FREE eBook

LEARNING pandas

#pandas

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: pandas

It is an unofficial and free pandas ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official pandas.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with pandas

Remarks

Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.

The official Pandas documentation can be found here.

Versions

Pandas

Version	Release Date
0.19.1	2016-11-03
0.19.0	2016-10-02
0.18.1	2016-05-03
0.18.0	2016-03-13
0.17.1	2015-11-21
0.17.0	2015-10-09
0.16.2	2015-06-12
0.16.1	2015-05-11
0.16.0	2015-03-22
0.15.2	2014-12-12
0.15.1	2014-11-09
0.15.0	2014-10-18
0.14.1	2014-07-11
0.14.0	2014-05-31
0.13.1	2014-02-03
0.13.0	2014-01-03

Version	Release Date
0.12.0	2013-07-23

Examples

Installation or Setup

Detailed instructions on getting pandas set up or installed can be found here in the official documentation.

Installing pandas with Anaconda

Installing pandas and the rest of the NumPy and SciPy stack can be a little difficult for inexperienced users.

The simplest way to install not only pandas, but Python and the most popular packages that make up the SciPy stack (IPython, NumPy, Matplotlib, ...) is with Anaconda, a cross-platform (Linux, Mac OS X, Windows) Python distribution for data analytics and scientific computing.

After running a simple installer, the user will have access to pandas and the rest of the SciPy stack without needing to install anything else, and without needing to wait for any software to be compiled.

Installation instructions for Anaconda can be found here.

A full list of the packages available as part of the Anaconda distribution can be found here.

An additional advantage of installing with Anaconda is that you don’t require admin rights to install it, it will install in the user’s home directory, and this also makes it trivial to delete Anaconda at a later date (just delete that folder).

Installing pandas with Miniconda

The previous section outlined how to get pandas installed as part of the Anaconda distribution. However this approach means you will install well over one hundred packages and involves downloading the installer which is a few hundred megabytes in size.

If you want to have more control on which packages, or have a limited internet bandwidth, then installing pandas with Miniconda may be a better solution.

Conda is the package manager that the Anaconda distribution is built upon. It is a package manager that is both cross-platform and language agnostic (it can play a similar role to a pip and virtualenv combination).

Miniconda allows you to create a minimal self contained Python installation, and then use the Conda command to install additional packages.

First you will need Conda to be installed and downloading and running the Miniconda will do this for you. The installer can be found here.

The next step is to create a new conda environment (these are analogous to a virtualenv but they also allow you to specify precisely which Python version to install also). Run the following commands from a terminal window:

```
conda create -n name_of_my_env python
```

This will create a minimal environment with only Python installed in it. To put your self inside this environment run:

```
source activate name_of_my_env
```

On Windows the command is:

```
activate name_of_my_env
```

The final step required is to install pandas. This can be done with the following command:

```
conda install pandas
```

To install a specific pandas version:

```
conda install pandas=0.13.1
```

To install other packages, IPython for example:

```
conda install ipython
```

To install the full Anaconda distribution:

```
conda install anaconda
```

If you require any packages that are available to pip but not conda, simply install pip, and use pip to install these packages:

```
conda install pip
pip install django
```

Usually, you would install pandas with one of packet managers.

pip example:

```
pip install pandas
```

This will likely require the installation of a number of dependencies, including NumPy, will require a compiler to compile required bits of code, and can take a few minutes to complete.

Install via anaconda

First download anaconda from the Continuum site. Either via the graphical installer (Windows/OSX) or running a shell script (OSX/Linux). This includes pandas!

If you don't want the 150 packages conveniently bundled in anaconda, you can install miniconda. Either via the graphical installer (Windows) or shell script (OSX/Linux).

Install pandas on miniconda using:

```
conda install pandas
```

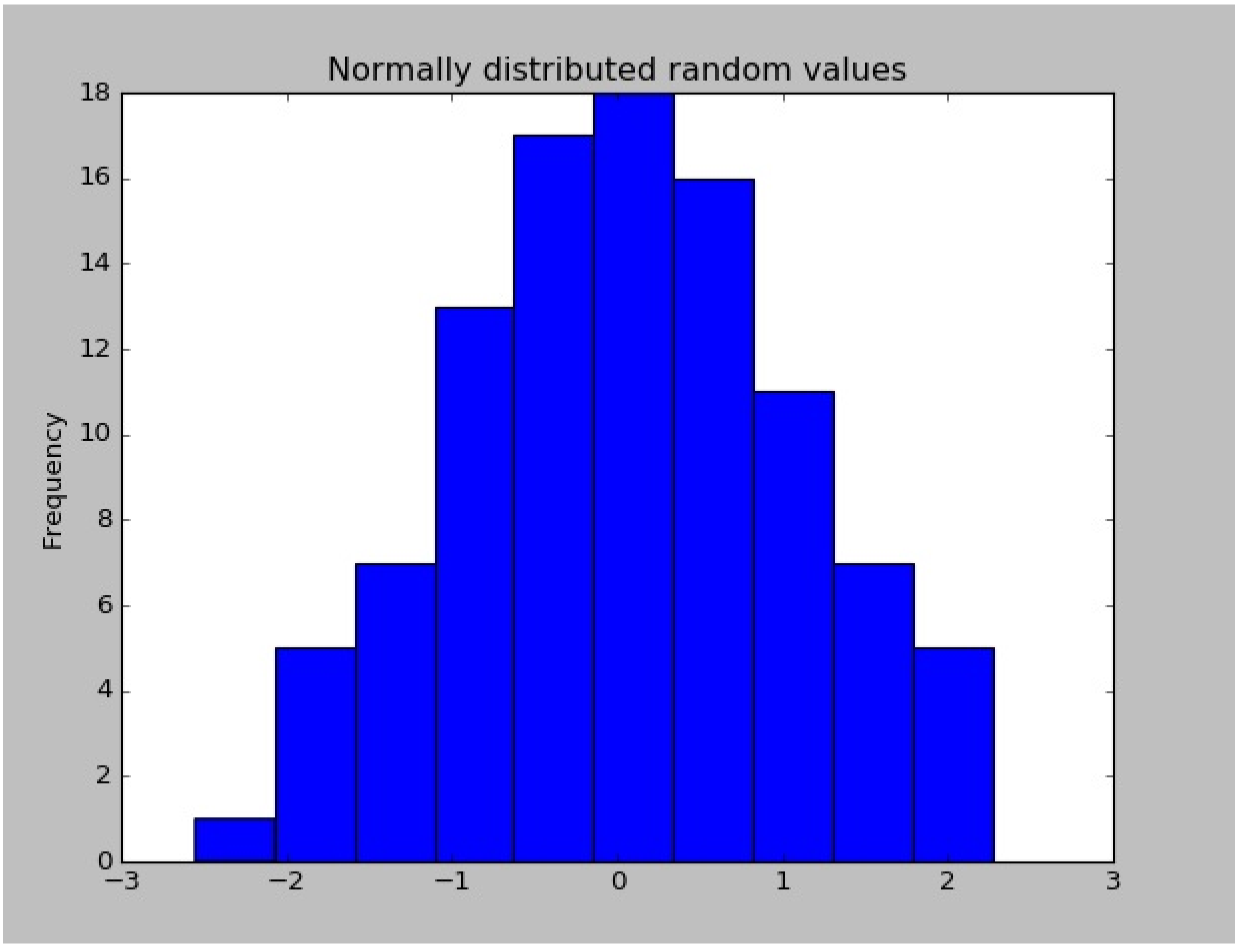
To update pandas to the latest version in anaconda or miniconda use:

```
conda update pandas
```

Hello World

Once Pandas has been installed, you can check if it is is working properly by creating a dataset of randomly distributed values and plotting its histogram.

```
import pandas as pd  # This is always assumed but is included here as an introduction.
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)

values = np.random.randn(100) # array of normally distributed random numbers
s = pd.Series(values) # generate a pandas series
s.plot(kind='hist', title='Normally distributed random values') # hist computes distribution
plt.show()
```



Check some of the data's statistics (mean, standard deviation, etc.)

```
s.describe()
# Output: count    100.000000
# mean       0.059808
# std        1.012960
# min       -2.552990
# 25%       -0.643857
# 50%        0.094096
# 75%        0.737077
# max        2.269755
# dtype: float64
```

Descriptive statistics

Descriptive statistics (mean, standard deviation, number of observations, minimum, maximum, and quartiles) of numerical columns can be calculated using the `.describe()` method, which returns a pandas dataframe of descriptive statistics.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 1, 4, 3, 5, 2, 3, 4, 1],
                           'B': [12, 14, 11, 16, 18, 18, 22, 13, 21, 17],
                           'C': ['a', 'a', 'b', 'a', 'b', 'c', 'b', 'a', 'b', 'a']})

In [2]: df
Out[2]:
   A   B  C
0  1  12  a
```

```
1  2  14  a
2  1  11  b
3  4  16  a
4  3  18  b
5  5  18  c
6  2  22  b
7  3  13  a
8  4  21  b
9  1  17  a

In [3]: df.describe()
Out[3]:
               A          B
count  10.000000  10.000000
mean    2.600000  16.200000
std     1.429841   3.705851
min     1.000000  11.000000
25%     1.250000  13.250000
50%     2.500000  16.500000
75%     3.750000  18.000000
max     5.000000  22.000000
```

Note that since `C` is not a numerical column, it is excluded from the output.

```
In [4]: df['C'].describe()
Out[4]:
count     10
unique     3
freq       5
Name: C, dtype: object
```

In this case the method summarizes categorical data by number of observations, number of unique elements, mode, and frequency of the mode.

Read Getting started with pandas online: https://riptutorial.com/pandas/topic/796/getting-started-with-pandas

Chapter 2: Analysis: Bringing it all together and making decisions

Examples

Quintile Analysis: with random data

Quintile analysis is a common framework for evaluating the efficacy of security factors.

What is a factor

A factor is a method for scoring/ranking sets of securities. For a particular point in time and for a particular set of securities, a factor can be represented as a pandas series where the index is an array of the security identifiers and the values are the scores or ranks.

If we take factor scores over time, we can, at each point in time, split the set of securities into 5 equal buckets, or quintiles, based on the order of the factor scores. There is nothing particularly sacred about the number 5. We could have used 3 or 10. But we use 5 often. Finally, we track the performance of each of the five buckets to determine if there is a meaningful difference in the returns. We tend to focus more intently on the difference in returns of the bucket with the highest rank relative to that of the lowest rank.

Let's start by setting some parameters and generating random data.

To facilitate the experimentation with the mechanics, we provide simple code to create random data to give us an idea how this works.

Random Data Includes

- **Returns**: generate random returns for specified number of securities and periods.
- **Signals**: generate random signals for specified number of securities and periods and with prescribed level of correlation with **Returns**. In order for a factor to be useful, there must be some information or correlation between the scores/ranks and subsequent returns. If there weren't correlation, we would see it. That would be a good exercise for the reader, duplicate this analysis with random data generated with `0` correlation.

Initialization

```
import pandas as pd
import numpy as np

num_securities = 1000
num_periods = 1000
period_frequency = 'W'
```

```
start_date = '2000-12-31'

np.random.seed([3,1415])

means = [0, 0]
covariance = [[  1., 5e-3],
              [5e-3,   1.]]

# generates to sets of data m[0] and m[1] with ~0.005 correlation
m = np.random.multivariate_normal(means, covariance,
                                  (num_periods, num_securities)).T
```

Let's now generate a time series index and an index representing security ids. Then use them to create dataframes for returns and signals

```
ids = pd.Index(['s{:05d}'.format(s) for s in range(num_securities)], 'ID')
tidx = pd.date_range(start=start_date, periods=num_periods, freq=period_frequency)
```

I divide `m[0]` by `25` to scale down to something that looks like stock returns. I also add `1e-7` to give a modest positive mean return.

```
security_returns = pd.DataFrame(m[0] / 25 + 1e-7, tidx, ids)
security_signals = pd.DataFrame(m[1], tidx, ids)
```

`pd.qcut` - Create Quintile Buckets

Let's use `pd.qcut` to divide my signals into quintile buckets for each period.

```
def qcut(s, q=5):
    labels = ['q{}'.format(i) for i in range(1, 6)]
    return pd.qcut(s, q, labels=labels)

cut = security_signals.stack().groupby(level=0).apply(qcut)
```

Use these cuts as an index on our returns

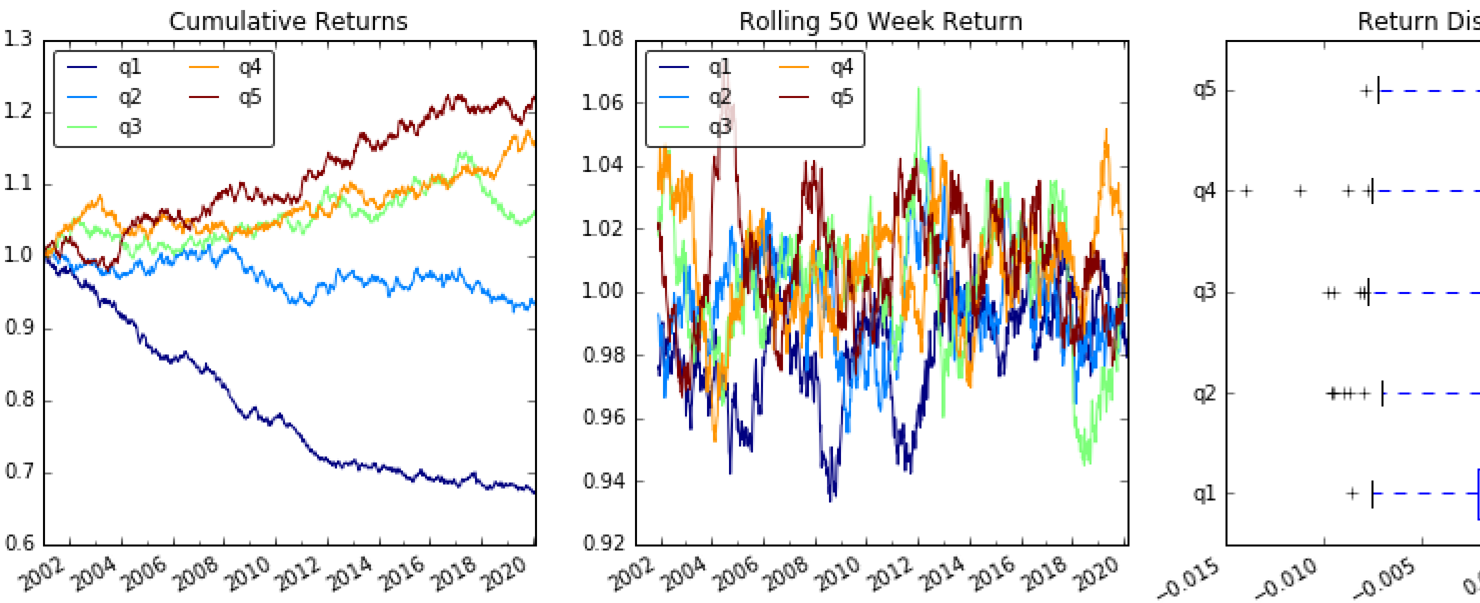
```
returns_cut = security_returns.stack().rename('returns') \
    .to_frame().set_index(cut, append=True) \
    .swaplevel(2, 1).sort_index().squeeze() \
    .groupby(level=[0, 1]).mean().unstack()
```

Analysis

Plot Returns

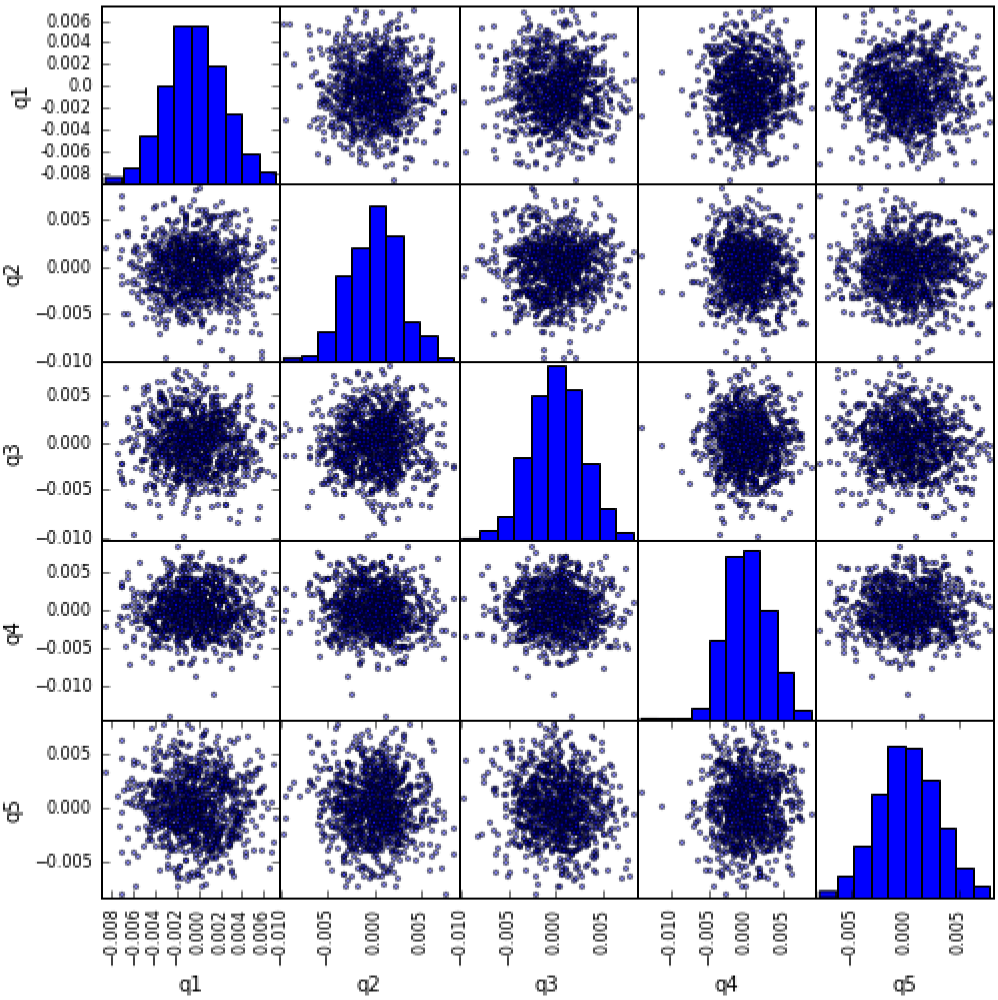
```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(15, 5))
```

```
ax1 = plt.subplot2grid((1,3), (0,0))
ax2 = plt.subplot2grid((1,3), (0,1))
ax3 = plt.subplot2grid((1,3), (0,2))

# Cumulative Returns
returns_cut.add(1).cumprod() \
    .plot(colormap='jet', ax=ax1, title="Cumulative Returns")
leg1 = ax1.legend(loc='upper left', ncol=2, prop={'size': 10}, fancybox=True)
leg1.get_frame().set_alpha(.8)

# Rolling 50 Week Return
returns_cut.add(1).rolling(50).apply(lambda x: x.prod()) \
    .plot(colormap='jet', ax=ax2, title="Rolling 50 Week Return")
leg2 = ax2.legend(loc='upper left', ncol=2, prop={'size': 10}, fancybox=True)
leg2.get_frame().set_alpha(.8)

# Return Distribution
returns_cut.plot.box(vert=False, ax=ax3, title="Return Distribution")

fig.autofmt_xdate()

plt.show()
```



Visualize Quintile Correlation with `scatter_matrix`

```
from pandas.tools.plotting import scatter_matrix

scatter_matrix(returns_cut, alpha=0.5, figsize=(8, 8), diagonal='hist')
plt.show()
```



Calculate and visualize Maximum Draw Down

```
def max_dd(returns):
    """returns is a series"""
    r = returns.add(1).cumprod()
    dd = r.div(r.cummax()).sub(1)
    mdd = dd.min()
    end = dd.argmin()
    start = r.loc[:end].argmax()
    return mdd, start, end

def max_dd_df(returns):
    """returns is a dataframe"""
    series = lambda x: pd.Series(x, ['Draw Down', 'Start', 'End'])
    return returns.apply(max_dd).apply(series)
```

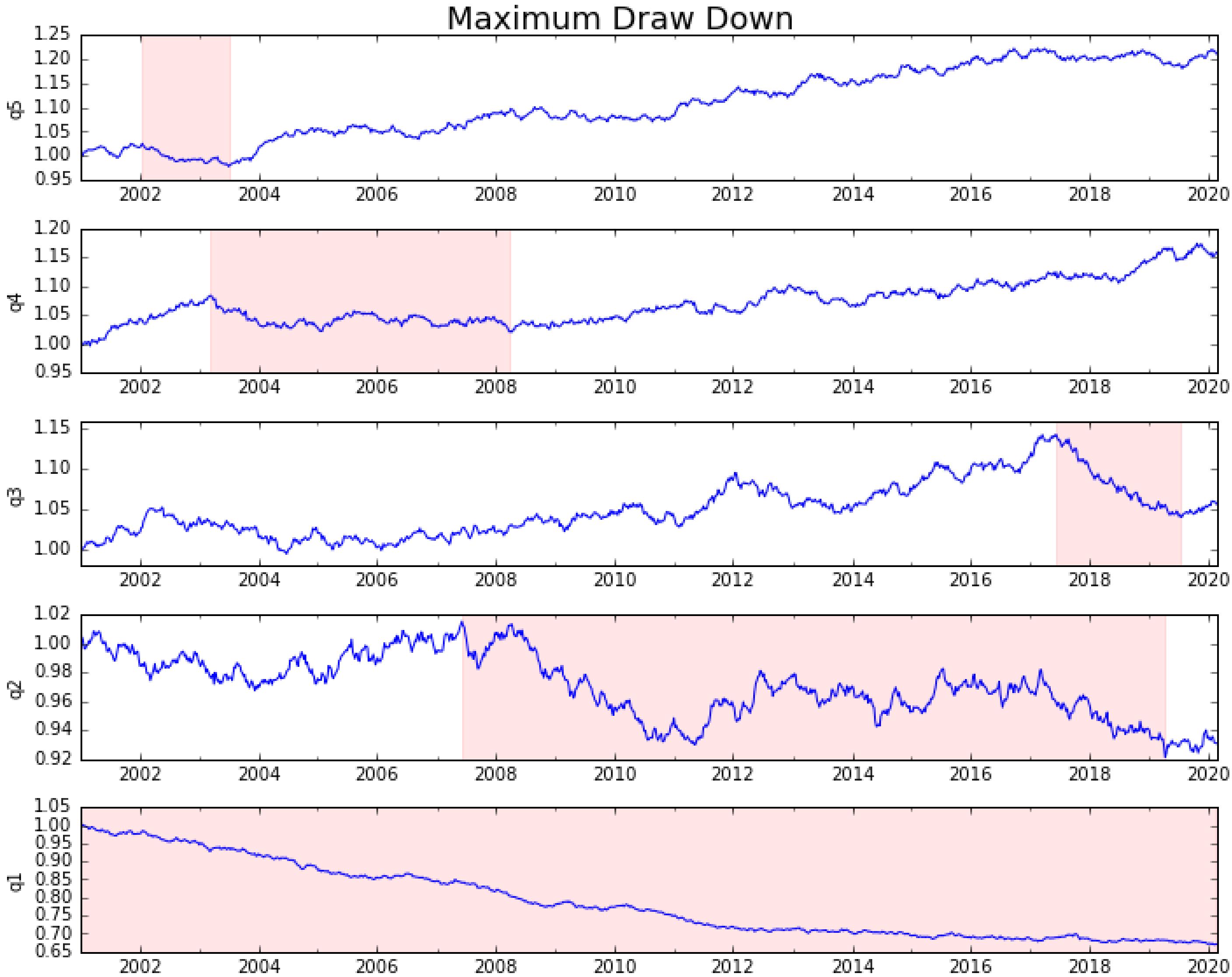
What does this look like

```
max_dd_df(returns_cut)
```

	Draw Down	Start	End
q1	-0.333527	2001-01-07	2020-02-16
q2	-0.092659	2007-06-10	2019-04-14
q3	-0.089682	2017-06-11	2019-07-21
q4	-0.058225	2003-03-16	2008-03-30
q5	-0.046822	2002-01-20	2003-07-06

Let's plot it

```
draw_downs = max_dd_df(returns_cut)

fig, axes = plt.subplots(5, 1, figsize=(10, 8))
for i, ax in enumerate(axes[::-1]):
    returns_cut.iloc[:, i].add(1).cumprod().plot(ax=ax)
    sd, ed = draw_downs[['Start', 'End']].iloc[i]
    ax.axvspan(sd, ed, alpha=0.1, color='r')
    ax.set_ylabel(returns_cut.columns[i])

fig.suptitle('Maximum Draw Down', fontsize=18)
fig.tight_layout()
plt.subplots_adjust(top=.95)
```



Calculate Statistics

There are many potential statistics we can include. Below are just a few, but demonstrate how simply we can incorporate new statistics into our summary.

```
def frequency_of_time_series(df):
    start, end = df.index.min(), df.index.max()
    delta = end - start
    return round((len(df) - 1.) * 365.25 / delta.days, 2)

def annualized_return(df):
    freq = frequency_of_time_series(df)
    return df.add(1).prod() ** (1 / freq) - 1

def annualized_volatility(df):
    freq = frequency_of_time_series(df)
    return df.std().mul(freq ** .5)

def sharpe_ratio(df):
    return annualized_return(df) / annualized_volatility(df)

def describe(df):
```

```
    r = annualized_return(df).rename('Return')
    v = annualized_volatility(df).rename('Volatility')
    s = sharpe_ratio(df).rename('Sharpe')
    skew = df.skew().rename('Skew')
    kurt = df.kurt().rename('Kurtosis')
    desc = df.describe().T

    return pd.concat([r, v, s, skew, kurt, desc], axis=1).T.drop('count')
```

We'll end up using just the `describe` function as it pulls all the others together.

