In addition to what’s in Anaconda, this lecture will need the following libraries:

In [1]: !pip install --upgrade pandas-datareader

2 Overview

Pandas is a package of fast, efficient data analysis tools for Python.

Its popularity has surged in recent years, coincident with the rise of fields such as data science and machine learning.

Here’s a popularity comparison over time against STATA, SAS, and dplyr courtesy of Stack Overflow Trends
Just as NumPy provides the basic array data type plus core array operations, pandas

1. defines fundamental structures for working with data and

2. endows them with methods that facilitate operations such as

   • reading in data
   • adjusting indices
   • working with dates and time series
   • sorting, grouping, re-ordering and general data munging Section ??
   • dealing with missing values, etc., etc.

More sophisticated statistical functionality is left to other packages, such as statsmodels and scikit-learn, which are built on top of pandas.

This lecture will provide a basic introduction to pandas.

Throughout the lecture, we will assume that the following imports have taken place

```python
In [2]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import requests
```

## 3 Series

Two important data types defined by pandas are **Series** and **DataFrame**.

You can think of a **Series** as a “column” of data, such as a collection of observations on a single variable.

A **DataFrame** is an object for storing related columns of data.

Let’s start with Series
In [3]: s = pd.Series(np.random.randn(4), name='daily returns')
   s

Out[3]:  0   -0.308441
       1   -1.174221
       2    0.318572
       3   -0.430371
Name: daily returns, dtype: float64

Here you can imagine the indices 0, 1, 2, 3 as indexing four listed companies, and the values being daily returns on their shares.

Pandas Series are built on top of NumPy arrays and support many similar operations

In [4]: s * 100

Out[4]:  0   -30.844062
       1   -117.422073
       2    31.857166
       3   -43.037064
Name: daily returns, dtype: float64

In [5]: np.abs(s)

Out[5]:  0    0.308441
       1   1.174221
       2    0.318572
       3    0.430371
Name: daily returns, dtype: float64

But Series provide more than NumPy arrays.
Not only do they have some additional (statistically oriented) methods

In [6]: s.describe()

Out[6]:  count   4.000000
        mean  -0.398615
        std   0.612389
       min   -1.174221
      25%    -0.616333
      50%   -0.369406
     75%    -0.151688
        max   0.318572
Name: daily returns, dtype: float64

But their indices are more flexible

In [7]: s.index = ['AMZN', 'AAPL', 'MSFT', 'GOOG']
   s

3
Out[7]:
AMZN   -0.308441
AAPL   -1.174221
MSFT   0.318572
GOOG   -0.430371
Name: daily returns, dtype: float64

Viewed in this way, Series are like fast, efficient Python dictionaries (with the restriction that the items in the dictionary all have the same type—in this case, floats).

In fact, you can use much of the same syntax as Python dictionaries.

```
In [8]: s['AMZN']
```

Out[8]: -0.3084406210440365

```
In [9]: s['AMZN'] = 0
```

```
Out[9]: AMZN 0.000000
    AAPL -1.174221
    MSFT 0.318572
    GOOG -0.430371
Name: daily returns, dtype: float64
```

```
In [10]: 'AAPL' in s
```

Out[10]: True

## 4 DataFrames

While a Series is a single column of data, a DataFrame is several columns, one for each variable.

In essence, a DataFrame in pandas is analogous to a (highly optimized) Excel spreadsheet.

Thus, it is a powerful tool for representing and analyzing data that are naturally organized into rows and columns, often with descriptive indexes for individual rows and individual columns.

Let’s look at an example that reads data from the CSV file `pandas/data/test_pwt.csv` that can be downloaded here.

Here’s the content of test_pwt.csv

```
"country","country isocode","year","POP","XRAT","tcgdp","cc","cg"
"Argentina","ARG","2000","3735.653","0.9995","295072.21869","75.716805379","5.5788042896"
"Australia","AUS","2000","19053.186","1.72483","541804.6521","67.759025993","6.7200975332"
"India","IND","2000","1006300.297","44.9416","1728144.3748","64.575551328","14.072205773"
"Israel","ISR","2000","6114.57","4.07733","129253.89423","64.436450847","10.266688415"
"Malawi","MWI","2000","11801.505","59.543808333","5026.2217836","74.707624181","11.658954494"
"South Africa","ZAF","2000","45064.098","6.93983","227424.36949","72.718710427","5.7265463933"
"United States","USA","2000","282171.957","1","9898700","72.347054303","6.0324539789"
"Uruguay","URY","2000","3219.793","12.099591667","25255.961693","78.978740282","5.108067988"
```

