  1.320181  0.328169
2    2.5  3.5 -0.519552  1.776190
3    2.5  3.5  0.651012  0.493790
4    2.5  3.5 -1.841215  0.548953
..   ...  ...       ...       ...
995  2.5  3.5  0.355734  0.741537
996  2.5  3.5  2.380434 -1.266545
997  2.5  3.5  0.385439 -1.116222
998  2.5  3.5  0.515891  2.126045
999  2.5  3.5  0.619449 -1.109658
[1000 rows x 4 columns]
'''
```

Extracting a Specific Column in a DataFrame

```
print(df3['z'])
#---OR---
print(df3.z)
'''
0     -0.463550
1      0.328169
2      1.776190
3      0.493790
4      0.548953
         ...
995    0.741537
996   -1.266545
997   -1.116222
998    2.126045
999   -1.109658
Name: z, Length: 1000, dtype: float64
'''
```

```
print(df3['x'] )              # result is a series
print(df3[df3.columns[1]])    # same as above; df3.columns[1] returns 'x'
'''
0      3.5
1      3.5
2      3.5
3      3.5
4      3.5
      ...
995    3.5
996    3.5
997    3.5
998    3.5
999    3.5
Name: x, Length: 1000, dtype: float64
'''
```

```
print(df3[['x']])              # result is a dataframe
print(df3[df3.columns[[1]]])
'''
       x
0    3.5
1    3.5
2    3.5
3    3.5
4    3.5
..   ...
995  3.5
996  3.5
997  3.5
998  3.5
999  3.5

[1000 rows x 1 columns]
'''
```

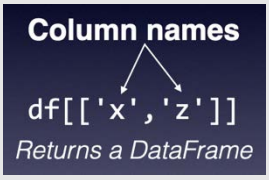


When you specify a value in a DataFrame indexer (**[]**), it will be treated as *column* name

Extracting Multiple Columns in a DataFrame

```
print(df3[['x','y']])              # result is a DataFrame
print(df3[df3.columns[[1,2]]])     # same as above
'''
       x         y
0    3.5  0.360326
1    3.5  1.320181
2    3.5 -0.519552
3    3.5  0.651012
```

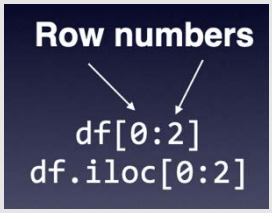
```
4    3.5 -1.841215
..   ...       ...
995  3.5  0.355734
996  3.5  2.380434
997  3.5  0.385439
998  3.5  0.515891
999  3.5  0.619449

[1000 rows x 2 columns]
'''
```



When you specify a list in a DataFrame indexer (**[]**), it will be treated as *column* names

Extracting Rows (by Position) and Columns from a DataFrame

```
print(df3[0:2])           # extract row 0 to 2 (exclusive); result is DataFrame
print(df3.iloc[0:2])      # same as above
'''
     w    x         y         z
0  2.5  3.5  0.360326 -0.463550
1  2.5  3.5  1.320181  0.328169
'''
```



If you specify an integer slice in a DataFrame indexer (**[]**), it will be treated as row numbers

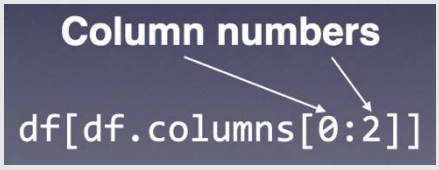
If you specify an string slice in a DataFrame indexer (**[]**), it will be treated as row values

The **iloc[]** indexer allows you to extract rows from a DataFrame

```
# extracting rows and single column
print(df3[0:2]["w"])                  # result is Series
print(df3[0:2][df3.columns[0]])       # same as above
print(df3.iloc[0:2]["w"])             # same as above
```

```
'''
0    2.5
1    2.5
Name: w, dtype: float64
'''

# extracting rows and single column
print(df3[0:2][["w"]])                # result is dataframe
print(df3.iloc[0:2][["w"]])           # same as above
'''
     w
0  2.5
1  2.5
'''

# extracting rows and multiple columns
print(df3[0:2][["w","x"]])            # result is dataframe
print(df3[0:2][df3.columns[[0,1]]])   # same as above
print(df3.iloc[0:2][["w","x"]])       # same as above
'''
     w    x
0  2.5  3.5
1  2.5  3.5
'''

# the following statements are not allowed!
print(df3[0:2][["w":"x"]])            # column slicing by name not allowed
print(df3.iloc[0:2][["w":"x"]])       # same as above

# this is OK
print(df3[0:2][df3.columns[0:2]])     # result is DataFrame; column slicing
'''
     w    x
0  2.5  3.5
1  2.5  3.5
'''
```

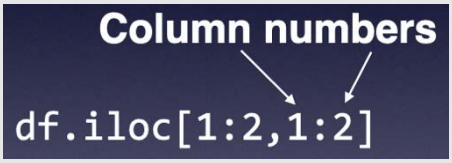
If you are performing column slicing on a Dataframe directly, you need to use the column index - **df.columns[0:2]**




```
print(df3.iloc[0:2, 1])       # get specific column; result is Series
'''
0    3.5
1    3.5
Name: x, dtype: float64
'''
```

```
print(df3.iloc[0:2, [1,3]])   # get specific columns; result is DataFrame
'''
     x         z
0  3.5 -0.463550
1  3.5  0.328169
'''

print(df3.iloc[0:2, 1:3])     # column slicing; result is DataFrame
'''
     x         y
0  3.5  0.360326
1  3.5  1.320181
'''

# error
print(df3.iloc[0:2, "x":"y"]) # cannot perform column slicing using column names
```

When extracting rows using the **iloc[]** indexer, you need to use the column numbers instead of their names when performing column slicing

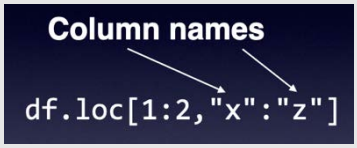


Extracting Rows (by Index Value) and Columns from a DataFrame

```
print(df3.loc[0:2])               # extract by index value; result is DataFrame
'''
     w    x         y         z
0  2.5  3.5  0.360326 -0.463550
1  2.5  3.5  1.320181  0.328169
2  2.5  3.5 -0.519552  1.776190
'''

print(df3.loc[1:2, "x"])          # extract by index value; specific column to
                                  # retrieve; result is Series
'''
1    3.5
2    3.5
Name: x, dtype: float64
'''