```
describe(returns_cut)
```

	q1	q2	q3	q4	q5
Return	-0.007609	-0.001375	0.001067	0.002821	0.003687
Volatility	0.019584	0.020445	0.020629	0.021185	0.020172
Sharpe	-0.388525	-0.067278	0.051709	0.133176	0.182792
Skew	0.040430	-0.085828	-0.078071	-0.067522	0.005652
Kurtosis	-0.174206	0.203038	0.026385	0.370249	-0.160678
mean	-0.000395	-0.000068	0.000060	0.000151	0.000196
std	0.002711	0.002830	0.002856	0.002933	0.002792
min	-0.008608	-0.009614	-0.009845	-0.014037	-0.007913
25%	-0.002196	-0.002018	-0.001956	-0.001833	-0.001694
50%	-0.000434	0.000065	0.000210	0.000029	0.000146
75%	0.001444	0.001768	0.001989	0.002107	0.002081
max	0.007070	0.008432	0.008100	0.008687	0.007791

This is not meant to be comprehensive. It's meant to bring many of pandas' features together and demonstrate how you can use it to help answer questions important to you. This is a subset of the types of metrics I use to evaluate the efficacy of quantitative factors.

Read Analysis: Bringing it all together and making decisions online: https://riptutorial.com/pandas/topic/5238/analysis--bringing-it-all-together-and-making-decisions

Chapter 3: Appending to DataFrame

Examples

Appending a new row to DataFrame

```
In [1]: import pandas as pd

In [2]: df = pd.DataFrame(columns = ['A', 'B', 'C'])

In [3]: df
Out[3]:
Empty DataFrame
Columns: [A, B, C]
Index: []
```

Appending a row by a single column value:

```
In [4]: df.loc[0, 'A'] = 1

In [5]: df
Out[5]:
   A    B    C
0  1  NaN  NaN
```

Appending a row, given list of values:

```
In [6]: df.loc[1] = [2, 3, 4]

In [7]: df
Out[7]:
   A    B    C
0  1  NaN  NaN
1  2    3    4
```

Appending a row given a dictionary:

```
In [8]: df.loc[2] = {'A': 3, 'C': 9, 'B': 9}

In [9]: df
Out[9]:
   A    B    C
0  1  NaN  NaN
1  2    3    4
2  3    9    9
```

The first input in .loc[] is the index. If you use an existing index, you will overwrite the values in that row:

```
In [17]: df.loc[1] = [5, 6, 7]
```

```
In [18]: df
Out[18]:
   A    B    C
0  1  NaN  NaN
1  5    6    7
2  3    9    9


In [19]: df.loc[0, 'B'] = 8

In [20]: df
Out[20]:
   A  B    C
0  1  8  NaN
1  5  6    7
2  3  9    9
```

Append a DataFrame to another DataFrame

Let us assume we have the following two DataFrames:

```
In [7]: df1
Out[7]:
    A   B
0  a1  b1
1  a2  b2

In [8]: df2
Out[8]:
    B   C
0  b1  c1
```

The two DataFrames are not required to have the same set of columns. The append method does not change either of the original DataFrames. Instead, it returns a new DataFrame by appending the original two. Appending a DataFrame to another one is quite simple:

```
In [9]: df1.append(df2)
Out[9]:
     A   B    C
0   a1  b1  NaN
1   a2  b2  NaN
0  NaN  b1   c1
```

As you can see, it is possible to have duplicate indices (0 in this example). To avoid this issue, you may ask Pandas to reindex the new DataFrame for you:

```
In [10]: df1.append(df2, ignore_index = True)
Out[10]:
     A   B    C
0   a1  b1  NaN
1   a2  b2  NaN
2  NaN  b1   c1
```

Read Appending to DataFrame online: https://riptutorial.com/pandas/topic/6456/appending-to-dataframe

Chapter 4: Boolean indexing of dataframes

Introduction

Accessing rows in a dataframe using the DataFrame indexer objects `.ix`, `.loc`, `.iloc` and how it differentiates itself from using a boolean mask.

Examples

Accessing a DataFrame with a boolean index

This will be our example data frame:

```
df = pd.DataFrame({"color": ['red', 'blue', 'red', 'blue']},
                  index=[True, False, True, False])
      color
True    red
False  blue
True    red
False  blue
```

Accessing with `.loc`

```
df.loc[True]
     color
True   red
True   red
```

Accessing with `.iloc`

```
df.iloc[True]
>> TypeError

df.iloc[1]
color    blue
dtype: object
```

> Important to note is that older pandas versions did not distinguish between boolean and integer input, thus `.iloc[True]` would return the same as `.iloc[1]`

Accessing with `.ix`

```
df.ix[True]
     color
True   red
True   red

df.ix[1]
color    blue
```

```
dtype: object
```

As you can see, `.ix` has two behaviors. This is very bad practice in code and thus it should be avoided. Please use `.iloc` or `.loc` to be more explicit.

Applying a boolean mask to a dataframe

This will be our example data frame:

```
  color      name   size
0   red      rose    big
1  blue    violet    big
2   red     tulip  small
3  blue  harebell  small
```

Using the magic `__getitem__` or `[]` accessor. Giving it a list of True and False of the same length as the dataframe will give you:

```
df[[True, False, True, False]]
  color   name   size
0   red   rose    big
2   red  tulip  small
```

Masking data based on column value

This will be our example data frame:

```
  color      name   size
0   red      rose    big
1  blue    violet  small
2   red     tulip  small
3  blue  harebell  small
```

Accessing a single column from a data frame, we can use a simple comparison `==` to compare every element in the column to the given variable, producing a `pd.Series` of True and False

```
df['size'] == 'small'
0    False
1     True
2     True
3     True
Name: size, dtype: bool
```

This `pd.Series` is an extension of an `np.array` which is an extension of a simple `list`, Thus we can hand this to the `__getitem__` or `[]` accessor as in the above example.

```
size_small_mask = df['size'] == 'small'
df[size_small_mask]
  color      name   size
1  blue    violet  small
2   red     tulip  small
```

```
3  blue   harebell   small
```

Masking data based on index value

This will be our example data frame:

```
         color   size
name
rose       red    big
violet    blue  small
tulip      red  small
harebell  blue  small
```

We can create a mask based on the index values, just like on a column value.

```
rose_mask = df.index == 'rose'
df[rose_mask]
     color size
name
rose   red  big
```

But doing this is *almost* the same as

```
df.loc['rose']
color    red
size     big
Name: rose, dtype: object
```

The important difference being, when `.loc` only encounters one row in the index that matches, it will return a `pd.Series`, if it encounters more rows that matches, it will return a `pd.DataFrame`. This makes this method rather unstable.

This behavior can be controlled by giving the `.loc` a list of a single entry. This will force it to return a data frame.

```
df.loc[['rose']]
         color   size
name
rose       red    big
```

Read Boolean indexing of dataframes online: https://riptutorial.com/pandas/topic/9589/boolean-indexing-of-dataframes

Chapter 5: Categorical data

Introduction

Categoricals are a pandas data type, which correspond to categorical variables in statistics: a variable, which can take on only a limited, and usually fixed, number of possible values (categories; levels in R). Examples are gender, social class, blood types, country affiliations, observation time or ratings via Likert scales. Source: Pandas Docs

Examples

Object Creation

```
In [188]: s = pd.Series(["a","b","c","a","c"], dtype="category")

In [189]: s
Out[189]:
0    a
1    b
2    c
3    a
4    c
dtype: category
Categories (3, object): [a, b, c]

In [190]: df = pd.DataFrame({"A":["a","b","c","a", "c"]})

In [191]: df["B"] = df["A"].astype('category')

In [192]: df["C"] = pd.Categorical(df["A"])

In [193]: df
Out[193]:
   A  B  C
0  a  a  a
1  b  b  b
2  c  c  c
3  a  a  a
4  c  c  c

In [194]: df.dtypes
Out[194]:
A      object
B    category
C    category
dtype: object
```

Creating large random datasets

```
In [1]: import pandas as pd
        import numpy as np
```

```
In [2]: df = pd.DataFrame(np.random.choice(['foo','bar','baz'], size=(100000,3)))
        df = df.apply(lambda col: col.astype('category'))

In [3]: df.head()
Out[3]:
     0    1    2
0  bar  foo  baz
1  baz  bar  baz
2  foo  foo  bar
3  bar  baz  baz
4  foo  bar  baz

In [4]: df.dtypes
Out[4]:
0    category
1    category
2    category
dtype: object

In [5]: df.shape
Out[5]: (100000, 3)
```

Read Categorical data online: https://riptutorial.com/pandas/topic/3887/categorical-data

Chapter 6: Computational Tools

Examples

Find The Correlation Between Columns

Suppose you have a DataFrame of numerical values, for example:

```
df = pd.DataFrame(np.random.randn(1000, 3), columns=['a', 'b', 'c'])
```

Then

```
>>> df.corr()
    a    b    c
a    1.000000    0.018602    0.038098
b    0.018602    1.000000    -0.014245
c    0.038098    -0.014245    1.000000
```

will find the Pearson correlation between the columns. Note how the diagonal is 1, as each column is (obviously) fully correlated with itself.

`pd.DataFrame.correlation` takes an optional `method` parameter, specifying which algorithm to use. The default is `pearson`. To use Spearman correlation, for example, use

```
>>> df.corr(method='spearman')
    a    b    c
a    1.000000    0.007744    0.037209
b    0.007744    1.000000    -0.011823
c    0.037209    -0.011823    1.000000
```

Read Computational Tools online: https://riptutorial.com/pandas/topic/5620/computational-tools

Chapter 7: Creating DataFrames

Introduction

DataFrame is a data structure provided by pandas library,apart from *Series* & *Panel*. It is a 2-dimensional structure & can be compared to a table of rows and columns.

Each row can be identified by an integer index (0..N) or a label explicitly set when creating a DataFrame object. Each column can be of distinct type and is identified by a label.

This topic covers various ways to construct/create a DataFrame object. Ex. from Numpy arrays, from list of tuples, from dictionary.

Examples

Create a sample DataFrame

```
import pandas as pd
```

Create a DataFrame from a dictionary, containing two columns: `numbers` and `colors`. Each key represent a column name and the value is a series of data, the content of the column:

```
df = pd.DataFrame({'numbers': [1, 2, 3], 'colors': ['red', 'white', 'blue']})
```

Show contents of dataframe:

```
print(df)
# Output:
#   colors  numbers
# 0    red        1
# 1  white        2
# 2   blue        3
```

Pandas orders columns alphabetically as `dict` are not ordered. To specify the order, use the `columns` parameter.

```
df = pd.DataFrame({'numbers': [1, 2, 3], 'colors': ['red', 'white', 'blue']},
                  columns=['numbers', 'colors'])

print(df)
# Output:
#    numbers colors
# 0        1    red
# 1        2  white
# 2        3   blue
```

Create a sample DataFrame using Numpy

Create a `DataFrame` of random numbers:

```
import numpy as np
import pandas as pd

# Set the seed for a reproducible sample
np.random.seed(0)

df = pd.DataFrame(np.random.randn(5, 3), columns=list('ABC'))

print(df)
# Output:
#           A         B         C
# 0  1.764052  0.400157  0.978738
# 1  2.240893  1.867558 -0.977278
# 2  0.950088 -0.151357 -0.103219
# 3  0.410599  0.144044  1.454274
# 4  0.761038  0.121675  0.443863
```

Create a `DataFrame` with integers:

```
df = pd.DataFrame(np.arange(15).reshape(5,3),columns=list('ABC'))

print(df)
# Output:
#     A   B   C
# 0   0   1   2
# 1   3   4   5
# 2   6   7   8
# 3   9  10  11
# 4  12  13  14
```

Create a `DataFrame` and include nans (`NaT, NaN, 'nan', None`) across columns and rows:

```
df = pd.DataFrame(np.arange(48).reshape(8,6),columns=list('ABCDEF'))

print(df)
# Output:
#     A   B   C   D   E   F
# 0   0   1   2   3   4   5
# 1   6   7   8   9  10  11
# 2  12  13  14  15  16  17
# 3  18  19  20  21  22  23
# 4  24  25  26  27  28  29
# 5  30  31  32  33  34  35
# 6  36  37  38  39  40  41
# 7  42  43  44  45  46  47

df.ix[::2,0] = np.nan # in column 0, set elements with indices 0,2,4, ... to NaN
df.ix[::4,1] = pd.NaT # in column 1, set elements with indices 0,4, ... to np.NaT
df.ix[:3,2] = 'nan'   # in column 2, set elements with index from 0 to 3 to 'nan'
df.ix[:,5] = None     # in column 5, set all elements to None
df.ix[5,:] = None     # in row 5, set all elements to None
df.ix[7,:] = np.nan   # in row 7, set all elements to NaN

print(df)
# Output:
#     A     B     C   D   E     F
```

```
# 0 NaN   NaT   nan   3   4  None
# 1   6     7   nan   9  10  None
# 2 NaN    13   nan  15  16  None
# 3  18    19   nan  21  22  None
# 4 NaN   NaT    26  27  28  None
# 5 NaN  None  None NaN NaN  None
# 6 NaN    37    38  39  40  None
# 7 NaN   NaN   NaN NaN NaN   NaN
```

Create a sample DataFrame from multiple collections using Dictionary

```
import pandas as pd
import numpy as np

np.random.seed(123)
x = np.random.standard_normal(4)
y = range(4)
df = pd.DataFrame({'X':x, 'Y':y})
>>> df
          X  Y
0 -1.085631  0
1  0.997345  1
2  0.282978  2
3 -1.506295  3
```

Create a DataFrame from a list of tuples

You can create a DataFrame from a list of simple tuples, and can even choose the specific elements of the tuples you want to use. Here we will create a DataFrame using all of the data in each tuple except for the last element.

```
import pandas as pd

data = [
('p1', 't1', 1, 2),
('p1', 't2', 3, 4),
('p2', 't1', 5, 6),
('p2', 't2', 7, 8),
('p2', 't3', 2, 8)
]

df = pd.DataFrame(data)

print(df)
#     0   1  2  3
# 0  p1  t1  1  2
# 1  p1  t2  3  4
# 2  p2  t1  5  6
# 3  p2  t2  7  8
# 4  p2  t3  2  8
```

Create a DataFrame from a dictionary of lists

Create a DataFrame from multiple lists by passing a dict whose values lists. The keys of the dictionary are used as column labels. The lists can also be ndarrays. The lists/ndarrays must all be the same length.

```
import pandas as pd

# Create DF from dict of lists/ndarrays
df = pd.DataFrame({'A' : [1, 2, 3, 4],
                       'B' : [4, 3, 2, 1]})
df
# Output:
#       A  B
#    0  1  4
#    1  2  3
#    2  3  2
#    3  4  1
```

If the arrays are not the same length an error is raised

```
df = pd.DataFrame({'A' : [1, 2, 3, 4], 'B' : [5, 5, 5]}) # a ValueError is raised
```

Using ndarrays

```
import pandas as pd
import numpy as np

np.random.seed(123)
x = np.random.standard_normal(4)
y = range(4)
df = pd.DataFrame({'X':x, 'Y':y})
df
# Output:           X  Y
#         0 -1.085631  0
#         1  0.997345  1
#         2  0.282978  2
#         3 -1.506295  3
```

See additional details at: http://pandas.pydata.org/pandas-docs/stable/dsintro.html#from-dict-of-ndarrays-lists

Create a sample DataFrame with datetime

```
import pandas as pd
import numpy as np

np.random.seed(0)
# create an array of 5 dates starting at '2015-02-24', one per minute
rng = pd.date_range('2015-02-24', periods=5, freq='T')
df = pd.DataFrame({ 'Date': rng, 'Val': np.random.randn(len(rng)) })

print (df)
# Output:
#                  Date       Val
# 0 2015-02-24 00:00:00  1.764052
# 1 2015-02-24 00:01:00  0.400157
# 2 2015-02-24 00:02:00  0.978738
# 3 2015-02-24 00:03:00  2.240893
```

```
# 4 2015-02-24 00:04:00  1.867558

# create an array of 5 dates starting at '2015-02-24', one per day
rng = pd.date_range('2015-02-24', periods=5, freq='D')
df = pd.DataFrame({ 'Date': rng, 'Val' : np.random.randn(len(rng))})

print (df)
# Output:
#          Date       Val
# 0 2015-02-24 -0.977278
# 1 2015-02-25  0.950088
# 2 2015-02-26 -0.151357
# 3 2015-02-27 -0.103219
# 4 2015-02-28  0.410599

# create an array of 5 dates starting at '2015-02-24', one every 3 years
rng = pd.date_range('2015-02-24', periods=5, freq='3A')
df = pd.DataFrame({ 'Date': rng, 'Val' : np.random.randn(len(rng))})

print (df)
# Output:
#          Date       Val
# 0 2015-12-31  0.144044
# 1 2018-12-31  1.454274
# 2 2021-12-31  0.761038
# 3 2024-12-31  0.121675
# 4 2027-12-31  0.443863
```

DataFrame with `DatetimeIndex`:

```
import pandas as pd
import numpy as np

np.random.seed(0)
rng = pd.date_range('2015-02-24', periods=5, freq='T')
df = pd.DataFrame({ 'Val' : np.random.randn(len(rng)) }, index=rng)

print (df)
# Output:
#                            Val
# 2015-02-24 00:00:00  1.764052
# 2015-02-24 00:01:00  0.400157
# 2015-02-24 00:02:00  0.978738
# 2015-02-24 00:03:00  2.240893
# 2015-02-24 00:04:00  1.867558
```

`Offset-aliases` for parameter `freq` in `date_range`:

```
Alias     Description
B         business day frequency
C         custom business day frequency (experimental)
D         calendar day frequency
W         weekly frequency
M         month end frequency
BM        business month end frequency
CBM       custom business month end frequency
MS        month start frequency
BMS       business month start frequency
CBMS      custom business month start frequency
```

```
Q         quarter end frequency
BQ        business quarter endfrequency
QS        quarter start frequency
BQS       business quarter start frequency
A         year end frequency
BA        business year end frequency
AS        year start frequency
BAS       business year start frequency
BH        business hour frequency
H         hourly frequency
T, min    minutely frequency
S         secondly frequency
L, ms     milliseconds
U, us     microseconds
N         nanoseconds
```

Create a sample DataFrame with MultiIndex

```
import pandas as pd
import numpy as np
```

Using `from_tuples`:

```
np.random.seed(0)
tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
                     'foo', 'foo', 'qux', 'qux'],
                      ['one', 'two', 'one', 'two',
                       'one', 'two', 'one', 'two']]))

idx = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
```

Using `from_product`:

```
idx = pd.MultiIndex.from_product([['bar', 'baz', 'foo', 'qux'],['one','two']])
```

Then, use this MultiIndex:

```
df = pd.DataFrame(np.random.randn(8, 2), index=idx, columns=['A', 'B'])
print (df)
                     A         B
first second
bar   one     1.764052  0.400157
      two     0.978738  2.240893
baz   one     1.867558 -0.977278
      two     0.950088 -0.151357
foo   one    -0.103219  0.410599
      two     0.144044  1.454274
qux   one     0.761038  0.121675
      two     0.443863  0.333674
```

Save and Load a DataFrame in pickle (.plk) format

```
import pandas as pd
```

```
# Save dataframe to pickled pandas object
df.to_pickle(file_name) # where to save it usually as a .plk

# Load dataframe from pickled pandas object
df= pd.read_pickle(file_name)
```

Create a DataFrame from a list of dictionaries

A DataFrame can be created from a list of dictionaries. Keys are used as column names.

```
import pandas as pd
L = [{'Name': 'John', 'Last Name': 'Smith'},
         {'Name': 'Mary', 'Last Name': 'Wood'}]
pd.DataFrame(L)
# Output:  Last Name  Name
# 0     Smith  John
# 1      Wood  Mary
```

Missing values are filled with `NaN`s

```
L = [{'Name': 'John', 'Last Name': 'Smith', 'Age': 37},
     {'Name': 'Mary', 'Last Name': 'Wood'}]
pd.DataFrame(L)
# Output:     Age Last Name  Name
#          0   37     Smith  John
#          1  NaN      Wood  Mary
```

Read Creating DataFrames online: https://riptutorial.com/pandas/topic/1595/creating-dataframes

Chapter 8: Cross sections of different axes with MultiIndex

Examples

Selection of cross-sections using .xs

```
In [1]:
import pandas as pd
import numpy as np
arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
          ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
idx_row = pd.MultiIndex.from_arrays(arrays, names=['Row_First', 'Row_Second'])
idx_col = pd.MultiIndex.from_product([['A','B'], ['i', 'ii']],
names=['Col_First','Col_Second'])
df = pd.DataFrame(np.random.randn(8,4), index=idx_row, columns=idx_col)

Out[1]:
Col_First                   A                   B
Col_Second                  i        ii         i        ii
Row_First Row_Second
bar       one       -0.452982 -1.872641  0.248450 -0.319433
          two       -0.460388 -0.136089 -0.408048  0.998774
baz       one        0.358206 -0.319344 -2.052081 -0.424957
          two       -0.823811 -0.302336  1.158968  0.272881
foo       one       -0.098048 -0.799666  0.969043 -0.595635
          two       -0.358485  0.412011 -0.667167  1.010457
qux       one        1.176911  1.578676  0.350719  0.093351
          two        0.241956  1.082138 -0.516898 -0.196605
```

`.xs` accepts a `level` (either the name of said level or an integer), and an `axis`: 0 for rows, 1 for columns.

`.xs` is available for both `pandas.Series` and `pandas.DataFrame`.

Selection on rows:

```
In [2]: df.xs('two', level='Row_Second', axis=0)
Out[2]:
Col_First          A                   B
Col_Second         i        ii         i        ii
Row_First
bar        -0.460388 -0.136089 -0.408048  0.998774
baz        -0.823811 -0.302336  1.158968  0.272881
foo        -0.358485  0.412011 -0.667167  1.010457
qux         0.241956  1.082138 -0.516898 -0.196605
```

Selection on columns:

```
In [3]: df.xs('ii', level=1, axis=1)
Out[3]:
```

```
Col_First                    A         B
Row_First Row_Second
bar       one        -1.872641 -0.319433
          two        -0.136089  0.998774
baz       one        -0.319344 -0.424957
          two        -0.302336  0.272881
foo       one        -0.799666 -0.595635
          two         0.412011  1.010457
qux       one         1.578676  0.093351
          two         1.082138 -0.196605
```

`.xs` only works for selection , assignment is NOT possible (getting, not setting):¨

```
In [4]: df.xs('ii', level='Col_Second', axis=1) = 0
  File "<ipython-input-10-92e0785187ba>", line 1
    df.xs('ii', level='Col_Second', axis=1) = 0
                                           ^
SyntaxError: can't assign to function call
```

Using .loc and slicers

Unlike the `.xs` method, this allows you to assign values. Indexing using slicers is available since version `0.14.0`.

```
In [1]:
import pandas as pd
import numpy as np
arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
          ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
idx_row = pd.MultiIndex.from_arrays(arrays, names=['Row_First', 'Row_Second'])
idx_col = pd.MultiIndex.from_product([['A','B'], ['i', 'ii']],
names=['Col_First','Col_Second'])
df = pd.DataFrame(np.random.randn(8,4), index=idx_row, columns=idx_col)

Out[1]:
Col_First                    A                   B
Col_Second                   i        ii         i        ii
Row_First Row_Second
bar       one        -0.452982 -1.872641  0.248450 -0.319433
          two        -0.460388 -0.136089 -0.408048  0.998774
baz       one         0.358206 -0.319344 -2.052081 -0.424957
          two        -0.823811 -0.302336  1.158968  0.272881
foo       one        -0.098048 -0.799666  0.969043 -0.595635
          two        -0.358485  0.412011 -0.667167  1.010457
qux       one         1.176911  1.578676  0.350719  0.093351
          two         0.241956  1.082138 -0.516898 -0.196605
```

Selection on rows:

```
In [2]: df.loc[(slice(None),'two'),:]
Out[2]:
Col_First                    A                   B
Col_Second                   i        ii         i        ii
Row_First Row_Second
bar       two        -0.460388 -0.136089 -0.408048  0.998774
baz       two        -0.823811 -0.302336  1.158968  0.272881
```

```
foo       two         -0.358485  0.412011 -0.667167  1.010457
qux       two          0.241956  1.082138 -0.516898 -0.196605
```

Selection on columns:

```
In [3]: df.loc[:,(slice(None),'ii')]
Out[3]:
Col_First                    A         B
Col_Second                  ii        ii
Row_First Row_Second
bar       one        -1.872641 -0.319433
          two        -0.136089  0.998774
baz       one        -0.319344 -0.424957
          two        -0.302336  0.272881
foo       one        -0.799666 -0.595635
          two         0.412011  1.010457
qux       one         1.578676  0.093351
          two         1.082138 -0.196605
```

Selection on both axis::

```
In [4]: df.loc[(slice(None),'two'),(slice(None),'ii')]
Out[4]:
Col_First                    A         B
Col_Second                  ii        ii
Row_First Row_Second
bar       two        -0.136089  0.998774
baz       two        -0.302336  0.272881
foo       two         0.412011  1.010457
qux       two         1.082138 -0.196605
```

Assignment works (unlike `.xs`):

```
In [5]: df.loc[(slice(None),'two'),(slice(None),'ii')]=0
         df
Out[5]:
Col_First                    A                   B
Col_Second                   i        ii         i        ii
Row_First Row_Second
bar       one        -0.452982 -1.872641  0.248450 -0.319433
          two        -0.460388  0.000000 -0.408048  0.000000
baz       one         0.358206 -0.319344 -2.052081 -0.424957
          two        -0.823811  0.000000  1.158968  0.000000
foo       one        -0.098048 -0.799666  0.969043 -0.595635
          two        -0.358485  0.000000 -0.667167  0.000000
qux       one         1.176911  1.578676  0.350719  0.093351
          two         0.241956  0.000000 -0.516898  0.000000
```

Read Cross sections of different axes with MultiIndex online: https://riptutorial.com/pandas/topic/8099/cross-sections-of-different-axes-with-multiindex

Chapter 9: Data Types

Remarks

dtypes are not native to pandas. They are a result of pandas close architectural coupling to numpy.

the dtype of a column does not in any way have to correlate to the python type of the object contained in the column.

Here we have a `pd.Series` with floats. The dtype will be `float`.

Then we use `astype` to "cast" it to object.

```
pd.Series([1.,2.,3.,4.,5.]).astype(object)
0    1
1    2
2    3
3    4
4    5
dtype: object
```

The dtype is now object, but the objects in the list are still float. Logical if you know that in python, everything is an object, and can be upcasted to object.

```
type(pd.Series([1.,2.,3.,4.,5.]).astype(object)[0])
float
```

Here we try "casting" the floats to strings.

```
pd.Series([1.,2.,3.,4.,5.]).astype(str)
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
dtype: object
```

The dtype is now object, but the type of the entries in the list are string. This is because `numpy` does not deal with strings, and thus acts as if they are just objects and of no concern.

