Introduction-Pandas

December 1, 2021

1 Introduction to Pandas and other libraries

(Many thanks to Evimaria Terzi and Mark Crovella for their code and examples)

1.1 Pandas

Pandas is the Python Data Analysis Library.

Pandas is an extremely versatile tool for manipulating datasets, mostly tabular data. You can think of Pandas as the evolution of excel spreadsheets, with more capabilities for coding, and SQL queries such as joins and group-by.

It also produces high quality plots with matplotlib, and integrates nicely with other libraries that expect NumPy arrays.

You can find more details here

1.1.1 Storing data tables

Most data can be viewed as tables or matrices (in the case where all entries are numeric). The rows correspond to objects and the columns correspond to the attributes or features.

There are different ways we can store such data tables in Python

Two-dimensional lists

[1]:
```
D = [[0.3, 10, 1000],[0.5,2,509],[0.4, 8, 789]]
print(D)
```

```
[[0.3, 10, 1000], [0.5, 2, 509], [0.4, 8, 789]]
```

[2]:
```
D = [[30000, 'Married', 1],[20000,'Single', 0],[45000, 'Maried', 0]]
print(D)
```

```
[[30000, 'Married', 1], [20000, 'Single', 0], [45000, 'Maried', 0]]
```

Numpy Arrays

Numpy is a the library of Python for numerical computations and matrix manipulations. It has a lot of the functionality of Matlab but also allows for data analysis operations (similar to Pandas). Read more for Numpy here: http://www.numpy.org/

The Array is the main data structure for numpy. It stores multidimensional **numeric** tables.

We can create numpy arrays from lists

[1]:
```
import numpy as np

#1-dimensional array
x = np.array([2,5,18,14,4])
print ("\n Deterministic 1-dimensional array \n")
print (x)

#2-dimensional array
x = np.array([[2,5,18,14,4], [12,15,1,2,8]])
print ("\n Deterministic 2-dimensional array \n")
print (x)
```

```
 Deterministic 1-dimensional array

[ 2  5 18 14  4]

 Deterministic 2-dimensional array

[[ 2  5 18 14  4]
 [12 15  1  2  8]]
```

There are also numpy operations that create arrays of different types

[2]:
```
x = np.random.rand(5,5)
print ("\n Random 5x5 2-dimensional array \n")
print (x)

x = np.ones((4,4))
print ("\n 4x4 array with ones \n")
print (x)

x = np.diag([1,2,3])
print ("\n Diagonal matrix\n")
print(x)
```

```
 Random 5x5 2-dimensional array

[[0.35784701 0.37334321 0.33161931 0.72063771 0.05463151]
 [0.57375842 0.2691378  0.68523771 0.78832574 0.76253267]
 [0.86422761 0.6888991  0.38571199 0.79552667 0.09740792]
 [0.81403013 0.77283582 0.5921944  0.9840318  0.54518905]
 [0.35168858 0.69825623 0.50968492 0.39713053 0.52311601]]

 4x4 array with ones
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

 Diagonal matrix

[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

Why do we need numpy arrays? Because we can do different linear algebra operations on the numeric arrays

For example:

[3]:
```
x = np.random.randint(10,size=(2,3))
print("\n Random 2x3 array with integers")
print(x)

#Matrix transpose
print ("\n Transpose of the matrix \n")
print (x.T)

#multiplication and addition with scalar value
print("\n Matrix 2x+1 \n")
print(2*x+1)
```

```
 Random 2x3 array with integers
[[7 5 1]
 [6 3 4]]

 Transpose of the matrix

[[7 6]
 [5 3]
 [1 4]]

 Matrix 2x+1

[[15 11  3]
 [13  7  9]]
```

Transform back to list of lists

[4]:
```
lx = [list(y) for y in x]
lx
```

[4]:
```
[[7, 5, 1], [6, 3, 4]]
```

Pandas data frames

A data frame is a table in which each row and column is given a label. Very similar to a spreahsheet or a SQL table.

Pandas DataFrames are documented at: http://pandas.pydata.org/pandas-docs/dev/generated/pandas.DataFrame.html

Pandas dataframes enable different data analysis operations

1.1.2 Creating Data Frames

A dataframe has names for the columns and the rows of the tables. The column names are stored in the attribute **columns**, while the row names in the attribute **index**. When these are not speficied, they are just indexed by default with the numbers 0,1,...

There are multiple ways we can create a data frame. Here we list just a few.

[5]:
```
import pandas as pd #The pandas library
from pandas import Series, DataFrame #Main pandas data structures
```

[6]:
```
#Creating a data frame from a list of lists

df = pd.DataFrame([[1,2,3],[9,10,12]])
print(df)

# Each list becomes a row
# Names of columns are 0,1,3
# Rows are indexed by position numbers 0,1
```

```
   0   1   2
0  1   2   3
1  9  10  12
```

[7]:
```
#Creating a data frame from a numpy array

df = pd.DataFrame(np.array([[1,2,3],[9,10,12]]))
print(df)
```

```
   0   1   2
0  1   2   3
1  9  10  12
```

[8]:
```
# Specifying column names
df = pd.DataFrame(np.array([[1,2,3],[9,10,12]]), columns=['A','B','C'])
print(df)
```

```
   A   B   C
0  1   2   3
1  9  10  12
```

[9]:
```
#Creating a data frame from a dictionary
# Keys are column names, values are lists with column values

dfe = pd.DataFrame({'A':[1,2,3], 'B':['a','b','c']})
print(dfe)
```

```
   A  B
0  1  a
1  2  b
2  3  c
```

[10]:
```
# Reading from a csv file:
df = pd.read_csv('example.csv')
print(df)

# The first row of the file is used for the column names
# The property columns gives us the column names
print(df.columns)
print(list(df.columns))

# Reading from a csv file without header:
df = pd.read_csv('no-header.csv',header = None)
print(df)
```

```
   NUMBER CHAR
0       1    a
1       2    b
2       3    c
Index(['NUMBER', 'CHAR'], dtype='object')
['NUMBER', 'CHAR']
   0  1
0  1  a
1  2  b
2  3  c
```

[11]:
```
# Reading from am excel file:
df = pd.read_excel('example.xlsx')
print(df)
```

```
   NUMBER CHAR
0       1    a
1       2    b
2       3    c
```

[12]:
```
#Writing to a csv file:
df.to_csv('example2.csv')
for x in open('example2.csv').readlines():
    print(x.strip())

# By default the row index is added as a column, we can remove it by seting␣
 ↪index=False
df.to_csv('example2.csv',index = False)
for x in open('example2.csv').readlines():
    print(x.strip())
```

```
,NUMBER,CHAR
0,1,a
1,2,b
2,3,c
NUMBER,CHAR
1,a
2,b
3,c
```

Fetching data

For demonstration purposes, we'll use a library built-in to Pandas that fetches data from standard online sources. More information on what types of data you can fetch is at: https://pandas-datareader.readthedocs.io/en/latest/remote_data.html

We will use stock quotes from IEX. To make use of these you need to first create an account and obtain an API key. Then you set the environment variable IEX_API_KEY to the value of the key as it is snown below

[13]:
```
import os
os.environ["IEX_API_KEY"] =␣
 ↪"pk_4f1eb9a770e04d2ebc44123e297618bb"#"pk_*****************************"
```

[14]:
```
import pandas_datareader.data as web # For accessing web data - you need to␣
 ↪install this
from datetime import datetime #For handling dates
```

[15]:
```
stocks = 'FB'
data_source = 'iex'
start = datetime(2018,1,1)
end = datetime(2018,12,31)

stocks_data = web.DataReader(stocks, data_source, start, end)