print(df3.loc[1:2, ["x"]])        # extract by index value; specific column to
                                  # retrieve; result is DataFrame
'''
     x
1  3.5
2  3.5
'''
```

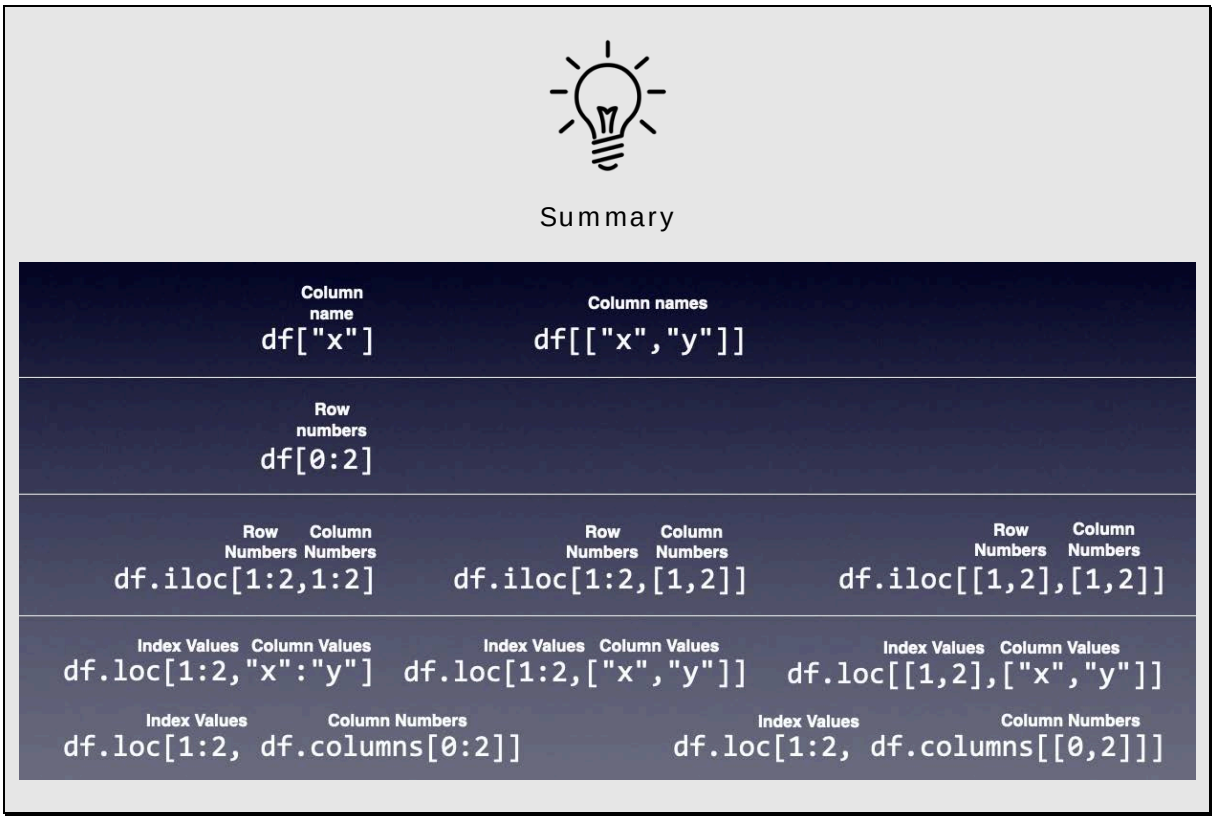
```
print(df3.loc[1:2, ["x","y"]])   # extract by index value; specific columns to
                                 # retrieve; result is DataFrame
'''
     x         y
1  3.5  1.320181
2  3.5 -0.519552
'''

print(df3.loc[1:2, "x":"z"])     # extract by index value; column slicing;
                                 # result is DataFrame
'''
     x         y         z
1  3.5  1.320181  0.328169
2  3.5 -0.519552  1.776190
'''
```

When extracting rows using the **loc[]** indexer, you can also perform column slicing via the column names




```
print(df3.loc[1:2, df3.columns[0]])   # extract by index value; specific column;
                                      # result is Series
'''
1    2.5
2    2.5
Name: w, dtype: float64
'''

print(df3.loc[1:2, df3.columns[[0,1]]])   # extract by index value; specific
                                          # columns; result is DataFrame
print(df3.loc[1:2, df3.columns[0:2]])     # extract by index value; column
                                          # slicing; result is DataFrame
'''
     w    x
1  2.5  3.5
2  2.5  3.5
'''

print(df3.loc[1:2]['x'])                  # result is Series
'''
1    3.5
2    3.5
Name: x, dtype: float64
'''

print(df3.loc[1:2][['x']])                # result is DataFrame
'''
     x
1  3.5
2  3.5
'''
```

```
print(df3.loc[1:2][['x','y']])            # result is DataFrame
'''
     x         y
1  3.5  1.320181
2  3.5 -0.519552
'''

print(df3.loc[2][df3.columns[0:2]])       # result is Series
'''
w    2.5
x    3.5
Name: 2, dtype: float64
'''
```



Test Yourself

1. Given the following code:

```
df = pd.DataFrame(3 * np.random.rand(6, 3),
                  index=[11, 22, 33, 44, 55, 66],
                  columns=[4, 5, 6])
print(df)
```

   a. Print the second columns as a series
   b. Print the second columns as a dataframe
   c. Print the first and last columns
   d. Print the second to fourth rows
   e. Print the last 2 columns in the last 2 rows

Solutions

1.

a. Print the second columns as a series

```
df[5]
df[df.columns[1]]
```

b. Print the second columns as a dataframe

```
df[[5]]
```

c. Print the first and last columns

```
df[df.columns[[0,2]]]
```

d. Print the second to fourth rows

```
df[1:4]
```

e. Print the last 2 columns in the last 2 rows

```
df.iloc[-2:, -2:]
```

Lab 5. Pandas DataFrame – Sorting Rows and Columns

Description	*In this lab, you will learn how to sort rows and columns in DataFrames.*
What You Will Learn	• How to sort a DataFrame by axis • How to sort a DataFrame by value
Duration	20 minutes

Creating the DataFrame

```
dict = {
  'w': [1,1,3,4,5],
  'x': pd.Series(3.5, index=list(range(5)), dtype='float'),
  'y': pd.Timestamp('20160517'),
  'z': [3,2,4,1,3],
}

df = pd.DataFrame(dict2)
print(df)
'''
       w    x         y         z
0    2.5  3.5  0.360326 -0.463550
1    2.5  3.5  1.320181  0.328169
2    2.5  3.5 -0.519552  1.776190
3    2.5  3.5  0.651012  0.493790
4    2.5  3.5 -1.841215  0.548953
..   ...  ...       ...       ...
995  2.5  3.5  0.355734  0.741537
996  2.5  3.5  2.380434 -1.266545
997  2.5  3.5  0.385439 -1.116222
998  2.5  3.5  0.515891  2.126045
999  2.5  3.5  0.619449 -1.109658

[1000 rows x 4 columns]
'''
```

Sorting by Axis

```
print(df.sort_index(axis=0, ascending=False))  # axis = 0 means sort by index
'''
       w    x         y         z
999  2.5  3.5  0.619449 -1.109658
998  2.5  3.5  0.515891  2.126045
997  2.5  3.5  0.385439 -1.116222
996  2.5  3.5  2.380434 -1.266545
995  2.5  3.5  0.355734  0.741537
..   ...  ...       ...       ...
4    2.5  3.5 -1.841215  0.548953
3    2.5  3.5  0.651012  0.493790
2    2.5  3.5 -0.519552  1.776190
1    2.5  3.5  1.320181  0.328169
0    2.5  3.5  0.360326 -0.463550

[1000 rows x 4 columns]
'''
```

The **sort_index()** function sorts the rows of a DataFrame based on the axis specified – 0 means sort by index, 1 means sort by columns

```
print(df.sort_index(axis=1, ascending=False))  # axis = 1 means sort by column
'''
            z         y    x    w
0   -0.463550  0.360326  3.5  2.5
1    0.328169  1.320181  3.5  2.5
2    1.776190 -0.519552  3.5  2.5
3    0.493790  0.651012  3.5  2.5
4    0.548953 -1.841215  3.5  2.5
..        ...       ...  ...  ...
995  0.741537  0.355734  3.5  2.5
996 -1.266545  2.380434  3.5  2.5
997 -1.116222  0.385439  3.5  2.5
998  2.126045  0.515891  3.5  2.5
999 -1.109658  0.619449  3.5  2.5

[1000 rows x 4 columns]
'''
```

Sorting by Values

```
print(df.sort_values(by=['z']))   # sort by the column named z
'''
       w    x         y         z
788  2.5  3.5 -0.061771 -2.996636
391  2.5  3.5  1.084164 -2.801160
742  2.5  3.5  0.333071 -2.783199
781  2.5  3.5  0.322948 -2.705775
501  2.5  3.5  0.159321 -2.652255
..   ...  ...       ...       ...
94   2.5  3.5 -0.032171  2.868832
411  2.5  3.5  0.118202  2.926211
435  2.5  3.5 -0.487092  3.111315
255  2.5  3.5 -1.143315  3.346644
60   2.5  3.5 -0.372109  3.983153

[1000 rows x 4 columns]
'''
```

The sort_values() function sorts the rows of a DataFrame based on the column specified

Sorting by Mulitple Columns

