RIT REST API Tutorial

Table of Contents

Introduction 2

Python/Environment Setup 3
- Rotman Interactive Trader Install 3
- Text Editor 3
- Python Distribution 3
- Verifying Your Installation 3
- Python Virtual Environments 5

Introduction to Python 7
- Create a Work Directory 7
- Hello World 7
- Hello Input 8
- Mathematical Expressions 11
- Tuples, Lists, and Dictionaries 11
- Summary 12

Using Pandas/NumPy Package – Stock Returns Example 13
- Running the Python Interpreter 13
- Importing Packages 13
- Reading In Data From CSV 14
- DataFrames 15
- Viewing Data From DataFrames 15
- Manipulating Data In DataFrames 17
- Summary 20

Introduction to the RIT REST API 21
- Setting Up Python 21
- Basic Use 21
- Important Notes 22
- Submitting Orders 22
- Cancelling Orders 24

Algorithmic Trading Example – Arbitrage 26
- Basic Setup 26
- Arbitrage Logic 27
- Running the Algorithm 29

Introduction

The Rotman Interactive Trader (RIT) allows users to query for market data and submit trading instructions through a REST API, as well as through a Microsoft Excel VBA-specific API. The purpose of this is to allow for program or 'algorithmic' trading, where the computer executes trades based on a pre-defined set of instructions or parameters.

This tutorial focuses on interacting with the REST API, which allows a language-agnostic way for programs to interact with the RIT. In effect, most programming languages capable of submitting HTTP requests to a pre-defined web address will be able to interact with the RIT. Specifically, this tutorial will use Python, as it is a general-purpose language that is commonly used in the data science/finance domains. This tutorial assumes no previous knowledge of Python, and provides an introduction to the concepts of programming, Python, and data manipulation before introducing users to the RIT REST API and an in-depth example of an algorithmic arbitrage trading strategy. Those users who are already familiar with Python or interacting with a REST API through their language of choice should skip to the Introduction to the RIT REST API section, or to the detailed documentation available through the RIT Client.

This tutorial does not discuss the strategies behind algorithmic trading. Rather, it introduces the user to the tools that are available through the RIT REST API. Users are encouraged to explore possible strategies and techniques and use the building blocks here to implement them.

Python/Environment Setup

Note: this tutorial is for individual users working with Python/the RIT on their own computers. For mass Python/the RIT deployments, please contact your local IT administration.

Rotman Interactive Trader Install

The Rotman Interactive Trader Client download and install information is available here. To use the RIT REST API, only the Client is required. To use the Microsoft Office Excel RTD links/VBA API (not used in this tutorial), the RTD links toolkit is also required (available from the same link above).

Text Editor

A text editor like Notepad, Notepad++, Notepad2, Atom, etc. is required to write and save the Python code presented in the tutorial into .py files. Notepad is already available on all versions of Windows. The recommended Anaconda/Miniconda installers (next section) include the option to install VSCode, another text editor from Microsoft.

Python Distribution

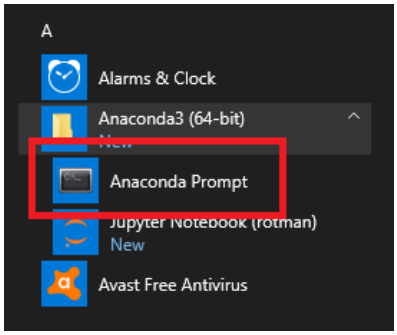
The recommended way to set up your Python environment/workspace is to use either the Anaconda distribution or the Miniconda distribution of Python 3.6+

Anaconda already includes many of the most commonly used data science packages (essentially additional tools) like NumPy (support for multidimensional arrays) and Pandas (easy to use data structures and tools for data analysis), as well as a package and virtual environment manager. Miniconda only contains the package and virtual environment manager, and users can manually decide on which packages to download and install for use.

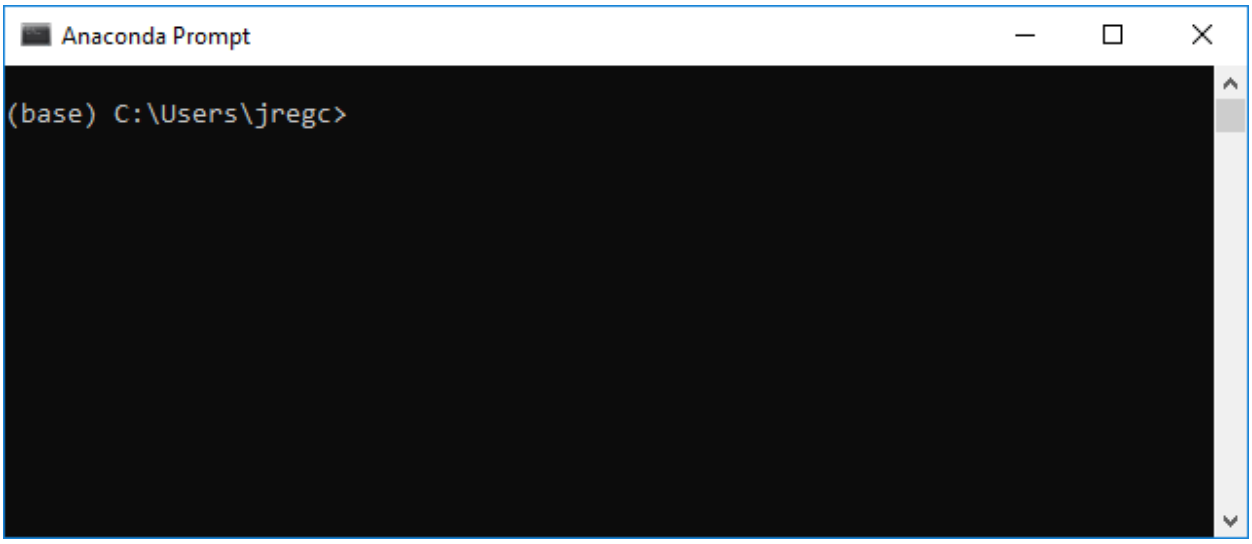
Note: when installing Anaconda or Miniconda, choose to leave the option 'Add Anaconda to my PATH variable' ***unchecked****, and the option 'Register Anaconda as my default Python 3.x'* ***checked***