#If you want to load only some of the attributes:
#stocks_data = web.DataReader(stocks, data_source, start, end)[['open','close']]
```

[16]:
```
# the method info() outputs basic information for our data frame
stocks_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 251 entries, 2018-01-02 to 2018-12-31
Data columns (total 5 columns):
open      251 non-null float64
high      251 non-null float64
low       251 non-null float64
close     251 non-null float64
volume    251 non-null int64
dtypes: float64(4), int64(1)
memory usage: 11.8+ KB
```

[17]:
```
#the medthod head() outputs the top rows of the data frame
stocks_data.head()
```

[17]:

date	open	high	low	close	volume
2018-01-02	177.68	181.58	177.5500	181.42	18151903
2018-01-03	181.88	184.78	181.3300	184.67	16886563
2018-01-04	184.90	186.21	184.0996	184.33	13880896
2018-01-05	185.59	186.90	184.9300	186.85	13574535
2018-01-08	187.20	188.90	186.3300	188.28	17994726

[18]:
```
#the medthod tail() outputs the last rows of the data frame
stocks_data.tail()
```

[18]:

date	open	high	low	close	volume
2018-12-24	123.10	129.74	123.02	124.06	22066002
2018-12-26	126.00	134.24	125.89	134.18	39723370
2018-12-27	132.44	134.99	129.67	134.52	31202509
2018-12-28	135.34	135.92	132.20	133.20	22627569
2018-12-31	134.45	134.64	129.95	131.09	24625308

Note that the date attribute is the index of the rows, not an attribute.

[19]:
```
#trying to access the date column will give an error

stocks_data.date
```

```
␣
↪---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call␣
↪last)
```

```
        <ipython-input-19-d893f040ef09> in <module>
          1 #trying to access the date column will give an error
          2
    ----> 3 stocks_data.date


        C:\ProgramData\Anaconda3\lib\site-packages\pandas\core\generic.py in␣
↪__getattr__(self, name)
       5177             if self._info_axis.
↪_can_hold_identifiers_and_holds_name(name):
       5178                 return self[name]
    -> 5179             return object.__getattribute__(self, name)
       5180
       5181     def __setattr__(self, name, value):


        AttributeError: 'DataFrame' object has no attribute 'date'
```

The number of rows in the DataFrame:

[20]:
```
len(stocks_data)
```

[20]:
```
251
```

[21]:
```
stocks_data.to_csv('stocks_data.csv')
for x in open('stocks_data.csv').readlines()[0:10]:
    print(x.strip())
df = pd.read_csv('stocks_data.csv')
df.head()
```

```
date,open,high,low,close,volume
2018-01-02,177.68,181.58,177.55,181.42,18151903
2018-01-03,181.88,184.78,181.33,184.67,16886563
2018-01-04,184.9,186.21,184.0996,184.33,13880896
2018-01-05,185.59,186.9,184.93,186.85,13574535
2018-01-08,187.2,188.9,186.33,188.28,17994726
2018-01-09,188.7,188.8,187.1,187.87,12393057
2018-01-10,186.94,187.89,185.63,187.84,10529894
2018-01-11,188.4,188.4,187.38,187.77,9588587
2018-01-12,178.06,181.48,177.4,179.37,77551299
```

[21]:

	date	open	high	low	close	volume
0	2018-01-02	177.68	181.58	177.5500	181.42	18151903
1	2018-01-03	181.88	184.78	181.3300	184.67	16886563
2	2018-01-04	184.90	186.21	184.0996	184.33	13880896
3	2018-01-05	185.59	186.90	184.9300	186.85	13574535

```
4  2018-01-08  187.20  188.90  186.3300  188.28  17994726
```

Note that in the new dataframe, there is now a date column, while the index values are numbers 0,1,...

1.1.3 Working with data columns

The columns are the "features" in your data

[22]:
```
df.columns
```

[22]:
```
Index(['date', 'open', 'high', 'low', 'close', 'volume'], dtype='object')
```

We can also assign a list to the columns property in order to change the attribute names.

Alternatively, you can change the name of an attribute using rename:

[23]:
```
df = df.rename(columns = {'volume':'V'})
print(list(df.columns))
df.columns = ['date', 'open', 'high', 'low', 'close', 'vol']
df.head()
```

```
['date', 'open', 'high', 'low', 'close', 'V']
```

[23]:
```
         date    open    high       low   close       vol
0  2018-01-02  177.68  181.58  177.5500  181.42  18151903
1  2018-01-03  181.88  184.78  181.3300  184.67  16886563
2  2018-01-04  184.90  186.21  184.0996  184.33  13880896
3  2018-01-05  185.59  186.90  184.9300  186.85  13574535
4  2018-01-08  187.20  188.90  186.3300  188.28  17994726
```

Selecting a single column from your data.

It is important to keep in mind that this selection process returns a new data frame.

[24]:
```
df['open'].head()
```

[24]:
```
0    177.68
1    181.88
2    184.90
3    185.59
4    187.20
Name: open, dtype: float64
```

Another way of selecting **a single column** from your data

[25]:
```
df.open.head()
```

[25]:
```
0    177.68
1    181.88
```

```
2    184.90
3    185.59
4    187.20
Name: open, dtype: float64
```

Selecting multiple columns

[26]:
```
df[['open','close']].head()
```

[26]:
```
     open   close
0  177.68  181.42
1  181.88  184.67
2  184.90  184.33
3  185.59  186.85
4  187.20  188.28
```

We can use the values method to obtain the values of one or more attributes. It returns a numpy array. You can trasform it into a list, by applying the list() operator.

[27]:
```
df.open.values
```

[27]:
```
array([177.68  , 181.88  , 184.9   , 185.59  , 187.2   , 188.7   ,
       186.94  , 188.4   , 178.06  , 181.5   , 179.26  , 178.13  ,
       180.85  , 180.8   , 186.05  , 189.89  , 187.95  , 187.75  ,
       188.75  , 183.01  , 188.37  , 188.22  , 192.04  , 186.93  ,
       178.57  , 184.15  , 181.01  , 174.76  , 177.06  , 175.62  ,
       173.45  , 180.5   , 178.99  , 175.77  , 176.71  , 178.7   ,
       179.9   , 184.58  , 184.45  , 182.3   , 179.01  , 173.29  ,
       176.2   , 181.78  , 178.74  , 183.56  , 183.91  , 185.23  ,
       185.61  , 182.6   , 183.24  , 184.49  , 177.01  , 167.47  ,
       164.8   , 166.13  , 165.44  , 160.82  , 156.31  , 151.65  ,
       155.15  , 157.81  , 156.55  , 152.025 , 161.56  , 157.73  ,
       157.82  , 157.93  , 165.36  , 166.98  , 164.58  , 165.7249,
       165.83  , 166.88  , 166.2   , 167.79  , 167.27  , 165.43  ,
       160.1448, 173.22  , 176.81  , 173.79  , 172.    , 174.246 ,
       175.13  , 173.08  , 177.35  , 178.25  , 179.67  , 183.15  ,
       184.85  , 187.71  , 184.88  , 183.6952, 182.68  , 183.49  ,
       183.77  , 184.93  , 182.5   , 185.88  , 186.02  , 184.34  ,
       186.54  , 187.87  , 193.065 , 191.84  , 194.3   , 191.0252,
       190.75  , 187.53  , 188.81  , 192.17  , 192.74  , 193.1   ,
       195.79  , 194.8   , 196.2352, 199.1   , 202.76  , 201.16  ,
       200.    , 197.6   , 199.18  , 195.18  , 197.32  , 193.37  ,
       194.55  , 194.74  , 198.45  , 204.93  , 204.5   , 202.22  ,
       203.43  , 207.81  , 207.5   , 204.9   , 209.82  , 208.77  ,
       208.85  , 210.58  , 215.11  , 215.715 , 174.89  , 179.87  ,
       175.3   , 170.67  , 173.93  , 170.68  , 177.69  , 178.97  ,
       186.5   , 184.75  , 185.8492, 182.04  , 180.1   , 180.71  ,
       179.34  , 180.42  , 174.5   , 174.04  , 172.81  , 172.21  ,
```