```
print(df.sort_values(by=['w', 'z']))
'''
       w    x         y         z
788  2.5  3.5 -0.061771 -2.996636
391  2.5  3.5  1.084164 -2.801160
742  2.5  3.5  0.333071 -2.783199
781  2.5  3.5  0.322948 -2.705775
501  2.5  3.5  0.159321 -2.652255
..   ...  ...       ...       ...
94   2.5  3.5 -0.032171  2.868832
411  2.5  3.5  0.118202  2.926211
435  2.5  3.5 -0.487092  3.111315
255  2.5  3.5 -1.143315  3.346644
60   2.5  3.5 -0.372109  3.983153

[1000 rows x 4 columns]
'''
```

```
print(df.sort_values(by=['z', 'w']))
'''
       w    x         y         z
788  2.5  3.5 -0.061771 -2.996636
391  2.5  3.5  1.084164 -2.801160
742  2.5  3.5  0.333071 -2.783199
781  2.5  3.5  0.322948 -2.705775
501  2.5  3.5  0.159321 -2.652255
..   ...  ...       ...       ...
94   2.5  3.5 -0.032171  2.868832
411  2.5  3.5  0.118202  2.926211
435  2.5  3.5 -0.487092  3.111315
255  2.5  3.5 -1.143315  3.346644
60   2.5  3.5 -0.372109  3.983153

[1000 rows x 4 columns]
'''
```

Lab 6. Pandas DataFrame – Applying Functions

Description	*In this lab, you will learn how to apply functions to DataFrames.*
What You Will Learn	• How to apply a lambda function to a DataFrame • How to apply a built-in function to a DataFrame • How to apply your user-defined function to a DataFrame • How to apply aggregate functions to a DataFrame
Duration	30 minutes

Creating the DataFrame

```
import pandas as pd

matrix = [
    (1, 2, 3),
    (4, 5, 6),
    (7, 8, 9),
    (10, 11, 12),
    (13, 14, 15),
    (16, 17, 18)
]

df = pd.DataFrame(matrix, columns=list('abc'))
print(df)
'''
    a   b   c
0   1   2   3
1   4   5   6
2   7   8   9
3  10  11  12
4  13  14  15
5  16  17  18
'''
```

Applying a Function

```
df_modified = df.apply(lambda x : x * 2)
print(df_modified)
'''
    a   b   c
0   2   4   6
1   8  10  12
2  14  16  18
3  20  22  24
4  26  28  30
5  32  34  36
'''
```

Summing the Rows

```
sum = df.apply(np.sum, axis=0)        # apply the sum() function on the items
                                      # over each row
print(sum)
'''
a    51
b    57
c    63
dtype: int64
'''
```

```
df_row_summed = df.append(sum, ignore_index=True)     # append the series (sum)
                                                      # to the df
print(df_row_summed)
'''
    a   b   c
0   1   2   3
1   4   5   6
2   7   8   9
3  10  11  12
4  13  14  15
5  16  17  18
6  51  57  63
'''
```

Summing the Columns

```
sum = df.apply(np.sum, axis=1)        # apply the sum() function on the items
                                      # over each column
print(sum)
'''
a    51
b    57
c    63
dtype: int64
'''

df['s'] = sum                         # append the series (sum) to the df
print(df)
'''
    a   b   c    s
0   1   2   3   12
1   4   5   6   30
2   7   8   9   48
3  10  11  12   66
4  13  14  15   84
5  16  17  18  102
'''
```

Applying Aggregate Functions

```
df = pd.DataFrame(
    [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]
    ],
    columns=['A', 'B', 'C'])

print(df)
'''
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
'''

print(df[['A','B']].agg(['sum', 'min', 'max', 'mean']))
'''
         A     B
sum   12.0  15.0
min    1.0   2.0
max    7.0   8.0
mean   4.0   5.0
'''
```

Understanding the apply() Function

```
import pandas as pd

matrix = [
    (1, 2, 3),
    (4, 5, 6),
    (7, 8, 9),
]

df = pd.DataFrame(matrix, columns=list('abc'))

def check_values(x):
    print(x)

df.apply(check_values, axis=0)
'''
0    1
1    4
2    7
Name: a, dtype: int64
0    2
1    5
2    8
Name: b, dtype: int64
0    3
1    6
2    9
Name: c, dtype: int64
'''

df.apply(check_values, axis=1)
'''
a    1
b    2
c    3
Name: 0, dtype: int64
a    4
b    5
c    6
Name: 1, dtype: int64
a    7
b    8
c    9
Name: 2, dtype: int64
'''
```

```
import pandas as pd

matrix = [
    (1, 2, 3),
    (4, 5, 6),
    (7, 8, 9),
]

df = pd.DataFrame(matrix, columns=list('abc'))

def check_values(x):
    if x[0]>2 and x[1]>2 and x[2]>2:
        return x*2
    else:
        return x

df = df.apply(check_values, axis=0)
print(df)
'''
```

```
   a  b   c
0  1  2   6
1  4  5  12
2  7  8  18
'''
```

Lab 7. Pandas DataFrame – Generating Statistics

Description	*In this lab, you will learn how to generate statistics from a DataFrame.*
What You Will Learn	• How to generate statistics from a DataFrame • How to generate statistics from a specific column
Duration	25 minutes

```
import pandas as pd

matrix = [
    (1, "r", 3),
    (4, "g", 6),
    (7, "b", 9),
    (2, "r", 4),
    (8, "g", 2),
]

df = pd.DataFrame(matrix, columns=list('abc'))

print(df)
'''
   a  b  c
0  1  r  3
1  4  g  6
2  7  b  9
3  2  r  4
4  8  g  2
'''

print(df.describe())
'''
              a         c
count  5.00000  5.000000
mean   4.40000  4.800000
std    3.04959  2.774887
min    1.00000  2.000000
25%    2.00000  3.000000
50%    4.00000  4.000000
75%    7.00000  6.000000
max    8.00000  9.000000
'''

print(df.mean())
'''
a    4.4
c    4.8
dtype: float64
'''

print(df.count())
'''
a    5
b    5
c    5
dtype: int64
'''

print(df["b"].value_counts())
'''
g    2
r    2
b    1
Name: b, dtype: int64
'''
```

Lab 8. Pandas DataFrame – Adding and Removing Rows and Columns

Description	*In this lab, you will learn how to add/remove rows/columns to a DataFrame.*
What You Will Learn	• How to use the crosstab() function to generate a frequency table • How to add column(s) to a DataFrame • How to add row(s) to a DataFrame • How to remove row(s) from a DataFrame • How to remove column(s) from a DataFrame
Duration	40 minutes

Adding Columns to a DataFrame

```
import numpy as np
import pandas as pd

df = pd.DataFrame(
  {
    "Gender": ['Male','Male','Female','Female','Female'],
    "Team"  : [1,2,3,3,1]
  })

print(df)
'''
   Gender  Team
0    Male     1
1    Male     2
2  Female     3
3  Female     3
4  Female     1
'''
```

```
print("Displaying the distribution of genders in each team")
print(pd.crosstab(df.Gender, df.Team))
'''
Displaying the distribution of genders in each team
Team    1  2  3
Gender
Female  1  0  2
Male    1  1  0
'''
```

The **crosstab()** function computes a simple cross-tabulation of two (or more) factors

```
schools = np.array(["Cambridge","Oxford","Oxford","Cambridge","Oxford"])

df["School"] = schools
print(df)
'''
   Gender  Team     School
0    Male     1  Cambridge
```

```
1    Male     2     Oxford
2  Female     3     Oxford
3  Female     3  Cambridge
4  Female     1     Oxford
'''
```

Adding Rows