```
type(pd.Series([1.,2.,3.,4.,5.]).astype(str)[0])
str
```

Do not trust dtypes, they are an artifact of an architectural flaw in pandas. Specify them as you must, but do not rely on what dtype is set on a column.

Examples

Checking the types of columns

Types of columns can be checked by `.dtypes` atrribute of DataFrames.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0], 'C': [True, False, True]})

In [2]: df
Out[2]:
   A    B      C
0  1  1.0   True
1  2  2.0  False
2  3  3.0   True

In [3]: df.dtypes
Out[3]:
A      int64
B    float64
C       bool
dtype: object
```

For a single series, you can use `.dtype` attribute.

```
In [4]: df['A'].dtype
Out[4]: dtype('int64')
```

Changing dtypes

`astype()` method changes the dtype of a Series and returns a new Series.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0],
                           'C': ['1.1.2010', '2.1.2011', '3.1.2011'],
                           'D': ['1 days', '2 days', '3 days'],
                           'E': ['1', '2', '3']})
In [2]: df
Out[2]:
   A    B          C       D  E
0  1  1.0   1.1.2010  1 days  1
1  2  2.0   2.1.2011  2 days  2
2  3  3.0   3.1.2011  3 days  3

In [3]: df.dtypes
Out[3]:
A      int64
B    float64
C     object
D     object
E     object
dtype: object
```

Change the type of column A to float, and type of column B to integer:

```
In [4]: df['A'].astype('float')
Out[4]:
0    1.0
1    2.0
```

```
2    3.0
Name: A, dtype: float64

In [5]: df['B'].astype('int')
Out[5]:
0    1
1    2
2    3
Name: B, dtype: int32
```

`astype()` method is for specific type conversion (i.e. you can specify `.astype(float64')`, `.astype(float32)`, or `.astype(float16)`). For general conversion, you can use `pd.to_numeric`, `pd.to_datetime` and `pd.to_timedelta`.

Changing the type to numeric

`pd.to_numeric` changes the values to a numeric type.

```
In [6]: pd.to_numeric(df['E'])
Out[6]:
0    1
1    2
2    3
Name: E, dtype: int64
```

By default, `pd.to_numeric` raises an error if an input cannot be converted to a number. You can change that behavior by using the `errors` parameter.

```
# Ignore the error, return the original input if it cannot be converted
In [7]: pd.to_numeric(pd.Series(['1', '2', 'a']), errors='ignore')
Out[7]:
0    1
1    2
2    a
dtype: object

# Return NaN when the input cannot be converted to a number
In [8]: pd.to_numeric(pd.Series(['1', '2', 'a']), errors='coerce')
Out[8]:
0    1.0
1    2.0
2    NaN
dtype: float64
```

If need check all rows with input cannot be converted to numeric use `boolean indexing` with `isnull`:

```
In [9]: df = pd.DataFrame({'A': [1, 'x', 'z'],
                           'B': [1.0, 2.0, 3.0],
                           'C': [True, False, True]})

In [10]: pd.to_numeric(df.A, errors='coerce').isnull()
Out[10]:
0    False
1     True
```

```
2     True
Name: A, dtype: bool

In [11]: df[pd.to_numeric(df.A, errors='coerce').isnull()]
Out[11]:
   A    B      C
1  x  2.0  False
2  z  3.0   True
```

Changing the type to datetime

```
In [12]: pd.to_datetime(df['C'])
Out[12]:
0   2010-01-01
1   2011-02-01
2   2011-03-01
Name: C, dtype: datetime64[ns]
```

Note that 2.1.2011 is converted to February 1, 2011. If you want January 2, 2011 instead, you need to use the `dayfirst` parameter.

```
In [13]: pd.to_datetime('2.1.2011', dayfirst=True)
Out[13]: Timestamp('2011-01-02 00:00:00')
```

Changing the type to timedelta

```
In [14]: pd.to_timedelta(df['D'])
Out[14]:
0   1 days
1   2 days
2   3 days
Name: D, dtype: timedelta64[ns]
```

Selecting columns based on dtype

`select_dtypes` method can be used to select columns based on dtype.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0], 'C': ['a', 'b', 'c'],
                           'D': [True, False, True]})

In [2]: df
Out[2]:
   A    B  C      D
0  1  1.0  a   True
1  2  2.0  b  False
2  3  3.0  c   True
```

With `include` and `exclude` parameters you can specify which types you want:

```
# Select numbers
In [3]: df.select_dtypes(include=['number'])  # You need to use a list
```

```
Out[3]:
   A    B
0  1  1.0
1  2  2.0
2  3  3.0

# Select numbers and booleans
In [4]: df.select_dtypes(include=['number', 'bool'])
Out[4]:
   A    B      D
0  1  1.0   True
1  2  2.0  False
2  3  3.0   True

# Select numbers and booleans but exclude int64
In [5]: df.select_dtypes(include=['number', 'bool'], exclude=['int64'])
Out[5]:
     B      D
0  1.0   True
1  2.0  False
2  3.0   True
```

Summarizing dtypes

`get_dtype_counts` method can be used to see a breakdown of dtypes.

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0], 'C': ['a', 'b', 'c'],
                           'D': [True, False, True]})

In [2]: df.get_dtype_counts()
Out[2]:
bool       1
float64    1
int64      1
object     1
dtype: int64
```

Read Data Types online: https://riptutorial.com/pandas/topic/2959/data-types

Chapter 10: Dealing with categorical variables

Examples

One-hot encoding with `get_dummies()`

```
>>> df = pd.DataFrame({'Name':['John Smith', 'Mary Brown'],
                     'Gender':['M', 'F'], 'Smoker':['Y', 'N']})
>>> print(df)
```

```
  Gender        Name Smoker
0      M  John Smith      Y
1      F  Mary Brown      N
```

```
>>> df_with_dummies = pd.get_dummies(df, columns=['Gender', 'Smoker'])
>>> print(df_with_dummies)
```

```
         Name  Gender_F  Gender_M  Smoker_N  Smoker_Y
0  John Smith       0.0       1.0       0.0       1.0
1  Mary Brown       1.0       0.0       1.0       0.0
```

Read Dealing with categorical variables online: https://riptutorial.com/pandas/topic/5999/dealing-with-categorical-variables

Chapter 11: Duplicated data

Examples

Select duplicated

If need set value `0` to column `B`, where in column `A` are duplicated data first create mask by `Series.duplicated` and then use `DataFrame.ix` or `Series.mask`:

```
In [224]: df = pd.DataFrame({'A':[1,2,3,3,2],
     ...:                    'B':[1,7,3,0,8]})

In [225]: mask = df.A.duplicated(keep=False)

In [226]: mask
Out[226]:
0    False
1     True
2     True
3     True
4     True
Name: A, dtype: bool

In [227]: df.ix[mask, 'B'] = 0

In [228]: df['C'] = df.A.mask(mask, 0)

In [229]: df
Out[229]:
   A  B  C
0  1  1  1
1  2  0  0
2  3  0  0
3  3  0  0
4  2  0  0
```

If need invert mask use `~`:

```
In [230]: df['C'] = df.A.mask(~mask, 0)

In [231]: df
Out[231]:
   A  B  C
0  1  1  0
1  2  0  2
2  3  0  3
3  3  0  3
4  2  0  2
```

Drop duplicated

Use `drop_duplicates`:

```
In [216]: df = pd.DataFrame({'A':[1,2,3,3,2],
     ...:                    'B':[1,7,3,0,8]})

In [217]: df
Out[217]:
   A  B
0  1  1
1  2  7
2  3  3
3  3  0
4  2  8

# keep only the last value
In [218]: df.drop_duplicates(subset=['A'], keep='last')
Out[218]:
   A  B
0  1  1
3  3  0
4  2  8

# keep only the first value, default value
In [219]: df.drop_duplicates(subset=['A'], keep='first')
Out[219]:
   A  B
0  1  1
1  2  7
2  3  3

# drop all duplicated values
In [220]: df.drop_duplicates(subset=['A'], keep=False)
Out[220]:
   A  B
0  1  1
```

When you don't want to get a copy of a data frame, but to modify the existing one:

```
In [221]: df = pd.DataFrame({'A':[1,2,3,3,2],
     ...:                    'B':[1,7,3,0,8]})

In [222]: df.drop_duplicates(subset=['A'], inplace=True)

In [223]: df
Out[223]:
   A  B
0  1  1
1  2  7
2  3  3
```

Counting and getting unique elements

Number of unique elements in a series:

```
In [1]: id_numbers = pd.Series([111, 112, 112, 114, 115, 118, 114, 118, 112])
In [2]: id_numbers.nunique()
Out[2]: 5
```

Get unique elements in a series:

```
In [3]: id_numbers.unique()
Out[3]: array([111, 112, 114, 115, 118], dtype=int64)

In [4]: df = pd.DataFrame({'Group': list('ABAABABAAB'),
                           'ID': [1, 1, 2, 3, 3, 2, 1, 2, 1, 3]})

In [5]: df
Out[5]:
  Group  ID
0     A   1
1     B   1
2     A   2
3     A   3
4     B   3
5     A   2
6     B   1
7     A   2
8     A   1
9     B   3
```

Number of unique elements in each group:

```
In [6]: df.groupby('Group')['ID'].nunique()
Out[6]:
Group
A    3
B    2
Name: ID, dtype: int64
```

Get of unique elements in each group:

```
In [7]: df.groupby('Group')['ID'].unique()
Out[7]:
Group
A    [1, 2, 3]
B       [1, 3]
Name: ID, dtype: object
```

Get unique values from a column.

```
In [15]: df = pd.DataFrame({"A":[1,1,2,3,1,1],"B":[5,4,3,4,6,7]})

In [21]: df
Out[21]:
   A  B
0  1  5
1  1  4
2  2  3
3  3  4
4  1  6
5  1  7
```

To get unique values in column A and B.

```
In [22]: df["A"].unique()
```

```
Out[22]: array([1, 2, 3])

In [23]: df["B"].unique()
Out[23]: array([5, 4, 3, 6, 7])
```

To get the unique values in column A as a list (note that `unique()` can be used in two slightly different ways)

```
In [24]: pd.unique(df['A']).tolist()
Out[24]: [1, 2, 3]
```

Here is a more complex example. Say we want to find the unique values from column 'B' where 'A' is equal to 1.

First, let's introduce a duplicate so you can see how it works. Let's replace the 6 in row '4', column 'B' with a 4:

```
In [24]: df.loc['4', 'B'] = 4
Out[24]:
   A  B
0  1  5
1  1  4
2  2  3
3  3  4
4  1  4
5  1  7
```

Now select the data:

```
In [25]: pd.unique(df[df['A'] == 1 ]['B']).tolist()
Out[25]: [5, 4, 7]
```

This can be broken down by thinking of the inner DataFrame first:

```
df['A'] == 1
```

This finds values in column A that are equal to 1, and applies True or False to them. We can then use this to select values from column 'B' of the DataFrame (the outer DataFrame selection)

For comparison, here is the list if we don't use unique. It retrieves every value in column 'B' where column 'A' is 1

```
In [26]: df[df['A'] == 1]['B'].tolist()
Out[26]: [5, 4, 4, 7]
```

Read Duplicated data online: https://riptutorial.com/pandas/topic/2082/duplicated-data

Chapter 12: Getting information about DataFrames

Examples

Get DataFrame information and memory usage

To get basic information about a DataFrame including the column names and datatypes:

```
import pandas as pd

df = pd.DataFrame({'integers': [1, 2, 3],
                   'floats': [1.5, 2.5, 3],
                   'text': ['a', 'b', 'c'],
                   'ints with None': [1, None, 3]})

df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3 entries, 0 to 2
Data columns (total 4 columns):
floats            3 non-null float64
integers          3 non-null int64
ints with None    2 non-null float64
text              3 non-null object
dtypes: float64(2), int64(1), object(1)
memory usage: 120.0+ bytes
```

To get the memory usage of the DataFrame:

```
>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3 entries, 0 to 2
Data columns (total 4 columns):
floats            3 non-null float64
integers          3 non-null int64
ints with None    2 non-null float64
text              3 non-null object
dtypes: float64(2), int64(1), object(1)
memory usage: 234.0 bytes
```

List DataFrame column names

```
df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
```

To list the column names in a DataFrame:

```
>>> list(df)
['a', 'b', 'c']
```

This list comprehension method is especially useful when using the debugger:

```
>>> [c for c in df]
['a', 'b', 'c']
```

This is the long way:

```
sampledf.columns.tolist()
```

You can also print them as an index instead of a list (this won't be very visible for dataframes with many columns though):

```
df.columns
```

Dataframe's various summary statistics.

```
import pandas as pd
df = pd.DataFrame(np.random.randn(5, 5), columns=list('ABCDE'))
```

To generate various summary statistics. For numeric values the number of non-NA/null values (`count`), the mean (`mean`), the standard deviation `std` and values known as the five-number summary :

- `min`: minimum (smallest observation)
- `25%`: lower quartile or first quartile (Q1)
- `50%`: median (middle value, Q2)
- `75%`: upper quartile or third quartile (Q3)
- `max`: maximum (largest observation)

```
>>> df.describe()
              A         B         C         D         E
count  5.000000  5.000000  5.000000  5.000000  5.000000
mean  -0.456917 -0.278666  0.334173  0.863089  0.211153
std    0.925617  1.091155  1.024567  1.238668  1.495219
min   -1.494346 -2.031457 -0.336471 -0.821447 -2.106488
25%   -1.143098 -0.407362 -0.246228 -0.087088 -0.082451
50%   -0.536503 -0.163950 -0.004099  1.509749  0.313918
75%    0.092630  0.381407  0.120137  1.822794  1.060268
max    0.796729  0.828034  2.137527  1.891436  1.870520
```

Read Getting information about DataFrames online:
https://riptutorial.com/pandas/topic/6697/getting-information-about-dataframes

Chapter 13: Gotchas of pandas

Remarks

Gotcha in general is a construct that is although documented, but not intuitive. Gotchas produce some output that is normally not expected because of its counter-intuitive character.

Pandas package has several gotchas, that can confuse someone, who is not aware of them, and some of them are presented on this documentation page.

Examples

Detecting missing values with np.nan

If you want to detect missings with

```
df=pd.DataFrame({'col':[1,np.nan]})
df==np.nan
```

you will get the following result:

```
col
0    False
1    False
```

This is because comparing missing value to anything results in a False - instead of this you should use

```
df=pd.DataFrame({'col':[1,np.nan]})
df.isnull()
```

which results in:

```
col
0    False
1    True
```

Integer and NA

Pandas don't support missing in attributes of type integer. For example if you have missings in the grade column:

```
df= pd.read_csv("data.csv", dtype={'grade': int})
error: Integer column has NA values
```

In this case you just should use float instead of integers or set the object dtype.

Automatic Data Alignment (index-awared behaviour)

If you want to append a series of values [1,2] to the column of dataframe df, you will get NaNs:

```
import pandas as pd

series=pd.Series([1,2])
df=pd.DataFrame(index=[3,4])
df['col']=series
df

   col
3    NaN
4    NaN
```

because setting a new column automatically aligns the data by the indexe, and your values 1 and 2 would get the indexes 0 and 1, and not 3 and 4 as in your data frame:

```
df=pd.DataFrame(index=[1,2])
df['col']=series
df

   col
1      2.0
2      NaN
```

If you want to ignore index, you should set the .values at the end:

```
df['col']=series.values

   col
3    1
4    2
```

Read Gotchas of pandas online: https://riptutorial.com/pandas/topic/6425/gotchas-of-pandas

Chapter 14: Graphs and Visualizations

Examples

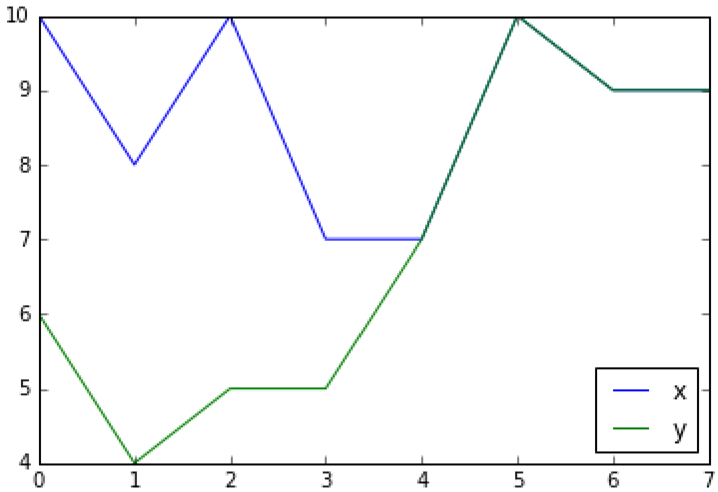
Basic Data Graphs

Pandas uses provides multiple ways to make graphs of the data inside the data frame. It uses matplotlib for that purpose.

The basic graphs have their wrappers for both DataFrame and Series objects:

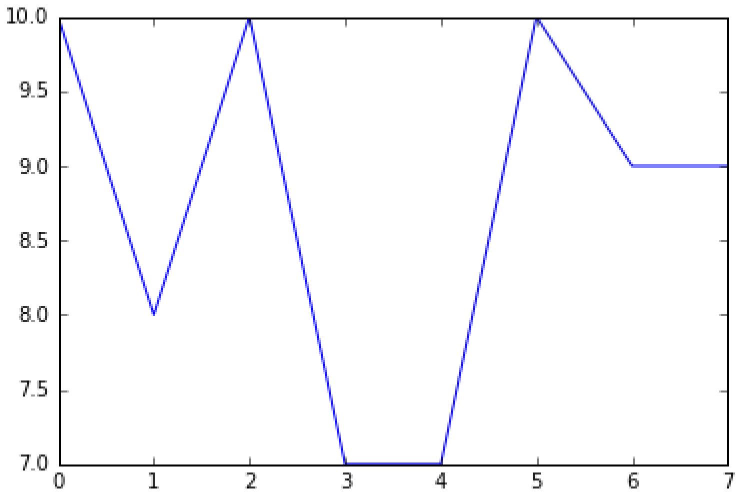
Line Plot

```
df = pd.DataFrame({'x': [10, 8, 10, 7, 7, 10, 9, 9],
                   'y': [6, 4, 5, 5, 7, 10, 9, 9]})
df.plot()
```



You can call the same method for a Series object to plot a subset of the Data Frame:

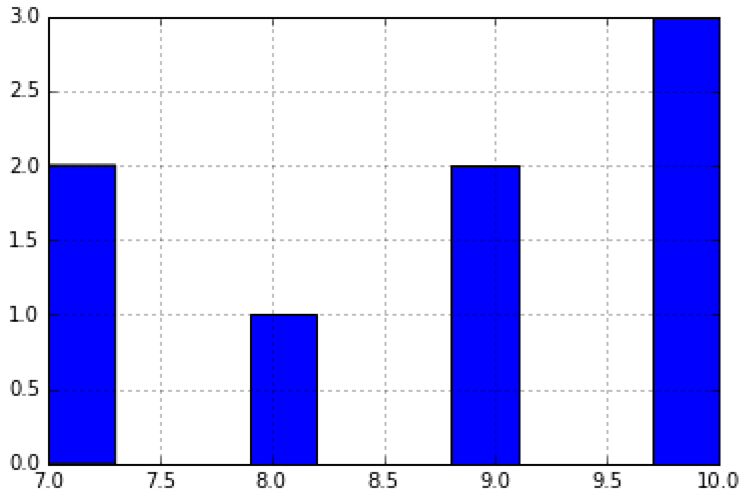
```
df['x'].plot()
```



Bar Chart

If you want to explore the distribution of your data, you can use the `hist()` method.

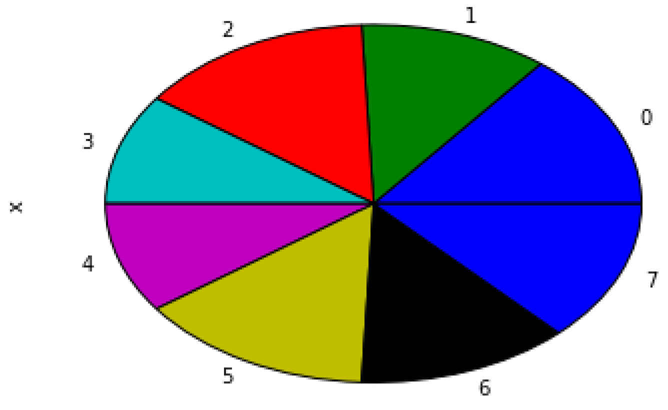
```
df['x'].hist()
```



General method for plotting plot()

All the possible graphs are available through the plot method. The kind of chart is selected by the **kind** argument.

```
df['x'].plot(kind='pie')
```



Note In many environments, the pie chart will come out an oval. To make it a circle, use the following:

```
from matplotlib import pyplot

pyplot.axis('equal')
df['x'].plot(kind='pie')
```

Styling the plot

`plot()` can take arguments that get passed on to matplotlib to style the plot in different ways.

```
df.plot(style='o')  # plot as dots, not lines
df.plot(style='g--')  # plot as green dashed line
df.plot(style='o', markeredgecolor='white')  # plot as dots with white edge
```

Plot on an existing matplotlib axis

By default, `plot()` creates a new figure each time it is called. It is possible to plot on an existing axis by passing the `ax` parameter.

```
plt.figure()  # create a new figure
ax = plt.subplot(121)  # create the left-side subplot
df1.plot(ax=ax)  # plot df1 on that subplot
ax = plt.subplot(122)  # create the right-side subplot
df2.plot(ax=ax)  # and plot df2 there
plt.show()  # show the plot
```

Read Graphs and Visualizations online: https://riptutorial.com/pandas/topic/3839/graphs-and-visualizations

Chapter 15: Grouping Data

Examples

Basic grouping

Group by one column

Using the following DataFrame

```
df = pd.DataFrame({'A': ['a', 'b', 'c', 'a', 'b', 'b'],
                   'B': [2, 8, 1, 4, 3, 8],
                   'C': [102, 98, 107, 104, 115, 87]})

df
# Output:
#    A  B    C
# 0  a  2  102
# 1  b  8   98
# 2  c  1  107
# 3  a  4  104
# 4  b  3  115
# 5  b  8   87
```

Group by column A and get the mean value of other columns:

```
df.groupby('A').mean()
# Output:
#            B    C
# A
# a  3.000000  103
# b  6.333333  100
# c  1.000000  107
```

Group by multiple columns

```
df.groupby(['A','B']).mean()
# Output:
#          C
# A B
# a 2  102.0
#   4  104.0
# b 3  115.0
#   8   92.5
# c 1  107.0
```

Note how after grouping each row in the resulting DataFrame is indexed by a tuple or MultiIndex (in this case a pair of elements from columns A and B).

To apply several aggregation methods at once, for instance to count the number of items in each group and compute their mean, use the `agg` function:

```
df.groupby(['A','B']).agg(['count', 'mean'])
# Output:
#         C
#     count   mean
# A B
# a 2     1  102.0
#   4     1  104.0
# b 3     1  115.0
#   8     2   92.5
# c 1     1  107.0
```

Grouping numbers

For the following DataFrame:

```
import numpy as np
import pandas as pd
np.random.seed(0)
df = pd.DataFrame({'Age': np.random.randint(20, 70, 100),
                   'Sex': np.random.choice(['Male', 'Female'], 100),
                   'number_of_foo': np.random.randint(1, 20, 100)})
df.head()
# Output:

#    Age     Sex  number_of_foo
# 0   64  Female             14
# 1   67  Female             14
# 2   20  Female             12
# 3   23    Male             17
# 4   23  Female             15
```

Group `Age` into three categories (or bins). Bins can be given as

- an integer `n` indicating the number of bins—in this case the dataframe's data is divided into `n` intervals of equal size
- a sequence of integers denoting the endpoint of the left-open intervals in which the data is divided into—for instance `bins=[19, 40, 65, np.inf]` creates three age groups `(19, 40]`, `(40, 65]`, and `(65, np.inf]`.

Pandas assigns automatically the string versions of the intervals as label. It is also possible to define own labels by defining a `labels` parameter as a list of strings.

```
pd.cut(df['Age'], bins=4)
# this creates four age groups: (19.951, 32.25] < (32.25, 44.5] < (44.5, 56.75] < (56.75, 69]
Name: Age, dtype: category
Categories (4, object): [(19.951, 32.25] < (32.25, 44.5] < (44.5, 56.75] < (56.75, 69]]

pd.cut(df['Age'], bins=[19, 40, 65, np.inf])
# this creates three age groups: (19, 40], (40, 65] and (65, infinity)
Name: Age, dtype: category
Categories (3, object): [(19, 40] < (40, 65] < (65, inf]]
```

Use it in `groupby` to get the mean number of foo:

```
age_groups = pd.cut(df['Age'], bins=[19, 40, 65, np.inf])
df.groupby(age_groups)['number_of_foo'].mean()
# Output:
# Age
# (19, 40]     9.880000
# (40, 65]     9.452381
# (65, inf]    9.250000
# Name: number_of_foo, dtype: float64
```

Cross tabulate age groups and gender:

```
pd.crosstab(age_groups, df['Sex'])
# Output:
# Sex        Female  Male
# Age
# (19, 40]       22    28
# (40, 65]       18    24
# (65, inf]       3     5
```

Column selection of a group

When you do a groupby you can select either a single column or a list of columns:

```
In [11]: df = pd.DataFrame([[1, 1, 2], [1, 2, 3], [2, 3, 4]], columns=["A", "B", "C"])

In [12]: df
Out[12]:
   A  B  C
0  1  1  2
1  1  2  3
2  2  3  4

In [13]: g = df.groupby("A")

In [14]: g["B"].mean()           # just column B
Out[14]:
A
1    1.5
2    3.0
Name: B, dtype: float64

In [15]: g[["B", "C"]].mean()    # columns B and C
Out[15]:
     B    C
A
1  1.5  2.5
2  3.0  4.0
```

You can also use `agg` to specify columns and aggregation to perform:

```
In [16]: g.agg({'B': 'mean', 'C': 'count'})
Out[16]:
   C    B
A
1  2  1.5
2  1  3.0
```

Aggregating by size versus by count

The difference between `size` and `count` is:

`size` counts `NaN` values, `count` does not.