Verifying Your Installation

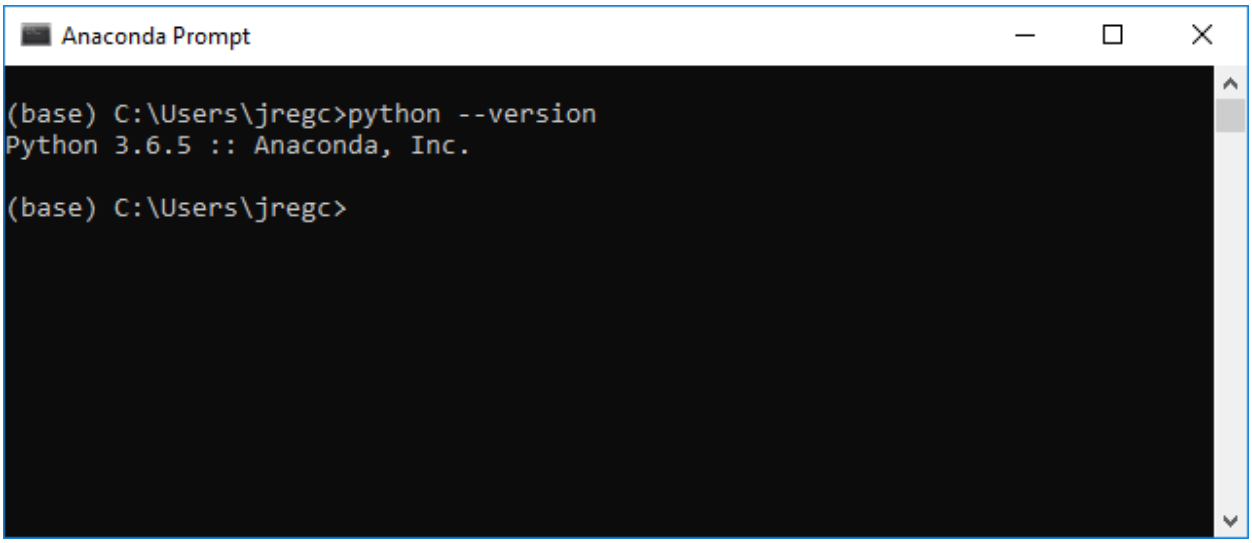
After installing Anaconda or Miniconda, please open the 'Anaconda Prompt' from the Start Menu, or the Command Prompt/PowerShell if you are using a different Python distribution.



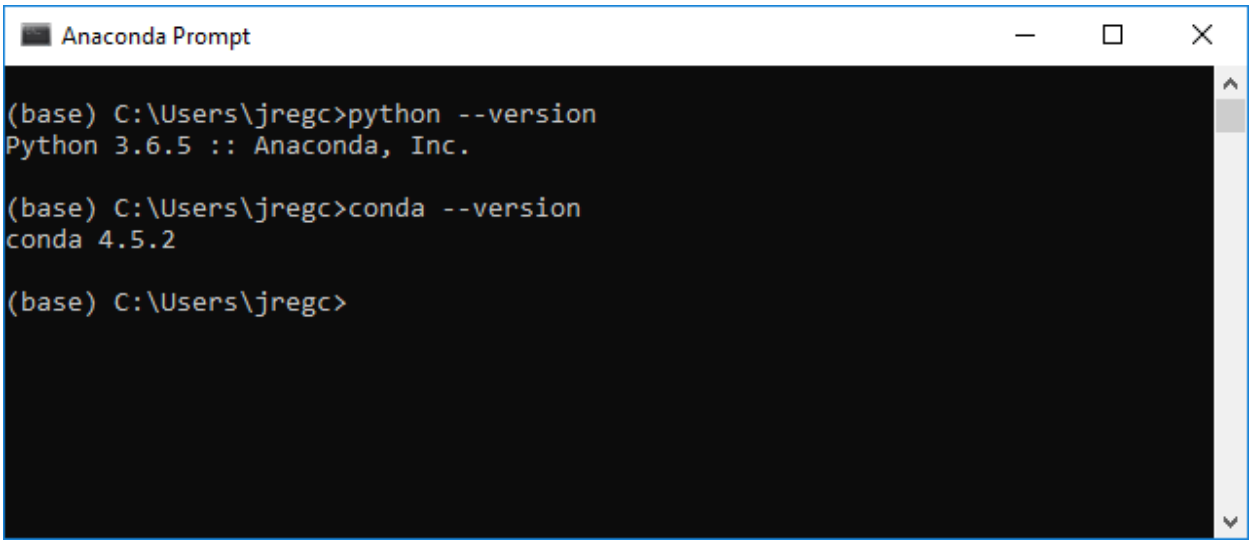
This should open a window looking similar to the following, with 'jregc' being replaced by your user ID. This tutorial will refer to this window as 'the prompt' from here onwards.