```
print(df.append({'Gender':'Male', 'Team':2, 'School':'Cambridge'},
      ignore_index=True))
'''
   Gender  Team     School
0    Male     1  Cambridge
1    Male     2     Oxford
2  Female     3     Oxford
3  Female     3  Cambridge
4  Female     1     Oxford
5    Male     2  Cambridge
'''
```

Be aware that the **append()** function does not modify the original dataframe unless you specify the **inplace=True** parameter

```
print(df.append(pd.Series(['Female',2, 'Oxford'],
                          index = df.columns),
                ignore_index=True))
'''
   Gender  Team     School
0    Male     1  Cambridge
1    Male     2     Oxford
2  Female     3     Oxford
3  Female     3  Cambridge
4  Female     1     Oxford
5  Female     2     Oxford
'''
```

If ignore_index is set to False, you need to specify the index value, like this:

```
print(df.append(pd.Series(['Female', 2, 'Oxford'],
                          index = df.columns, name=5),
                ignore_index=False))
```

Dropping Rows

```
import pandas as pd

data = {'name': ['Janet', 'Nad', 'Timothy', 'June', 'Amy'],
        'year': [2012, 2012, 2013, 2014, 2014],
        'reports': [6, 13, 14, 1, 7]}

df = pd.DataFrame(data, index =
       ['Singapore', 'China', 'Japan', 'Sweden', 'Norway'])
print(df)
'''
              name  reports  year
Singapore    Janet        6  2012
China          Nad       13  2012
Japan      Timothy       14  2013
Sweden        June        1  2014
Norway         Amy        7  2014
'''

print(df.drop('China'))                # drop rows based on index value
'''
              name  year  reports
Singapore    Janet  2012        6
Japan      Timothy  2013       14
Sweden        June  2014        1
Norway         Amy  2014        7
'''

print(df.drop(['China', 'Japan']))     # drop rows based on index value
'''
            name  year  reports
Singapore  Janet  2012        6
Sweden      June  2014        1
Norway       Amy  2014        7
'''

print(df.drop(df.index[0]))            # drop row based on row position
'''
           name  year  reports
China       Nad  2012       13
Japan   Timothy  2013       14
Sweden     June  2014        1
Norway      Amy  2014        7
'''

print(df.drop(df.index[2:4]))          # drop row 2 through 4 (not inclusive)
'''
            name  year  reports
Singapore  Janet  2012        6
China        Nad  2012       13
Norway       Amy  2014        7
'''

print(df.drop(df.index[[2,4]]))        # drop row 2 and 4
'''
            name  year  reports
Singapore  Janet  2012        6
China        Nad  2012       13
Sweden      June  2014        1
'''
```

```
print(df.drop(df.index[-2]))            # drop second last row
'''
              name  year  reports
Singapore    Janet  2012        6
China          Nad  2012       13
Japan      Timothy  2013       14
Norway         Amy  2014        7
'''

df.drop(df.index)                       # drop all rows; operation is not inplace,
                                        # i.e. the original df is not modified,
                                        # the function returns the modified df

print(df.drop(df.index[0:2])) # drop first 2 rows
'''
        name  year  reports
Sweden  June  2014        1
Norway   Amy  2014        7
'''
```

Note that the `drop()` function returns a dataframe that has the rows dropped; the original dataframe is unchanged; if you want to modify the original dataframe, use the `inplace=True` argument in the `drop()` function

Dropping Columns

```
print(df.drop('reports', axis=1))              # drop column
'''
              name  year
Singapore    Janet  2012
China          Nad  2012
Japan      Timothy  2013
Sweden        June  2014
Norway         Amy  2014
'''
```

Lab 9. Pandas DataFrame – Querying

Description	*In this lab, you will learn how to select rows and columns based on cell values.*
What You Will Learn	• How to select rows based on cell value • How to specify multiple search conditions • How to use the any() and all() functions to check for items over a specified axis
Duration	40 minutes

Selecting Rows Based on Cell Value

```
print(df[df.name != 'Nad'])
'''
              name  year  reports
Singapore    Janet  2012        6
Japan      Timothy  2013       14
Sweden        June  2014        1
Norway         Amy  2014        7
'''

print(df[df.name == 'Janet'])
'''
            name  year  reports
Singapore  Janet  2012        6
'''
```

Specifying Multiple Conditions

```
janet = (df.name=="Janet")
print(janet)
'''
Singapore     True
China        False
Japan        False
Sweden       False
Norway       False
Name: name, dtype: bool
'''

gt_2013 = df.year>2013
print(gt_2013)
'''
Singapore    False
China        False
Japan        False
Sweden        True
Norway        True
Name: year, dtype: bool
'''

print(janet | gt_2013)
'''
Singapore     True
China        False
Japan        False
Sweden        True
Norway        True
dtype: bool
'''
```

```
print(df[(df.name=="Janet") | (df.year>2013)])      # get row whose name is Janet
                                                    # OR whose's year is greater
                                                    # than 2013
'''
            name  year  reports
Singapore  Janet  2012        6
Sweden      June  2014        1
Norway       Amy  2014        7
'''
```

```
condition = (df['name']=="Janet") & (df['year']==2012)
print(df[condition])
'''
            name  year  reports
Singapore  Janet  2012        6
'''

print(df[df.name.str.contains("un")])          # get row(s) whose name contains
                                               # "un"
'''
        name  year  reports
Sweden  June  2014        1
'''

print(df[df.name.str.contains("un|my")])       # get row(s) whose name contains
                                               # "un" or "my"
'''
        name  year  reports
Sweden  June  2014        1
Norway   Amy  2014        7
'''

names = ["Timothy","Amy"]
print(df[df.name.isin(names)])                 # get row(s) whose name is either
                                               # Timothy or Amy
'''
           name  year  reports
Japan   Timothy  2013       14
Norway      Amy  2014        7
'''
```

Using the any() and all() function

```
df = pd.DataFrame([[1,2,3],[3,4,5],[3,1,4],[1,2,1]],
                  columns=['A','B','C'])
print(df)
'''
   A  B  C
0  1  2  3
1  3  4  5
2  3  1  4
3  1  2  1
'''

# get all rows with at least one column containing at minimum value of 2
print(df[(df > 2).any(axis=1)])
'''
   A  B  C
0  1  2  3
1  3  4  5
2  3  1  4
'''

# get all rows with at all columns containing at least a value of more than 1
print(df[(df > 1).all(axis=1)])
'''
   A  B  C
1  3  4  5
'''
```

Lab 10. Pandas DataFrame – Summarizing Data using GroupBy

Description	*In this lab, you will learn how to group data using the groupby() function.*
What You Will Learn	• How to load a JSON string into a DataFrame • How to use the groupby() function • How to iterate through a GroupBy object • How to summarize a GroupBy object
Duration	60 minutes

Example 1

```
import pandas as pd

scores = {'Zone': ['North', 'North', 'South', 'South',
                   'East', 'East', 'West', 'West'],
          'School': ['Rushmore', 'Rushmore','Bayside','Rydell',
                     'Shermer','Shermer','Ridgemont','Hogwarts'],
          'Name': ['Jonny', 'Mary', 'Joe', 'Jakob',
                   'Jimmy', 'Erik', 'Lam', 'Yip'],
          'Math': [78, 39, 76, 56, 67, 89, 100, 55],
          'Science': [70, 45, 68, 90, 45, 66, 89, 32]}