```
       173.09  , 173.7   , 175.99  , 178.1   , 176.295 , 175.9   ,
       177.15  , 173.5   , 169.49  , 166.98  , 160.31  , 163.51  ,
       163.94  , 163.25  , 162.    , 161.715 , 161.92  , 159.39  ,
       160.08  , 164.5   , 166.64  , 161.03  , 161.99  , 164.3   ,
       167.55  , 168.33  , 163.03  , 161.58  , 160.    , 161.46  ,
       159.21  , 155.54  , 157.69  , 156.82  , 150.13  , 156.73  ,
       153.32  , 155.4   , 159.56  , 158.51  , 155.86  , 154.76  ,
       151.22  , 154.28  , 147.73  , 145.82  , 148.5   , 139.935 ,
       155.    , 151.52  , 151.8   , 150.1   , 149.31  , 151.57  ,
       150.49  , 146.75  , 144.48  , 142.    , 143.7   , 142.33  ,
       141.07  , 137.61  , 127.03  , 134.4   , 133.65  , 133.    ,
       135.75  , 136.28  , 135.92  , 138.26  , 143.    , 140.73  ,
       133.82  , 139.25  , 139.6   , 143.88  , 143.08  , 145.57  ,
       143.34  , 143.08  , 141.08  , 141.21  , 130.7   , 133.39  ,
       123.1   , 126.    , 132.44  , 135.34  , 134.45  ])
```

[28]:
```
df[['open','close']].values
```

[28]:
```
array([[177.68  , 181.42  ],
       [181.88  , 184.67  ],
       [184.9   , 184.33  ],
       [185.59  , 186.85  ],
       [187.2   , 188.28  ],
       [188.7   , 187.87  ],
       [186.94  , 187.84  ],
       [188.4   , 187.77  ],
       [178.06  , 179.37  ],
       [181.5   , 178.39  ],
       [179.26  , 177.6   ],
       [178.13  , 179.8   ],
       [180.85  , 181.29  ],
       [180.8   , 185.37  ],
       [186.05  , 189.35  ],
       [189.89  , 186.55  ],
       [187.95  , 187.48  ],
       [187.75  , 190.    ],
       [188.75  , 185.98  ],
       [183.01  , 187.12  ],
       [188.37  , 186.89  ],
       [188.22  , 193.09  ],
       [192.04  , 190.28  ],
       [186.93  , 181.26  ],
       [178.57  , 185.31  ],
       [184.15  , 180.18  ],
       [181.01  , 171.58  ],
       [174.76  , 176.11  ],
       [177.06  , 176.41  ],
```

```
[175.62  , 173.15  ],
[173.45  , 179.52  ],
[180.5   , 179.96  ],
[178.99  , 177.36  ],
[175.77  , 176.01  ],
[176.71  , 177.91  ],
[178.7   , 178.99  ],
[179.9   , 183.29  ],
[184.58  , 184.93  ],
[184.45  , 181.46  ],
[182.3   , 178.32  ],
[179.01  , 175.94  ],
[173.29  , 176.62  ],
[176.2   , 180.4   ],
[181.78  , 179.78  ],
[178.74  , 183.71  ],
[183.56  , 182.34  ],
[183.91  , 185.23  ],
[185.23  , 184.76  ],
[185.61  , 181.88  ],
[182.6   , 184.19  ],
[183.24  , 183.86  ],
[184.49  , 185.09  ],
[177.01  , 172.56  ],
[167.47  , 168.15  ],
[164.8   , 169.39  ],
[166.13  , 164.89  ],
[165.44  , 159.39  ],
[160.82  , 160.06  ],
[156.31  , 152.22  ],
[151.65  , 153.03  ],
[155.15  , 159.79  ],
[157.81  , 155.39  ],
[156.55  , 156.11  ],
[152.025 , 155.1   ],
[161.56  , 159.34  ],
[157.73  , 157.2   ],
[157.82  , 157.93  ],
[157.93  , 165.04  ],
[165.36  , 166.32  ],
[166.98  , 163.87  ],
[164.58  , 164.52  ],
[165.7249, 164.83  ],
[165.83  , 168.66  ],
[166.88  , 166.36  ],
[166.2   , 168.1   ],
[167.79  , 166.28  ],
```

```
[167.27  , 165.84  ],
[165.43  , 159.69  ],
[160.1448, 159.69  ],
[173.22  , 174.16  ],
[176.81  , 173.59  ],
[173.79  , 172.    ],
[172.    , 173.86  ],
[174.246 , 176.07  ],
[175.13  , 174.02  ],
[173.08  , 176.61  ],
[177.35  , 177.97  ],
[178.25  , 178.92  ],
[179.67  , 182.66  ],
[183.15  , 185.53  ],
[184.85  , 186.99  ],
[187.71  , 186.64  ],
[184.88  , 184.32  ],
[183.6952, 183.2   ],
[182.68  , 183.76  ],
[183.49  , 182.68  ],
[183.77  , 184.49  ],
[184.93  , 183.8   ],
[182.5   , 186.9   ],
[185.88  , 185.93  ],
[186.02  , 184.92  ],
[184.34  , 185.74  ],
[186.54  , 187.67  ],
[187.87  , 191.78  ],
[193.065 , 193.99  ],
[191.84  , 193.28  ],
[194.3   , 192.94  ],
[191.0252, 191.34  ],
[190.75  , 188.18  ],
[187.53  , 189.1   ],
[188.81  , 191.54  ],
[192.17  , 192.4   ],
[192.74  , 192.41  ],
[193.1   , 196.81  ],
[195.79  , 195.85  ],
[194.8   , 198.31  ],
[196.2352, 197.49  ],
[199.1   , 202.    ],
[202.76  , 201.5   ],
[201.16  , 201.74  ],
[200.    , 196.35  ],
[197.6   , 199.    ],
[199.18  , 195.84  ],
```