Type `python --version` into the prompt and press `enter`. This command asks Python for its current version number. The output should look similar to the following if everything has been installed correctly, where the version number is 3.6 or greater.



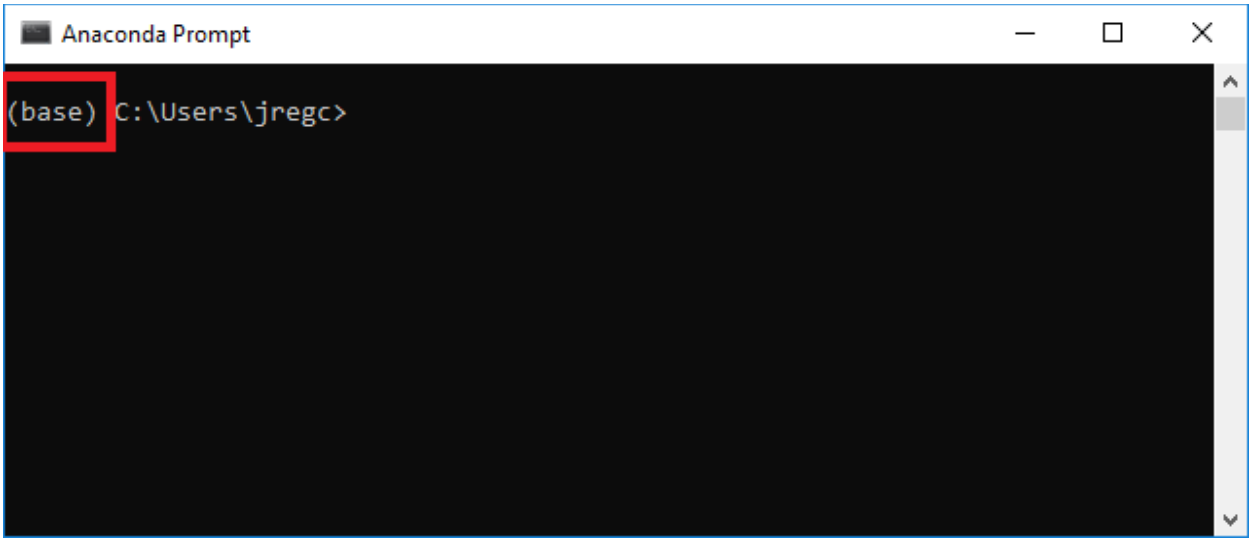
Then type `conda --version` into the prompt and press `enter`. This command asks Anaconda/Miniconda for its current version number. The output should look similar to the following if everything has been installed correctly, where the version number is 4.5 or greater. In the case where the version number is lower than 4.5, type `conda update -n base conda` to get the latest version.



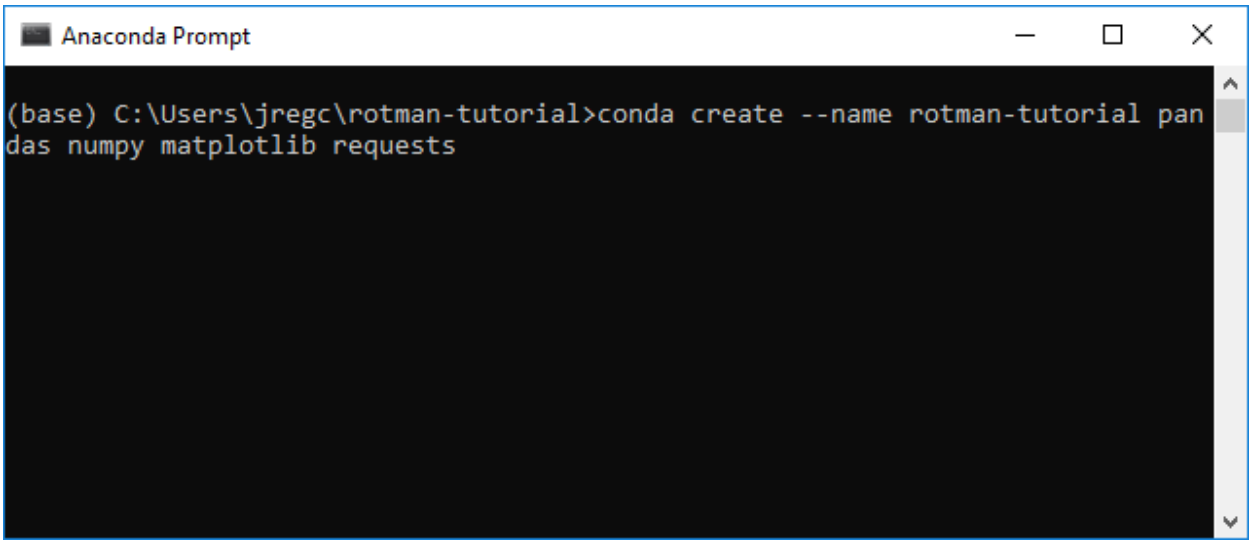
Python Virtual Environments

Anaconda and Miniconda come with the conda package and virtual environment manager. Different Python applications that users write may require different files and packages, and virtual environments help solve this problem. A virtual environment is a self-contained environment/directory that contains its own files, installed packages, and their dependencies that will not interact with other environments' files, packages, and dependencies.