```
df = pd.DataFrame(
        {"Name":["Alice", "Bob", "Mallory", "Mallory", "Bob" , "Mallory"],
         "City":["Seattle", "Seattle", "Portland", "Seattle", "Seattle", "Portland"],
         "Val": [4, 3, 3, np.nan, np.nan, 4]})

df
# Output:
#        City     Name  Val
# 0   Seattle    Alice  4.0
# 1   Seattle      Bob  3.0
# 2  Portland  Mallory  3.0
# 3   Seattle  Mallory  NaN
# 4   Seattle      Bob  NaN
# 5  Portland  Mallory  4.0


df.groupby(["Name", "City"])['Val'].size().reset_index(name='Size')
# Output:
#       Name      City  Size
# 0    Alice   Seattle     1
# 1      Bob   Seattle     2
# 2  Mallory  Portland     2
# 3  Mallory   Seattle     1

df.groupby(["Name", "City"])['Val'].count().reset_index(name='Count')
# Output:
#       Name      City  Count
# 0    Alice   Seattle      1
# 1      Bob   Seattle      1
# 2  Mallory  Portland      2
# 3  Mallory   Seattle      0
```

Aggregating groups

```
In [1]: import numpy as np
In [2]: import pandas as pd

In [3]: df = pd.DataFrame({'A': list('XYZXYZXYZX'), 'B': [1, 2, 1, 3, 1, 2, 3, 3, 1, 2],
                           'C': [12, 14, 11, 12, 13, 14, 16, 12, 10, 19]})

In [4]: df.groupby('A')['B'].agg({'mean': np.mean, 'standard deviation': np.std})
Out[4]:
   standard deviation      mean
A
X            0.957427  2.250000
Y            1.000000  2.000000
Z            0.577350  1.333333
```

For multiple columns:

```
In [5]: df.groupby('A').agg({'B': [np.mean, np.std], 'C': [np.sum, 'count']})
Out[5]:
    C               B
  sum count      mean       std
A
X  59     4  2.250000  0.957427
Y  39     3  2.000000  1.000000
Z  35     3  1.333333  0.577350
```

Export groups in different files

You can iterate on the object returned by `groupby()`. The iterator contains `(Category, DataFrame)` tuples.

```
# Same example data as in the previous example.
import numpy as np
import pandas as pd
np.random.seed(0)
df = pd.DataFrame({'Age': np.random.randint(20, 70, 100),
                   'Sex': np.random.choice(['Male', factor'Female'], 100),
                   'number_of_foo': np.random.randint(1, 20, 100)})

# Export to Male.csv and Female.csv files.
for sex, data in df.groupby('Sex'):
    data.to_csv("{}.csv".format(sex))
```

using transform to get group-level statistics while preserving the original dataframe

example:

```
df = pd.DataFrame({'group1' :  ['A', 'A', 'A', 'A',
                               'B', 'B', 'B', 'B'],
                   'group2' :  ['C', 'C', 'C', 'D',
                               'E', 'E', 'F', 'F'],
                   'B'      :  ['one', np.NaN, np.NaN, np.NaN,
                                np.NaN, 'two', np.NaN, np.NaN],
                   'C'      :  [np.NaN, 1, np.NaN, np.NaN,
                               np.NaN, np.NaN, np.NaN, 4]})

 df
Out[34]:
     B    C group1 group2
0  one  NaN      A      C
1  NaN  1.0      A      C
2  NaN  NaN      A      C
3  NaN  NaN      A      D
4  NaN  NaN      B      E
5  two  NaN      B      E
6  NaN  NaN      B      F
7  NaN  4.0      B      F
```

I want to get the count of non-missing observations of B for each combination of `group1` and `group2`. `groupby.transform` is a very powerful function that does exactly that.

```
df['count_B']=df.groupby(['group1','group2']).B.transform('count')

df
Out[36]:
     B    C group1 group2  count_B
0  one  NaN      A      C        1
1  NaN  1.0      A      C        1
2  NaN  NaN      A      C        1
3  NaN  NaN      A      D        0
4  NaN  NaN      B      E        1
5  two  NaN      B      E        1
6  NaN  NaN      B      F        0
7  NaN  4.0      B      F        0
```

Read Grouping Data online: https://riptutorial.com/pandas/topic/1822/grouping-data

Chapter 16: Grouping Time Series Data

Examples

Generate time series of random numbers then down sample

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# I want 7 days of 24 hours with 60 minutes each
periods = 7 * 24 * 60
tidx = pd.date_range('2016-07-01', periods=periods, freq='T')
#                     ^                                   ^
#                     |                                   |
#                 Start Date        Frequency Code for Minute
# This should get me 7 Days worth of minutes in a datetimeindex

# Generate random data with numpy.  We'll seed the random
# number generator so that others can see the same results.
# Otherwise, you don't have to seed it.
np.random.seed([3,1415])

# This will pick a number of normally distributed random numbers
# where the number is specified by periods
data = np.random.randn(periods)

ts = pd.Series(data=data, index=tidx, name='HelloTimeSeries')

ts.describe()

count    10080.000000
mean        -0.008853
std          0.995411
min         -3.936794
25%         -0.683442
50%          0.002640
75%          0.654986
max          3.906053
Name: HelloTimeSeries, dtype: float64
```

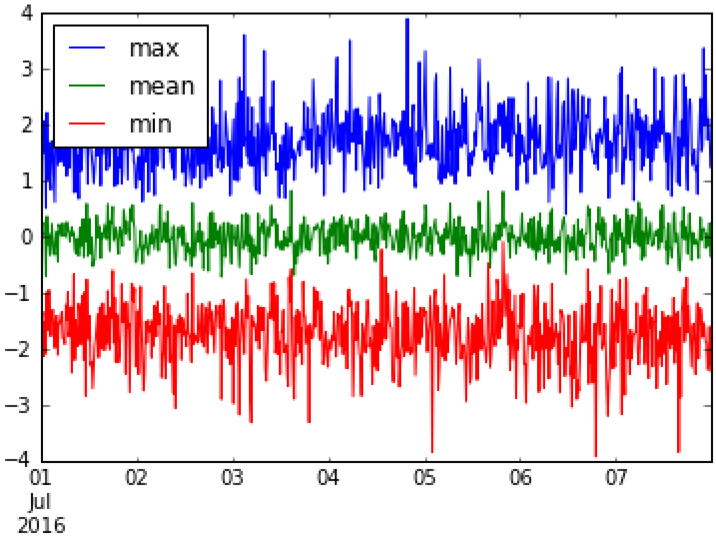
Let's take this 7 days of per minute data and down sample to every 15 minutes. All frequency codes can be found here.

```
# resample says to group by every 15 minutes.  But now we need
# to specify what to do within those 15 minute chunks.

# We could take the last value.
ts.resample('15T').last()
```

Or any other thing we can do to a `groupby` object, documentation.

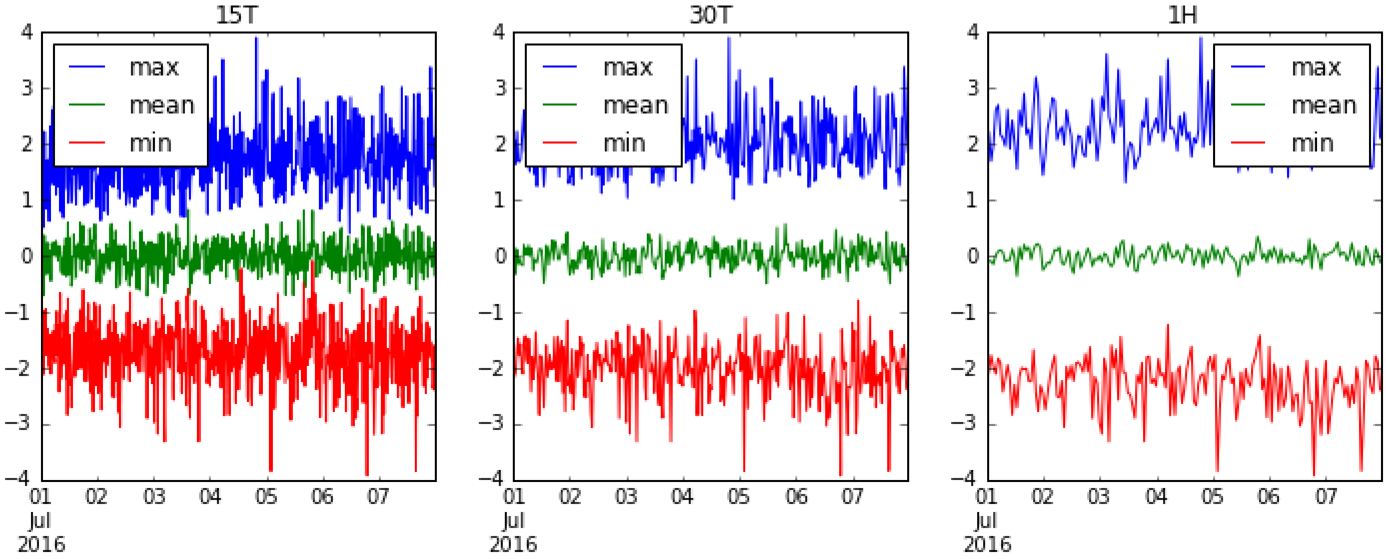
We can even aggregate several useful things. Let's plot the `min`, `mean`, and `max` of this `resample('15M')` data.

```
ts.resample('15T').agg(['min', 'mean', 'max']).plot()
```



Let's resample over `'15T'` (15 minutes), `'30T'` (half hour), and `'1H'` (1 hour) and see how our data gets smoother.

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
for i, freq in enumerate(['15T', '30T', '1H']):
    ts.resample(freq).agg(['max', 'mean', 'min']).plot(ax=axes[i], title=freq)
```



Read Grouping Time Series Data online: https://riptutorial.com/pandas/topic/4747/grouping-time-series-data

Chapter 17: Holiday Calendars

Examples

Create a custom calendar

Here is how to create a custom calendar. The example given is a french calendar -- so it provides many examples.

```
from pandas.tseries.holiday import AbstractHolidayCalendar, Holiday, EasterMonday, Easter
from pandas.tseries.offsets import Day, CustomBusinessDay

class FrBusinessCalendar(AbstractHolidayCalendar):
    """ Custom Holiday calendar for France based on
        https://en.wikipedia.org/wiki/Public_holidays_in_France
      - 1 January: New Year's Day
      - Moveable: Easter Monday (Monday after Easter Sunday)
      - 1 May: Labour Day
      - 8 May: Victory in Europe Day
      - Moveable Ascension Day (Thursday, 39 days after Easter Sunday)
      - 14 July: Bastille Day
      - 15 August: Assumption of Mary to Heaven
      - 1 November: All Saints' Day
      - 11 November: Armistice Day
      - 25 December: Christmas Day
    """
    rules = [
        Holiday('New Years Day', month=1, day=1),
        EasterMonday,
        Holiday('Labour Day', month=5, day=1),
        Holiday('Victory in Europe Day', month=5, day=8),
        Holiday('Ascension Day', month=1, day=1, offset=[Easter(), Day(39)]),
        Holiday('Bastille Day', month=7, day=14),
        Holiday('Assumption of Mary to Heaven', month=8, day=15),
        Holiday('All Saints Day', month=11, day=1),
        Holiday('Armistice Day', month=11, day=11),
        Holiday('Christmas Day', month=12, day=25)
    ]
```

Use a custom calendar

Here is how to use the custom calendar.

Get the holidays between two dates

```
import pandas as pd
from datetime import date

# Creating some boundaries
year = 2016
start = date(year, 1, 1)
```

```
end = start + pd.offsets.MonthEnd(12)

# Creating a custom calendar
cal = FrBusinessCalendar()
# Getting the holidays (off-days) between two dates
cal.holidays(start=start, end=end)

# DatetimeIndex(['2016-01-01', '2016-03-28', '2016-05-01', '2016-05-05',
#                 '2016-05-08', '2016-07-14', '2016-08-15', '2016-11-01',
#                 '2016-11-11', '2016-12-25'],
#                dtype='datetime64[ns]', freq=None)
```

Count the number of working days between two dates

It is sometimes useful to get the number of working days by month whatever the year in the future or in the past. Here is how to do that with a custom calendar.

```
from pandas.tseries.offsets import CDay

# Creating a series of dates between the boundaries
# by using the custom calendar
se = pd.bdate_range(start=start,
                    end=end,
                    freq=CDay(calendar=cal)).to_series()
# Counting the number of working days by month
se.groupby(se.dt.month).count().head()

# 1    20
# 2    21
# 3    22
# 4    21
# 5    21
```

Read Holiday Calendars online: https://riptutorial.com/pandas/topic/7976/holiday-calendars

Chapter 18: Indexing and selecting data

Examples

Select column by label

```
# Create a sample DF
df = pd.DataFrame(np.random.randn(5, 3), columns=list('ABC'))

# Show DF
df
          A         B         C
0 -0.467542  0.469146 -0.861848
1 -0.823205 -0.167087 -0.759942
2 -1.508202  1.361894 -0.166701
3  0.394143 -0.287349 -0.978102
4 -0.160431  1.054736 -0.785250

# Select column using a single label, 'A'
df['A']
0   -0.467542
1   -0.823205
2   -1.508202
3    0.394143
4   -0.160431

# Select multiple columns using an array of labels, ['A', 'C']
df[['A', 'C']]
          A         C
0 -0.467542 -0.861848
1 -0.823205 -0.759942
2 -1.508202 -0.166701
3  0.394143 -0.978102
4 -0.160431 -0.785250
```

Additional details at: http://pandas.pydata.org/pandas-docs/version/0.18.0/indexing.html#selection-by-label

Select by position

The `iloc` (short for *integer location*) method allows to select the rows of a dataframe based on their position index. This way one can slice dataframes just like one does with Python's list slicing.

```
df = pd.DataFrame([[11, 22], [33, 44], [55, 66]], index=list("abc"))

df
# Out:
#     0   1
# a  11  22
# b  33  44
# c  55  66

df.iloc[0]  # the 0th index (row)
```

```
# Out:
# 0     11
# 1     22
# Name: a, dtype: int64

df.iloc[1]  # the 1st index (row)
# Out:
# 0     33
# 1     44
# Name: b, dtype: int64

df.iloc[:2] # the first 2 rows
#     0   1
# a  11  22
# b  33  44

df[::-1]    # reverse order of rows
#     0   1
# c  55  66
# b  33  44
# a  11  22
```

Row location can be combined with column location

```
df.iloc[:, 1]  # the 1st column
# Out[15]:
# a    22
# b    44
# c    66
# Name: 1, dtype: int64
```

See also: Selection by Position

Slicing with labels

When using labels, both the start and the stop are included in the results.

```
import pandas as pd
import numpy as np
np.random.seed(5)
df = pd.DataFrame(np.random.randint(100, size=(5, 5)), columns = list("ABCDE"),
                  index = ["R" + str(i) for i in range(5)])

# Out:
#      A   B   C   D   E
# R0  99  78  61  16  73
# R1   8  62  27  30  80
# R2   7  76  15  53  80
# R3  27  44  77  75  65
# R4  47  30  84  86  18
```

Rows `R0` to `R2`:

```
df.loc['R0':'R2']
# Out:
#      A   B   C   D   E
```

```
# R0   9  41  62   1  82
# R1  16  78   5  58   0
# R2  80   4  36  51  27
```

Notice how `loc` differs from `iloc` because `iloc` excludes the end index

```
df.loc['R0':'R2'] # rows labelled R0, R1, R2
# Out:
#      A   B   C   D   E
# R0   9  41  62   1  82
# R1  16  78   5  58   0
# R2  80   4  36  51  27


# df.iloc[0:2] # rows indexed by 0, 1
#      A   B   C   D   E
# R0  99  78  61  16  73
# R1   8  62  27  30  80
```

Columns `C` to `E`:

```
df.loc[:, 'C':'E']
# Out:
#      C   D   E
# R0  62   1  82
# R1   5  58   0
# R2  36  51  27
# R3  68  38  83
# R4   7  30  62
```

Mixed position and label based selection

DataFrame:

```
import pandas as pd
import numpy as np
np.random.seed(5)
df = pd.DataFrame(np.random.randint(100, size=(5, 5)), columns = list("ABCDE"),
                  index = ["R" + str(i) for i in range(5)])


df
Out[12]:
     A   B   C   D   E
R0  99  78  61  16  73
R1   8  62  27  30  80
R2   7  76  15  53  80
R3  27  44  77  75  65
R4  47  30  84  86  18
```

Select rows by position, and columns by label:

```
df.ix[1:3, 'C':'E']
Out[19]:
     C   D   E
```

```
R1   5  58   0
R2  36  51  27
```

If the index is integer, `.ix` will use labels rather than positions:

```
df.index = np.arange(5, 10)

df
Out[22]:
    A   B   C   D   E
5   9  41  62   1  82
6  16  78   5  58   0
7  80   4  36  51  27
8  31   2  68  38  83
9  19  18   7  30  62

#same call returns an empty DataFrame because now the index is integer
df.ix[1:3, 'C':'E']
Out[24]:
Empty DataFrame
Columns: [C, D, E]
Index: []
```

Boolean indexing

One can select rows and columns of a dataframe using boolean arrays.

```
import pandas as pd
import numpy as np
np.random.seed(5)
df = pd.DataFrame(np.random.randint(100, size=(5, 5)), columns = list("ABCDE"),
                  index = ["R" + str(i) for i in range(5)])
print (df)
#      A   B   C   D   E
# R0  99  78  61  16  73
# R1   8  62  27  30  80
# R2   7  76  15  53  80
# R3  27  44  77  75  65
# R4  47  30  84  86  18
```

```
mask = df['A'] > 10
print (mask)
# R0     True
# R1    False
# R2    False
# R3     True
# R4     True
# Name: A, dtype: bool

print (df[mask])
#      A   B   C   D   E
# R0  99  78  61  16  73
# R3  27  44  77  75  65
# R4  47  30  84  86  18

print (df.ix[mask, 'C'])
# R0    61
```

```
# R3    77
# R4    84
# Name: C, dtype: int32

print(df.ix[mask, ['C', 'D']])
#      C   D
# R0  61  16
# R3  77  75
# R4  84  86
```

More in pandas documentation.

Filtering columns (selecting "interesting", dropping unneeded, using RegEx, etc.)

generate sample DF

```
In [39]: df = pd.DataFrame(np.random.randint(0, 10, size=(5, 6)),
columns=['a10','a20','a25','b','c','d'])

In [40]: df
Out[40]:
   a10  a20  a25  b  c  d
0    2    3    7  5  4  7
1    3    1    5  7  2  6
2    7    4    9  0  8  7
3    5    8    8  9  6  8
4    8    1    0  4  4  9
```

show columns containing letter 'a'

```
In [41]: df.filter(like='a')
Out[41]:
   a10  a20  a25
0    2    3    7
1    3    1    5
2    7    4    9
3    5    8    8
4    8    1    0
```

show columns using RegEx filter `(b|c|d)` - `b` or `c` or `d`:

```
In [42]: df.filter(regex='(b|c|d)')
Out[42]:
   b  c  d
0  5  4  7
1  7  2  6
```

```
2   0   8   7
3   9   6   8
4   4   4   9
```

show all columns except those beginning with `a` (in other word remove / drop all columns satisfying given RegEx)

```
In [43]: df.ix[:, ~df.columns.str.contains('^a')]
Out[43]:
   b  c  d
0  5  4  7
1  7  2  6
2  0  8  7
3  9  6  8
4  4  4  9
```

Filtering / selecting rows using `.query()` method

```
import pandas as pd
```

generate random DF

```
df = pd.DataFrame(np.random.randint(0,10,size=(10, 3)), columns=list('ABC'))

In [16]: print(df)
   A  B  C
0  4  1  4
1  0  2  0
2  7  8  8
3  2  1  9
4  7  3  8
5  4  0  7
6  1  5  5
7  6  7  8
8  6  7  3
9  6  4  5
```

select rows where values in column `A > 2` and values in column `B < 5`

```
In [18]: df.query('A > 2 and B < 5')
Out[18]:
   A  B  C
0  4  1  4
4  7  3  8
5  4  0  7
```

```
9   6   4   5
```

using `.query()` method with variables for filtering

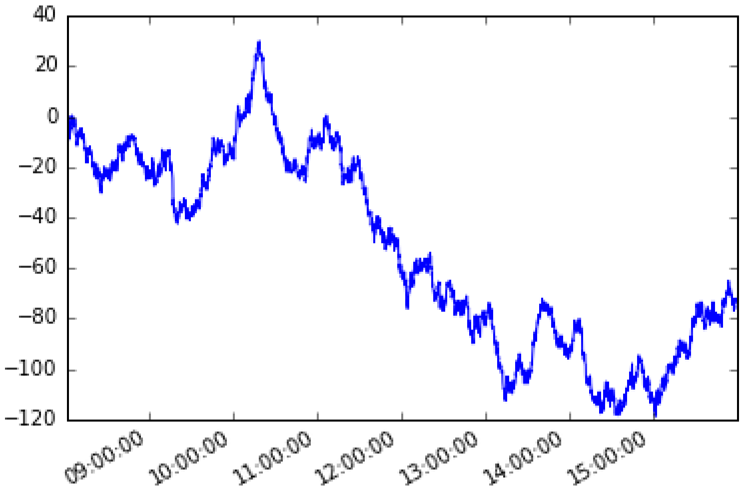
```
In [23]: B_filter = [1,7]

In [24]: df.query('B == @B_filter')
Out[24]:
   A  B  C
0  4  1  4
3  2  1  9
7  6  7  8
8  6  7  3

In [25]: df.query('@B_filter in B')
Out[25]:
   A  B  C
0  4  1  4
```

Path Dependent Slicing

It may become necessary to traverse the elements of a series or the rows of a dataframe in a way that the next element or next row is dependent on the previously selected element or row. This is called path dependency.

Consider the following time series `s` with irregular frequency.

```
#starting python community conventions
import numpy    as np
import pandas   as pd

# n is number of observations
n = 5000

day = pd.to_datetime(['2013-02-06'])
# irregular seconds spanning 28800 seconds (8 hours)
seconds = np.random.rand(n) * 28800 * pd.Timedelta(1, 's')
# start at 8 am
start = pd.offsets.Hour(8)
# irregular timeseries
tidx = day + start + seconds
tidx = tidx.sort_values()

s = pd.Series(np.random.randn(n), tidx, name='A').cumsum()
s.plot();
```



Let's assume a path dependent condition. Starting with the first member of the series, I want to grab each subsequent element such that the absolute difference between that element and the current element is greater than or equal to `x`.

We'll solve this problem using python generators.

Generator function

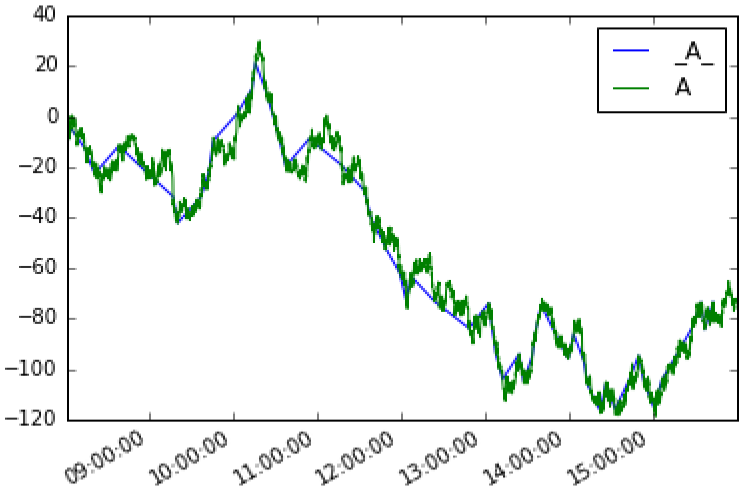
```
def mover(s, move_size=10):
    """Given a reference, find next value with
    an absolute difference >= move_size"""
    ref = None
    for i, v in s.iteritems():
        if ref is None or (abs(ref - v) >= move_size):
            yield i, v
            ref = v
```

Then we can define a new series `moves` like so

```
moves = pd.Series({i:v for i, v in mover(s, move_size=10)},
                  name='_{}_'.format(s.name))
```

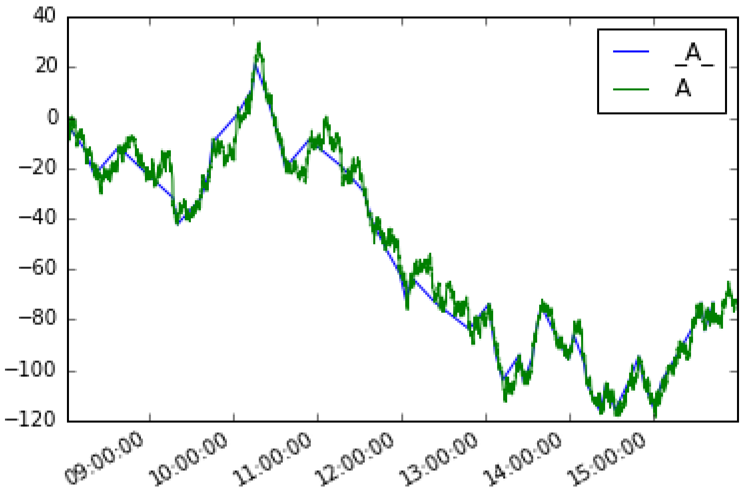
Plotting them both

```
moves.plot(legend=True)
s.plot(legend=True)
```



The analog for dataframes would be:

```
def mover_df(df, col, move_size=2):
    ref = None
    for i, row in df.iterrows():
        if ref is None or (abs(ref - row.loc[col]) >= move_size):
            yield row
            ref = row.loc[col]

df = s.to_frame()
moves_df = pd.concat(mover_df(df, 'A', 10), axis=1).T

moves_df.A.plot(label='_A_', legend=True)
df.A.plot(legend=True)
```



Get the first/last n rows of a dataframe

To view the first or last few records of a dataframe, you can use the methods `head` and `tail`

To return the first n rows use `DataFrame.head([n])`

```
df.head(n)
```

To return the last n rows use `DataFrame.tail([n])`

```
df.tail(n)
```

Without the argument n, these functions return 5 rows.

Note that the slice notation for `head`/`tail` would be:

```
df[:10]  # same as df.head(10)
df[-10:] # same as df.tail(10)
```

Select distinct rows across dataframe

Let

```
df = pd.DataFrame({'col_1':['A','B','A','B','C'], 'col_2':[3,4,3,5,6]})
df
# Output:
#   col_1  col_2
# 0     A      3
# 1     B      4
# 2     A      3
# 3     B      5
# 4     C      6
```

To get the distinct values in `col_1` you can use `Series.unique()`

```
df['col_1'].unique()
# Output:
# array(['A', 'B', 'C'], dtype=object)
```

But Series.unique() works only for a single column.

To simulate the *select unique col_1, col_2* of SQL you can use `DataFrame.drop_duplicates()`:

```
df.drop_duplicates()
#   col_1  col_2
# 0     A      3
# 1     B      4
# 3     B      5
# 4     C      6
```

This will get you all the unique rows in the dataframe. So if

```
df = pd.DataFrame({'col_1':['A','B','A','B','C'], 'col_2':[3,4,3,5,6],
'col_3':[0,0.1,0.2,0.3,0.4]})
df
# Output:
#   col_1  col_2  col_3
# 0     A      3    0.0
# 1     B      4    0.1
# 2     A      3    0.2
```

```
# 3      B       5     0.3
# 4      C       6     0.4

df.drop_duplicates()
#   col_1  col_2  col_3
# 0     A      3    0.0
# 1     B      4    0.1
# 2     A      3    0.2
# 3     B      5    0.3
# 4     C      6    0.4
```

To specify the columns to consider when selecting unique records, pass them as arguments

```
df = pd.DataFrame({'col_1':['A','B','A','B','C'], 'col_2':[3,4,3,5,6],
'col_3':[0,0.1,0.2,0.3,0.4]})
df.drop_duplicates(['col_1','col_2'])
# Output:
#   col_1  col_2  col_3
# 0     A      3    0.0
# 1     B      4    0.1
# 3     B      5    0.3
# 4     C      6    0.4

# skip last column
# df.drop_duplicates(['col_1','col_2'])[['col_1','col_2']]
#   col_1  col_2
# 0     A      3
# 1     B      4
# 3     B      5
# 4     C      6
```

Source: How to “select distinct” across multiple data frame columns in pandas?.

Filter out rows with missing data (NaN, None, NaT)

If you have a dataframe with missing data (`NaN`, `pd.NaT`, `None`) you can filter out incomplete rows

```
df = pd.DataFrame([[0,1,2,3],
                  [None,5,None,pd.NaT],
                  [8,None,10,None],
                  [11,12,13,pd.NaT]],columns=list('ABCD'))
df
# Output:
#     A   B   C     D
# 0   0   1   2     3
# 1 NaN   5 NaN   NaT
# 2   8 NaN  10  None
# 3  11  12  13   NaT
```

`DataFrame.dropna` drops all rows containing at least one field with missing data

```
df.dropna()
# Output:
#    A  B  C  D
# 0  0  1  2  3
```

To just drop the rows that are missing data at specified columns use `subset`

```
df.dropna(subset=['C'])
# Output:
#     A   B   C     D
# 0   0   1   2     3
# 2   8 NaN  10  None
# 3  11  12  13   NaT
```

Use the option `inplace = True` for in-place replacement with the filtered frame.

Read Indexing and selecting data online: https://riptutorial.com/pandas/topic/1751/indexing-and-selecting-data

Chapter 19: IO for Google BigQuery

Examples

Reading data from BigQuery with user account credentials

```
In [1]: import pandas as pd
```

In order to run a query in BigQuery you need to have your own BigQuery project. We can request some public sample data:

```
In [2]: data = pd.read_gbq('''SELECT title, id, num_characters
   ...:                        FROM [publicdata:samples.wikipedia]
   ...:                        LIMIT 5'''
   ...:                     , project_id='<your-project-id>')
```

This will print out:

```
Your browser has been opened to visit:

    https://accounts.google.com/o/oauth2/v2/auth...[looong url cutted]

If your browser is on a different machine then exit and re-run this
application with the command-line parameter

  --noauth_local_webserver
```

If your are operating from local machine than browser will pop-up. After granting privileges pandas will continue with output:

```
Authentication successful.
Requesting query... ok.
Query running...
Query done.
Processed: 13.8 Gb

Retrieving results...
Got 5 rows.