4
Supposing you have this data saved as `test_pwt.csv` in the present working directory (type `%pwd` in Jupyter to see what this is), it can be read in as follows:

In [11]:
   df = pd.read_csv('https://raw.githubusercontent.com/QuantEcon/lecture-source-py/master/source/_static/lecture_specific/pandas/data/test_pwt.csv')
   type(df)

Out[11]: pandas.core.frame.DataFrame

In [12]:
   df

Out[12]:
   [country    country isocode year POP XRAT tcgdp  
    0 Argentina ARG 2000 37335.653 0.999500 2.950722e+05
    1 Australia AUS 2000 19053.186 1.724830 5.418047e+05
    2 India IND 2000 1006300.297 44.941600 1.728144e+06
    3 Israel ISR 2000 6114.570 4.077330 1.292539e+05
    4 Malawi MWI 2000 11801.505 59.543808 5.026222e+03
    5 South Africa ZAF 2000 45064.098 6.939830 2.272424e+05
    6 United States USA 2000 282171.957 1.000000 9.898700e+06
    7 Uruguay URY 2000 3219.793 12.099592 2.525596e+04

   cc    cg
   0 75.716805 5.578804
   1 67.759026 6.720098
   2 64.575551 14.072206
   3 64.436451 10.266688
   4 74.707624 11.658954
   5 72.718710 5.726546
   6 72.347054 6.032454
   7 78.978740 5.108068

We can select particular rows using standard Python array slicing notation

In [13]:
   df[2:5]

Out[13]:
   [country    country isocode year POP XRAT tcgdp  
    2 India IND 2000 1006300.297 44.941600 1.728144e+06
    3 Israel ISR 2000 6114.570 4.077330 1.292539e+05
    4 Malawi MWI 2000 11801.505 59.543808 5.026222e+03

   cc    cg
   2 64.575551 14.072206
   3 64.436451 10.266688
   4 74.707624 11.658954

To select columns, we can pass a list containing the names of the desired columns represented as strings

In [14]:
   df[['country', 'tcgdp']]
To select both rows and columns using integers, the `iloc` attribute should be used with the format `.iloc[rows, columns]`

```python
In [15]: df.iloc[2:5, 0:4]
```

Out[15]:
```
          country  isocode  year  POP
     2    India   IND  2000  1006300.297
     3      Israel  ISR  2000    6114.570
     4  Malawi    MWI  2000  11801.505
```

To select rows and columns using a mixture of integers and labels, the `loc` attribute can be used in a similar way

```python
In [16]: df.loc[df.index[2:5], ['country', 'tcgdp']]
```

Out[16]:
```
   country   tcgdp
     India  1.728144e+06
     Israel  1.292539e+05
     Malawi  5.026222e+03
```

Let’s imagine that we’re only interested in population (POP) and total GDP (tcgdp).

One way to strip the data frame `df` down to only these variables is to overwrite the dataframe using the selection method described above

```python
In [17]: df = df[['country', 'POP', 'tcgdp']]
df
```

Out[17]:
```
          country  POP  tcgdp
     0  Argentina  37335.653  2.950722e+05
     1  Australia  19053.186   5.418047e+05
     2    India  1006300.297  1.728144e+06
     3      Israel  6114.570   1.292539e+05
     4  Malawi  11801.505  5.026222e+03
     5  South Africa  45064.098  2.272424e+05
     6  United States  282171.957   9.898700e+06
     7    Uruguay   3219.793  2.525596e+04
```

Here the index 0, 1, ..., 7 is redundant because we can use the country names as an index.

To do this, we set the index to be the `country` variable in the dataframe.
```python
In [18]: df = df.set_index('country')
   ...:
   ...:
   ...: df
   ...:
         POP  tcgdp
    country
        Argentina 37335.653  2.950722e+05
       Australia 19053.186  5.418047e+05
        India 1006300.297  1.728144e+06
        Israel  6114.570  1.292539e+05
       Malawi  11801.505  5.026222e+03
     South Africa  45064.098  2.272424e+05
     United States 282171.957  9.898700e+06
      Uruguay  3219.793  2.525596e+04
   ...:
   ...:
   ...:
   ...:
   ...:
   ...:
```  