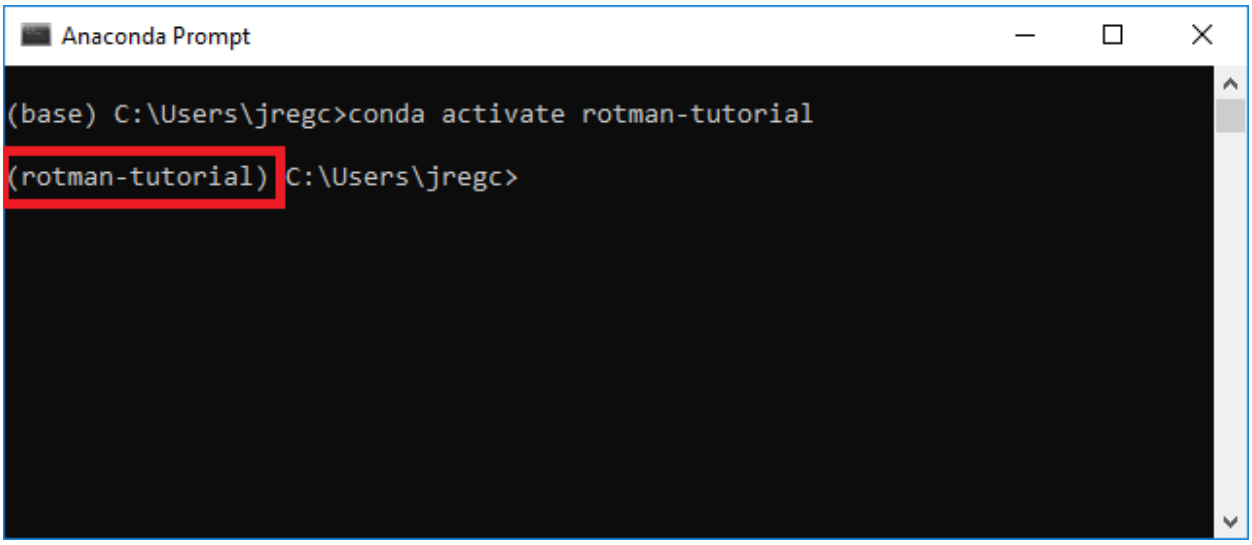
When a user initially starts the prompt, it starts in the 'base' environment, as indicated on the left side of the prompt.



However, it is not recommended to install additional packages in the 'base' environment. To create a new environment, enter `conda create --name <ENV NAME> pandas numpy matplotlib requests`. This will create a new virtual environment, with the name supplied in `<ENV NAME>`, and with the 'pandas', 'numpy', 'matplotlib', and 'requests' packages needed in this tutorial, plus any dependencies for those packages.



In this case, the virtual environment is named 'rotman-tutorial'. Enter `y` into the prompt after `conda` lists the packages that must be downloaded and installed to proceed and create the environment. After the environment is created, enter `conda activate <ENV NAME>` or simply `activate <ENV NAME>` into the prompt to switch the context of the prompt to that environment.

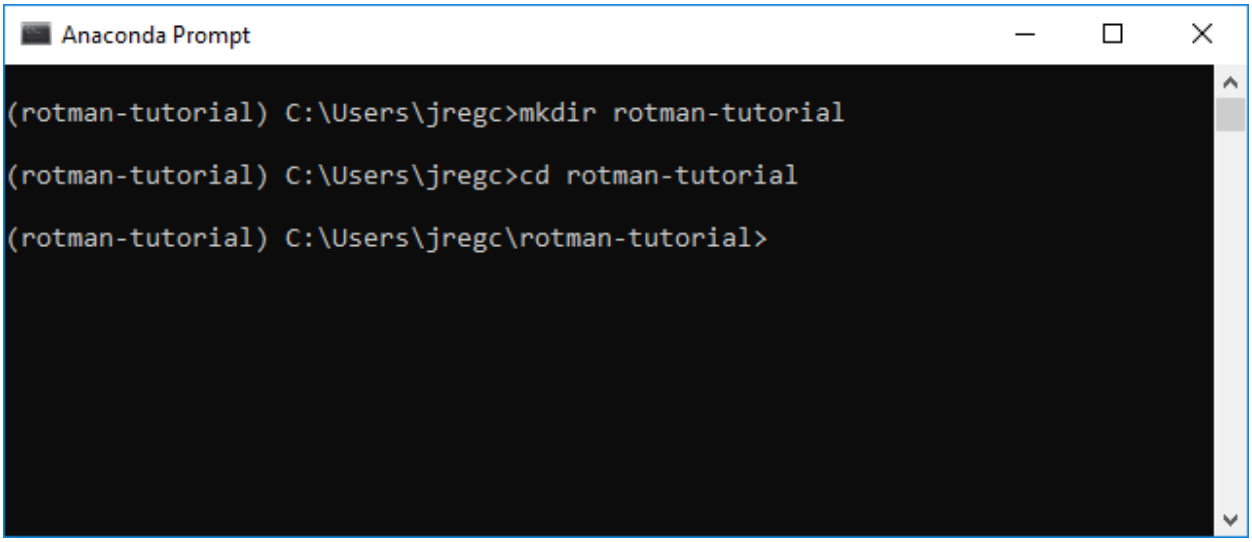


As shown in the above screenshot, after entering `conda activate rotman-tutorial` into the prompt, the prompt indicates that the current environment is 'rotman-tutorial'. If a user wants to deactivate the current environment and go back to the 'base' environment, enter `conda deactivate`.

Introduction to Python

Create a Work Directory

In the local user directory, create a work directory to store the tutorial files. Users can do this from the prompt by entering `mkdir <WORK DIR NAME>` to create a directory in the current location. Then, enter `cd <PATH TO WORK DIR>` to change locations to that directory.