df = pd.DataFrame(scores, columns =
                  ['Zone', 'School', 'Name',
                   'Science', 'Math'])
print(df)
'''
    Zone     School   Name  Science  Math
0  North   Rushmore  Jonny       70    78
1  North   Rushmore   Mary       45    39
2  South    Bayside    Joe       68    76
3  South     Rydell  Jakob       90    56
4   East    Shermer  Jimmy       45    67
5   East    Shermer   Erik       66    89
6   West  Ridgemont    Lam       89   100
7   West   Hogwarts    Yip       32    55
'''
```

When importing JSON data into a DataFrame, you can specify the keys in the JSON string to import as columns in the DataFrame

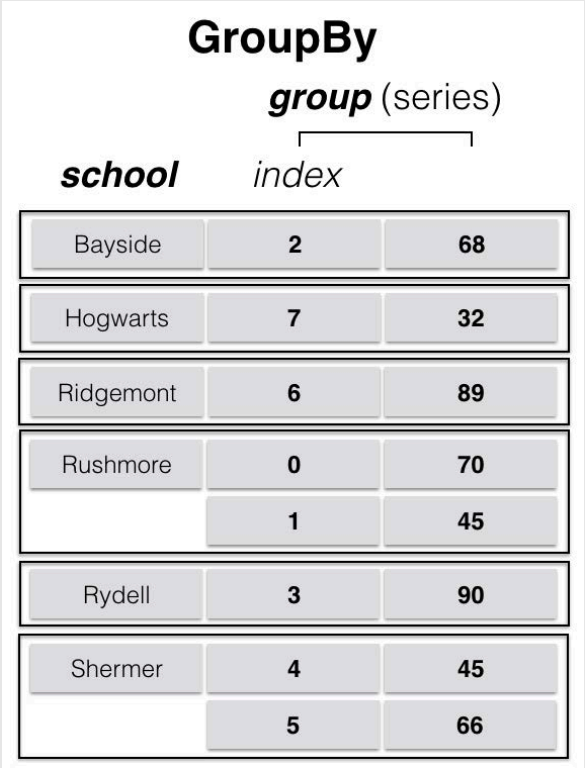
Grouping the Science Score by School

```
# Science score grouped by School
group_science_by_school = df['Science'].groupby(df['School']) # returns a GroupBy
                                                              # object
print(list(group_science_by_school))
'''
# formatted for clarity
[('Bayside', 2    68 Name: Science, dtype: int64),
 ('Hogwarts', 7    32 Name: Science, dtype: int64),
```

```
 ('Ridgemont', 6    89 Name: Science, dtype: int64),
 ('Rushmore', 0    70 1    45 Name: Science, dtype: int64),
 ('Rydell', 3    90 Name: Science, dtype: int64),
 ('Shermer', 4    45 5    66 Name: Science, dtype: int64)]
'''
```

The `groupby()` function returns a `GroupBy` object

You can visualize the `group_science_by_school` object as follows:



Iterating Through a GroupBy Object

```
for school, group in group_science_by_school:
    print('===', school, '===')        # print the school
    print (group)                      # group is a Series;
    print()
'''
=== Bayside ===
2    68
Name: Science, dtype: int64

=== Hogwarts ===
7    32
Name: Science, dtype: int64

=== Ridgemont ===
6    89
Name: Science, dtype: int64
```

```
=== Rushmore ===
0    70
1    45
Name: Science, dtype: int64

=== Rydell ===
3    90
Name: Science, dtype: int64

=== Shermer ===
4    45
5    66
Name: Science, dtype: int64
'''
```

```
for school, group in group_science_by_school:
    print('===', school, '===')        # print the school
    for i in group.index:              # .index returns an Index
                                       # pointing to the original dataframe
        print(df.values[i][2],end=' ') # column 2 in dataframe is Name
        print(df.values[i][4])         # column 4 in dataframe is Math
    print()
'''
=== Bayside ===
Joe 76

=== Hogwarts ===
Yip 55

=== Ridgemont ===
Lam 100

=== Rushmore ===
Jonny 78
Mary 39

=== Rydell ===
Jakob 56

=== Shermer ===
Jimmy 67
Erik 89
'''
```

Summarizing a GroupBy Object

```
# descriptive statistics by group
print(group_science_by_school.describe())
'''
           count  mean        std   min    25%   50%    75%   max
School
Bayside      1.0  68.0        NaN  68.0  68.00  68.0  68.00  68.0
Hogwarts     1.0  32.0        NaN  32.0  32.00  32.0  32.00  32.0
Ridgemont    1.0  89.0        NaN  89.0  89.00  89.0  89.00  89.0
Rushmore     2.0  57.5  17.677670  45.0  51.25  57.5  63.75  70.0
Rydell       1.0  90.0        NaN  90.0  90.00  90.0  90.00  90.0
Shermer      2.0  55.5  14.849242  45.0  50.25  55.5  60.75  66.0
'''
```

The describe() function generates various summary statistics, excluding NaN values

Finding the Average of Science for Each School

```
# show the Science mean for each school
print(group_science_by_school.mean())
'''
School
Bayside      68.0
Hogwarts     32.0
Ridgemont    89.0
Rushmore     57.5
Rydell       90.0
Shermer      55.5
Name: Science, dtype: float64
'''
```

Example 2

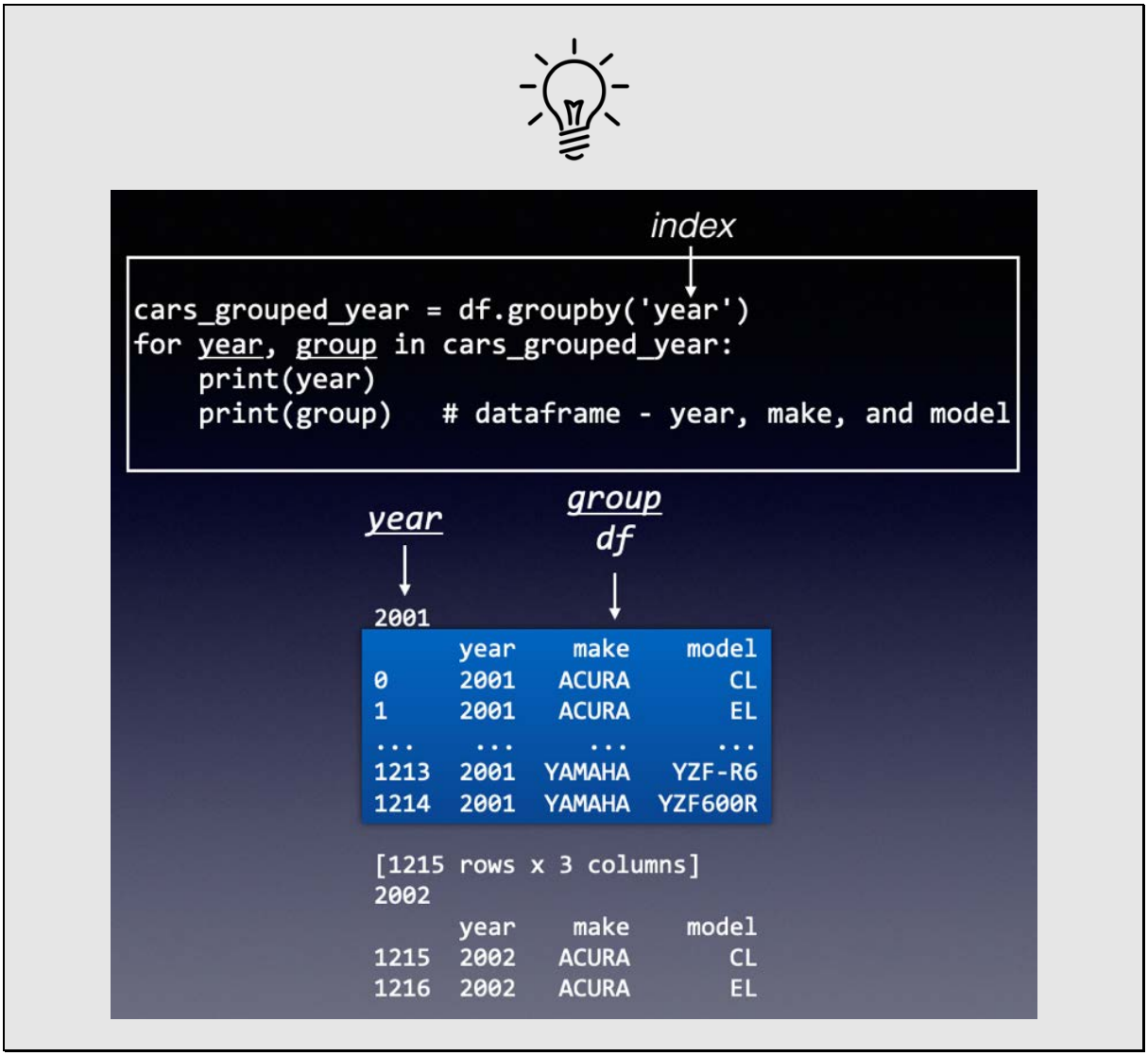
This example uses the dataset "vehicle-make-model-data" from
https://github.com/arthurkao/vehicle-make-model-data