```
[195.18  , 196.23  ],
[197.32  , 194.32  ],
[193.37  , 197.36  ],
[194.55  , 192.73  ],
[194.74  , 198.45  ],
[198.45  , 203.23  ],
[204.93  , 204.74  ],
[204.5   , 203.54  ],
[202.22  , 202.54  ],
[203.43  , 206.92  ],
[207.81  , 207.32  ],
[207.5   , 207.23  ],
[204.9   , 209.99  ],
[209.82  , 209.36  ],
[208.77  , 208.09  ],
[208.85  , 209.94  ],
[210.58  , 210.91  ],
[215.11  , 214.67  ],
[215.715 , 217.5   ],
[174.89  , 176.26  ],
[179.87  , 174.89  ],
[175.3   , 171.06  ],
[170.67  , 172.58  ],
[173.93  , 171.65  ],
[170.68  , 176.37  ],
[177.69  , 177.78  ],
[178.97  , 185.69  ],
[186.5   , 183.81  ],
[184.75  , 185.18  ],
[185.8492, 183.09  ],
[182.04  , 180.26  ],
[180.1   , 180.05  ],
[180.71  , 181.11  ],
[179.34  , 179.53  ],
[180.42  , 174.7   ],
[174.5   , 173.8   ],
[174.04  , 172.5   ],
[172.81  , 172.62  ],
[172.21  , 173.64  ],
[173.09  , 172.9   ],
[173.7   , 174.645 ],
[175.99  , 177.46  ],
[178.1   , 176.26  ],
[176.295 , 175.9   ],
[175.9   , 177.64  ],
[177.15  , 175.73  ],
[173.5   , 171.16  ],
```

```
[169.49  , 167.18  ],
[166.98  , 162.53  ],
[160.31  , 163.04  ],
[163.51  , 164.18  ],
[163.94  , 165.94  ],
[163.25  , 162.    ],
[162.    , 161.36  ],
[161.715 , 162.32  ],
[161.92  , 160.58  ],
[159.39  , 160.3   ],
[160.08  , 163.06  ],
[164.5   , 166.02  ],
[166.64  , 162.93  ],
[161.03  , 165.41  ],
[161.99  , 164.91  ],
[164.3   , 166.95  ],
[167.55  , 168.84  ],
[168.33  , 164.46  ],
[163.03  , 162.44  ],
[161.58  , 159.33  ],
[160.    , 162.43  ],
[161.46  , 158.85  ],
[159.21  , 157.33  ],
[155.54  , 157.25  ],
[157.69  , 157.9   ],
[156.82  , 151.38  ],
[150.13  , 153.35  ],
[156.73  , 153.74  ],
[153.32  , 153.52  ],
[155.4   , 158.78  ],
[159.56  , 159.42  ],
[158.51  , 154.92  ],
[155.86  , 154.05  ],
[154.76  , 154.78  ],
[151.22  , 154.39  ],
[154.28  , 146.04  ],
[147.73  , 150.95  ],
[145.82  , 145.37  ],
[148.5   , 142.09  ],
[139.935 , 146.22  ],
[155.    , 151.79  ],
[151.52  , 151.75  ],
[151.8   , 150.35  ],
[150.1   , 148.68  ],
[149.31  , 149.94  ],
[151.57  , 151.53  ],
[150.49  , 147.87  ],
```

```
       [146.75  , 144.96  ],
       [144.48  , 141.55  ],
       [142.    , 142.16  ],
       [143.7   , 144.22  ],
       [142.33  , 143.85  ],
       [141.07  , 139.53  ],
       [137.61  , 131.55  ],
       [127.03  , 132.43  ],
       [134.4   , 134.82  ],
       [133.65  , 131.73  ],
       [133.    , 136.38  ],
       [135.75  , 135.    ],
       [136.28  , 136.76  ],
       [135.92  , 138.68  ],
       [138.26  , 140.61  ],
       [143.    , 141.09  ],
       [140.73  , 137.93  ],
       [133.82  , 139.63  ],
       [139.25  , 137.42  ],
       [139.6   , 141.85  ],
       [143.88  , 142.08  ],
       [143.08  , 144.5   ],
       [145.57  , 145.01  ],
       [143.34  , 144.06  ],
       [143.08  , 140.19  ],
       [141.08  , 143.66  ],
       [141.21  , 133.24  ],
       [130.7   , 133.4   ],
       [133.39  , 124.95  ],
       [123.1   , 124.06  ],
       [126.    , 134.18  ],
       [132.44  , 134.52  ],
       [135.34  , 133.2   ],
       [134.45  , 131.09  ]])
```

1.2 Data Frame methods

A DataFrame object has many useful methods.

[29]:
```
df.mean() #produces the mean of the columns/features
```

[29]:
```
open      1.714544e+02
high      1.736153e+02
low       1.693031e+02
close     1.715109e+02
vol       2.768798e+07
dtype: float64
```

Note that date did not appear in the list. This is because it stores Strings

[30]:
```
df.std() #produces the standard deviation of the columns/features
```

[30]:
```
open     1.968343e+01
high     1.942384e+01
low      2.007438e+01
close    1.997745e+01
vol      1.922117e+07
dtype: float64
```

[31]:
```
df.sem() #produces the standard error of the mean of the columns/features
```

[31]:
```
open     1.242407e+00
high     1.226022e+00
low      1.267084e+00
close    1.260965e+00
vol      1.213230e+06
dtype: float64
```

[32]:
```
#confidence interval
import scipy.stats as stats
conf = 0.95
t = stats.t.ppf((1+conf)/2.0, len(df)-1)
(df.mean()-t*df.sem(), df.mean()+t*df.sem())
```

[32]:
```
(open     1.690075e+02
 high     1.712006e+02
 low      1.668076e+02
 close    1.690275e+02
 vol      2.529852e+07
 dtype: float64, open     1.739013e+02
 high     1.760299e+02
 low      1.717986e+02
 close    1.739944e+02
 vol      3.007743e+07
 dtype: float64)
```

[33]:
```
df.median() #produces the median of the columns/features
```

[33]:
```
open          174.89
high          176.98
low           172.83
close         174.70
vol      21860931.00
dtype: float64
```

[36]:
```
df.open.mean()
```

[36]:
```
171.4544243027888
```

[34]:
```
#95%-confidence interval
(df.open.mean()-t*df.open.sem(), df.open.mean()+t*df.open.sem())
```

[34]:
```
(169.00750496130627, 173.90134364427132)
```

Use describe to get all statistics for the data

[35]:
```
stocks_data.describe()
```

[35]:

	open	high	low	close	volume
count	251.000000	251.000000	251.000000	251.000000	2.510000e+02
mean	171.454424	173.615298	169.303110	171.510936	2.768798e+07
std	19.683435	19.423837	20.074382	19.977448	1.922117e+07
min	123.100000	129.740000	123.020000	124.060000	9.588587e+06
25%	157.815000	160.745000	155.525000	157.915000	1.782839e+07
50%	174.890000	176.980000	172.830000	174.700000	2.186093e+07
75%	184.890000	186.450000	183.420000	185.270000	3.031384e+07
max	215.715000	218.620000	214.270000	217.500000	1.698037e+08

[36]:
```
stocks_data.sum()
```

[36]:
```
open      4.303506e+04
high      4.357744e+04
low       4.249508e+04
close     4.304924e+04
volume    6.949682e+09
dtype: float64
```

The functions we have seen work on columns. We can apply them to rows as well by specifying the **axis** of the data.

axis = 1 means columns, and it is the default behavior

axis = 0 means rows

[37]:
```
stocks_data.sum(axis=1)
```

[37]:
```
date
2018-01-02    1.815262e+07
2018-01-03    1.688730e+07
2018-01-04    1.388164e+07
2018-01-05    1.357528e+07
2018-01-08    1.799548e+07
                  ...
2018-12-24    2.206650e+07
2018-12-26    3.972389e+07
2018-12-27    3.120304e+07
```