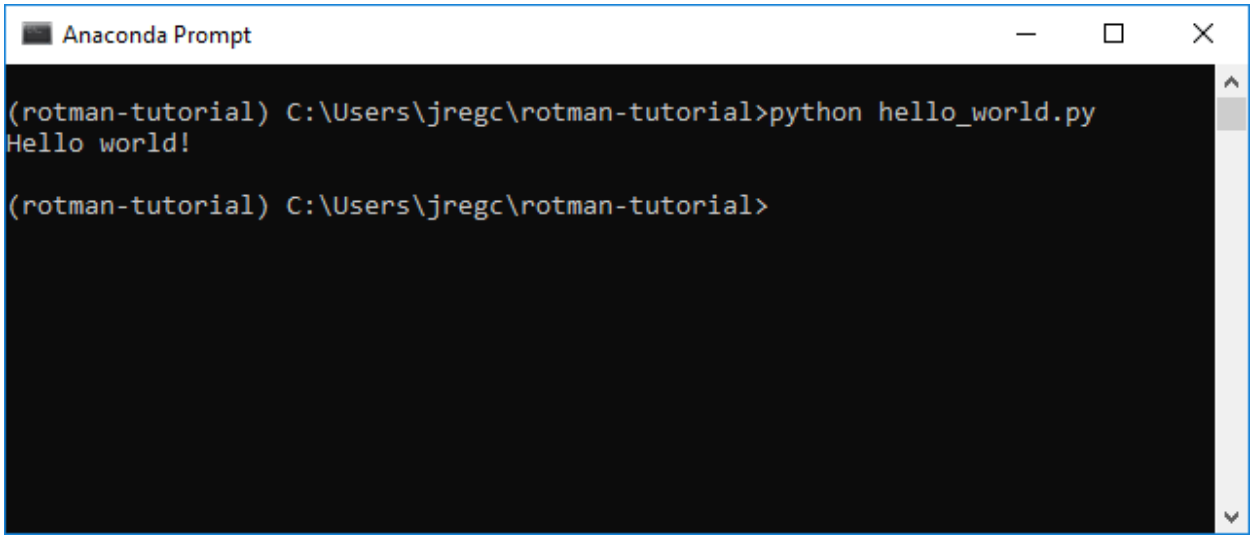
In the above screenshot, a directory called 'rotman-tutorial' was created in the directory `C:\Users\jregc`

Hello World

Open your preferred text editor, type the following into a new file, and save the file in the work directory as `hello_world.py`.

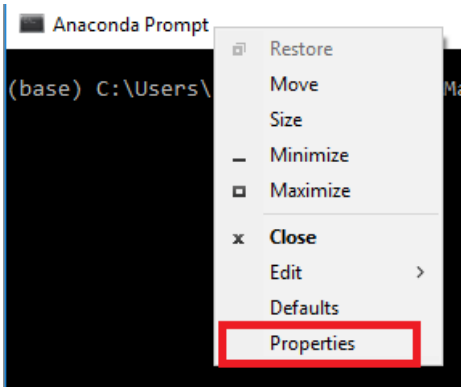
```
def main():
    print('Hello world!')

# this if-block tells Python to call the main() method when it runs the file
from the prompt
if __name__ == '__main__':
    main()
```

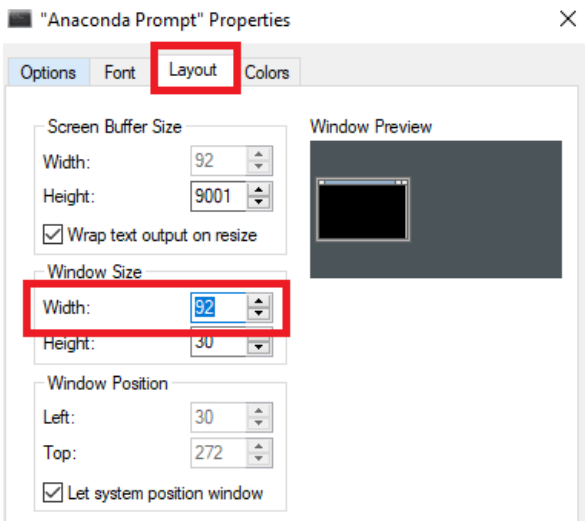
Then in the prompt, enter `python hello_world.py`.



This command tells Python to run the file in the local directory called `hello_world.py`. Inside that file, there is a method called **main** that calls the **print** method. The **print** method takes in the text 'Hello world' as a parameter and prints it out to the prompt as `Hello world!`.

In case the prompt window size needs to be changed, right-click on the top module bar from the prompt window, choose “Properties”, click on “Layout”, and change “Width” under “Window Size” to display any contents properly.