Let's give the columns slightly better names

```python
In [19]: df.columns = 'population', 'total GDP'
   ...: df
   ...:
         population  total GDP
    country
        Argentina  37335.653  2.950722e+05
       Australia  19053.186  5.418047e+05
        India  1006300.297  1.728144e+06
        Israel  6114.570  1.292539e+05
       Malawi  11801.505  5.026222e+03
     South Africa  45064.098  2.272424e+05
     United States  282171.957  9.898700e+06
      Uruguay  3219.793  2.525596e+04
   ...:
   ...:
   ...:
   ...:
   ...:
   ...:
```  

Population is in thousands, let's revert to single units

```python
In [20]: df['population'] = df['population'] * 1e3
   ...: df
   ...:
         population  total GDP
    country
        Argentina  3.733565e+07  2.950722e+05
       Australia  1.905319e+07  5.418047e+05
        India  1.006300e+09  1.728144e+06
        Israel  6.114570e+06  1.292539e+05
       Malawi  1.180150e+07  5.026222e+03
     South Africa  4.506409e+07  2.272424e+05
     United States  2.821720e+08  9.898700e+06
      Uruguay  3.219793e+06  2.525596e+04
   ...:
   ...:
   ...:
   ...:
   ...:
   ...:
```  

Next, we're going to add a column showing real GDP per capita, multiplying by 1,000,000 as we go because total GDP is in millions

```python
In [21]: df['GDP percap'] = df['total GDP'] * 1e6 / df['population']
   ...: df
   ...:
         GDP percap  population  total GDP
    country
        Argentina  3.733565e+07  3.733565e+07  2.950722e+05
       Australia  1.905319e+07  1.905319e+07  5.418047e+05
        India  1.006300e+09  1.006300e+09  1.728144e+06
        Israel  6.114570e+06  6.114570e+06  1.292539e+05
       Malawi  1.180150e+07  1.180150e+07  5.026222e+03
     South Africa  4.506409e+07  4.506409e+07  2.272424e+05
     United States  2.821720e+08  2.821720e+08  9.898700e+06
      Uruguay  3.219793e+06  3.219793e+06  2.525596e+04
   ...:
   ...:
   ...:
   ...:
   ...:
   ...:
```
One of the nice things about pandas `DataFrame` and `Series` objects is that they have methods for plotting and visualization that work through Matplotlib.

For example, we can easily generate a bar plot of GDP per capita:

```
In [22]: ax = df['GDP percap'].plot(kind='bar')
ax.set_xlabel('country', fontsize=12)
ax.set_ylabel('GDP per capita', fontsize=12)
plt.show()
```

At the moment the data frame is ordered alphabetically on the countries—let’s change it to GDP per capita:

```
In [23]: df = df.sort_values(by='GDP percap', ascending=False)
df
```
Out[23]:

<table>
<thead>
<tr>
<th>country</th>
<th>population</th>
<th>total GDP</th>
<th>GDP per cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>2.821720e+08</td>
<td>9.898700e+06</td>
<td>35080.381854</td>
</tr>
<tr>
<td>Australia</td>
<td>1.905319e+07</td>
<td>5.418047e+05</td>
<td>28436.433261</td>
</tr>
<tr>
<td>Israel</td>
<td>6.114570e+06</td>
<td>1.292539e+05</td>
<td>21138.672749</td>
</tr>
<tr>
<td>Argentina</td>
<td>3.733565e+07</td>
<td>2.950722e+05</td>
<td>7903.229085</td>
</tr>
<tr>
<td>Uruguay</td>
<td>3.219793e+06</td>
<td>2.525596e+04</td>
<td>7843.976029</td>
</tr>
<tr>
<td>South Africa</td>
<td>4.506410e+07</td>
<td>2.272424e+05</td>
<td>5042.647686</td>
</tr>
<tr>
<td>India</td>
<td>1.006300e+09</td>
<td>1.728144e+06</td>
<td>1717.324719</td>
</tr>
<tr>
<td>Malawi</td>
<td>1.180150e+07</td>
<td>5.026222e+03</td>
<td>425.896679</td>
</tr>
</tbody>
</table>

Plotting as before now yields

In [24]: ax = df['GDP per cap'].plot(kind='bar')
   : ax.set_xlabel('country', fontsize=12)
   : ax.set_ylabel('GDP per capita', fontsize=12)
   : plt.show()

5 On-Line Data Sources

Python makes it straightforward to query online databases programmatically.