```
2018-12-28    2.262811e+07
2018-12-31    2.462584e+07
Length: 251, dtype: float64
```

Sorting: You can sort by a specific column, ascending (default) or descending. You can also sort inplace.

[38]:
```
stocks_data.sort_values(by = 'open', ascending =False).head()
```

[38]:

date	open	high	low	close	volume
2018-07-25	215.715	218.62	214.27	217.50	64592585
2018-07-24	215.110	216.20	212.60	214.67	28468681
2018-07-23	210.580	211.62	208.80	210.91	16731969
2018-07-18	209.820	210.99	208.44	209.36	15334907
2018-07-20	208.850	211.50	208.50	209.94	16241508

1.2.1 Bulk Operations

Methods like **sum( )** and **std( )** work on entire columns.

We can run our own functions across all values in a column (or row) using **apply( )**.

[39]:
```
df.date.head()
```

[39]:
```
0    2018-01-02
1    2018-01-03
2    2018-01-04
3    2018-01-05
4    2018-01-08
Name: date, dtype: object
```

The **values** property of the column returns a list of values for the column. Inspecting the first value reveals that these are strings with a particular format.

[40]:
```
first_date = df.date.values[0]
first_date
#returns a string
```

[40]:
```
'2018-01-02'
```

The datetime library handles dates. The method strptime transforms a string into a date (according to a format given as parameter).

[41]:
```
datetime.strptime(first_date, "%Y-%m-%d")
```

[41]:
```
datetime.datetime(2018, 1, 2, 0, 0)
```

We will now make use of two operations:

The **apply** method takes a dataframe and applies a function that is given as input to apply to all the entries in the data frame. In the case below we apply it to just one column.

The **lambda** function allows to define an anonymus function that takes some parameters (d) and uses them to compute some expression.

Using the lambda function with apply, we can apply the function to all the entries of the data frame (in this case the column values)

[42]:
```
df.date = df.date.apply(lambda d: datetime.strptime(d, "%Y-%m-%d"))
date_series = df.date # We want to keep the dates
df.date.head()

#Another way to do the same thing, by applying the function to every row (axis
 ↪= 1)
#df.date = df.apply(lambda row: datetime.strptime(row.date, "%Y-%m-%d"), axis=1)
```

[42]:
```
0   2018-01-02
1   2018-01-03
2   2018-01-04
3   2018-01-05
4   2018-01-08
Name: date, dtype: datetime64[ns]
```

[43]:
```
df.date.head()
```

[43]:
```
0   2018-01-02
1   2018-01-03
2   2018-01-04
3   2018-01-05
4   2018-01-08
Name: date, dtype: datetime64[ns]
```

For example, we can obtain the integer part of the open value

[44]:
```
#dftest = df[['open','close']]
#dftest.apply(lambda x: int(x))
df.apply(lambda r: int(r.open), axis=1)
```

[44]:
```
0      177
1      181
2      184
3      185
4      187
      ...
246    123
247    126
248    132
249    135
```

```
250    134
Length: 251, dtype: int64
```

Each row in a DataFrame is associated with an index, which is a label that uniquely identifies a row.

The row indices so far have been auto-generated by pandas, and are simply integers starting from 0.

From now on we will use dates instead of integers for indices – the benefits of this will show later.

Overwriting the index is as easy as assigning to the **index** property of the DataFrame.

[45]:
```
df.index = df.date
df.head()
```

[45]:

	date	open	high	low	close	vol
date						
2018-01-02	2018-01-02	177.68	181.58	177.5500	181.42	18151903
2018-01-03	2018-01-03	181.88	184.78	181.3300	184.67	16886563
2018-01-04	2018-01-04	184.90	186.21	184.0996	184.33	13880896
2018-01-05	2018-01-05	185.59	186.90	184.9300	186.85	13574535
2018-01-08	2018-01-08	187.20	188.90	186.3300	188.28	17994726

Another example using the simple example.csv data we loaded

[46]:
```
dfe
```

[46]:

	A	B
0	1	a
1	2	b
2	3	c

[47]:
```
dfe.index = dfe.B
```

[48]:
```
dfe
```

[48]:

	A	B
B		
a	1	a
b	2	b
c	3	c

Now that we have made an index based on date, we can drop the original `date` column. We will not do it in this example to use it later on.

[49]:
```
df = df.drop(columns = ['date']) #Equivalent to df = df.drop(columns =␣
↪['date']), axis=1)
#axis = 0 refers to dropping labels from rows (or you can use index = labels)
```

```
#axis = 1 refers to dropping labels from columns.
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 251 entries, 2018-01-02 to 2018-12-31
Data columns (total 5 columns):
open     251 non-null float64
high     251 non-null float64
low      251 non-null float64
close    251 non-null float64
vol      251 non-null int64
dtypes: float64(4), int64(1)
memory usage: 11.8 KB
```

1.2.2 Accessing rows of the DataFrame

So far we've seen how to access a column of the DataFrame. To access a row we use a different notation.

To access a row by its index value, use the **`.loc()`** method.

[50]:
```
df.loc[datetime(2018,5,7)]
```

[50]:
```
open          177.35
high          179.50
low           177.17
close         177.97
vol      18697195.00
Name: 2018-05-07 00:00:00, dtype: float64
```

To access a row by its sequence number (ie, like an array index), use **`.iloc()`** ('Integer Location')

[51]:
```
df.iloc[10:20] #dataframe with rows from 10 to 20
```

[51]:

date	open	high	low	close	vol
2018-01-17	179.26	179.32	175.8000	177.60	27992376
2018-01-18	178.13	180.98	177.0800	179.80	23304901
2018-01-19	180.85	182.37	180.1702	181.29	26826540
2018-01-22	180.80	185.39	180.4100	185.37	21059464
2018-01-23	186.05	189.55	185.5500	189.35	25678781
2018-01-24	189.89	190.66	186.5200	186.55	24334548
2018-01-25	187.95	188.62	186.6000	187.48	17377740
2018-01-26	187.75	190.00	186.8100	190.00	17759212
2018-01-29	188.75	188.84	185.6301	185.98	20453172
2018-01-30	183.01	188.18	181.8400	187.12	20858556

[52]:
```
df.iloc[0:2,[1,3]] #dataframe with rows 0:2, and the second and fourth columns
```

[52]:
```
              high   close
date
2018-01-02  181.58  181.42
2018-01-03  184.78  184.67
```

[53]:
```
df[['high','close']].iloc[0:2]
```

[53]:
```
              high   close
date
2018-01-02  181.58  181.42
2018-01-03  184.78  184.67
```

1.2.3 To iterate over the rows, use .iterrows()

[54]:
```
num_positive_days = 0
for idx, row in df.iterrows(): #returns the index name and the row
    if row.close > row.open:
        num_positive_days += 1

print("The total number of positive-gain days is {}.".format(num_positive_days))
```