```
import pandas as pd

df = pd.read_csv("csv_data.csv")
print(df)
'''
       year    make    model
0      2001   ACURA       CL
1      2001   ACURA       EL
2      2001   ACURA  INTEGRA
3      2001   ACURA      MDX
4      2001   ACURA      NSX
...     ...     ...      ...
19767  2015  YAMAHA    SR400
19768  2016     KIA  SORENTO
19769  2016   MAZDA        6
19770  2016   MAZDA     CX-5
19771  2016   VOLVO     XC90

[19772 rows x 3 columns]
'''
```

Display all the Cars Grouped by Year

```
cars_grouped_year = df.groupby('year')
for year, group in cars_grouped_year:
    print(year)
    print(group)                          # dataframe - year, make, and model
```

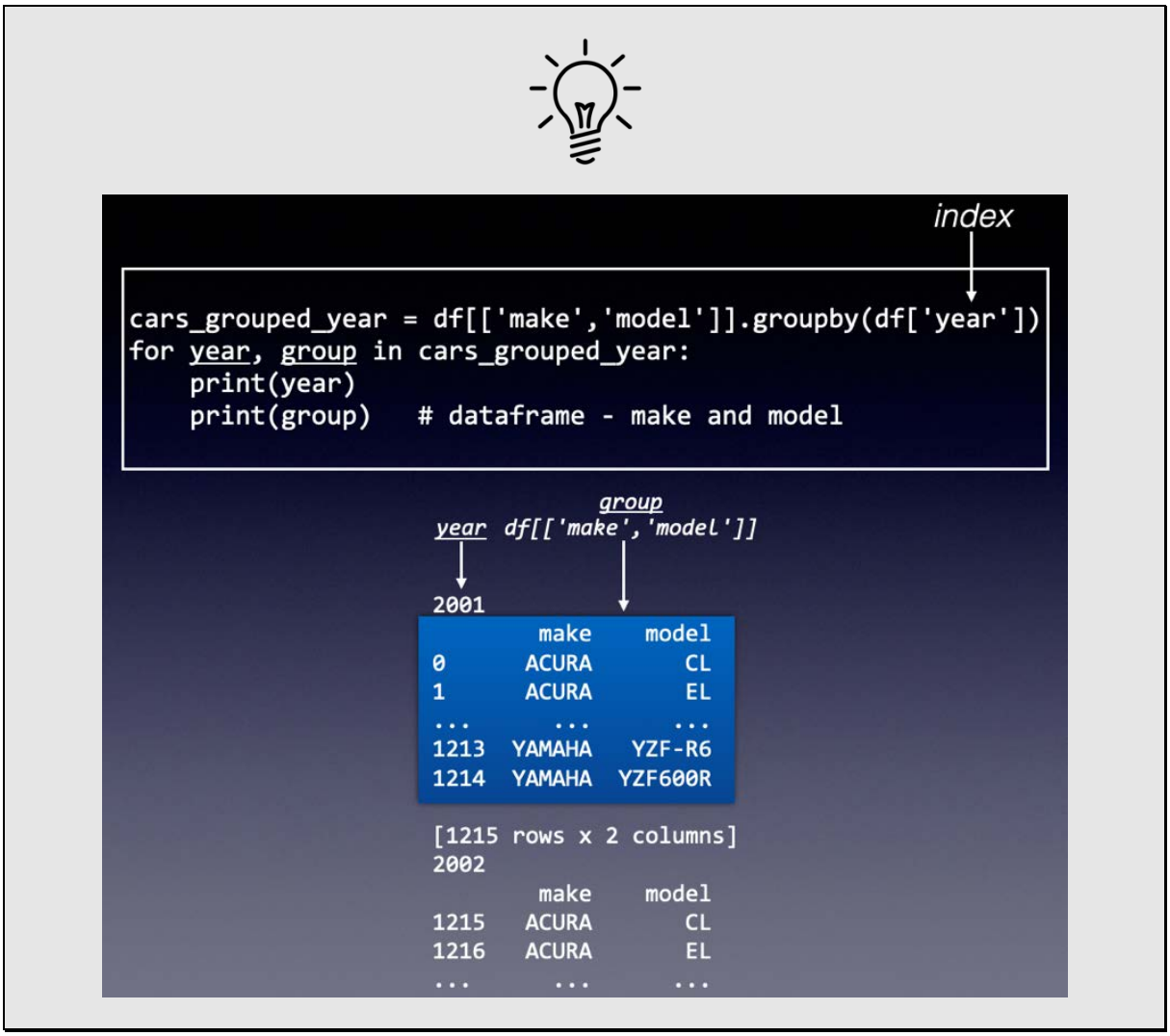
```
'''
2001
      year    make    model
0     2001   ACURA       CL
1     2001   ACURA       EL
...    ...     ...      ...
1213  2001  YAMAHA   YZF-R6
1214  2001  YAMAHA  YZF600R

[1215 rows x 3 columns]
2002
      year    make    model
1215  2002   ACURA       CL
1216  2002   ACURA       EL
...    ...     ...      ...
2504  2002  YAMAHA   YZF-R6
2505  2002  YAMAHA  YZF600R

[1291 rows x 3 columns]
2003
...
'''
```



Display the Make and Model of Each Car Grouped by Year

```
cars_grouped_year = df[['make','model']].groupby(df['year'])
for year, group in cars_grouped_year:
    print(year)
    print(group)                                # dataframe - make and model
'''
2001
        make    model
0      ACURA       CL
1      ACURA       EL
...      ...      ...
1213  YAMAHA    YZF-R6
1214  YAMAHA   YZF600R