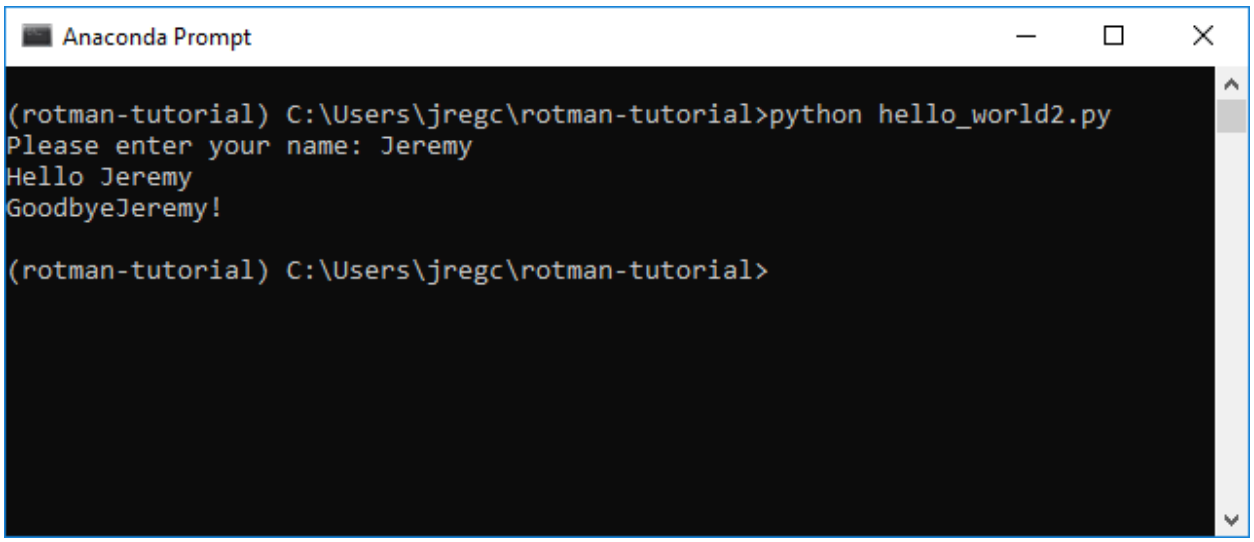


Hello Input

Python can also take in user input. For example, try saving the following into a file called `hello_world2.py` and running it:

```
def main():
    name = input('Please enter your name: ')
    print('Hello', name)
    print('Goodbye' + name + '!')

if __name__ == '__main__':
    main()
```

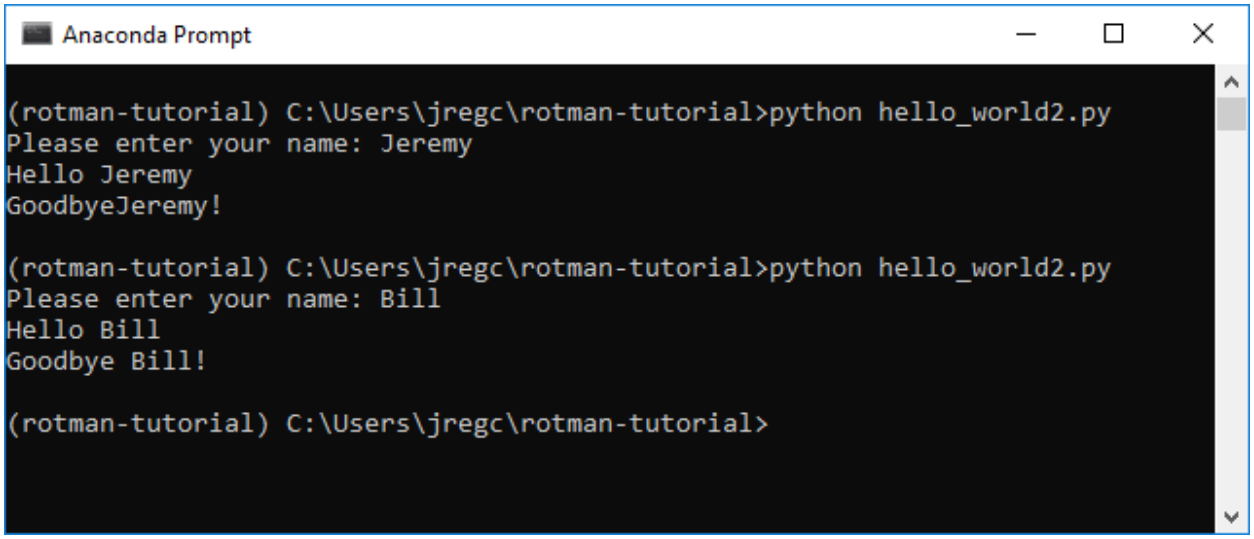

This time, a prompt should be displayed, asking for your name. In effect, the first line of code tells Python to print to the prompt the text `Please enter your name:`, wait for an input to be typed in, and then save that input into the variable called **name**. The second line then tells Python to print `Hello` and the value saved in the variable **name**. The third line shows another way of combining text together to be printed out.



But if you look at what's displayed on the third line of the output, it looks a little messy. Let's fix that:

```
def main():
    name = input('Please enter your name: ')
    print('Hello', name)
    print('Goodbye ' + name + '!')

if __name__ == '__main__':
    main()
```

Note the space in the quoted text 'Goodbye '.