Total time taken 1.5 s.
Finished at 2016-08-23 11:26:03.
```

Result:

```
In [3]: data
Out[3]:
               title       id  num_characters
0       Fusidic acid   935328            1112
1     Clark Air Base   426241            8257
2  Watergate scandal    52382           25790
3               2005    35984           75813
```

```
4               .BLP  2664340            1659
```

As a side effect pandas will create json file `bigquery_credentials.dat` which will allow you to run further queries without need to grant privileges any more:

```
In [9]: pd.read_gbq('SELECT count(1) cnt FROM [publicdata:samples.wikipedia]'
                   , project_id='<your-project-id>')
Requesting query... ok.
[rest of output cutted]

Out[9]:
         cnt
0  313797035
```

Reading data from BigQuery with service account credentials

If you have created service account and have private key json file for it, you can use this file to authenticate with pandas

```
In [5]: pd.read_gbq('''SELECT corpus, sum(word_count) words
                       FROM [bigquery-public-data:samples.shakespeare]
                       GROUP BY corpus
                       ORDER BY words desc
                       LIMIT 5'''
                   , project_id='<your-project-id>'
                   , private_key='<private key json contents or file path>')
Requesting query... ok.
[rest of output cutted]

Out[5]:
           corpus  words
0          hamlet  32446
1  kingrichardiii  31868
2      coriolanus  29535
3       cymbeline  29231
4    2kinghenryiv  28241
```

Read IO for Google BigQuery online: https://riptutorial.com/pandas/topic/5610/io-for-google-bigquery

Chapter 20: JSON

Examples

Read JSON

can either pass string of the json, or a filepath to a file with valid json

```
In [99]: pd.read_json('[{"A": 1, "B": 2}, {"A": 3, "B": 4}]')
Out[99]:
   A  B
0  1  2
1  3  4
```

Alternatively to conserve memory:

```
with open('test.json') as f:
    data = pd.DataFrame(json.loads(line) for line in f)
```

Dataframe into nested JSON as in flare.js files used in D3.js

```
def to_flare_json(df, filename):
    """Convert dataframe into nested JSON as in flare files used for D3.js"""
    flare = dict()
    d = {"name":"flare", "children": []}

    for index, row in df.iterrows():
        parent = row[0]
        child = row[1]
        child_size = row[2]

        # Make a list of keys
        key_list = []
        for item in d['children']:
            key_list.append(item['name'])

        #if 'parent' is NOT a key in flare.JSON, append it
        if not parent in key_list:
            d['children'].append({"name": parent, "children":[{"value": child_size, "name":
child}]})
        # if parent IS a key in flare.json, add a new child to it
        else:
            d['children'][key_list.index(parent)]['children'].append({"value": child_size,
"name": child})
    flare = d
    # export the final result to a json file
    with open(filename +'.json', 'w') as outfile:
        json.dump(flare, outfile, indent=4)
```

```
    return ("Done")
```

Read JSON from file

Content of file.json (one JSON object per line):

```
{"A": 1, "B": 2}
{"A": 3, "B": 4}
```

How to read directly from a local file:

```
pd.read_json('file.json', lines=True)
# Output:
#    A  B
# 0  1  2
# 1  3  4
```

Read JSON online: https://riptutorial.com/pandas/topic/4752/json

Chapter 21: Making Pandas Play Nice With Native Python Datatypes

Examples

Moving Data Out of Pandas Into Native Python and Numpy Data Structures

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [1.0, 2.0, 3.0], 'C': ['a', 'b', 'c'],
                           'D': [True, False, True]})

In [2]: df
Out[2]:
   A    B  C      D
0  1  1.0  a   True
1  2  2.0  b  False
2  3  3.0  c   True
```

Getting a python list from a series:

```
In [3]: df['A'].tolist()
Out[3]: [1, 2, 3]
```

DataFrames do not have a `tolist()` method. Trying it results in an AttributeError:

```
In [4]: df.tolist()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-fc6763af1ff7> in <module>()
----> 1 df.tolist()

//anaconda/lib/python2.7/site-packages/pandas/core/generic.pyc in __getattr__(self, name)
   2742             if name in self._info_axis:
   2743                 return self[name]
-> 2744             return object.__getattribute__(self, name)
   2745
   2746     def __setattr__(self, name, value):

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

Getting a numpy array from a series:

```
In [5]: df['B'].values
Out[5]: array([ 1.,  2.,  3.])
```

You can also get an array of the columns as individual numpy arrays from an entire dataframe:

```
In [6]: df.values
Out[6]:
array([[1, 1.0, 'a', True],
       [2, 2.0, 'b', False],
```

```
       [3, 3.0, 'c', True]], dtype=object)
```

Getting a dictionary from a series (uses the index as the keys):

```
In [7]: df['C'].to_dict()
Out[7]: {0: 'a', 1: 'b', 2: 'c'}
```

You can also get the entire DataFrame back as a dictionary:

```
In [8]: df.to_dict()
Out[8]:
{'A': {0: 1, 1: 2, 2: 3},
 'B': {0: 1.0, 1: 2.0, 2: 3.0},
 'C': {0: 'a', 1: 'b', 2: 'c'},
 'D': {0: True, 1: False, 2: True}}
```

The `to_dict` method has a few different parameters to adjust how the dictionaries are formatted. To get a list of dicts for each row:

```
In [9]: df.to_dict('records')
Out[9]:
[{'A': 1, 'B': 1.0, 'C': 'a', 'D': True},
 {'A': 2, 'B': 2.0, 'C': 'b', 'D': False},
 {'A': 3, 'B': 3.0, 'C': 'c', 'D': True}]
```

See the documentation for the full list of options available to create dictionaries.

Read Making Pandas Play Nice With Native Python Datatypes online: https://riptutorial.com/pandas/topic/8008/making-pandas-play-nice-with-native-python-datatypes

Chapter 22: Map Values

Remarks

it should be mentioned that if the key value does not exist then this will raise `KeyError`, in those situations it maybe better to use `merge` or `get` which allows you to specify a default value if the key doesn't exist

Examples

Map from Dictionary

Starting from a dataframe `df`:

```
  U   L
111  en
112  en
112  es
113  es
113  ja
113  zh
114  es
```

Imagine you want to add a new column called `S` taking values from the following dictionary:

```
d = {112: 'en', 113: 'es', 114: 'es', 111: 'en'}
```

You can use `map` to perform a lookup on keys returning the corresponding values as a new column:

```
df['S'] = df['U'].map(d)
```

that returns:

```
  U   L   S
111  en  en
112  en  en
112  es  en
113  es  es
113  ja  es
113  zh  es
114  es  es
```

Read Map Values online: https://riptutorial.com/pandas/topic/3928/map-values

Chapter 23: Merge, join, and concatenate

Syntax

- DataFrame.**merge**(*right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False*)
- Merge DataFrame objects by performing a database-style join operation by columns or indexes.
- If joining columns on columns, the DataFrame indexes will be ignored. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters

Parameters	Explanation
right	DataFrame
how	{'left', 'right', 'outer', 'inner'}, default 'inner'
left_on	label or list, or array-like. Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns
right_on	label or list, or array-like. Field names to join on in right DataFrame or vector/list of vectors per left_on docs
left_index	boolean, default False. Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels
right_index	boolean, default False. Use the index from the right DataFrame as the join key. Same caveats as left_index
sort	boolean, default Fals. Sort the join keys lexicographically in the result DataFrame
suffixes	2-length sequence (tuple, list, ...). Suffix to apply to overlapping column names in the left and right side, respectively
copy	boolean, default True. If False, do not copy data unnecessarily
indicator	boolean or string, default False. If True, adds a column to output DataFrame called "_merge" with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and

Parameters	Explanation
	column will be named value of string. Information column is Categorical-type and takes on a value of “left_only” for observations whose merge key only appears in ‘left’ DataFrame, “right_only” for observations whose merge key only appears in ‘right’ DataFrame, and “both” if the observation’s merge key is found in both.

Examples

Merge

For instance, two tables are given,

T1

```
id    x        y
8    42        1.9
9    30        1.9
```

T2

```
id    signal
8    55
8    56
8    59
9    57
9    58
9    60
```

The goal is to get the new table T3:

```
id    x        y        s1        s2        s3
8    42        1.9        55        56        58
9    30        1.9        57        58        60
```

Which is to create columns `s1`, `s2` and `s3`, each corresponding to a row (the number of rows per `id` is always fixed and equal to 3)

By applying `join` (which takes an optional on argument which may be a column or multiple column names, which specifies that the passed DataFrame is to be aligned on that column in the DataFrame). So the solution can be as shown below:

df = df1.merge(df2.groupby('id')['signal'].apply(lambda x: x.reset_index(drop=True)).unstack().reset_index())

```
df
Out[63]:
   id   x    y   0   1   2
0   8  42  1.9  55  56  59
```

```
1   9  30  1.9  57  58  60
```

If I separate them:

```
df2t = df2.groupby('id')['signal'].apply(lambda x:
x.reset_index(drop=True)).unstack().reset_index()

df2t
Out[59]:
   id   0   1   2
0   8  55  56  59
1   9  57  58  60

df = df1.merge(df2t)

df
Out[61]:
   id   x    y   0   1   2
0   8  42  1.9  55  56  59
1   9  30  1.9  57  58  60
```

Merging two DataFrames

```
In [1]: df1 = pd.DataFrame({'x': [1, 2, 3], 'y': ['a', 'b', 'c']})

In [2]: df2 = pd.DataFrame({'y': ['b', 'c', 'd'], 'z': [4, 5, 6]})

In [3]: df1
Out[3]:
   x  y
0  1  a
1  2  b
2  3  c

In [4]: df2
Out[4]:
   y  z
0  b  4
1  c  5
2  d  6
```

Inner join:

Uses the intersection of keys from two DataFrames.

```
In [5]: df1.merge(df2) # by default, it does an inner join on the common column(s)
Out[5]:
   x  y  z
0  2  b  4
1  3  c  5
```

Alternatively specify intersection of keys from two Dataframes.

```
In [5]: merged_inner = pd.merge(left=df1, right=df2, left_on='y', right_on='y')
Out[5]:
   x  y  z
0  2  b  4
1  3  c  5
```

Outer join:

Uses the union of the keys from two DataFrames.

```
In [6]: df1.merge(df2, how='outer')
Out[6]:
     x  y    z
0  1.0  a  NaN
1  2.0  b  4.0
2  3.0  c  5.0
3  NaN  d  6.0
```

Left join:

Uses only keys from left DataFrame.

```
In [7]: df1.merge(df2, how='left')
Out[7]:
   x  y    z
0  1  a  NaN
1  2  b  4.0
2  3  c  5.0
```

Right Join

Uses only keys from right DataFrame.

```
In [8]: df1.merge(df2, how='right')
Out[8]:
     x  y  z
0  2.0  b  4
1  3.0  c  5
2  NaN  d  6
```

Merging / concatenating / joining multiple data frames (horizontally and vertically)

generate sample data frames:

```
In [57]: df3 = pd.DataFrame({'col1':[211,212,213], 'col2': [221,222,223]})
```

```
In [58]: df1 = pd.DataFrame({'col1':[11,12,13], 'col2': [21,22,23]})

In [59]: df2 = pd.DataFrame({'col1':[111,112,113], 'col2': [121,122,123]})

In [60]: df3 = pd.DataFrame({'col1':[211,212,213], 'col2': [221,222,223]})

In [61]: df1
Out[61]:
   col1  col2
0    11    21
1    12    22
2    13    23

In [62]: df2
Out[62]:
   col1  col2
0   111   121
1   112   122
2   113   123

In [63]: df3
Out[63]:
   col1  col2
0   211   221
1   212   222
2   213   223
```

merge / join / concatenate data frames [df1, df2, df3] vertically - add rows

```
In [64]: pd.concat([df1,df2,df3], ignore_index=True)
Out[64]:
   col1  col2
0    11    21
1    12    22
2    13    23
3   111   121
4   112   122
5   113   123
6   211   221
7   212   222
8   213   223
```

merge / join / concatenate data frames horizontally (aligning by index):

```
In [65]: pd.concat([df1,df2,df3], axis=1)
Out[65]:
   col1  col2  col1  col2  col1  col2
0    11    21   111   121   211   221
1    12    22   112   122   212   222
2    13    23   113   123   213   223
```

Merge, Join and Concat

Merging key names are same

```
pd.merge(df1, df2, on='key')
```

Merging key names are different

```
pd.merge(df1, df2, left_on='l_key', right_on='r_key')
```

Different types of joining

```
pd.merge(df1, df2, on='key', how='left')
```

Merging on multiple keys

```
pd.merge(df1, df2, on=['key1', 'key2'])
```

Treatment of overlapping columns

```
pd.merge(df1, df2, on='key', suffixes=('_left', '_right'))
```

Using row index instead of merging keys

```
pd.merge(df1, df2, right_index=True, left_index=True)
```

Avoid use of `.join` syntax as it gives exception for overlapping columns

Merging on left dataframe index and right dataframe column

```
pd.merge(df1, df2, right_index=True, left_on='l_key')
```

Concate dataframes

Glued vertically

```
pd.concat([df1, df2, df3], axis=0)
```

Glued horizontally

```
pd.concat([df1, df2, df3], axis=1)
```

What is the difference between join and merge

Consider the dataframes `left` and `right`

```
left = pd.DataFrame([['a', 1], ['b', 2]], list('XY'), list('AB'))
left

   A  B
X  a  1
Y  b  2
```

```
right = pd.DataFrame([['a', 3], ['b', 4]], list('XY'), list('AC'))
right

   A  C
X  a  3
Y  b  4
```

join

Think of `join` as wanting to combine to dataframes based on their respective indexes. If there are overlapping columns, `join` will want you to add a suffix to the overlapping column name from left dataframe. Our two dataframes do have an overlapping column name `A`.

```
left.join(right, lsuffix='_')

  A_  B  A  C
X  a  1  a  3
Y  b  2  b  4
```

Notice the index is preserved and we have 4 columns. 2 columns from `left` and 2 from `right`.

If the indexes did not align

```
left.join(right.reset_index(), lsuffix='_', how='outer')

    A_    B index    A    C
0  NaN  NaN     X    a  3.0
1  NaN  NaN     Y    b  4.0
X    a  1.0   NaN  NaN  NaN
Y    b  2.0   NaN  NaN  NaN
```

I used an outer join to better illustrate the point. If the indexes do not align, the result will be the union of the indexes.

We can tell `join` to use a specific column in the left dataframe to use as the join key, but it will still use the index from the right.

```
left.reset_index().join(right, on='index', lsuffix='_')

  index A_  B  A  C
0     X  a  1  a  3
1     Y  b  2  b  4
```

merge

Think of `merge` as aligning on columns. By default `merge` will look for overlapping columns in which to merge on. `merge` gives better control over merge keys by allowing the user to specify a subset of the overlapping columns to use with parameter `on`, or to separately allow the specification of which columns on the left and which columns on the right to merge by.

`merge` will return a combined dataframe in which the index will be destroyed.

This simple example finds the overlapping column to be `'A'` and combines based on it.

```
left.merge(right)

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

Note the index is `[0, 1]` and no longer `['X', 'Y']`

You can explicitly specify that you are merging on the index with the `left_index` or `right_index` paramter

```
left.merge(right, left_index=True, right_index=True, suffixes=['_', ''])

  A_  B  A  C
X  a  1  a  3
Y  b  2  b  4
```

And this looks exactly like the `join` example above.

Read Merge, join, and concatenate online: https://riptutorial.com/pandas/topic/1966/merge--join--and-concatenate

Chapter 24: Meta: Documentation Guidelines

Remarks

This meta post is similar to the python version http://stackoverflow.com/documentation/python/394/meta-documentation-guidelines#t=201607240058406359521.

Please make edit suggestions, and comment on those (in lieu of proper comments), so we can flesh out/iterate on these suggestions :)

Examples

Showing code snippets and output

Two popular options are to use:

ipython notation:

```
In [11]: df = pd.DataFrame([[1, 2], [3, 4]])

In [12]: df
Out[12]:
   0  1
0  1  2
1  3  4
```

Alternatively (this is popular over in the python documentation) and more concisely:

```
df.columns  # Out: RangeIndex(start=0, stop=2, step=1)

df[0]
# Out:
# 0    1
# 1    3
# Name: 0, dtype: int64

for col in df:
    print(col)
# prints:
# 0
# 1
```

Generally, this is better for smaller examples.

Note: The distinction between output and printing. ipython makes this clear (the prints occur before the output is returned):

```
In [21]: [print(col) for col in df]
```

```
0
1
Out[21]: [None, None]
```

style

Use the pandas library as `pd`, this can be assumed (the import does not need to be in every example)

```
import pandas as pd
```

PEP8!

- 4 space indentation
- kwargs should use no spaces `f(a=1)`
- 80 character limit (the entire line fitting in the rendered code snippet should be strongly preferred)

Pandas version support

Most examples will work across multiple versions, if you are using a "new" feature you should mention when this was introduced.

Example: `sort_values`.

print statements

Most of the time printing should be avoided as it can be a distraction (Out should be preferred). That is:

```
a
# Out: 1
```

is always better than

```
print(a)
# prints: 1
```

Prefer supporting python 2 and 3:

```
print(x)    # yes! (works same in python 2 and 3)
print x     # no! (python 2 only)
print(x, y) # no! (works differently in python 2 and 3)
```

Read Meta: Documentation Guidelines online: https://riptutorial.com/pandas/topic/3253/meta--documentation-guidelines

Chapter 25: Missing Data

Remarks

Should we include the non-documented `ffill` and `bfill`?

Examples

Filling missing values

```
In [11]: df = pd.DataFrame([[1, 2, None, 3], [4, None, 5, 6],
                            [7, 8, 9, 10], [None, None, None, None]])

Out[11]:
     0    1    2     3
0  1.0  2.0  NaN   3.0
1  4.0  NaN  5.0   6.0
2  7.0  8.0  9.0  10.0
3  NaN  NaN  NaN   NaN
```

Fill missing values with a single value:

```
In [12]: df.fillna(0)
Out[12]:
     0    1    2     3
0  1.0  2.0  0.0   3.0
1  4.0  0.0  5.0   6.0
2  7.0  8.0  9.0  10.0
3  0.0  0.0  0.0   0.0
```

This returns a new DataFrame. If you want to change the original DataFrame, either use the `inplace` parameter (`df.fillna(0, inplace=True)`) or assign it back to original DataFrame (`df = df.fillna(0)`).

Fill missing values with the previous ones:

```
In [13]: df.fillna(method='pad')  # this is equivalent to both method='ffill' and .ffill()
Out[13]:
     0    1    2     3
0  1.0  2.0  NaN   3.0
1  4.0  2.0  5.0   6.0
2  7.0  8.0  9.0  10.0
3  7.0  8.0  9.0  10.0
```

Fill with the next ones:

```
In [14]: df.fillna(method='bfill')  # this is equivalent to .bfill()
Out[14]:
     0    1    2     3
0  1.0  2.0  5.0   3.0
1  4.0  8.0  5.0   6.0
2  7.0  8.0  9.0  10.0
3  NaN  NaN  NaN   NaN
```

Fill using another DataFrame:

```
In [15]: df2 = pd.DataFrame(np.arange(100, 116).reshape(4, 4))
         df2
Out[15]:
     0    1    2    3
0  100  101  102  103
1  104  105  106  107
2  108  109  110  111
3  112  113  114  115

In [16]: df.fillna(df2) #  takes the corresponding cells in df2 to fill df
Out[16]:
       0      1      2      3
0    1.0    2.0  102.0    3.0
1    4.0  105.0    5.0    6.0
2    7.0    8.0    9.0   10.0
3  112.0  113.0  114.0  115.0
```

Dropping missing values

When creating a DataFrame `None` (python's missing value) is converted to `NaN` (pandas' missing value):

```
In [11]: df = pd.DataFrame([[1, 2, None, 3], [4, None, 5, 6],
                            [7, 8, 9, 10], [None, None, None, None]])

Out[11]:
     0    1    2     3
0  1.0  2.0  NaN   3.0
1  4.0  NaN  5.0   6.0
2  7.0  8.0  9.0  10.0
3  NaN  NaN  NaN   NaN
```

Drop rows if at least one column has a missing value

```
In [12]: df.dropna()
Out[12]:
     0    1    2     3
2  7.0  8.0  9.0  10.0
```

This returns a new DataFrame. If you want to change the original DataFrame, either use the `inplace` parameter (`df.dropna(inplace=True)`) or assign it back to original DataFrame (`df = df.dropna()`).

Drop rows if all values in that row are missing

```
In [13]: df.dropna(how='all')
Out[13]:
     0    1    2     3
0  1.0  2.0  NaN   3.0
1  4.0  NaN  5.0   6.0
2  7.0  8.0  9.0  10.0
```

Drop *columns* that don't have at least 3 non-missing values

```
In [14]: df.dropna(axis=1, thresh=3)
Out[14]:
     0     3
0  1.0   3.0
1  4.0   6.0
2  7.0  10.0
3  NaN   NaN
```

Interpolation

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'A':[1,2,np.nan,3,np.nan],
                   'B':[1.2,7,3,0,8]})

df['C'] = df.A.interpolate()
df['D'] = df.A.interpolate(method='spline', order=1)

print (df)
     A    B    C         D
0  1.0  1.2  1.0  1.000000
1  2.0  7.0  2.0  2.000000
2  NaN  3.0  2.5  2.428571
3  3.0  0.0  3.0  3.000000
4  NaN  8.0  3.0  3.714286
```

Checking for missing values

In order to check whether a value is NaN, `isnull()` or `notnull()` functions can be used.

```
In [1]: import numpy as np
In [2]: import pandas as pd
In [3]: ser = pd.Series([1, 2, np.nan, 4])
In [4]: pd.isnull(ser)
Out[4]:
0    False
1    False
2     True
3    False
dtype: bool
```

Note that `np.nan == np.nan` returns False so you should avoid comparison against np.nan:

```
In [5]: ser == np.nan
Out[5]:
0    False
1    False
2    False
3    False
dtype: bool
```

Both functions are also defined as methods on Series and DataFrames.