[1215 rows x 2 columns]
2002
        make    model
1215   ACURA       CL
1216   ACURA       EL
...      ...      ...
2504  YAMAHA    YZF-R6
2505  YAMAHA   YZF600R
```

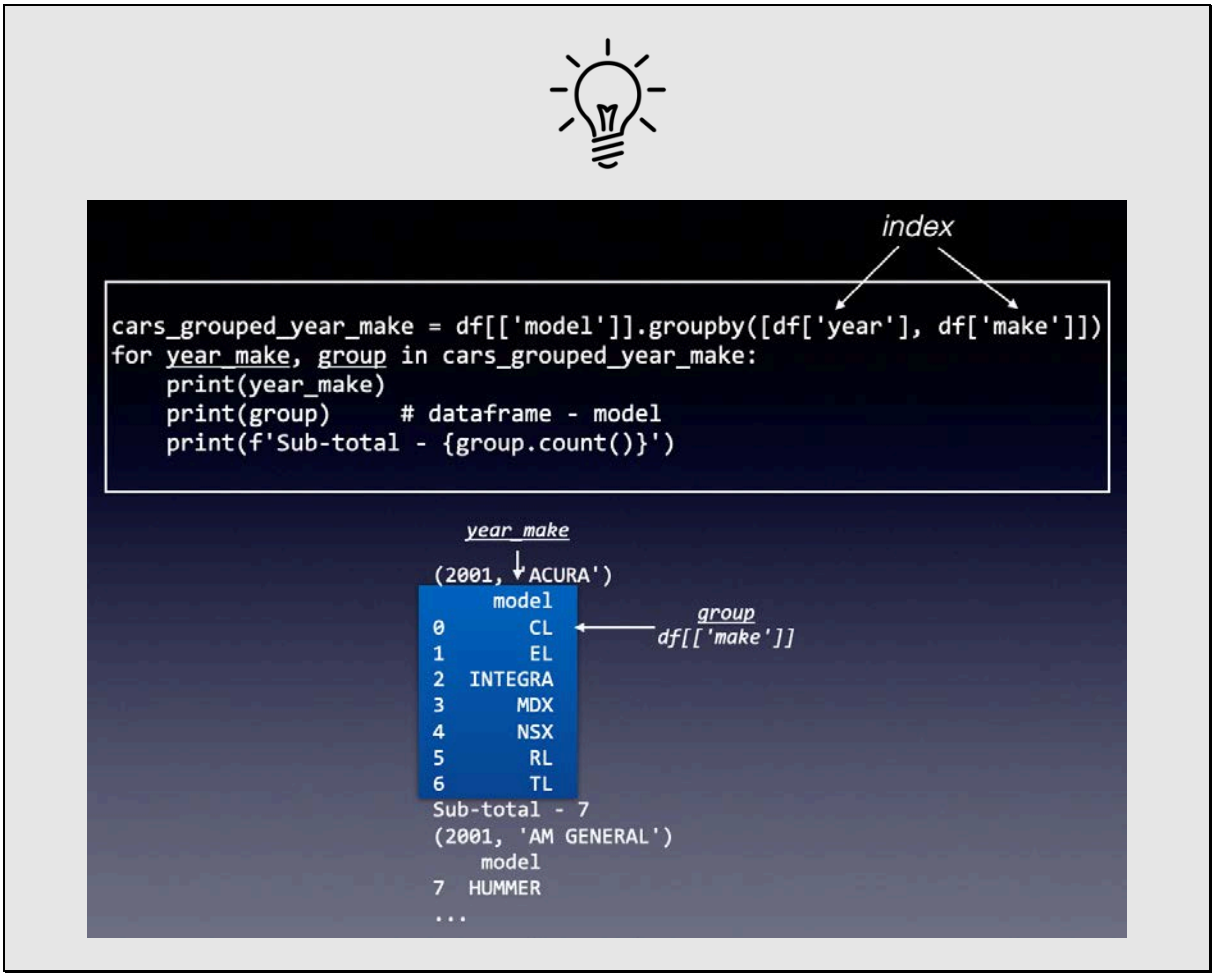
```
[1291 rows x 2 columns]
2003
...
'''
```



Display all the Models Grouped by Year and Make

```
cars_grouped_year_make = df[['model']].groupby([df['year'], df['make']])
for year_make, group in cars_grouped_year_make:
    print(year_make)
    print(group)     # dataframe - model
    print(f'Sub-total - {group.model.count()}')
'''
(2001, 'ACURA')
     model
0       CL
1       EL
2  INTEGRA
3      MDX
4      NSX
5       RL
6       TL
Sub-total - 7
(2001, 'AM GENERAL')
    model
7  HUMMER
Sub-total - model    1
```

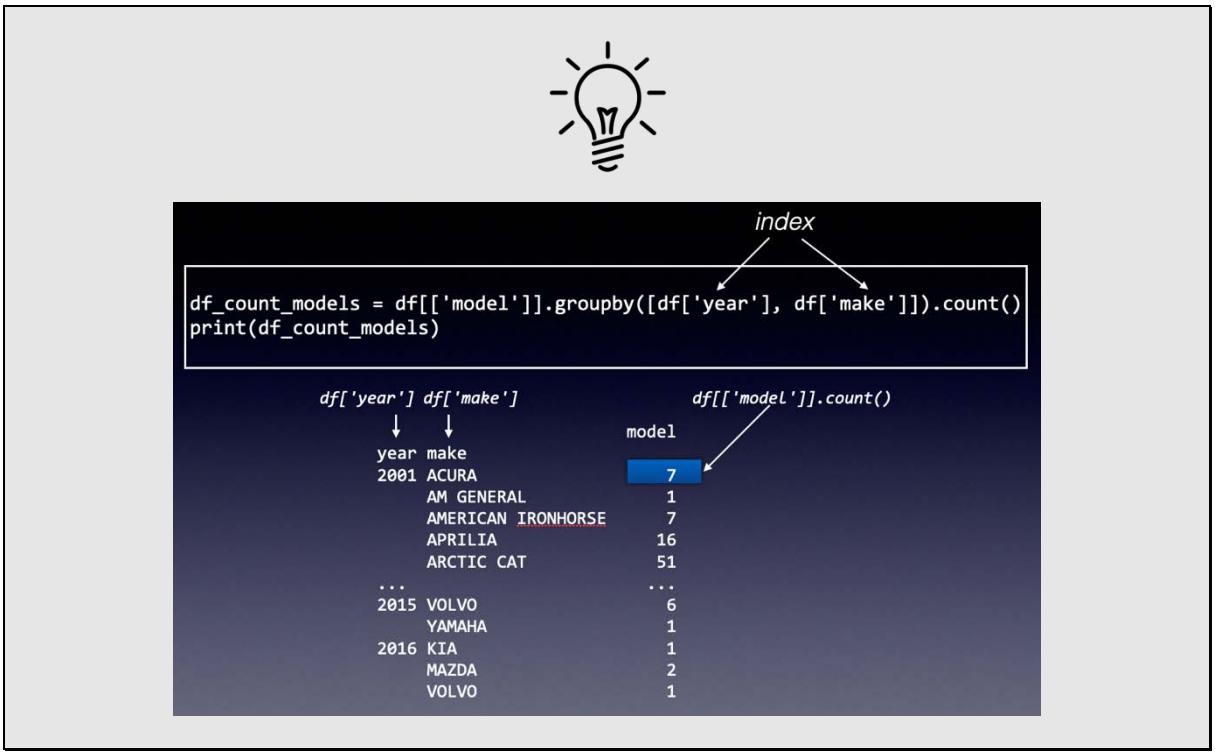
```
dtype: int64
(2001, 'AMERICAN IRONHORSE')
      model
8   CLASSIC
9    LEGEND
10   OUTLAW
11   RANGER
12  SLAMMER
13    TEJAS
14  THUNDER
Sub-total - 7
dtype: int64
(2001, 'APRILIA')
...
'''
```



Count the Total Number of Makes for Each Model

```
df_count_models = df[['model']].groupby([df['year'], df['make']]).count()
print(df_count_models)
'''
                          model
year make
2001 ACURA                    7
     AM GENERAL               1
     AMERICAN IRONHORSE       7
     APRILIA                 16
     ARCTIC CAT              51
...                         ...
2015 VOLVO                    6
     YAMAHA                   1
```

```
2016 KIA                    1
     MAZDA                  2
     VOLVO                  1

[1467 rows x 1 columns]
'''
```



Example 3

This example uses the dataset "Car Features and MSRP" from
https://www.kaggle.com/CooperUnion/cardataset

```
df = pd.read_csv("data.csv")
print(df)
'''
          Make       Model  Year                Engine Fuel Type  Engine HP  \
0          BMW  1 Series M  2011     premium unleaded (required)      335.0
1          BMW    1 Series  2011     premium unleaded (required)      300.0
2          BMW    1 Series  2011     premium unleaded (required)      300.0
3          BMW    1 Series  2011     premium unleaded (required)      230.0
4          BMW    1 Series  2011     premium unleaded (required)      230.0
...        ...         ...   ...                             ...        ...
11909    Acura         ZDX  2012     premium unleaded (required)      300.0
11910    Acura         ZDX  2012     premium unleaded (required)      300.0
11911    Acura         ZDX  2012     premium unleaded (required)      300.0
11912    Acura         ZDX  2013  premium unleaded (recommended)      300.0
11913  Lincoln      Zephyr  2006                regular unleaded      221.0