An important database for economists is FRED — a vast collection of time series data maintained by the St. Louis Fed.
For example, suppose that we are interested in the unemployment rate. Via FRED, the entire series for the US civilian unemployment rate can be downloaded directly by entering this URL into your browser (note that this requires an internet connection)

https://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv

(Equivalently, click here: https://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv)

This request returns a CSV file, which will be handled by your default application for this class of files.

Alternatively, we can access the CSV file from within a Python program.

This can be done with a variety of methods.

We start with a relatively low-level method and then return to pandas.

### 5.1 Accessing Data with requests

One option is to use requests, a standard Python library for requesting data over the Internet.

To begin, try the following code on your computer

```python
In [25]: r = requests.get('http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv')
```

If there’s no error message, then the call has succeeded.

If you do get an error, then there are two likely causes

1. You are not connected to the Internet — hopefully, this isn’t the case.
2. Your machine is accessing the Internet through a proxy server, and Python isn’t aware of this.

In the second case, you can either

- switch to another machine
- solve your proxy problem by reading the documentation

Assuming that all is working, you can now proceed to use the `source` object returned by the call

```python
requests.get('http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv')
```

```python
In [26]: url = 'http://research.stlouisfed.org/fred2/series/UNRATE/downloaddata/UNRATE.csv'
    
    source = requests.get(url).content.decode().split("\n")
    
    source[0]
```

```
'DATE,VALUE\r'
```

Out[26]: 'DATE,VALUE\r'
We could now write some additional code to parse this text and store it as an array. But this is unnecessary — pandas’ `read_csv` function can handle the task for us. We use `parse_dates=True` so that pandas recognizes our dates column, allowing for simple date filtering.

```python
In [29]: data = pd.read_csv(url, index_col=0, parse_dates=True)
```

The data has been read into a pandas DataFrame called `data` that we can now manipulate in the usual way.

```python
In [30]: type(data)
Out[30]: pandas.core.frame.DataFrame
```

```python
In [31]: data.head()  # A useful method to get a quick look at a data frame
```

```python
Out[31]:
<table>
<thead>
<tr>
<th>DATE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1948-01-01</td>
<td>3.4</td>
</tr>
<tr>
<td>1948-02-01</td>
<td>3.8</td>
</tr>
<tr>
<td>1948-03-01</td>
<td>4.0</td>
</tr>
<tr>
<td>1948-04-01</td>
<td>3.9</td>
</tr>
<tr>
<td>1948-05-01</td>
<td>3.5</td>
</tr>
</tbody>
</table>
```

```python
In [32]: pd.set_option('precision', 1)
data.describe()  # Your output might differ slightly
```

```python
Out[32]:
<table>
<thead>
<tr>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
</tr>
<tr>
<td>mean</td>
</tr>
<tr>
<td>std</td>
</tr>
<tr>
<td>min</td>
</tr>
<tr>
<td>25%</td>
</tr>
<tr>
<td>50%</td>
</tr>
<tr>
<td>75%</td>
</tr>
<tr>
<td>max</td>
</tr>
</tbody>
</table>
```

We can also plot the unemployment rate from 2006 to 2012 as follows.

```python
In [33]: ax = data['2006':'2012'].plot(title='US Unemployment Rate', legend=False)
ax.set_xlabel('year', fontsize=12)
ax.set_ylabel('%', fontsize=12)
plt.show()
```
Note that pandas offers many other file type alternatives.

Pandas has a wide variety of top-level methods that we can use to read, excel, json, parquet or plug straight into a database server.

5.2 Using pandas_datareader to Access Data

The maker of pandas has also authored a library called pandas_datareader that gives programmatic access to many data sources straight from the Jupyter notebook.

While some sources require an access key, many of the most important (e.g., FRED, OECD, EUROSTAT and the World Bank) are free to use.

For now let’s work through one example of downloading and plotting data — this time from the World Bank.

The World Bank collects and organizes data on a huge range of indicators.

For example, here’s some data on government debt as a ratio to GDP.

The next code example fetches the data for you and plots time series for the US and Australia

In [34]: from pandas_datareader import wb

    govt_debt = wb.download(indicator='GC.DOD.TOTL.GD.ZS', country=['US', 'AU'],
                   start=2005, end=2016).stack().unstack()[0]
    ind = govt_debt.index.droplevel(-1)
    govt_debt.index = ind
    ax = govt_debt.plot(lw=2)
ax.set_xlabel('year', fontsize=12)
plt.title("Government Debt to GDP (%)")
plt.show()

The documentation provides more details on how to access various data sources.