```
The total number of positive-gain days is 130.
```

You can also do it this way:

[55]:
```
num_positive_days = 0
for i in range(len(df)):
    row = df.iloc[i]
    if row.close > row.open:
        num_positive_days += 1

print("The total number of positive-gain days is {}.".format(num_positive_days))
```

```
The total number of positive-gain days is 130.
```

Or this way:

[56]:
```
pos_days = [idx for (idx,row) in df.iterrows() if row.close > row.open]
print("The total number of positive-gain days is "+str(len(pos_days)))
```

```
The total number of positive-gain days is 130
```

1.3 Filtering

It is very easy to select interesting rows from the data.

All these operations below return a new DataFrame, which itself can be treated the same way as all DataFrames we have seen so far.

[57]:
```
tmp_high = df.high > 170
tmp_high.head()
```

[57]:
```
date
2018-01-02    True
2018-01-03    True
2018-01-04    True
2018-01-05    True
2018-01-08    True
Name: high, dtype: bool
```

Summing a Boolean array is the same as counting the number of **True** values.

[58]:
```
sum(tmp_high)
```

[58]:
```
149
```

Now, let's select only the rows of **df** that correspond to **tmp_high**

[59]:
```
df[tmp_high].head()
```

[59]:

date	open	high	low	close	vol
2018-01-02	177.68	181.58	177.5500	181.42	18151903
2018-01-03	181.88	184.78	181.3300	184.67	16886563
2018-01-04	184.90	186.21	184.0996	184.33	13880896
2018-01-05	185.59	186.90	184.9300	186.85	13574535
2018-01-08	187.20	188.90	186.3300	188.28	17994726

Putting it all together, we have the following commonly-used patterns:

[60]:
```
positive_days = df[df.close > df.open]
positive_days.head()
```

[60]:

date	open	high	low	close	vol
2018-01-02	177.68	181.58	177.55	181.42	18151903
2018-01-03	181.88	184.78	181.33	184.67	16886563
2018-01-05	185.59	186.90	184.93	186.85	13574535
2018-01-08	187.20	188.90	186.33	188.28	17994726
2018-01-10	186.94	187.89	185.63	187.84	10529894

[61]:
```
very_positive_days = df[df.close-df.open > 5]
very_positive_days.head()
```

[61]:

date	open	high	low	close	vol
2018-02-06	178.57	185.77	177.7400	185.31	37758505

```
2018-02-14  173.45  179.81  173.2119  179.52  28929704
2018-04-10  157.93  165.98  157.0100  165.04  58947041
2018-07-17  204.90  210.46  204.8400  209.99  15349892
2018-08-02  170.68  176.79  170.2700  176.37  32399954
```

[62]:
```
df[(df.high<170)&(df.low>80)]
```

[62]:
```
              open    high     low   close        vol
date
2018-03-23  165.44  167.10  159.02  159.39   53609706
2018-03-26  160.82  161.10  149.02  160.06  126116634
2018-03-27  156.31  162.85  150.75  152.22   79116995
2018-03-28  151.65  155.88  150.80  153.03   60029170
2018-03-29  155.15  161.42  154.14  159.79   59434293
…              …       …       …       …          …
2018-12-24  123.10  129.74  123.02  124.06   22066002
2018-12-26  126.00  134.24  125.89  134.18   39723370
2018-12-27  132.44  134.99  129.67  134.52   31202509
2018-12-28  135.34  135.92  132.20  133.20   22627569
2018-12-31  134.45  134.64  129.95  131.09   24625308

[102 rows x 5 columns]
```

1.3.1 Creating new columns

To create a new column, simply assign values to it. Think of the columns as a dictionary:

[63]:
```
df['profit'] = (df.close - df.open)
df.head()
```

[63]:
```
              open    high       low   close       vol  profit
date
2018-01-02  177.68  181.58  177.5500  181.42  18151903    3.74
2018-01-03  181.88  184.78  181.3300  184.67  16886563    2.79
2018-01-04  184.90  186.21  184.0996  184.33  13880896   -0.57
2018-01-05  185.59  186.90  184.9300  186.85  13574535    1.26
2018-01-08  187.20  188.90  186.3300  188.28  17994726    1.08
```

[64]:
```
df.profit[df.profit>0].describe()
```

[64]:
```
count    130.000000
mean       2.193566
std        1.783093
min        0.020000
25%        0.720000
50%        1.630000
75%        3.280000
```

```
max        8.180000
Name: profit, dtype: float64
```

[65]:
```
for idx, row in df.iterrows():
    if row.close < row.open:
        df.loc[idx,'gain']='negative'
    elif (row.close - row.open) < 1:
        df.loc[idx,'gain']='small_gain'
    elif (row.close - row.open) < 3:
        df.loc[idx,'gain']='medium_gain'
    else:
        df.loc[idx,'gain']='large_gain'
df.head()
```

[65]:

date	open	high	low	close	vol	profit	gain
2018-01-02	177.68	181.58	177.5500	181.42	18151903	3.74	large_gain
2018-01-03	181.88	184.78	181.3300	184.67	16886563	2.79	medium_gain
2018-01-04	184.90	186.21	184.0996	184.33	13880896	-0.57	negative
2018-01-05	185.59	186.90	184.9300	186.85	13574535	1.26	medium_gain
2018-01-08	187.20	188.90	186.3300	188.28	17994726	1.08	medium_gain

Here is another, more "functional", way to accomplish the same thing.

Define a function that classifies rows, and **apply** it to each row.

[66]:
```
def gainrow(row):
    if row.close < row.open:
        return 'negative'
    elif (row.close - row.open) < 1:
        return 'small_gain'
    elif (row.close - row.open) < 3:
        return 'medium_gain'
    else:
        return 'large_gain'

df['test_column'] = df.apply(gainrow, axis = 1)
#axis = 0 means rows, axis =1 means columns
```

[67]:
```
df.head()
```

[67]:

date	open	high	low	close	vol	profit	gain	\
2018-01-02	177.68	181.58	177.5500	181.42	18151903	3.74	large_gain	
2018-01-03	181.88	184.78	181.3300	184.67	16886563	2.79	medium_gain	
2018-01-04	184.90	186.21	184.0996	184.33	13880896	-0.57	negative	
2018-01-05	185.59	186.90	184.9300	186.85	13574535	1.26	medium_gain	
2018-01-08	187.20	188.90	186.3300	188.28	17994726	1.08	medium_gain	