       Engine Cylinders Transmission Type      Driven_Wheels  Number of Doors  \
0                   6.0            MANUAL   rear wheel drive              2.0
1                   6.0            MANUAL   rear wheel drive              2.0
2                   6.0            MANUAL   rear wheel drive              2.0
3                   6.0            MANUAL   rear wheel drive              2.0
4                   6.0            MANUAL   rear wheel drive              2.0
...                 ...               ...                 ...              ...
11909               6.0         AUTOMATIC    all wheel drive              4.0
11910               6.0         AUTOMATIC    all wheel drive              4.0
11911               6.0         AUTOMATIC    all wheel drive              4.0
11912               6.0         AUTOMATIC    all wheel drive              4.0
11913               6.0         AUTOMATIC  front wheel drive              4.0

                           Market Category Vehicle Size  Vehicle Style  \
0      Factory Tuner,Luxury,High-Performance      Compact          Coupe
1                      Luxury,Performance      Compact    Convertible
2                Luxury,High-Performance      Compact          Coupe
3                      Luxury,Performance      Compact          Coupe
4                                 Luxury      Compact    Convertible
...                                  ...          ...            ...
11909            Crossover,Hatchback,Luxury      Midsize  4dr Hatchback
11910            Crossover,Hatchback,Luxury      Midsize  4dr Hatchback
11911            Crossover,Hatchback,Luxury      Midsize  4dr Hatchback
11912            Crossover,Hatchback,Luxury      Midsize  4dr Hatchback
11913                                 Luxury      Midsize          Sedan

       highway MPG  city mpg  Popularity   MSRP
0               26        19        3916  46135
1               28        19        3916  40650
2               28        20        3916  36350
3               28        18        3916  29450
4               28        18        3916  34500
...            ...       ...         ...    ...
11909           23        16         204  46120
11910           23        16         204  56670
11911           23        16         204  50620
11912           23        16         204  50920
11913           26        17          61  28995
```

```
[11914 rows x 16 columns]
'''
```

Group by Maker

```
cars_grouped_make = df.groupby('Make')
for make, group in cars_grouped_make:
    print(make)
    print(group[['Make','Model','Year', 'MSRP']])  # only display specific
                                                   # columns
'''
Acura
        Make Model  Year   MSRP
2696   Acura    CL  2001  29980
2697   Acura    CL  2001  27980
2698   Acura    CL  2002  28030
2699   Acura    CL  2002  30030
2700   Acura    CL  2003  32700
...      ...   ...   ...    ...
11908  Acura   ZDX  2011  50520
11909  Acura   ZDX  2012  46120
11910  Acura   ZDX  2012  56670
11911  Acura   ZDX  2012  50620
11912  Acura   ZDX  2013  50920

[252 rows x 4 columns]
Alfa Romeo
           Make Model  Year   MSRP
474  Alfa Romeo    4C  2015  63900
475  Alfa Romeo    4C  2015  68400
:::
'''
```

Find the Most Expensive, Cheapest, and the Mean Price for Each Maker

```
cars_grouped_make = df[['Vehicle Style','Market Category','MSRP']].groupby(
                        [df['Make'], df['Model'], df['Year']])
for make, group in cars_grouped_make:
    print(make)                                # make is a tuple and group is a df
    print(group)                               # group is a df
    print(f'Most Expensive : ${group["MSRP"].max()}')  # find the max for this gp
    print(f'Cheapest       : ${group["MSRP"].min()}')  # find the min for this gp
    print(f'Mean           : ${group["MSRP"].mean()}') # find the avg for this gp
    print()
'''
('Acura', 'CL', 2001)
     Vehicle Style Market Category   MSRP
2696         Coupe          Luxury  29980
2697         Coupe          Luxury  27980
Most Expensive : $29980
Cheapest       : $27980
Mean           : $28980.0

('Acura', 'CL', 2002)
     Vehicle Style Market Category   MSRP
2698         Coupe          Luxury  28030
2699         Coupe          Luxury  30030
Most Expensive : $30030
Cheapest       : $28030
Mean           : $29030.0
:::
'''
```

Find the Most Expensive Car

```
print(pd.DataFrame(df.loc[df['MSRP'].idxmax()])) # find the index of the maximum
                                                 # number in the MSRP column
'''
                                        11362
Make                                   Bugatti
Model                                Veyron 16.4
Year                                        2008
Engine Fuel Type   premium unleaded (required)
Engine HP                                   1001
Engine Cylinders                              16
Transmission Type                AUTOMATED_MANUAL
Driven_Wheels                     all wheel drive
Number of Doors                                2
Market Category          Exotic,High-Performance
Vehicle Size                             Compact
Vehicle Style                              Coupe
highway MPG                                   14
city mpg                                       8
Popularity                                   820
MSRP                                     2065902
'''
```

The idxmax() function returns the index of the maximum value in the dataframe

Most Expensive Car From Each Maker

```
cars_grouped_make = df.groupby('Make')
for make, group in cars_grouped_make:
    print(make)
    print("="* len(make))
    most_exp_each_make = group['MSRP'].idxmax()  # returns an index of all the
                                                 # max value in the MSRP column
    print(pd.DataFrame(df.iloc[most_exp_each_make]))
    print()
'''
Acura
=====
                                                 7263
Make                                           Acura
Model                                            NSX
Year                                            2017
Engine Fuel Type             premium unleaded (required)
Engine HP                                         573
Engine Cylinders                                    6
Transmission Type                      AUTOMATED_MANUAL
Driven_Wheels                           all wheel drive
Number of Doors                                     2
Market Category    Exotic,Luxury,High-Performance,Hybrid
Vehicle Size                                  Compact
Vehicle Style                                   Coupe
highway MPG                                        22
city mpg                                           21
Popularity                                        204
MSRP                                           156000

Alfa Romeo
```

```
==========
                                      475
Make                                Alfa Romeo
Model                                   4C
Year                                  2015
Engine Fuel Type   premium unleaded (required)
Engine HP                              237
Engine Cylinders                         4
Transmission Type             AUTOMATED_MANUAL
Driven_Wheels                 rear wheel drive
Number of Doors                          2
Market Category        Luxury,High-Performance
Vehicle Size                       Compact
Vehicle Style                        Coupe
highway MPG                             34
city mpg                                24
Popularity                             113
MSRP                                 68400

Aston Martin
============
                                     11213
Make                              Aston Martin
...
'''
```

Most Expensive Model and Vehicle Type For Each Make

```
df_result = df[['MSRP']].groupby(
    [df['Make'],df['Model'],df['Vehicle Style']]).max()
print(df_result)
'''
                                  MSRP
Make  Model      Vehicle Style
Acura CL         Coupe           32700
      ILX        Sedan           34980
      ILX Hybrid Sedan           34600
      Integra    2dr Hatchback   24450
                 Sedan           22600
...                                ...
Volvo V90        Wagon            2200
      XC         Wagon           36500
      XC60       4dr SUV         51300
      XC70       Wagon           48175
      XC90       4dr SUV         65700

[1168 rows x 1 columns]
'''
```

Finds the most expensive model and vehicle type for each make

For e.g. Acura has the model Integra, which has two vehicle types – 2dr Hatchback and Sedan; display the most expensive for each vehicle type

Most Expensive Model For Each Make

```
rows = df[['MSRP']].groupby([df['Make'], df['Model']]).idxmax()
print(df.iloc[rows.MSRP,[0,1,15]])
'''
        Make       Model   MSRP
2700   Acura          CL  32700
5811   Acura         ILX  34980
5797   Acura  ILX Hybrid  34600
5939   Acura     Integra  24450
6425   Acura      Legend   2506
...      ...         ...    ...
11181  Volvo         V90   2200
11640  Volvo          XC  36500
11597  Volvo        XC60  51300
11615  Volvo        XC70  48175
11631  Volvo        XC90  65700

[928 rows x 3 columns]
'''
```

Finds the most expensive model for each make

For e.g. Acura has the model Integra, which has two vehicle types – 2dr Hatchback and Sedan; only display the most expensive price for this model