```
In [6]: ser.isnull()
Out[6]:
0    False
1    False
2     True
3    False
dtype: bool
```

Testing on DataFrames:

```
In [7]: df = pd.DataFrame({'A': [1, np.nan, 3], 'B': [np.nan, 5, 6]})
In [8]: print(df)
Out[8]:
     A    B
0  1.0  NaN
1  NaN  5.0
2  3.0  6.0

In [9]: df.isnull()  # If the value is NaN, returns True.
Out[9]:
       A      B
0  False   True
1   True  False
2  False  False

In [10]: df.notnull()  # Opposite of .isnull(). If the value is not NaN, returns True.
Out[10]:
       A      B
0   True  False
1  False   True
2   True   True
```

Read Missing Data online: https://riptutorial.com/pandas/topic/1896/missing-data

Chapter 26: MultiIndex

Examples

Select from MultiIndex by Level

Given the following DataFrame:

```
In [11]: df = pd.DataFrame(np.random.randn(6, 3), columns=['A', 'B', 'C'])

In [12]: df.set_index(['A', 'B'], inplace=True)

In [13]: df
Out[13]:
                            C
A         B
 0.902764 -0.259656 -1.864541
-0.695893  0.308893  0.125199
 1.696989 -1.221131 -2.975839
-1.132069 -1.086189 -1.945467
 2.294835 -1.765507  1.567853
-1.788299  2.579029  0.792919
```

Get the values of `A`, by name:

```
In [14]: df.index.get_level_values('A')
Out[14]:
Float64Index([0.902764041011, -0.69589264969,  1.69698924476, -1.13206872067,
               2.29483481146,   -1.788298829],
             dtype='float64', name='A')
```

Or by number of level:

```
In [15]: df.index.get_level_values(level=0)
Out[15]:
Float64Index([0.902764041011, -0.69589264969,  1.69698924476, -1.13206872067,
               2.29483481146,   -1.788298829],
             dtype='float64', name='A')
```

And for a specific range:

```
In [16]: df.loc[(df.index.get_level_values('A') > 0.5) & (df.index.get_level_values('A') <
2.1)]
Out[16]:
                           C
A         B
0.902764 -0.259656 -1.864541
1.696989 -1.221131 -2.975839
```

Range can also include multiple columns:

```
In [17]: df.loc[(df.index.get_level_values('A') > 0.5) & (df.index.get_level_values('B') < 0)]
Out[17]:
                           C
A        B
0.902764 -0.259656 -1.864541
1.696989 -1.221131 -2.975839
2.294835 -1.765507  1.567853
```

To extract a specific value you can use xs (cross-section):

```
In [18]: df.xs(key=0.9027639999999999)
Out[18]:
                  C
B
-0.259656 -1.864541

In [19]: df.xs(key=0.9027639999999999, drop_level=False)
Out[19]:
                           C
A        B
0.902764 -0.259656 -1.864541
```

Iterate over DataFrame with MultiIndex

Given the following DataFrame:

```
In [11]: df = pd.DataFrame({'a':[1,1,1,2,2,3],'b':[4,4,5,5,6,7,],'c':[10,11,12,13,14,15]})

In [12]: df.set_index(['a','b'], inplace=True)

In [13]: df
Out[13]:
      c
a b
1 4  10
  4  11
  5  12
2 5  13
  6  14
3 7  15
```

You can iterate by any level of the MultiIndex. For example, `level=0` (you can also select the level by name e.g. `level='a'`):

```
In[21]: for idx, data in df.groupby(level=0):
            print('---')
            print(data)
---
      c
a b
1 4  10
  4  11
  5  12
---
      c
a b
```

```
2 5  13
  6  14
---
      c
a b
3 7  15
```

You can also select the levels by name e.g. `level='b':

```
In[22]: for idx, data in df.groupby(level='b'):
            print('---')
            print(data)
---
      c
a b
1 4  10
  4  11
---
      c
a b
1 5  12
2 5  13
---
      c
a b
2 6  14
---
      c
a b
3 7  15
```

Setting and sorting a MultiIndex

This example shows how to use column data to set a `MultiIndex` in a `pandas.DataFrame`.

```
In [1]: df = pd.DataFrame([['one', 'A', 100], ['two', 'A', 101], ['three', 'A', 102],
   ...:                    ['one', 'B', 103], ['two', 'B', 104], ['three', 'B', 105]],
   ...:                   columns=['c1', 'c2', 'c3'])


In [2]: df
Out[2]:
      c1 c2   c3
0    one  A  100
1    two  A  101
2  three  A  102
3    one  B  103
4    two  B  104
5  three  B  105



In [3]: df.set_index(['c1', 'c2'])
Out[3]:
           c3
c1    c2
one   A   100
two   A   101
```

```
three A   102
one   B   103
two   B   104
three B   105
```

You can sort the index right after you set it:

```
In [4]: df.set_index(['c1', 'c2']).sort_index()
Out[4]:
           c3
c1    c2
one   A   100
      B   103
three A   102
      B   105
two   A   101
      B   104
```

Having a sorted index, will result in slightly more efficient lookups on the first level:

```
In [5]: df_01 = df.set_index(['c1', 'c2'])

In [6]: %timeit df_01.loc['one']
1000 loops, best of 3: 607 µs per loop


In [7]: df_02 = df.set_index(['c1', 'c2']).sort_index()

In [8]: %timeit df_02.loc['one']
1000 loops, best of 3: 413 µs per loop
```

After the index has been set, you can perform lookups for specific records or groups of records:

```
In [9]: df_indexed = df.set_index(['c1', 'c2']).sort_index()

In [10]: df_indexed.loc['one']
Out[10]:
     c3
c2
A   100
B   103


In [11]: df_indexed.loc['one', 'A']
Out[11]:
c3    100
Name: (one, A), dtype: int64


In [12]: df_indexed.xs((slice(None), 'A'))
Out[12]:
        c3
c1
one    100
three  102
two    101
```

How to change MultiIndex columns to standard columns

Given a DataFrame with MultiIndex columns

```
# build an example DataFrame
midx = pd.MultiIndex(levels=[['zero', 'one'], ['x','y']], labels=[[1,1,0,],[1,0,1,]])
df = pd.DataFrame(np.random.randn(2,3), columns=midx)

In [2]: df
Out[2]:
        one                zero
          y         x         y
0  0.785806 -0.679039  0.513451
1 -0.337862 -0.350690 -1.423253
```

If you want to change the columns to standard columns (not MultiIndex), just rename the columns.

```
df.columns = ['A','B','C']
In [3]: df
Out[3]:
          A         B         C
0  0.785806 -0.679039  0.513451
1 -0.337862 -0.350690 -1.423253
```

How to change standard columns to MultiIndex

Start with a standard DataFrame

```
df = pd.DataFrame(np.random.randn(2,3), columns=['a','b','c'])

In [91]: df
Out[91]:
          a         b         c
0 -0.911752 -1.405419 -0.978419
1  0.603888 -1.187064 -0.035883
```

Now to change to MultiIndex, create a `MultiIndex` object and assign it to `df.columns`.

```
midx = pd.MultiIndex(levels=[['zero', 'one'], ['x','y']], labels=[[1,1,0,],[1,0,1,]])
df.columns = midx

In [94]: df
Out[94]:
            one                zero
              y         x         y
    0 -0.911752 -1.405419 -0.978419
    1  0.603888 -1.187064 -0.035883
```

MultiIndex Columns

MultiIndex can also be used to create DataFrames with multilevel columns. Just use the `columns` keyword in the DataFrame command.

```
midx = pd.MultiIndex(levels=[['zero', 'one'], ['x','y']], labels=[[1,1,0,],[1,0,1,]])
df = pd.DataFrame(np.random.randn(6,4), columns=midx)

In [86]: df
Out[86]:
        one              zero
          y         x         y
0  0.625695  2.149377  0.006123
1 -1.392909  0.849853  0.005477
```

Displaying all elements in the index

To view all elements in the index change the print options that “sparsifies” the display of the MultiIndex.

```
pd.set_option('display.multi_sparse', False)
df.groupby(['A','B']).mean()
# Output:
#        C
# A B
# a 1  107
# a 2  102
# a 3  115
# b 5   92
# b 8   98
# c 2   87
# c 4  104
# c 9  123
```

Read MultiIndex online: https://riptutorial.com/pandas/topic/3840/multiindex

Chapter 27: Pandas Datareader

Remarks

The Pandas datareader is a sub package that allows one to create a dataframe from various internet datasources, currently including:

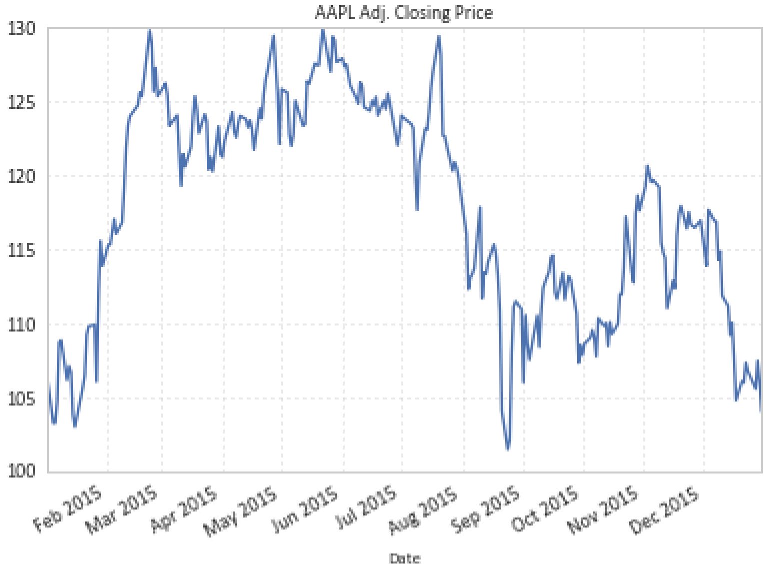
- Yahoo! Finance
- Google Finance
- St.Louis FED (FRED)
- Kenneth French's data library
- World Bank
- Google Analytics

For more information, see here.

Examples

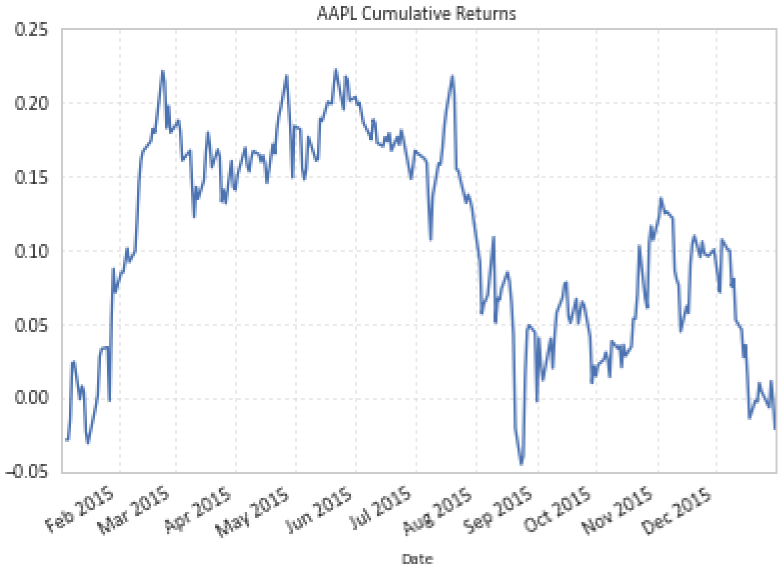
Datareader basic example (Yahoo Finance)

```
from pandas_datareader import data

# Only get the adjusted close.
aapl = data.DataReader("AAPL",
                       start='2015-1-1',
                       end='2015-12-31',
                       data_source='yahoo')['Adj Close']

>>> aapl.plot(title='AAPL Adj. Closing Price')
```




```
# Convert the adjusted closing prices to cumulative returns.
returns = aapl.pct_change()
```

```
>>> ((1 + returns).cumprod() - 1).plot(title='AAPL Cumulative Returns')
```



Reading financial data (for multiple tickers) into pandas panel - demo

```
from datetime import datetime
import pandas_datareader.data as wb

stocklist = ['AAPL','GOOG','FB','AMZN','COP']

start = datetime(2016,6,8)
end = datetime(2016,6,11)

p = wb.DataReader(stocklist, 'yahoo',start,end)
```

`p` - is a pandas panel, with which we can do funny things:

let's see what do we have in our panel

```
In [388]: p.axes
Out[388]:
[Index(['Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close'], dtype='object'),
 DatetimeIndex(['2016-06-08', '2016-06-09', '2016-06-10'], dtype='datetime64[ns]',
name='Date', freq='D'),
 Index(['AAPL', 'AMZN', 'COP', 'FB', 'GOOG'], dtype='object')]

In [389]: p.keys()
Out[389]: Index(['Open', 'High', 'Low', 'Close', 'Volume', 'Adj Close'], dtype='object')
```

selecting & slicing data

```
In [390]: p['Adj Close']
Out[390]:
                 AAPL        AMZN        COP          FB        GOOG
Date
2016-06-08  98.940002  726.640015  47.490002  118.389999  728.280029
2016-06-09  99.650002  727.650024  46.570000  118.559998  728.580017
2016-06-10  98.830002  717.909973  44.509998  116.620003  719.409973
```

```
In [391]: p['Volume']
Out[391]:
                  AAPL       AMZN        COP          FB       GOOG
Date
2016-06-08  20812700.0  2200100.0  9596700.0  14368700.0  1582100.0
2016-06-09  26419600.0  2163100.0  5389300.0  13823400.0   985900.0
2016-06-10  31462100.0  3409500.0  8941200.0  18412700.0  1206000.0

In [394]: p[:,:,'AAPL']
Out[394]:
                 Open       High        Low      Close      Volume  Adj Close
Date
2016-06-08  99.019997  99.559998  98.680000  98.940002  20812700.0  98.940002
2016-06-09  98.500000  99.989998  98.459999  99.650002  26419600.0  99.650002
2016-06-10  98.529999  99.349998  98.480003  98.830002  31462100.0  98.830002

In [395]: p[:,'2016-06-10']
Out[395]:
            Open        High         Low       Close      Volume   Adj Close
AAPL   98.529999   99.349998   98.480003   98.830002  31462100.0   98.830002
AMZN  722.349976  724.979980  714.210022  717.909973   3409500.0  717.909973
COP    45.900002   46.119999   44.259998   44.509998   8941200.0   44.509998
FB    117.540001  118.110001  116.260002  116.620003  18412700.0  116.620003
GOOG  719.469971  725.890015  716.429993  719.409973   1206000.0  719.409973
```

Read Pandas Datareader online: https://riptutorial.com/pandas/topic/1912/pandas-datareader

Chapter 28: Pandas IO tools (reading and saving data sets)

Remarks

The pandas official documentation includes a page on IO Tools with a list of relevant functions to read and write to files, as well as some examples and common parameters.

Examples

Reading csv file into DataFrame

Example for reading file `data_file.csv` such as:

File:

```
index,header1,header2,header3
1,str_data,12,1.4
3,str_data,22,42.33
4,str_data,2,3.44
2,str_data,43,43.34

7, str_data, 25, 23.32
```

Code:

```
pd.read_csv('data_file.csv')
```

Output:

```
   index    header1  header2  header3
0      1   str_data       12     1.40
1      3   str_data       22    42.33
2      4   str_data        2     3.44
3      2   str_data       43    43.34
4      7   str_data       25    23.32
```

Some useful arguments:

- **`sep`** The default field delimiter is a comma `,`. Use this option if you need a different delimiter, for instance `pd.read_csv('data_file.csv', sep=';')`
- **`index_col`** With `index_col = n` (`n` an integer) you tell pandas to use column `n` to index the

DataFrame. In the above example:

```
pd.read_csv('data_file.csv',  index_col=0)
```

Output:

```
          header1  header2  header3
index
 1        str_data       12     1.40
 3        str_data       22    42.33
 4        str_data        2     3.44
 2        str_data       43    43.34
 7        str_data       25    23.32
```

- **`skip_blank_lines`** By default blank lines are skipped. Use `skip_blank_lines=False` to include blank lines (they will be filled with `NaN` values)

```
pd.read_csv('data_file.csv',  index_col=0,skip_blank_lines=False)
```

Output:

```
         header1  header2  header3
index
 1       str_data       12     1.40
 3       str_data       22    42.33
 4       str_data        2     3.44
 2       str_data       43    43.34
NaN           NaN      NaN      NaN
 7       str_data       25    23.32
```

- **`parse_dates`** Use this option to parse date data.

File:

```
date_begin;date_end;header3;header4;header5
1/1/2017;1/10/2017;str_data;1001;123,45
2/1/2017;2/10/2017;str_data;1001;67,89
3/1/2017;3/10/2017;str_data;1001;0
```

Code to parse columns `0` and `1` as dates:

```
pd.read_csv('f.csv', sep=';', parse_dates=[0,1])
```

Output:

```
  date_begin   date_end   header3  header4 header5
0 2017-01-01 2017-01-10  str_data     1001  123,45
1 2017-02-01 2017-02-10  str_data     1001   67,89
2 2017-03-01 2017-03-10  str_data     1001       0
```

By default, the date format is inferred. If you want to specify a date format you can use for instance

```
dateparse = lambda x: pd.datetime.strptime(x, '%d/%m/%Y')
pd.read_csv('f.csv', sep=';',parse_dates=[0,1],date_parser=dateparse)
```

Output:

```
  date_begin   date_end   header3  header4 header5
0 2017-01-01 2017-10-01  str_data     1001  123,45
1 2017-01-02 2017-10-02  str_data     1001   67,89
2 2017-01-03 2017-10-03  str_data     1001       0
```

More information on the function's parameters can be found in the official documentation.

Basic saving to a csv file

```
raw_data = {'first_name': ['John', 'Jane', 'Jim'],
            'last_name': ['Doe', 'Smith', 'Jones'],
            'department': ['Accounting', 'Sales', 'Engineering'],}
df = pd.DataFrame(raw_data,columns=raw_data.keys())
df.to_csv('data_file.csv')
```

Parsing dates when reading from csv

You can specify a column that contains dates so pandas would automatically parse them when reading from the csv

```
pandas.read_csv('data_file.csv', parse_dates=['date_column'])
```

Spreadsheet to dict of DataFrames

```
with pd.ExcelFile('path_to_file.xls) as xl:
    d = {sheet_name: xl.parse(sheet_name) for sheet_name in xl.sheet_names}
```

Read a specific sheet

```
pd.read_excel('path_to_file.xls', sheetname='Sheet1')
```

There are many parsing options for `read_excel` (similar to the options in `read_csv`.

```
pd.read_excel('path_to_file.xls',
              sheetname='Sheet1', header=[0, 1, 2],
              skiprows=3, index_col=0)  # etc.
```

Testing read_csv

```
import pandas as pd
import io
```

```
temp=u"""index; header1; header2; header3
1; str_data; 12; 1.4
3; str_data; 22; 42.33
4; str_data; 2; 3.44
2; str_data; 43; 43.34
7; str_data; 25; 23.32"""
#after testing replace io.StringIO(temp) to filename
df = pd.read_csv(io.StringIO(temp),
                 sep = ';',
                 index_col = 0,
                 skip_blank_lines = True)
print (df)
         header1   header2   header3
index
1       str_data        12      1.40
3       str_data        22     42.33
4       str_data         2      3.44
2       str_data        43     43.34
7       str_data        25     23.32
```

List comprehension

All files are in folder `files`. First create list of DataFrames and then `concat` them:

```
import pandas as pd
import glob

#a.csv
#a,b
#1,2
#5,8

#b.csv
#a,b
#9,6
#6,4

#c.csv
#a,b
#4,3
#7,0

files = glob.glob('files/*.csv')
dfs = [pd.read_csv(fp) for fp in files]
```

```
#duplicated index inherited from each Dataframe
df = pd.concat(dfs)
print (df)
   a  b
0  1  2
1  5  8
0  9  6
1  6  4
0  4  3
1  7  0
#'reseting' index
df = pd.concat(dfs, ignore_index=True)
```

```
print (df)
   a  b
0  1  2
1  5  8
2  9  6
3  6  4
4  4  3
5  7  0
#concat by columns
df1 = pd.concat(dfs, axis=1)
print (df1)
   a  b  a  b  a  b
0  1  2  9  6  4  3
1  5  8  6  4  7  0
#reset column names
df1 = pd.concat(dfs, axis=1, ignore_index=True)
print (df1)
   0  1  2  3  4  5
0  1  2  9  6  4  3
1  5  8  6  4  7  0
```

Read in chunks

```
import pandas as pd

chunksize = [n]
for chunk in pd.read_csv(filename, chunksize=chunksize):
    process(chunk)
    delete(chunk)
```

Save to CSV file

Save with default parameters:

```
df.to_csv(file_name)
```

Write specific columns:

```
df.to_csv(file_name, columns =['col'])
```

Difault delimiter is ',' - to change it:

```
df.to_csv(file_name,sep="|")
```

Write without the header:

```
df.to_csv(file_name, header=False)
```

Write with a given header:

```
df.to_csv(file_name, header = ['A','B','C',...]
```

To use a specific encoding (e.g. 'utf-8') use the encoding argument:

df.to_csv(file_name, encoding='utf-8')

Parsing date columns with read_csv

Date always have a different format, they can be parsed using a specific parse_dates function.

This *input.csv*:

```
2016 06 10 20:30:00    foo
2016 07 11 19:45:30    bar
2013 10 12 4:30:00     foo
```

Can be parsed like this :

```
mydateparser = lambda x: pd.datetime.strptime(x, "%Y %m %d %H:%M:%S")
df = pd.read_csv("file.csv", sep='\t', names=['date_column', 'other_column'],
parse_dates=['date_column'], date_parser=mydateparser)
```

parse_dates argument is the column to be parsed
date_parser is the parser function

Read & merge multiple CSV files (with the same structure) into one DF

```
import os
import glob
import pandas as pd

def get_merged_csv(flist, **kwargs):
    return pd.concat([pd.read_csv(f, **kwargs) for f in flist], ignore_index=True)

path = 'C:/Users/csvfiles'
fmask = os.path.join(path, '*mask*.csv')

df = get_merged_csv(glob.glob(fmask), index_col=None, usecols=['col1', 'col3'])

print(df.head())
```

If you want to merge CSV files horizontally (adding columns), use `axis=1` when calling `pd.concat()` function:

```
def merged_csv_horizontally(flist, **kwargs):
    return pd.concat([pd.read_csv(f, **kwargs) for f in flist], axis=1)
```

Reading cvs file into a pandas data frame when there is no header row

If the file does not contain a header row, File:

```
1;str_data;12;1.4
3;str_data;22;42.33
4;str_data;2;3.44
2;str_data;43;43.34

7; str_data; 25; 23.32
```

you can use the keyword `names` to provide column names:

```
df = pandas.read_csv('data_file.csv', sep=';', index_col=0,
                     skip_blank_lines=True, names=['a', 'b', 'c'])

df
Out:
           a   b      c
1   str_data  12   1.40
3   str_data  22  42.33
4   str_data   2   3.44
2   str_data  43  43.34
7   str_data  25  23.32
```

Using HDFStore

```
import string
import numpy as np
import pandas as pd
```

generate sample DF with various dtypes

```
df = pd.DataFrame({
     'int32':    np.random.randint(0, 10**6, 10),
     'int64':    np.random.randint(10**7, 10**9, 10).astype(np.int64)*10,
     'float':    np.random.rand(10),
     'string':   np.random.choice([c*10 for c in string.ascii_uppercase], 10),
     })

In [71]: df
Out[71]:
      float   int32       int64      string
0  0.649978  848354  5269162190  DDDDDDDDDD
1  0.346963  490266  6897476700  OOOOOOOOOO
2  0.035069  756373  6711566750  ZZZZZZZZZZ
3  0.066692  957474  9085243570  FFFFFFFFFF
4  0.679182  665894  3750794810  MMMMMMMMMM
5  0.861914  630527  6567684430  TTTTTTTTTT
6  0.697691  825704  8005182860  FFFFFFFFFF
7  0.474501  942131  4099797720  QQQQQQQQQQ
8  0.645817  951055  8065980030  VVVVVVVVVV
9  0.083500  349709  7417288920  EEEEEEEEEE
```

make a bigger DF (10 * 100.000 = 1.000.000 rows)

```
df = pd.concat([df] * 10**5, ignore_index=True)
```

create (or open existing) HDFStore file

```
store = pd.HDFStore('d:/temp/example.h5')
```

save our data frame into `h5` (HDFStore) file, indexing [int32, int64, string] columns:

```
store.append('store_key', df, data_columns=['int32','int64','string'])
```

show HDFStore details

```
In [78]: store.get_storer('store_key').table
Out[78]:
/store_key/table (Table(10,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "int32": Int32Col(shape=(), dflt=0, pos=2),
  "int64": Int64Col(shape=(), dflt=0, pos=3),
  "string": StringCol(itemsize=10, shape=(), dflt=b'', pos=4)}
  byteorder := 'little'
  chunkshape := (1724,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "int32": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "int64": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

show indexed columns

```
In [80]: store.get_storer('store_key').table.colindexes
Out[80]:
{
    "int32": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False,
```

```
    "int64": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

close (flush to disk) our store file

```
store.close()
```

Read Nginx access log (multiple quotechars)

For multiple quotechars use regex in place of sep:

```
df = pd.read_csv(log_file,
              sep=r'\s(?=(?:[^"]*"[^"]*")*[^"]*$)(?![^\[]*\])',
              engine='python',
              usecols=[0, 3, 4, 5, 6, 7, 8],
              names=['ip', 'time', 'request', 'status', 'size', 'referer', 'user_agent'],
              na_values='-',
              header=None
                )
```

Read Pandas IO tools (reading and saving data sets) online:
https://riptutorial.com/pandas/topic/2896/pandas-io-tools--reading-and-saving-data-sets-

Chapter 29: pd.DataFrame.apply

Examples

pandas.DataFrame.apply Basic Usage

The pandas.DataFrame.apply() method is used to apply a given function to an entire `DataFrame` --- for example, computing the square root of every entry of a given `DataFrame` or summing across each row of a `DataFrame` to return a `Series`.

The below is a basic example of usage of this function:

```
# create a random DataFrame with 7 rows and 2 columns
df = pd.DataFrame(np.random.randint(0,100,size = (7,2)),
                  columns = ['fst','snd'])

>>> df
   fst  snd
0   40   94
1   58   93
2   95   95
3   88   40
4   25   27
5   62   64
6   18   92

# apply the square root function to each column:
# (this returns a DataFrame where each entry is the sqrt of the entry in df;
# setting axis=0 or axis=1 doesn't make a difference)
>>> df.apply(np.sqrt)
        fst       snd
0  6.324555  9.695360
1  7.615773  9.643651
2  9.746794  9.746794
3  9.380832  6.324555
4  5.000000  5.196152
5  7.874008  8.000000
6  4.242641  9.591663

# sum across the row (axis parameter now makes a difference):
>>> df.apply(np.sum, axis=1)
0    134
1    151
2    190
3    128
4     52
5    126
6    110
dtype: int64

>>> df.apply(np.sum)
fst    386
snd    505
dtype: int64
```

Read pd.DataFrame.apply online: https://riptutorial.com/pandas/topic/7024/pd-dataframe-apply

Chapter 30: Read MySQL to DataFrame

Examples

Using sqlalchemy and PyMySQL

```
from sqlalchemy import create_engine

cnx = create_engine('mysql+pymysql://username:password@server:3306/database').connect()
sql = 'select * from mytable'
df = pd.read_sql(sql, cnx)
```

To read mysql to dataframe, In case of large amount of data

To fetch large data we can use generators in pandas and load data in chunks.