6 Exercises

6.1 Exercise 1

With these imports:

In [35]: import datetime as dt
   from pandas_datareader import data

Write a program to calculate the percentage price change over 2019 for the following shares:

In [36]: ticker_list = {'INTC': 'Intel',
   'MSFT': 'Microsoft',
   'IBM': 'IBM',
   'BHP': 'BHP',
   'TM': 'Toyota',
   'AAPL': 'Apple',
   'AMZN': 'Amazon',
   'BA': 'Boeing',
   'QCOM': 'Qualcomm',
   'KO': 'Coca-Cola',
   'GOOG': 'Google',
   'SNE': 'Sony',
   'PTR': 'PetroChina'}
Here’s the first part of the program

```python
In [37]: def read_data(ticker_list,
                start=dt.datetime(2019, 1, 2),
                end=dt.datetime(2019, 12, 31)):
    
    """
    This function reads in closing price data from Yahoo
    for each tick in the ticker_list.
    """

    ticker = pd.DataFrame()

    for tick in ticker_list:
        prices = data.DataReader(tick, 'yahoo', start, end)
        closing_prices = prices['Close']
        ticker[tick] = closing_prices

    return ticker

ticker = read_data(ticker_list)
```

Complete the program to plot the result as a bar graph like this one:

![Bar graph showing percentage change in price](image)

### 7 Solutions

#### 7.1 Exercise 1

There are a few ways to approach this problem using Pandas to calculate the percentage change.

First, you can extract the data and perform the calculation such as:
In [38]:
p1 = ticker.iloc[0]  # Get the first set of prices as a Series
p2 = ticker.iloc[-1]  # Get the last set of prices as a Series
price_change = (p2 - p1) / p1 * 100
price_change

Out[38]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>INTC</td>
<td>27.1</td>
</tr>
<tr>
<td>MSFT</td>
<td>56.0</td>
</tr>
<tr>
<td>IBM</td>
<td>16.3</td>
</tr>
<tr>
<td>BHP</td>
<td>14.3</td>
</tr>
<tr>
<td>TM</td>
<td>20.9</td>
</tr>
<tr>
<td>AAPL</td>
<td>85.9</td>
</tr>
<tr>
<td>AMZN</td>
<td>20.1</td>
</tr>
<tr>
<td>BA</td>
<td>0.6</td>
</tr>
<tr>
<td>QCOM</td>
<td>53.7</td>
</tr>
<tr>
<td>KO</td>
<td>17.9</td>
</tr>
<tr>
<td>GOOG</td>
<td>27.8</td>
</tr>
<tr>
<td>SNE</td>
<td>39.6</td>
</tr>
<tr>
<td>PTR</td>
<td>-17.4</td>
</tr>
</tbody>
</table>

dtype: float64

Alternatively you can use an inbuilt method `pct_change` and configure it to perform the correct calculation using `periods` argument.

In [39]:
change = ticker.pct_change(periods=len(ticker)-1, axis='rows')*100
price_change = change.iloc[-1]
price_change

Out[39]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>INTC</td>
<td>27.1</td>
</tr>
<tr>
<td>MSFT</td>
<td>56.0</td>
</tr>
<tr>
<td>IBM</td>
<td>16.3</td>
</tr>
<tr>
<td>BHP</td>
<td>14.3</td>
</tr>
<tr>
<td>TM</td>
<td>20.9</td>
</tr>
<tr>
<td>AAPL</td>
<td>85.9</td>
</tr>
<tr>
<td>AMZN</td>
<td>20.1</td>
</tr>
<tr>
<td>BA</td>
<td>0.6</td>
</tr>
<tr>
<td>QCOM</td>
<td>53.7</td>
</tr>
<tr>
<td>KO</td>
<td>17.9</td>
</tr>
<tr>
<td>GOOG</td>
<td>27.8</td>
</tr>
<tr>
<td>SNE</td>
<td>39.6</td>
</tr>
<tr>
<td>PTR</td>
<td>-17.4</td>
</tr>
</tbody>
</table>

Name: 2019-12-31 00:00:00, dtype: float64

Then to plot the chart

In [40]:

price_change.sort_values(inplace=True)
price_change = price_change.rename(index=ticker_list)
fig, ax = plt.subplots(figsize=(10,8))
ax.set_xlabel('stock', fontsize=12)
ax.set_ylabel('percentage change in price', fontsize=12)
price_change.plot(kind='bar', ax=ax)
plt.show()
Footnotes

[1] Wikipedia defines munging as cleaning data from one raw form into a structured, purged one.