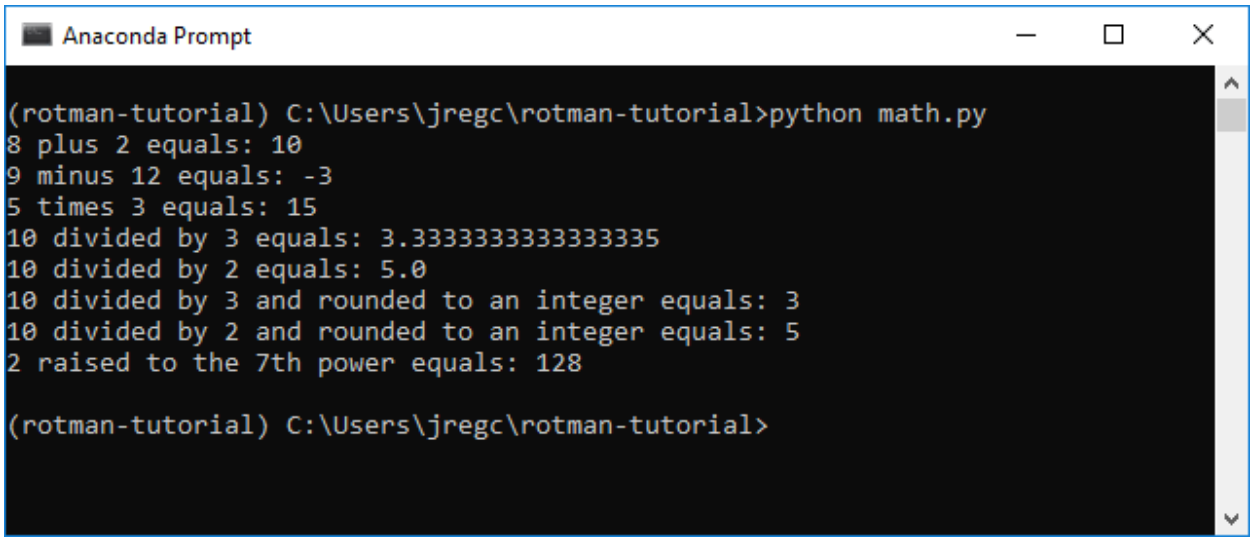


There, that's better!

Mathematical Expressions

Like many programming languages, Python can also perform mathematical calculations. Try saving and running the following as `math.py`:

```
def main():
    print('8 plus 2 equals:', 8 + 2)
    print('9 minus 12 equals:', 9 - 12)
    print('5 times 3 equals:', 5 * 3)
    print('10 divided by 3 equals:', 10 / 3)
    print('10 divided by 2 equals:', 10 / 2)
    print('10 divided by 3 and rounded to an integer equals:', 10 // 3)
    print('10 divided by 2 and rounded to an integer equals:', 10 // 2)
    print('2 raised to the 7th power equals:', 2 ** 7)

if __name__ == '__main__':
    main()
```

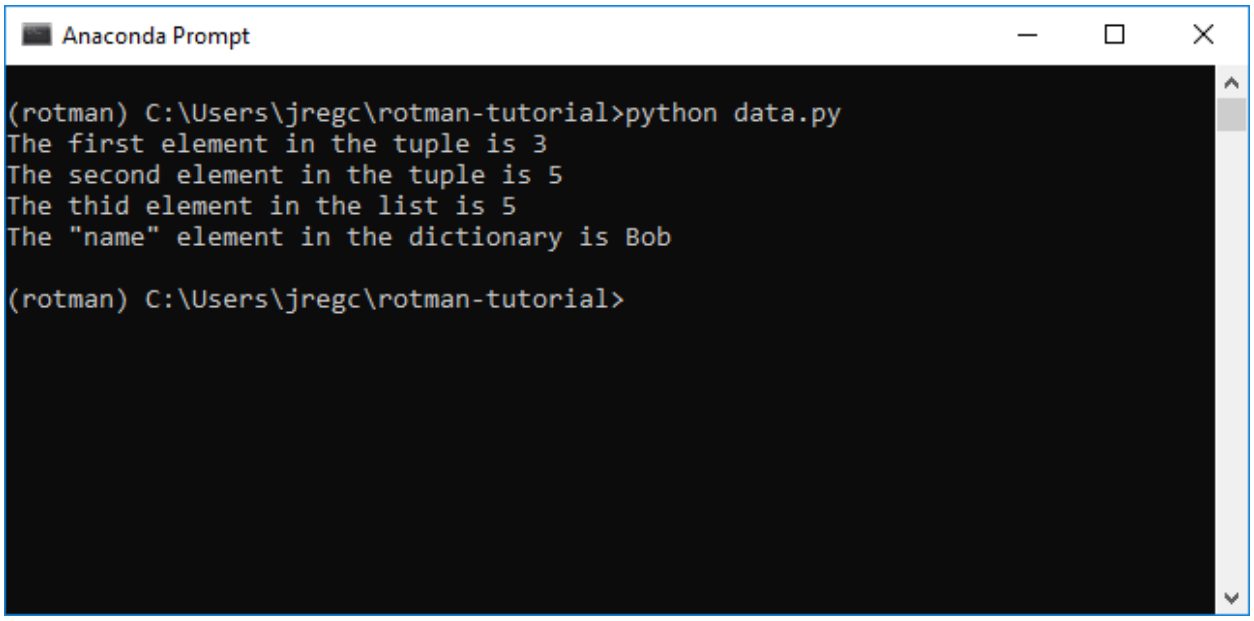


Note that there is a difference between integer and floating-point math, where floating-point numbers are representations of real numbers including decimals.

Tuples, Lists, and Dictionaries

There are also three common data structures that are used in Python: tuples, lists, and dictionaries. Tuples are comma-separated lists of values that cannot be changed once created, while lists are comma-separated lists of values that can be changed. Dictionaries are lists of key/value pairs that are associated with one another. In effect, the major difference is how to access values in the different data structures: usually one will index by number to access values in tuples and lists, while one will index by key to access a value in a dictionary. The following example illustrates how this works.

```
def main():
    t = (3, 5, 10, 9)
    l = [8, 9, 5]
    d = {'key': 'value', 'name': 'Bob'}

    print('The first element in the tuple is', t[0])
    print('The second element in the tuple is', t[1])
    print('The third element in the list is', l[2])
    print('The "name" element in the dictionary is', d['name'])

if __name__ == '__main__':
    main()
```



Note that python uses 0-based indexing, such that the first element is at position 0, the second is at position 1, etc.

Summary

This concludes a basic introduction to Python, necessary for the following sections on using Pandas/NumPy for simple stock return calculations, as well as on using the RIT REST API. You should now be able to write a simple set of instructions (a method) in Python, using a pre-defined method (print) and execute it from the prompt.