```
import pandas as pd
from sqlalchemy import create_engine
from sqlalchemy.engine.url import URL


# sqlalchemy engine
engine = create_engine(URL(
    drivername="mysql"
    username="user",
    password="password"
    host="host"
    database="database"
))

conn = engine.connect()

generator_df = pd.read_sql(sql=query,  # mysql query
                           con=conn,
                           chunksize=chunksize)  # size you want to fetch each time

for dataframe in generator_df:
    for row in dataframe:
        pass  # whatever you want to do
```

Read Read MySQL to DataFrame online: https://riptutorial.com/pandas/topic/8809/read-mysql-to-dataframe

Chapter 31: Read SQL Server to Dataframe

Examples

Using pyodbc

```
import pandas.io.sql
import pyodbc
import pandas as pd
```

Specify the parameters

```
# Parameters
server = 'server_name'
db = 'database_name'
UID = 'user_id'
```

Create the connection

```
# Create the connection
conn = pyodbc.connect('DRIVER={SQL Server};SERVER=' + server + ';DATABASE=' + db + '; UID = '
+ UID + '; PWD = ' + UID + 'Trusted_Connection=yes')
```

Query into pandas dataframe

```
# Query into dataframe
df= pandas.io.sql.read_sql('sql_query_string', conn)
```

Using pyodbc with connection loop

```
import os, time
import pyodbc
import pandas.io.sql as pdsql

def todf(dsn='yourdsn', uid=None, pwd=None, query=None, params=None):
    ''' if `query` is not an actual query but rather a path to a text file
        containing a query, read it in instead '''
    if query.endswith('.sql') and os.path.exists(query):
        with open(query,'r') as fin:
            query = fin.read()

    connstr = "DSN={};UID={};PWD={}".format(dsn,uid,pwd)
    connected = False
    while not connected:
        try:
            with pyodbc.connect(connstr,autocommit=True) as con:
                cur = con.cursor()
                if params is not None: df = pdsql.read_sql(query, con,
                                                           params=params)
                else: df = pdsql.read_sql(query, con)
                cur.close()
```

```
            break
        except pyodbc.OperationalError:
            time.sleep(60) # one minute could be changed
    return df
```

Read Read SQL Server to Dataframe online: https://riptutorial.com/pandas/topic/2176/read-sql-server-to-dataframe

Chapter 32: Reading files into pandas DataFrame

Examples

Read table into DataFrame

Table file with header, footer, row names, and index column:

file: table.txt

```
This is a header that discusses the table file
to show space in a generic table file

index  name     occupation
1      Alice    Salesman
2      Bob      Engineer
3      Charlie  Janitor

This is a footer because your boss does not understand data files
```

code:

```
import pandas as pd
# index_col=0 tells pandas that column 0 is the index and not data
pd.read_table('table.txt', delim_whitespace=True, skiprows=3, skipfooter=2, index_col=0)
```

output:

```
          name occupation
index
1        Alice   Salesman
2          Bob   Engineer
3      Charlie    Janitor
```

Table file without row names or index:

file: table.txt

```
Alice    Salesman
Bob      Engineer
Charlie  Janitor
```

code:

```
import pandas as pd
```

```
pd.read_table('table.txt', delim_whitespace=True, names=['name','occupation'])
```

output:

```
      name occupation
0    Alice   Salesman
1      Bob   Engineer
2  Charlie    Janitor
```

All options can be found in the pandas documentation here

Read CSV File

Data with header, separated by semicolons instead of commas

file: table.csv

```
index;name;occupation
1;Alice;Saleswoman
2;Bob;Engineer
3;Charlie;Janitor
```

code:

```
import pandas as pd
pd.read_csv('table.csv', sep=';', index_col=0)
```

output:

```
          name occupation
index
1        Alice   Salesman
2          Bob   Engineer
3      Charlie    Janitor
```

Table without row names or index and commas as separators

file: table.csv

```
Alice,Saleswoman
Bob,Engineer
Charlie,Janitor
```

code:

```
import pandas as pd
pd.read_csv('table.csv', names=['name','occupation'])
```

output:

```
      name occupation
0    Alice   Salesman
1      Bob   Engineer
2  Charlie    Janitor
```

further clarification can be found in the `read_csv` documentation page

Collect google spreadsheet data into pandas dataframe

Sometimes we need to collect data from google spreadsheets. We can use **gspread** and **oauth2client** libraries to collect data from google spreadsheets. Here is a example to collect data:

Code:

```
from __future__ import print_function
import gspread
from oauth2client.client import SignedJwtAssertionCredentials
import pandas as pd
import json

scope = ['https://spreadsheets.google.com/feeds']

credentials = ServiceAccountCredentials.from_json_keyfile_name('your-authorization-file.json',
scope)

gc = gspread.authorize(credentials)

work_sheet = gc.open_by_key("spreadsheet-key-here")
sheet = work_sheet.sheet1
data = pd.DataFrame(sheet.get_all_records())

print(data.head())
```

Read Reading files into pandas DataFrame online:
https://riptutorial.com/pandas/topic/1988/reading-files-into-pandas-dataframe

Chapter 33: Resampling

Examples

Downsampling and upsampling

```
import pandas as pd
import numpy as np

np.random.seed(0)
rng = pd.date_range('2015-02-24', periods=10, freq='T')
df = pd.DataFrame({'Val' : np.random.randn(len(rng))}, index=rng)
print (df)
                          Val
2015-02-24 00:00:00  1.764052
2015-02-24 00:01:00  0.400157
2015-02-24 00:02:00  0.978738
2015-02-24 00:03:00  2.240893
2015-02-24 00:04:00  1.867558
2015-02-24 00:05:00 -0.977278
2015-02-24 00:06:00  0.950088
2015-02-24 00:07:00 -0.151357
2015-02-24 00:08:00 -0.103219
2015-02-24 00:09:00  0.410599
```

```
#downsampling with aggregating sum
print (df.resample('5Min').sum())
                          Val
2015-02-24 00:00:00  7.251399
2015-02-24 00:05:00  0.128833

#5Min is same as 5T
print (df.resample('5T').sum())
                          Val
2015-02-24 00:00:00  7.251399
2015-02-24 00:05:00  0.128833

#upsampling and fill NaN values method forward filling
print (df.resample('30S').ffill())
                          Val
2015-02-24 00:00:00  1.764052
2015-02-24 00:00:30  1.764052
2015-02-24 00:01:00  0.400157
2015-02-24 00:01:30  0.400157
2015-02-24 00:02:00  0.978738
2015-02-24 00:02:30  0.978738
2015-02-24 00:03:00  2.240893
2015-02-24 00:03:30  2.240893
2015-02-24 00:04:00  1.867558
2015-02-24 00:04:30  1.867558
2015-02-24 00:05:00 -0.977278
2015-02-24 00:05:30 -0.977278
2015-02-24 00:06:00  0.950088
2015-02-24 00:06:30  0.950088
2015-02-24 00:07:00 -0.151357
2015-02-24 00:07:30 -0.151357
```

```
2015-02-24 00:08:00 -0.103219
2015-02-24 00:08:30 -0.103219
2015-02-24 00:09:00  0.410599
```

Read Resampling online: https://riptutorial.com/pandas/topic/2164/resampling

Chapter 34: Reshaping and pivoting

Examples

Simple pivoting

First try use `pivot`:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'Name':['Mary', 'Josh','Jon','Lucy', 'Jane', 'Sue'],
                   'Age':[34, 37, 29, 40, 29, 31],
                   'City':['Boston','New York', 'Chicago', 'Los Angeles', 'Chicago',
'Boston'],
                   'Position':['Manager','Programmer','Manager','Manager','Programmer',
'Programmer']},
                    columns=['Name','Position','City','Age'])

print (df)
   Name    Position         City  Age
0  Mary     Manager       Boston   34
1  Josh  Programmer     New York   37
2   Jon     Manager      Chicago   29
3  Lucy     Manager  Los Angeles   40
4  Jane  Programmer      Chicago   29
5   Sue  Programmer       Boston   31

print (df.pivot(index='Position', columns='City', values='Age'))
City        Boston  Chicago  Los Angeles  New York
Position
Manager       34.0     29.0         40.0       NaN
Programmer    31.0     29.0          NaN      37.0
```

If need reset index, remove columns names and fill NaN values:

```
#pivoting by numbers - column Age
print (df.pivot(index='Position', columns='City', values='Age')
         .reset_index()
         .rename_axis(None, axis=1)
         .fillna(0))

     Position  Boston  Chicago  Los Angeles  New York
0     Manager    34.0     29.0         40.0       0.0
1  Programmer    31.0     29.0          0.0      37.0


#pivoting by strings - column Name
print (df.pivot(index='Position', columns='City', values='Name'))

City       Boston Chicago Los Angeles New York
Position
Manager      Mary     Jon        Lucy     None
Programmer    Sue    Jane        None     Josh
```

Pivoting with aggregating

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'Name':['Mary', 'Jon','Lucy', 'Jane', 'Sue', 'Mary', 'Lucy'],
                   'Age':[35, 37, 40, 29, 31, 26, 28],
                   'City':['Boston', 'Chicago', 'Los Angeles', 'Chicago', 'Boston', 'Boston',
'Chicago'],
                   'Position':['Manager','Manager','Manager','Programmer',
'Programmer','Manager','Manager'],
                    'Sex':['Female','Male','Female','Female', 'Female','Female','Female']},
                    columns=['Name','Position','City','Age','Sex'])

print (df)
   Name    Position         City  Age   Sex
0  Mary     Manager       Boston   35  Female
1   Jon     Manager      Chicago   37  Male
2  Lucy     Manager  Los Angeles   40  Female
3  Jane  Programmer      Chicago   29  Female
4   Sue  Programmer       Boston   31  Female
5  Mary     Manager       Boston   26  Female
6  Lucy     Manager      Chicago   28  Female
```

If use `pivot`, get error:

```
print (df.pivot(index='Position', columns='City', values='Age'))
```

> ValueError: Index contains duplicate entries, cannot reshape

Use `pivot_table` with aggregating function:

```
#default aggfunc is np.mean
print (df.pivot_table(index='Position', columns='City', values='Age'))
City        Boston  Chicago  Los Angeles
Position
Manager       30.5     32.5         40.0
Programmer    31.0     29.0          NaN

print (df.pivot_table(index='Position', columns='City', values='Age', aggfunc=np.mean))
City        Boston  Chicago  Los Angeles
Position
Manager       30.5     32.5         40.0
Programmer    31.0     29.0          NaN
```

Another agg functions:

```
print (df.pivot_table(index='Position', columns='City', values='Age', aggfunc=sum))
City        Boston  Chicago  Los Angeles
Position
Manager       61.0     65.0         40.0
Programmer    31.0     29.0          NaN

#lost data !!!
print (df.pivot_table(index='Position', columns='City', values='Age', aggfunc='first'))
City        Boston  Chicago  Los Angeles
```

```
Position
Manager       35.0     37.0         40.0
Programmer    31.0     29.0          NaN
```

If need aggregate by columns with `string` values:

```
print (df.pivot_table(index='Position', columns='City', values='Name'))
```

> DataError: No numeric types to aggregate

You can use these aggragating functions:

```
print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc='first'))
City       Boston Chicago Los Angeles
Position
Manager      Mary     Jon        Lucy
Programmer    Sue    Jane        None

print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc='last'))
City       Boston Chicago Los Angeles
Position
Manager      Mary    Lucy        Lucy
Programmer    Sue    Jane        None

print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc='sum'))
City          Boston  Chicago Los Angeles
Position
Manager     MaryMary  JonLucy        Lucy
Programmer       Sue     Jane        None

print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc=', '.join))
City            Boston    Chicago Los Angeles
Position
Manager     Mary, Mary  Jon, Lucy        Lucy
Programmer         Sue       Jane        None

print (df.pivot_table(index='Position', columns='City', values='Name', aggfunc=', '.join,
fill_value='-')
         .reset_index()
         .rename_axis(None, axis=1))
     Position      Boston    Chicago Los Angeles
0     Manager  Mary, Mary  Jon, Lucy        Lucy
1  Programmer         Sue       Jane           -
```

The information regarding the *Sex* has yet not been used. It could be switched by one of the columns, or it could be added as another level:

```
print (df.pivot_table(index='Position', columns=['City','Sex'], values='Age',
aggfunc='first'))

City       Boston Chicago       Los Angeles
Sex        Female  Female  Male      Female
Position
Manager      35.0    28.0  37.0        40.0
Programmer   31.0    29.0   NaN         NaN
```

Multiple columns can be specified in any of the attributes index, columns and values.

```
print (df.pivot_table(index=['Position','Sex'], columns='City', values='Age',
aggfunc='first'))

City                Boston  Chicago  Los Angeles
Position   Sex
Manager    Female    35.0     28.0         40.0
           Male       NaN     37.0          NaN
Programmer Female    31.0     29.0          NaN
```

Applying several aggregating functions

You can easily apply multiple functions during a single pivot:

```
In [23]: import numpy as np

In [24]: df.pivot_table(index='Position', values='Age', aggfunc=[np.mean, np.std])
Out[24]:
                 mean       std
Position
Manager     34.333333  5.507571
Programmer  32.333333  4.163332
```

Sometimes, you may want to apply specific functions to specific columns:

```
In [35]: df['Random'] = np.random.random(6)
In [36]: df
Out[36]:
   Name    Position         City  Age    Random
0  Mary     Manager       Boston   34  0.678577
1  Josh  Programmer     New York   37  0.973168
2   Jon     Manager      Chicago   29  0.146668
3  Lucy     Manager  Los Angeles   40  0.150120
4  Jane  Programmer      Chicago   29  0.112769
5   Sue  Programmer       Boston   31  0.185198

For example, find the mean age, and standard deviation of random by Position:

In [37]: df.pivot_table(index='Position', aggfunc={'Age': np.mean, 'Random': np.std})
Out[37]:
                  Age    Random
Position
Manager     34.333333  0.306106
Programmer  32.333333  0.477219
```

One can pass a list of functions to apply to the individual columns as well:

```
In [38]: df.pivot_table(index='Position', aggfunc={'Age': np.mean, 'Random': [np.mean,
np.std]})]
Out[38]:
                  Age    Random
                 mean      mean       std
Position
Manager     34.333333  0.325122  0.306106
Programmer  32.333333  0.423712  0.477219
```

Stacking and unstacking

```
import pandas as pd
import numpy as np

np.random.seed(0)
tuples = list(zip(*[['bar', 'bar', 'foo', 'foo', 'qux', 'qux'],
                    ['one', 'two', 'one', 'two','one', 'two']]))

idx = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
df = pd.DataFrame(np.random.randn(6, 2), index=idx, columns=['A', 'B'])
print (df)
                     A         B
first second
bar   one     1.764052  0.400157
      two     0.978738  2.240893
foo   one     1.867558 -0.977278
      two     0.950088 -0.151357
qux   one    -0.103219  0.410599
      two     0.144044  1.454274
```

```
print (df.stack())
first  second
bar    one     A    1.764052
               B    0.400157
       two     A    0.978738
               B    2.240893
foo    one     A    1.867558
               B   -0.977278
       two     A    0.950088
               B   -0.151357
qux    one     A   -0.103219
               B    0.410599
       two     A    0.144044
               B    1.454274
dtype: float64

#reset index, rename column name
print (df.stack().reset_index(name='val2').rename(columns={'level_2': 'val1'}))
   first second val1      val2
0    bar    one    A  1.764052
1    bar    one    B  0.400157
2    bar    two    A  0.978738
3    bar    two    B  2.240893
4    foo    one    A  1.867558
5    foo    one    B -0.977278
6    foo    two    A  0.950088
7    foo    two    B -0.151357
8    qux    one    A -0.103219
9    qux    one    B  0.410599
10   qux    two    A  0.144044
11   qux    two    B  1.454274
```

```
print (df.unstack())
               A                   B
second        one       two       one       two
first
bar     1.764052  0.978738  0.400157  2.240893
```

```
foo     1.867558  0.950088 -0.977278 -0.151357
qux    -0.103219  0.144044  0.410599  1.454274
```

`rename_axis` (new in `pandas 0.18.0`):

```
#reset index, remove columns names
df1 = df.unstack().reset_index().rename_axis((None,None), axis=1)
#reset MultiIndex in columns with list comprehension
df1.columns = ['_'.join(col).strip('_') for col in df1.columns]
print (df1)
  first     A_one     A_two     B_one     B_two
0   bar  1.764052  0.978738  0.400157  2.240893
1   foo  1.867558  0.950088 -0.977278 -0.151357
2   qux -0.103219  0.144044  0.410599  1.454274
```

pandas bellow 0.18.0

```
#reset index
df1 = df.unstack().reset_index()
#remove columns names
df1.columns.names = (None, None)
#reset MultiIndex in columns with list comprehension
df1.columns = ['_'.join(col).strip('_') for col in df1.columns]
print (df1)
  first     A_one     A_two     B_one     B_two
0   bar  1.764052  0.978738  0.400157  2.240893
1   foo  1.867558  0.950088 -0.977278 -0.151357
2   qux -0.103219  0.144044  0.410599  1.454274
```

Cross Tabulation

```
import pandas as pd
df = pd.DataFrame({'Sex': ['M', 'M', 'F', 'M', 'F', 'F', 'M', 'M', 'F', 'F'],
               'Age': [20, 19, 17, 35, 22, 22, 12, 15, 17, 22],
               'Heart Disease': ['Y', 'N', 'Y', 'N', 'N', 'Y', 'N', 'Y', 'N', 'Y']})

df

  Age Heart Disease Sex
0   20             Y   M
1   19             N   M
2   17             Y   F
3   35             N   M
4   22             N   F
5   22             Y   F
6   12             N   M
7   15             Y   M
8   17             N   F
9   22             Y   F

pd.crosstab(df['Sex'], df['Heart Disease'])

Hearth Disease  N  Y
Sex
F               2  3
M               3  2
```

Using dot notation:

```
pd.crosstab(df.Sex, df.Age)

Age  12  15  17  19  20  22  35
Sex
F     0   0   2   0   0   3   0
M     1   1   0   1   1   0   1
```

Getting transpose of DF:

```
pd.crosstab(df.Sex, df.Age).T

Sex  F  M
Age
12   0  1
15   0  1
17   2  0
19   0  1
20   0  1
22   3  0
35   0  1
```

Getting margins or cumulatives:

```
pd.crosstab(df['Sex'], df['Heart Disease'], margins=True)

Heart Disease  N  Y  All
Sex
F              2  3    5
M              3  2    5
All            5  5   10
```

Getting transpose of cumulative:

```
pd.crosstab(df['Sex'], df['Age'], margins=True).T


Sex  F  M  All
Age
12   0  1    1
15   0  1    1
17   2  0    2
19   0  1    1
20   0  1    1
22   3  0    3
35   0  1    1
All  5  5   10
```

Getting percentages :

```
pd.crosstab(df["Sex"],df['Heart Disease']).apply(lambda r: r/len(df), axis=1)

Heart Disease    N    Y
Sex
```

```
F              0.2  0.3
M              0.3  0.2
```

Getting cumulative and multiplying by 100:

```
df2 = pd.crosstab(df["Age"],df['Sex'], margins=True ).apply(lambda r: r/len(df)*100, axis=1)

df2

Sex     F     M    All
Age
12    0.0  10.0   10.0
15    0.0  10.0   10.0
17   20.0   0.0   20.0
19    0.0  10.0   10.0
20    0.0  10.0   10.0
22   30.0   0.0   30.0
35    0.0  10.0   10.0
All  50.0  50.0  100.0
```

Removing a column from DF (one way):

```
df2[["F","M"]]

Sex     F     M
Age
12    0.0  10.0
15    0.0  10.0
17   20.0   0.0
19    0.0  10.0
20    0.0  10.0
22   30.0   0.0
35    0.0  10.0
All  50.0  50.0
```

Pandas melt to go from wide to long

```
>>> df
   ID  Year  Jan_salary  Feb_salary  Mar_salary
0   1  2016        4500        4200        4700
1   2  2016        3800        3600        4400
2   3  2016        5500        5200        5300

>>> melted_df = pd.melt(df,id_vars=['ID','Year'],
                        value_vars=['Jan_salary','Feb_salary','Mar_salary'],
                        var_name='month',value_name='salary')

>>> melted_df
   ID  Year       month  salary
0   1  2016  Jan_salary    4500
1   2  2016  Jan_salary    3800
2   3  2016  Jan_salary    5500
3   1  2016  Feb_salary    4200
4   2  2016  Feb_salary    3600
5   3  2016  Feb_salary    5200
6   1  2016  Mar_salary    4700
7   2  2016  Mar_salary    4400
```

```
8   3  2016  Mar_salary    5300

>>> melted_['month'] = melted_['month'].str.replace('_salary','')

>>> import calendar
>>> def mapper(month_abbr):
...     # from http://stackoverflow.com/a/3418092/42346
...     d = {v: str(k).zfill(2) for k,v in enumerate(calendar.month_abbr)}
...     return d[month_abbr]

>>> melted_df['month'] = melted_df['month'].apply(mapper)
>>> melted_df
   ID  Year month  salary
0   1  2016    01    4500
1   2  2016    01    3800
2   3  2016    01    5500
3   1  2016    02    4200
4   2  2016    02    3600
5   3  2016    02    5200
6   1  2016    03    4700
7   2  2016    03    4400
8   3  2016    03    5300
```

Split (reshape) CSV strings in columns into multiple rows, having one element per row

```
import pandas as pd

df = pd.DataFrame([{'var1': 'a,b,c', 'var2': 1, 'var3': 'XX'},
                   {'var1': 'd,e,f,x,y', 'var2': 2, 'var3': 'ZZ'}])

print(df)

reshaped = \
(df.set_index(df.columns.drop('var1',1).tolist())
   .var1.str.split(',', expand=True)
   .stack()
   .reset_index()
   .rename(columns={0:'var1'})
   .loc[:, df.columns]
)

print(reshaped)
```

Output:

```
        var1  var2 var3
0      a,b,c     1   XX
1  d,e,f,x,y     2   ZZ

  var1  var2 var3
0    a     1   XX
1    b     1   XX
2    c     1   XX
3    d     2   ZZ
4    e     2   ZZ
5    f     2   ZZ
```

```
6    x     2   ZZ
7    y     2   ZZ
```

Read Reshaping and pivoting online: https://riptutorial.com/pandas/topic/1463/reshaping-and-pivoting

Chapter 35: Save pandas dataframe to a csv file

Parameters

Parameter	Description
path_or_buf	string or file handle, default None File path or object, if None is provided the result is returned as a string.
sep	character, default ‘,’ Field delimiter for the output file.
na_rep	string, default ‘’ Missing data representation
float_format	string, default None Format string for floating point numbers
columns	sequence, optional Columns to write
header	boolean or list of string, default True Write out column names. If a list of string is given it is assumed to be aliases for the column names
index	boolean, default True Write row names (index)
index_label	string or sequence, or False, default None Column label for index column(s) if desired. If None is given, and header and index are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R
nanRep	None deprecated, use na_rep
mode	str Python write mode, default ‘w’
encoding	string, optional A string representing the encoding to use in the output file, defaults to ‘ascii’ on Python 2 and ‘utf-8’ on Python 3.
compression	string, optional a string representing the compression to use in the output file, allowed values are ‘gzip’, ‘bz2’, ‘xz’, only used when the first argument is a filename
line_terminator	string, default ‘n’ The newline character or character sequence to use in the output file
quoting	optional constant from csv module defaults to csv.QUOTE_MINIMAL
quotechar	string (length 1), default ‘”’ character used to quote fields

Parameter	Description
doublequote	boolean, default True Control quoting of quotechar inside a field
escapechar	string (length 1), default None character used to escape sep and quotechar when appropriate
chunksize	int or None rows to write at a time
tupleize_cols	boolean, default False write multi_index columns as a list of tuples (if True) or new (expanded format) if False)
date_format	string, default None Format string for datetime objects
decimal	string, default '.' Character recognized as decimal separator. E.g. use ',' for European data

Examples

Create random DataFrame and write to .csv

Create a simple DataFrame.

```
import numpy as np
import pandas as pd

# Set the seed so that the numbers can be reproduced.
np.random.seed(0)

df = pd.DataFrame(np.random.randn(5, 3), columns=list('ABC'))

# Another way to set column names is
"columns=['column_1_name','column_2_name','column_3_name']"

df

      A         B         C
0  1.764052  0.400157  0.978738
1  2.240893  1.867558 -0.977278
2  0.950088 -0.151357 -0.103219
3  0.410599  0.144044  1.454274
4  0.761038  0.121675  0.443863
```

Now, write to a CSV file:

```
df.to_csv('example.csv', index=False)
```

Contents of example.csv:

```
A,B,C
1.76405234597,0.400157208367,0.978737984106
2.2408931992,1.86755799015,-0.977277879876
```

```
0.950088417526,-0.151357208298,-0.103218851794
0.410598501938,0.144043571161,1.45427350696
0.761037725147,0.121675016493,0.443863232745
```

Note that we specify `index=False` so that the auto-generated indices (row #s 0,1,2,3,4) are not included in the CSV file. Include it if you need the index column, like so:

```
df.to_csv('example.csv', index=True)  # Or just leave off the index param; default is True
```

Contents of example.csv:

```
,A,B,C
0,1.76405234597,0.400157208367,0.978737984106
1,2.2408931992,1.86755799015,-0.977277879876
2,0.950088417526,-0.151357208298,-0.103218851794
3,0.410598501938,0.144043571161,1.45427350696
4,0.761037725147,0.121675016493,0.443863232745
```

Also note that you can remove the header if it's not needed with `header=False`. This is the simplest output:

```
df.to_csv('example.csv', index=False, header=False)
```

Contents of example.csv:

```
1.76405234597,0.400157208367,0.978737984106
2.2408931992,1.86755799015,-0.977277879876
0.950088417526,-0.151357208298,-0.103218851794
0.410598501938,0.144043571161,1.45427350696
0.761037725147,0.121675016493,0.443863232745
```

The delimiter can be set by `sep=` argument, although the standard separator for csv files is `','` .

`df.to_csv('example.csv', index=False, header=False, sep='\t')`

```
1.76405234597    0.400157208367    0.978737984106
2.2408931992    1.86755799015    -0.977277879876
0.950088417526    -0.151357208298    -0.103218851794
0.410598501938    0.144043571161    1.45427350696
0.761037725147    0.121675016493    0.443863232745
```

Save Pandas DataFrame from list to dicts to csv with no index and with data encoding

```
import pandas as pd
data = [
    {'name': 'Daniel', 'country': 'Uganda'},
    {'name': 'Yao', 'country': 'China'},
    {'name': 'James', 'country': 'Colombia'},
]
df = pd.DataFrame(data)
filename = 'people.csv'
```

```
df.to_csv(filename, index=False, encoding='utf-8')
```

Read Save pandas dataframe to a csv file online: https://riptutorial.com/pandas/topic/1558/save-pandas-dataframe-to-a-csv-file

Chapter 36: Series

Examples

Simple Series creation examples

A series is a one-dimension data structure. It's a bit like a supercharged array, or a dictionary.