```
            test_column
date
2018-01-02   large_gain
2018-01-03  medium_gain
2018-01-04     negative
2018-01-05  medium_gain
2018-01-08  medium_gain
```

OK, point made, let's get rid of that extraneous `test_column`:

[68]:
```
df = df.drop('test_column', axis = 1)
df.head()
```

[68]:
```
              open    high       low   close       vol  profit         gain
date
2018-01-02  177.68  181.58  177.5500  181.42  18151903    3.74   large_gain
2018-01-03  181.88  184.78  181.3300  184.67  16886563    2.79  medium_gain
2018-01-04  184.90  186.21  184.0996  184.33  13880896   -0.57     negative
2018-01-05  185.59  186.90  184.9300  186.85  13574535    1.26  medium_gain
2018-01-08  187.20  188.90  186.3300  188.28  17994726    1.08  medium_gain
```

1.3.2 Missing values

Data often has missing values. In Pandas these are denoted as NaN values. These may be part of our data (e.g. empty cells in an excel sheet), or they may appear as a result of a join. There are special methods for handling these values.

[69]:
```
mdf = pd.read_csv('example-missing.csv')
mdf
```

[69]:
```
     A    B    C
0  1.0    a    x
1  5.0    b  NaN
2  3.0    c    y
3  9.0  NaN    z
4  NaN    a    x
```

We can fill the values using the fillna method

[70]:
```
mdf.fillna(0)
```

[70]:
```
     A  B  C
0  1.0  a  x
1  5.0  b  0
2  3.0  c  y
3  9.0  0  z
4  0.0  a  x
```

[71]:
```
mdf.A = mdf.A.fillna(mdf.A.mean())
mdf = mdf.fillna('')
mdf
```

[71]:

	A	B	C
0	1.0	a	x
1	5.0	b	
2	3.0	c	y
3	9.0		z
4	4.5	a	x

We can drop the rows with missing values

[72]:
```
mdf = pd.read_csv('example-missing.csv')
mdf.dropna()
```

[72]:

	A	B	C
0	1.0	a	x
2	3.0	c	y

We can find those rows

[73]:
```
mdf[mdf.B.isnull()]
```

[73]:

	A	B	C
3	9.0	NaN	z

1.4 Grouping

An **extremely** powerful DataFrame method is `groupby()`.

This is entirely analagous to `GROUP BY` in SQL.

It will group the rows of a DataFrame by the values in one (or more) columns, and let you iterate through each group.

Here we will look at the average gain among the categories of gains (negative, small, medium and large) we defined above and stored in column `gain`.

[74]:
```
gain_groups = df.groupby('gain')
```

[75]:
```
type(gain_groups)
```

[75]:
```
pandas.core.groupby.generic.DataFrameGroupBy
```

Essentially, `gain_groups` behaves like a dictionary * The keys are the unique values found in the `gain` column, and * The values are DataFrames that contain only the rows having the corresponding unique values.

[76]:
```
for gain, gain_data in gain_groups:
    print(gain)
    print(gain_data.head())
    print('============================')
```

```
large_gain
              open    high     low   close       vol  profit        gain
date
2018-01-02  177.68  181.58  177.55  181.42  18151903    3.74  large_gain
2018-01-22  180.80  185.39  180.41  185.37  21059464    4.57  large_gain
2018-01-23  186.05  189.55  185.55  189.35  25678781    3.30  large_gain
2018-01-30  183.01  188.18  181.84  187.12  20858556    4.11  large_gain
2018-02-01  188.22  195.32  187.89  193.09  54211293    4.87  large_gain
============================
medium_gain
              open    high     low   close       vol  profit         gain
date
2018-01-03  181.88  184.78  181.33  184.67  16886563    2.79  medium_gain
2018-01-05  185.59  186.90  184.93  186.85  13574535    1.26  medium_gain
2018-01-08  187.20  188.90  186.33  188.28  17994726    1.08  medium_gain
2018-01-12  178.06  181.48  177.40  179.37  77551299    1.31  medium_gain
2018-01-18  178.13  180.98  177.08  179.80  23304901    1.67  medium_gain
============================
negative
              open    high       low   close       vol  profit      gain
date
2018-01-04  184.90  186.21  184.0996  184.33  13880896   -0.57  negative
2018-01-09  188.70  188.80  187.1000  187.87  12393057   -0.83  negative
2018-01-11  188.40  188.40  187.3800  187.77   9588587   -0.63  negative
2018-01-16  181.50  181.75  178.0400  178.39  36183842   -3.11  negative
2018-01-17  179.26  179.32  175.8000  177.60  27992376   -1.66  negative
============================
small_gain
              open    high       low   close       vol  profit        gain
date
2018-01-10  186.94  187.89  185.6300  187.84  10529894    0.90  small_gain
2018-01-19  180.85  182.37  180.1702  181.29  26826540    0.44  small_gain
2018-02-20  175.77  177.95  175.1100  176.01  21204921    0.24  small_gain
2018-02-22  178.70  180.21  177.4100  178.99  18464192    0.29  small_gain
2018-02-26  184.58  185.66  183.2228  184.93  17599703    0.35  small_gain
============================
```

We can obtain the dataframe that corresponds to a specific group by using the get_group method of the groupby object

[77]:
```
sm = gain_groups.get_group('small_gain')
sm.head()
```

[77]:
```
             open    high       low   close       vol  profit        gain
date
2018-01-10  186.94  187.89  185.6300  187.84  10529894    0.90  small_gain
2018-01-19  180.85  182.37  180.1702  181.29  26826540    0.44  small_gain
2018-02-20  175.77  177.95  175.1100  176.01  21204921    0.24  small_gain
2018-02-22  178.70  180.21  177.4100  178.99  18464192    0.29  small_gain
2018-02-26  184.58  185.66  183.2228  184.93  17599703    0.35  small_gain
```

[78]:
```
for gain, gain_data in df.groupby("gain"):
    print('The average closing value for the {} group is {}'.format(gain,
                                                    gain_data.close.mean()))
```

```
The average closing value for the large_gain group is 174.99081081081084
The average closing value for the medium_gain group is 174.18557692307695
The average closing value for the negative group is 169.2336363636363
The average closing value for the small_gain group is 171.6991463414634
```

[79]:
```
for gain, gain_data in df.groupby("gain"):
    print('The median volumn value for the {} group is {}'.format(gain,
                                                    gain_data.vol.median()))
```