```
import pandas as pd

s = pd.Series([10, 20, 30])

>>> s
0    10
1    20
2    30
dtype: int64
```

Every value in a series has an index. By default, the indices are integers, running from 0 to the series length minus 1. In the example above you can see the indices printed to the left of the values.

You can specify your own indices:

```
s2 = pd.Series([1.5, 2.5, 3.5], index=['a', 'b', 'c'], name='my_series')

>>> s2
a    1.5
b    2.5
c    3.5
Name: my_series, dtype: float64

s3 = pd.Series(['a', 'b', 'c'], index=list('ABC'))

>>> s3
A    a
B    b
C    c
dtype: object
```

Series with datetime

```
import pandas as pd
import numpy as np

np.random.seed(0)
rng = pd.date_range('2015-02-24', periods=5, freq='T')
s = pd.Series(np.random.randn(len(rng)), index=rng)
print (s)

2015-02-24 00:00:00    1.764052
2015-02-24 00:01:00    0.400157
```

```
2015-02-24 00:02:00    0.978738
2015-02-24 00:03:00    2.240893
2015-02-24 00:04:00    1.867558
Freq: T, dtype: float64

rng = pd.date_range('2015-02-24', periods=5, freq='T')
s1 = pd.Series(rng)
print (s1)

0   2015-02-24 00:00:00
1   2015-02-24 00:01:00
2   2015-02-24 00:02:00
3   2015-02-24 00:03:00
4   2015-02-24 00:04:00
dtype: datetime64[ns]
```

A few quick tips about Series in Pandas

Let us assume we have the following Series:

```
>>> import pandas as pd
>>> s = pd.Series([1, 4, 6, 3, 8, 7, 4, 5])
>>> s
0    1
1    4
2    6
3    3
4    8
5    7
6    4
7    5
dtype: int64
```

Followings are a few simple things which come handy when you are working with Series:

To get the length of s:

```
>>> len(s)
8
```

To access an element in s:

```
>>> s[4]
8
```

To access an element in s using the index:

```
>>> s.loc[2]
6
```

To access a sub-Series inside s:

```
>>> s[1:3]
```

```
1    4
2    6
dtype: int64
```

To get a sub-Series of s with values larger than 5:

```
>>> s[s > 5]
2    6
4    8
5    7
dtype: int64
```

To get the minimum, maximum, mean, and standard deviation:

```
>>> s.min()
1
>>> s.max()
8
>>> s.mean()
4.75
>>> s.std()
2.2519832529192065
```

To convert the Series type to float:

```
>>> s.astype(float)
0    1.0
1    4.0
2    6.0
3    3.0
4    8.0
5    7.0
6    4.0
7    5.0
dtype: float64
```

To get the values in s as a numpy array:

```
>>> s.values
array([1, 4, 6, 3, 8, 7, 4, 5])
```

To make a copy of s:

```
>>> d = s.copy()
>>> d
0    1
1    4
2    6
3    3
4    8
5    7
6    4
7    5
dtype: int64
```

Applying a function to a Series

Pandas provides an effective way to apply a function to every element of a Series and get a new Series. Let us assume we have the following Series:

```
>>> import pandas as pd
>>> s = pd.Series([3, 7, 5, 8, 9, 1, 0, 4])
>>> s
0    3
1    7
2    5
3    8
4    9
5    1
6    0
7    4
dtype: int64
```

and a square function:

```
>>> def square(x):
...     return x*x
```

We can simply apply square to every element of s and get a new Series:

```
>>> t = s.apply(square)
>>> t
0     9
1    49
2    25
3    64
4    81
5     1
6     0
7    16
dtype: int64
```

In some cases it is easier to use a lambda expression:

```
>>> s.apply(lambda x: x ** 2)
0     9
1    49
2    25
3    64
4    81
5     1
6     0
7    16
dtype: int64
```

or we can use any builtin function:

```
>>> q = pd.Series(['Bob', 'Jack', 'Rose'])
>>> q.apply(str.lower)
```

```
0     bob
1    jack
2    rose
dtype: object
```

If all the elements of the Series are strings, there is an easier way to apply string methods:

```
>>> q.str.lower()
0     bob
1    jack
2    rose
dtype: object
>>> q.str.len()
0    3
1    4
2    4
```

Read Series online: https://riptutorial.com/pandas/topic/1898/series

Chapter 37: Shifting and Lagging Data

Examples

Shifting or lagging values in a dataframe

```
import pandas as pd

df = pd.DataFrame({'eggs': [1,2,4,8,], 'chickens': [0,1,2,4,]})

df

#    chickens  eggs
# 0         0     1
# 1         1     2
# 2         2     4
# 3         4     8

df.shift()

#    chickens  eggs
# 0       NaN   NaN
# 1       0.0   1.0
# 2       1.0   2.0
# 3       2.0   4.0

df.shift(-2)

#    chickens  eggs
# 0       2.0   4.0
# 1       4.0   8.0
# 2       NaN   NaN
# 3       NaN   NaN

df['eggs'].shift(1) - df['chickens']

# 0     NaN
# 1     0.0
# 2     0.0
# 3     0.0
```

The first argument to `.shift()` is `periods`, the number of spaces to move the data. If not specified, defaults to `1`.

Read Shifting and Lagging Data online: https://riptutorial.com/pandas/topic/7554/shifting-and-lagging-data

Chapter 38: Simple manipulation of DataFrames

Examples

Delete a column in a DataFrame

There are a couple of ways to delete a column in a DataFrame.

```
import numpy as np
import pandas as pd

np.random.seed(0)

pd.DataFrame(np.random.randn(5, 6), columns=list('ABCDEF'))

print(df)
# Output:
#           A         B         C         D         E         F
# 0 -0.895467  0.386902 -0.510805 -1.180632 -0.028182  0.428332
# 1  0.066517  0.302472 -0.634322 -0.362741 -0.672460 -0.359553
# 2 -0.813146 -1.726283  0.177426 -0.401781 -1.630198  0.462782
# 3 -0.907298  0.051945  0.729091  0.128983  1.139401 -1.234826
# 4  0.402342 -0.684810 -0.870797 -0.578850 -0.311553  0.056165
```

1) Using `del`

```
del df['C']

print(df)
# Output:
#           A         B         D         E         F
# 0 -0.895467  0.386902 -1.180632 -0.028182  0.428332
# 1  0.066517  0.302472 -0.362741 -0.672460 -0.359553
# 2 -0.813146 -1.726283 -0.401781 -1.630198  0.462782
# 3 -0.907298  0.051945  0.128983  1.139401 -1.234826
# 4  0.402342 -0.684810 -0.578850 -0.311553  0.056165
```

2) Using `drop`

```
df.drop(['B', 'E'], axis='columns', inplace=True)
# or df = df.drop(['B', 'E'], axis=1) without the option inplace=True

print(df)
# Output:
#           A         D         F
# 0 -0.895467 -1.180632  0.428332
# 1  0.066517 -0.362741 -0.359553
# 2 -0.813146 -0.401781  0.462782
# 3 -0.907298  0.128983 -1.234826
# 4  0.402342 -0.578850  0.056165
```

3) Using `drop` with column numbers

To use column integer numbers instead of names (remember column indices start at zero):

```
df.drop(df.columns[[0, 2]], axis='columns')

print(df)
# Output:
#           D
# 0 -1.180632
# 1 -0.362741
# 2 -0.401781
# 3  0.128983
# 4 -0.578850
```

Rename a column

```
df = pd.DataFrame({'old_name_1': [1, 2, 3], 'old_name_2': [5, 6, 7]})

print(df)
# Output:
#    old_name_1  old_name_2
# 0           1           5
# 1           2           6
# 2           3           7
```

To rename one or more columns, pass the old names and new names as a dictionary:

```
df.rename(columns={'old_name_1': 'new_name_1', 'old_name_2': 'new_name_2'}, inplace=True)
print(df)
# Output:
#   new_name_1  new_name_2
# 0           1           5
# 1           2           6
# 2           3           7
```

Or a function:

```
df.rename(columns=lambda x: x.replace('old_', '_new'), inplace=True)
print(df)
# Output:
#   new_name_1  new_name_2
# 0           1           5
# 1           2           6
# 2           3           7
```

You can also set `df.columns` as the list of the new names:

```
df.columns = ['new_name_1','new_name_2']
print(df)
# Output:
#   new_name_1  new_name_2
# 0           1           5
# 1           2           6
```

```
# 2           3           7
```

More details can be found here.

Adding a new column

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

print(df)
# Output:
#    A  B
# 0  1  4
# 1  2  5
# 2  3  6
```

Directly assign

```
df['C'] = [7, 8, 9]

print(df)
# Output:
#    A  B  C
# 0  1  4  7
# 1  2  5  8
# 2  3  6  9
```

Add a constant column

```
df['C'] = 1

print(df)

# Output:
#    A  B  C
# 0  1  4  1
# 1  2  5  1
# 2  3  6  1
```

Column as an expression in other columns

```
df['C'] = df['A'] + df['B']

# print(df)
# Output:
#    A  B  C
# 0  1  4  5
# 1  2  5  7
# 2  3  6  9

df['C'] = df['A']**df['B']

print(df)
```

```
# Output:
#    A  B    C
# 0  1  4    1
# 1  2  5   32
# 2  3  6  729
```

Operations are computed component-wise, so if we would have columns as lists

```
a = [1, 2, 3]
b = [4, 5, 6]
```

the column in the last expression would be obtained as

```
c = [x**y for (x,y) in zip(a,b)]

print(c)
# Output:
# [1, 32, 729]
```

Create it on the fly

```
df_means = df.assign(D=[10, 20, 30]).mean()

print(df_means)
# Output:
# A     2.0
# B     5.0
# C     7.0
# D    20.0  # adds a new column D before taking the mean
# dtype: float64
```

add multiple columns

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
df[['A2','B2']] = np.square(df)

print(df)
# Output:
#    A  B  A2  B2
# 0  1  4   1  16
# 1  2  5   4  25
# 2  3  6   9  36
```

add multiple columns on the fly

```
new_df = df.assign(A3=df.A*df.A2, B3=5*df.B)

print(new_df)
# Output:
#    A  B  A2  B2  A3  B3
# 0  1  4   1  16   1  20
```

```
# 1   2  5   4  25   8  25
# 2   3  6   9  36  27  30
```

Locate and replace data in a column

```
import pandas as pd

df = pd.DataFrame({'gender': ["male", "female","female"],
                    'id': [1, 2, 3] })
>>> df
   gender  id
0    male   1
1  female   2
2  female   3
```

To encode the male to 0 and female to 1:

```
df.loc[df["gender"] == "male","gender"] = 0
df.loc[df["gender"] == "female","gender"] = 1

>>> df
       gender  id
    0       0   1
    1       1   2
    2       1   3
```

Adding a new row to DataFrame

Given a DataFrame:

```
s1 = pd.Series([1,2,3])
s2 = pd.Series(['a','b','c'])

df = pd.DataFrame([list(s1), list(s2)],  columns =  ["C1", "C2", "C3"])
print df
```

Output:

```
  C1 C2 C3
0  1  2  3
1  a  b  c
```

Lets add a new row, `[10,11,12]`:

```
df = pd.DataFrame(np.array([[10,11,12]]), \
        columns=["C1", "C2", "C3"]).append(df, ignore_index=True)
print df
```

Output:

```
   C1  C2  C3
0  10  11  12
```

```
1   1   2   3
2   a   b   c
```

Delete / drop rows from DataFrame

let's generate a DataFrame first:

```
df = pd.DataFrame(np.arange(10).reshape(5,2), columns=list('ab'))

print(df)
# Output:
#    a  b
# 0  0  1
# 1  2  3
# 2  4  5
# 3  6  7
# 4  8  9
```

drop rows with indexes: `0` and `4` using `drop([...], inplace=True)` method:

```
df.drop([0,4], inplace=True)

print(df)
# Output
#    a  b
# 1  2  3
# 2  4  5
# 3  6  7
```

drop rows with indexes: `0` and `4` using `df = drop([...])` method:

```
df = pd.DataFrame(np.arange(10).reshape(5,2), columns=list('ab'))

df = df.drop([0,4])

print(df)
# Output:
#    a  b
# 1  2  3
# 2  4  5
# 3  6  7
```

using negative selection method:

```
df = pd.DataFrame(np.arange(10).reshape(5,2), columns=list('ab'))

df = df[~df.index.isin([0,4])]

print(df)
# Output:
#    a  b
# 1  2  3
# 2  4  5
# 3  6  7
```

Reorder columns

```
# get a list of columns
cols = list(df)

# move the column to head of list using index, pop and insert
cols.insert(0, cols.pop(cols.index('listing')))

# use ix to reorder
df2 = df.ix[:, cols]
```

Read Simple manipulation of DataFrames online: https://riptutorial.com/pandas/topic/6694/simple-manipulation-of-dataframes

Chapter 39: String manipulation

Examples

Regular expressions

```
# Extract strings with a specific regex
df= df['col_name'].str.extract[r'[Aa-Zz]']

# Replace strings within a regex
df['col_name'].str.replace('Replace this', 'With this')
```

For information on how to match strings using regex, see Getting started with Regular Expressions
.

Slicing strings

Strings in a Series can be sliced using `.str.slice()` method, or more conveniently, using brackets (`.str[]`).

```
In [1]: ser = pd.Series(['Lorem ipsum', 'dolor sit amet', 'consectetur adipiscing elit'])
In [2]: ser
Out[2]:
0                  Lorem ipsum
1               dolor sit amet
2    consectetur adipiscing elit
dtype: object
```

Get the first character of each string:

```
In [3]: ser.str[0]
Out[3]:
0    L
1    d
2    c
dtype: object
```

Get the first three characters of each string:

```
In [4]: ser.str[:3]
Out[4]:
0    Lor
1    dol
2    con
dtype: object
```

Get the last character of each string:

```
In [5]: ser.str[-1]
```

```
Out[5]:
0    m
1    t
2    t
dtype: object
```

Get the last three characters of each string:

```
In [6]: ser.str[-3:]
Out[6]:
0    sum
1    met
2    lit
dtype: object
```

Get the every other character of the first 10 characters:

```
In [7]: ser.str[:10:2]
Out[7]:
0    Lrmis
1    dlrst
2    cnett
dtype: object
```

Pandas behaves similarly to Python when handling slices and indices. For example, if an index is outside the range, Python raises an error:

```
In [8]:'Lorem ipsum'[12]
# IndexError: string index out of range
```

However, if a slice is outside the range, an empty string is returned:

```
In [9]: 'Lorem ipsum'[12:15]
Out[9]: ''
```

Pandas returns NaN when an index is out of range:

```
In [10]: ser.str[12]
Out[10]:
0    NaN
1      e
2      a
dtype: object
```

And returns an empty string if a slice is out of range:

```
In [11]: ser.str[12:15]
Out[11]:
0
1     et
2    adi
dtype: object
```

Checking for contents of a string

`str.contains()` method can be used to check if a pattern occurs in each string of a Series. `str.startswith()` and `str.endswith()` methods can also be used as more specialized versions.

```
In [1]: animals = pd.Series(['cat', 'dog', 'bear', 'cow', 'bird', 'owl', 'rabbit', 'snake'])
```

Check if strings contain the letter 'a':

```
In [2]: animals.str.contains('a')
Out[2]:
0      True
1     False
2      True
3     False
4     False
5     False
6      True
7      True
8      True
dtype: bool
```

This can be used as a boolean index to return only the animals containing the letter 'a':

```
In [3]: animals[animals.str.contains('a')]
Out[3]:
0       cat
2      bear
6    rabbit
7     snake
dtype: object
```

`str.startswith` and `str.endswith` methods work similarly, but they also accept tuples as inputs.

```
In [4]: animals[animals.str.startswith(('b', 'c'))]
# Returns animals starting with 'b' or 'c'
Out[4]:
0     cat
2    bear
3     cow
4    bird
dtype: object
```

Capitalization of strings

```
In [1]: ser = pd.Series(['lORem ipSuM', 'Dolor sit amet', 'Consectetur Adipiscing Elit'])
```

Convert all to uppercase:

```
In [2]: ser.str.upper()
Out[2]:
0                     LOREM IPSUM
```

```
1                  DOLOR SIT AMET
2    CONSECTETUR ADIPISCING ELIT
dtype: object
```

All lowercase:

```
In [3]: ser.str.lower()
Out[3]:
0                     lorem ipsum
1                  dolor sit amet
2    consectetur adipiscing elit
dtype: object
```

Capitalize the first character and lowercase the remaining:

```
In [4]: ser.str.capitalize()
Out[4]:
0                     Lorem ipsum
1                  Dolor sit amet
2    Consectetur adipiscing elit
dtype: object
```

Convert each string to a titlecase (capitalize the first character of each word in each string, lowercase the remaining):

```
In [5]: ser.str.title()
Out[5]:
0                     Lorem Ipsum
1                  Dolor Sit Amet
2    Consectetur Adipiscing Elit
dtype: object
```

Swap cases (convert lowercase to uppercase and vice versa):

```
In [6]: ser.str.swapcase()
Out[6]:
0                     LorEM IPsUm
1                  dOLOR SIT AMET
2    cONSECTETUR aDIPISCING eLIT
dtype: object
```

Aside from these methods that change the capitalization, several methods can be used to check the capitalization of strings.

```
In [7]: ser = pd.Series(['LOREM IPSUM', 'dolor sit amet', 'Consectetur Adipiscing Elit'])
```

Check if it is all lowercase:

```
In [8]: ser.str.islower()
Out[8]:
0    False
1     True
2    False
```

```
dtype: bool
```

Is it all uppercase:

```
In [9]: ser.str.isupper()
Out[9]:
0     True
1    False
2    False
dtype: bool
```

Is it a titlecased string:

```
In [10]: ser.str.istitle()
Out[10]:
0    False
1    False
2     True
dtype: bool
```

Read String manipulation online: https://riptutorial.com/pandas/topic/2372/string-manipulation

Chapter 40: Using .ix, .iloc, .loc, .at and .iat to access a DataFrame

Examples

Using .iloc

.iloc uses integers to read and write data to a DataFrame.

First, let's create a DataFrame:

```
df = pd.DataFrame({'one': [1, 2, 3, 4, 5],
                   'two': [6, 7, 8, 9, 10],
              }, index=['a', 'b', 'c', 'd', 'e'])
```

This DataFrame looks like:

```
   one  two
a    1    6
b    2    7
c    3    8
d    4    9
e    5   10
```

Now we can use .iloc to read and write values. Let's read the first row, first column:

```
print df.iloc[0, 0]
```

This will print out:

```
1
```

We can also set values. Lets set the second column, second row to something new:

```
df.iloc[1, 1] = '21'
```

And then have a look to see what happened:

```
print df

   one two
a    1   6
b    2  21
c    3   8
d    4   9
e    5  10
```

Using .loc

.loc uses **labels** to read and write data.

Let's setup a DataFrame:

```
df = pd.DataFrame({'one': [1, 2, 3, 4, 5],
                   'two': [6, 7, 8, 9, 10],
                  }, index=['a', 'b', 'c', 'd', 'e'])
```

Then we can print the DataFrame to have a look at the shape:

```
print df
```

This will output

```
   one  two
a    1    6
b    2    7
c    3    8
d    4    9
e    5   10
```

We use the column and row **labels** to access data with .loc. Let's set row 'c', column 'two' to the value 33:

```
df.loc['c', 'two'] = 33
```

This is what the DataFrame now looks like:

```
   one  two
a    1    6
b    2    7
c    3   33
d    4    9
e    5   10
```

Of note, using `df['two'].loc['c'] = 33` may not report a warning, and may even work, however, using `df.loc['c', 'two']` is guaranteed to work correctly, while the former is not.

We can read slices of data, for example

```
print df.loc['a':'c']
```

will print rows a to c. This is inclusive.

```
   one  two
a    1    6
b    2    7
c    3    8
```

And finally, we can do both together:

```
print df.loc['b':'d', 'two']
```

Will output rows b to c of column 'two'. Notice that the column label is not printed.

```
b    7
c    8
d    9
```

If .loc is supplied with an integer argument that is not a label it reverts to integer indexing of axes (the behaviour of .iloc). This makes mixed label and integer indexing possible:

```
df.loc['b', 1]
```

will return the value in 2nd column (index starting at 0) in row 'b':

```
7
```

Read Using .ix, .iloc, .loc, .at and .iat to access a DataFrame online:
https://riptutorial.com/pandas/topic/7074/using--ix---iloc---loc---at-and--iat-to-access-a-dataframe

Chapter 41: Working with Time Series

Examples

Creating Time Series

Here is how to create a simple Time Series.

```
import pandas as pd
import numpy as np

# The number of sample to generate
nb_sample = 100

# Seeding to obtain a reproductible dataset
np.random.seed(0)

se = pd.Series(np.random.randint(0, 100, nb_sample),
                  index = pd.date_range(start = pd.to_datetime('2016-09-24'),
                                        periods = nb_sample, freq='D'))
se.head(2)

# 2016-09-24    44
# 2016-09-25    47

se.tail(2)

# 2016-12-31    85
# 2017-01-01    48
```

Partial String Indexing

A very handy way to subset Time Series is to use **partial string indexing**. It permits to select range of dates with a clear syntax.

Getting Data

We are using the dataset in the Creating Time Series example

Displaying head and tail to see the boundaries

```
se.head(2).append(se.tail(2))

# 2016-09-24    44
# 2016-09-25    47
# 2016-12-31    85
# 2017-01-01    48
```

Subsetting

Now we can subset by year, month, day very intuitively.

By year

```
se['2017']

# 2017-01-01    48
```

By month

```
se['2017-01']

# 2017-01-01    48
```

By day

```
se['2017-01-01']

# 48
```

With a range of year, month, day according to your needs.

```
se['2016-12-31':'2017-01-01']

# 2016-12-31    85
# 2017-01-01    48
```

pandas also provides a dedicated `truncate` function for this usage through the `after` and `before` parameters -- but I think it's less clear.

```
se.truncate(before='2017')

# 2017-01-01    48

se.truncate(before='2016-12-30', after='2016-12-31')

# 2016-12-30    13
# 2016-12-31    85
```

Read Working with Time Series online: https://riptutorial.com/pandas/topic/7029/working-with-time-series

Credits

S. No	Chapters	Contributors
1	Getting started with pandas	Alexander, Andy Hayden, ayhan, Bryce Frank, Community, hashcode55, Nikita Pestrov, user2314737
2	Analysis: Bringing it all together and making decisions	piRSquared
3	Appending to DataFrame	shahins
4	Boolean indexing of dataframes	firelynx
5	Categorical data	jezrael, Julien Marrec
6	Computational Tools	Ami Tavory
7	Creating DataFrames	Ahamed Mustafa M, Alexander, ayhan, Ayush Kumar Singh, bernie, Gal Dreiman, Geeklhem, Gorkem Ozkaya, jasimpson, jezrael, JJD, Julien Marrec, MaxU, Merlin, pylang, Romain, SerialDev, user2314737, vaerek, ysearka
8	Cross sections of different axes with MultiIndex	Julien Marrec
9	Data Types	Andy Hayden, ayhan, firelynx, jezrael
10	Dealing with categorical variables	Gorkem Ozkaya
11	Duplicated data	ayhan, Ayush Kumar Singh, bee-sting, jezrael
12	Getting information about DataFrames	Alexander, ayhan, Ayush Kumar Singh, bernie, Romain, ysearka
13	Gotchas of pandas	vlad.rad
14	Graphs and Visualizations	Ami Tavory, Nikita Pestrov, Scimonster

15	Grouping Data	Andy Hayden, ayhan, danio, Geeklhem, jezrael, ℕʘʘḆḽḘ, QM.py, Romain, user2314737
16	Grouping Time Series Data	ayhan, piRSquared
17	Holiday Calendars	Romain
18	Indexing and selecting data	amin, Andy Hayden, ayhan, double0darbo, jasimpson, jezrael, Joseph Dasenbrock, MaxU, Merlin, piRSquared, SerialDev, user2314737
19	IO for Google BigQuery	ayhan, tworec
20	JSON	PinoSan, SerialDev, user2314737
21	Making Pandas Play Nice With Native Python Datatypes	DataSwede
22	Map Values	EdChum, Fabio Lamanna
23	Merge, join, and concatenate	ayhan, Josh Garlitos, MaThMaX, MaxU, piRSquared, SerialDev, varunsinghal
24	Meta: Documentation Guidelines	Andy Hayden, ayhan, Stephen Leppik
25	Missing Data	Andy Hayden, ayhan, EdChum, jezrael, Zdenek
26	MultiIndex	Andy Hayden, benten, danielhadar, danio, Pedro M Duarte
27	Pandas Datareader	Alexander, MaxU
28	Pandas IO tools (reading and saving data sets)	amin, Andy Hayden, bernie, Fabich, Gal Dreiman, jezrael, João Almeida, Julien Spronck, MaxU, Nikita Pestrov, SerialDev, user2314737
29	pd.DataFrame.apply	ptsw, Romain
30	Read MySQL to DataFrame	andyabel, rrawat
31	Read SQL Server to Dataframe	bernie, SerialDev
32	Reading files into	Arthur Camara, bee-sting, Corey Petty, Sirajus Salayhin

	pandas DataFrame	
33	Resampling	jezrael
34	Reshaping and pivoting	Albert Camps, ayhan, bernie, DataSwede, jezrael, MaxU, Merlin
35	Save pandas dataframe to a csv file	amin, bernie, eraoul, Gal Dreiman, maxliving, Musafir Safwan, Nikita Pestrov, Olel Daniel, Stephan
36	Series	Alexander, daphshez, EdChum, jezrael, shahins
37	Shifting and Lagging Data	ASGM
38	Simple manipulation of DataFrames	Alexander, ayhan, Ayush Kumar Singh, Gal Dreiman, Geeklhem, MaxU, paulo.filip3, R.M., SerialDev, user2314737, ysearka
39	String manipulation	ayhan, mnoronha, SerialDev
40	Using .ix, .iloc, .loc, .at and .iat to access a DataFrame	bee-sting, DataSwede, farleytpm
41	Working with Time Series	Romain