```
The median volumn value for the large_gain group is 21059464.0
The median volumn value for the medium_gain group is 23155130.0
The median volumn value for the negative group is 22627569.0
The median volumn value for the small_gain group is 20197680.0
```

We often want to do a typical SQL-like group by, where we group by one or more attributes, and **aggreagate** the values (of some) of the other attributes.

For example group by "gain" and take the average of the values for open, high, low, close, volume.

You can also use other aggregators such as count, sum, median, max, min.

Pandas is now returning a new dataframe indexed by the values of the group-by attribute(s), with columns the other attributes

[80]:
```
gdf= df[['open','low','high','close','vol','gain']].groupby('gain').mean()
type(gdf)
```

[80]:
```
pandas.core.frame.DataFrame
```

[81]:
```
gdf
```

[81]:
```
                open         low        high       close           vol
gain
large_gain   170.459459  169.941454  175.660722  174.990811  3.034571e+07
medium_gain  172.305504  171.410923  175.321108  174.185577  2.795407e+07
negative     171.473133  168.024464  172.441342  169.233636  2.771124e+07
small_gain   171.217688  169.827283  173.070561  171.699146  2.488339e+07
```

If you want to remove the (hierarchical) index and have the group-by atrribute(s) to be part of the table, you can use the **reset_index** method. The result of this method is to make the index attribute one more column in the dataframe, and use the default index

[82]:
```
#This can be used to remove the hiearchical index, if necessary
gdf = gdf.reset_index()
gdf
```

[82]:

	gain	open	low	high	close	vol
0	large_gain	170.459459	169.941454	175.660722	174.990811	3.034571e+07
1	medium_gain	172.305504	171.410923	175.321108	174.185577	2.795407e+07
2	negative	171.473133	168.024464	172.441342	169.233636	2.771124e+07
3	small_gain	171.217688	169.827283	173.070561	171.699146	2.488339e+07

[83]:
```
gdf.set_index('gain')
```

[83]:

gain	open	low	high	close	vol
large_gain	170.459459	169.941454	175.660722	174.990811	3.034571e+07
medium_gain	172.305504	171.410923	175.321108	174.185577	2.795407e+07
negative	171.473133	168.024464	172.441342	169.233636	2.771124e+07
small_gain	171.217688	169.827283	173.070561	171.699146	2.488339e+07

Another example:

[84]:
```
test = pd.DataFrame({'A':[1,2,3,4],'B':['a','b','b','a'],'C':['a','a','b','a']})
test
```

[84]:

	A	B	C
0	1	a	a
1	2	b	a
2	3	b	b
3	4	a	a

[85]:
```
gtest = test.groupby(['B','C']).mean()
gtest
#note that in this case we get a hierarchical index
```

[85]:

		A
B	C	
a	a	2.5
b	a	2.0
	b	3.0

[86]:
```
gtest = gtest.reset_index()
gtest
#the hierarchical index is flattened out
```

[86]:
```
   B  C    A
0  a  a  2.5
1  b  a  2.0
2  b  b  3.0
```

1.5 Joins

We can join data frames in a similar way that we can do joins in SQL

[90]:
```
data_source = 'iex'
start = datetime(2018,1,1)
end = datetime(2018,12,31)

dfb = web.DataReader('FB', data_source, start, end)
dgoog = web.DataReader('GOOG', data_source, start, end)

print(dfb.head())
print(dgoog.head())
```

```
              open    high       low   close    volume
date
2018-01-02  177.68  181.58  177.5500  181.42  18151903
2018-01-03  181.88  184.78  181.3300  184.67  16886563
2018-01-04  184.90  186.21  184.0996  184.33  13880896
2018-01-05  185.59  186.90  184.9300  186.85  13574535
2018-01-08  187.20  188.90  186.3300  188.28  17994726
               open       high        low    close   volume
date
2018-01-02  1048.34  1066.9400  1045.2300  1065.00  1237564
2018-01-03  1064.31  1086.2900  1063.2100  1082.48  1430170
2018-01-04  1088.00  1093.5699  1084.0017  1086.40  1004605
2018-01-05  1094.00  1104.2500  1092.0000  1102.23  1279123
2018-01-08  1102.23  1111.2700  1101.6200  1106.94  1047603
```

Perform join on the date (the index value)

[93]:
```
common_dates = pd.merge(dfb,dgoog,on='date')
common_dates.head()
```

[93]:
```
            open_x  high_x     low_x  close_x  volume_x   open_y     high_y  \
date
2018-01-02  177.68  181.58  177.5500   181.42  18151903  1048.34  1066.9400
2018-01-03  181.88  184.78  181.3300   184.67  16886563  1064.31  1086.2900
2018-01-04  184.90  186.21  184.0996   184.33  13880896  1088.00  1093.5699
2018-01-05  185.59  186.90  184.9300   186.85  13574535  1094.00  1104.2500
2018-01-08  187.20  188.90  186.3300   188.28  17994726  1102.23  1111.2700

                low_y  close_y  volume_y
```

```
date
2018-01-02  1045.2300  1065.00   1237564
2018-01-03  1063.2100  1082.48   1430170
2018-01-04  1084.0017  1086.40   1004605
2018-01-05  1092.0000  1102.23   1279123
2018-01-08  1101.6200  1106.94   1047603
```

Compute gain and perform join on the data AND gain

[94]:
```
dfb['gain'] = dfb.apply(gainrow, axis = 1)
dgoog['gain'] = dgoog.apply(gainrow, axis = 1)
dfb['profit'] = dfb.close-dfb.open
dgoog['profit'] = dgoog.close-dgoog.open
```

[95]:
```
common_gain_dates = pd.merge(dfb, dgoog, on=['date','gain'])
common_gain_dates.head()
```

[95]:
```
            open_x  high_x     low_x  close_x  volume_x        gain  profit_x  \
date
2018-01-02  177.68  181.58  177.5500   181.42  18151903  large_gain      3.74
2018-01-04  184.90  186.21  184.0996   184.33  13880896    negative     -0.57
2018-01-09  188.70  188.80  187.1000   187.87  12393057    negative     -0.83
2018-01-11  188.40  188.40  187.3800   187.77   9588587    negative     -0.63
2018-01-16  181.50  181.75  178.0400   178.39  36183842    negative     -3.11

             open_y     high_y      low_y  close_y  volume_y  profit_y
date
2018-01-02  1048.34  1066.9400  1045.2300  1065.00   1237564     16.66
2018-01-04  1088.00  1093.5699  1084.0017  1086.40   1004605     -1.60
2018-01-09  1109.40  1110.5700  1101.2307  1106.26    902541     -3.14
2018-01-11  1106.30  1106.5250  1099.5900  1105.52    978292     -0.78
2018-01-16  1132.51  1139.9100  1117.8316  1121.76   1575261    -10.75
```

More join examples, including left outer join

[96]:
```
left = pd.DataFrame({'key': ['foo', 'foo', 'boo'], 'lval': [1, 2,3]})
print(left)
print('\n')
right = pd.DataFrame({'key': ['foo', 'hoo'], 'rval': [4, 5]})
print(right)
print('\n')
dfm = pd.merge(left, right, on='key') #keeps only the common key 'foo'
print(dfm)
```

```
   key  lval
0  foo     1
1  foo     2
2  boo     3
```

```
   key  rval
0  foo     4
1  hoo     5
```

```
   key  lval  rval
0  foo     1     4
1  foo     2     4
```

Left outer join

[97]:
```
dfm = pd.merge(left, right, on='key', how='left') #keeps all the keys from the␣
 ↪left and puts NaN for missing values
print(dfm)
print('\n')
dfm = dfm.fillna(0) #fills the NaN values with specified value
dfm
```

```
   key  lval  rval
0  foo     1   4.0
1  foo     2   4.0
2  boo     3   NaN
```

[97]:
```
   key  lval  rval
0  foo     1   4.0
1  foo     2   4.0
2  boo     3   0.0
```