For a more detailed introduction to Python, please see The Python Tutorial.

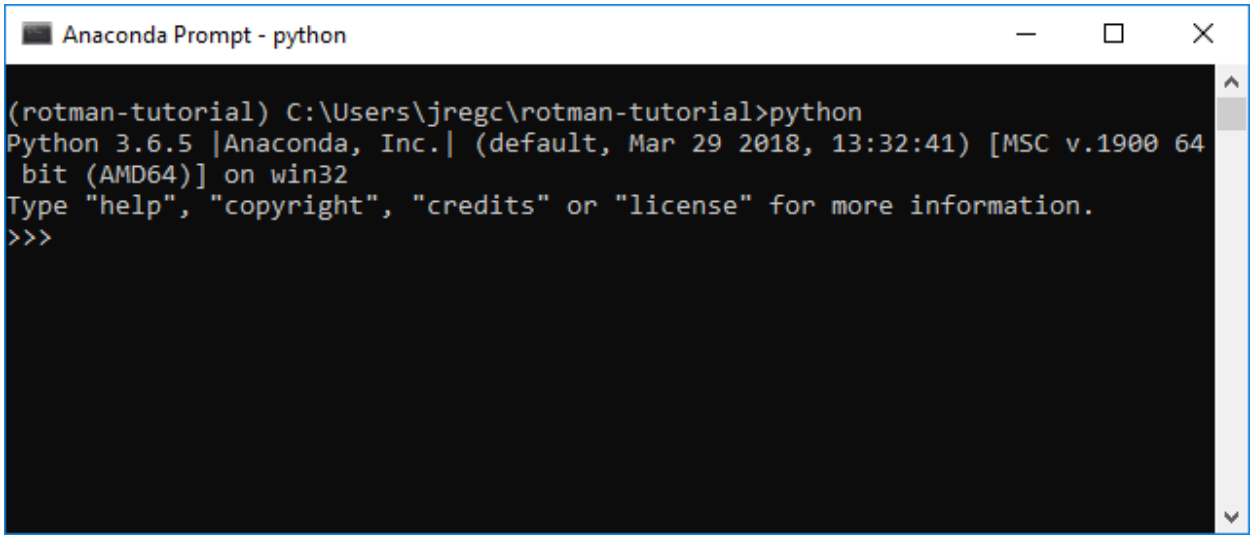
Using Pandas/NumPy Package – Stock Returns Example

Pandas is a commonly used open-source data analysis package for Python. It provides a comprehensive set of easy-to-use data structures and analysis tools. We'll take a quick look at how to use Pandas to read in CSV data from Yahoo Finance and perform some common calculations like returns and summary statistics.

Instead of writing the code into a file and then running it via `python <FILE NAME>.py`, we'll use the interactive Python interpreter available via the prompt. Note however that the code can also be saved into a `.py` file and run, as demonstrated in the Introduction to Python section.

Running the Python Interpreter

To run the Python interpreter, simply enter `python` into the prompt, first ensuring that the 'rotman-tutorial' (or other) virtual environment is active and the prompt is in your working directory.



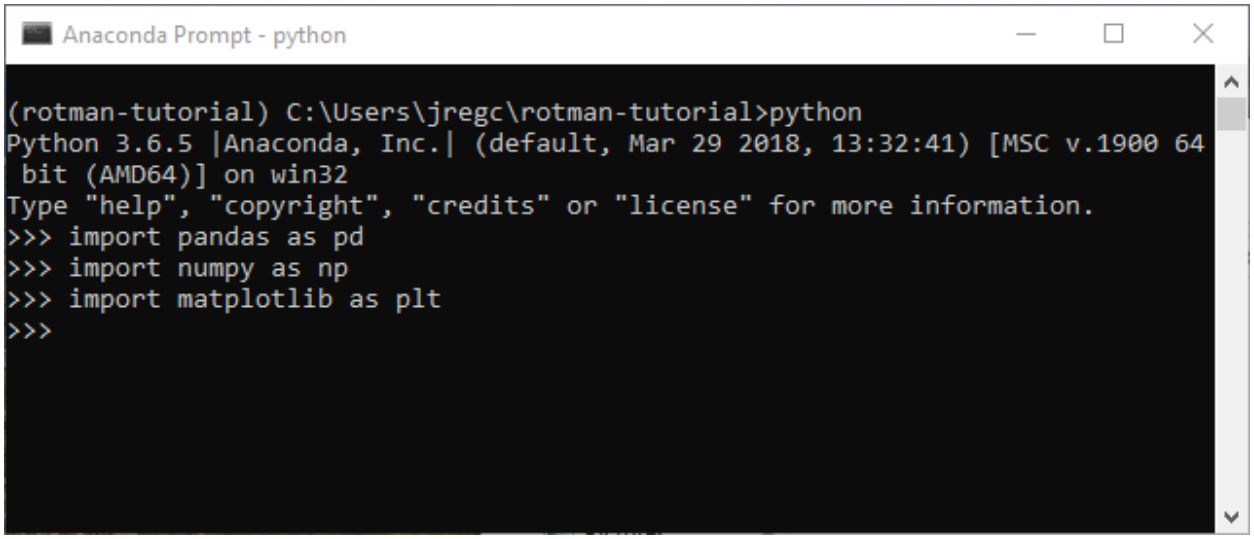
As the screenshot shows, the Python interpreter is active, running Python version 3.6.5. The `>>>` shows that we are in interactive mode, and can enter commands to be interpreted by Python.

To exit the Python interpreter, enter the command `exit()`.

Importing Packages

To import packages, either into a Python file or into the interpreter, type the following lines:

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



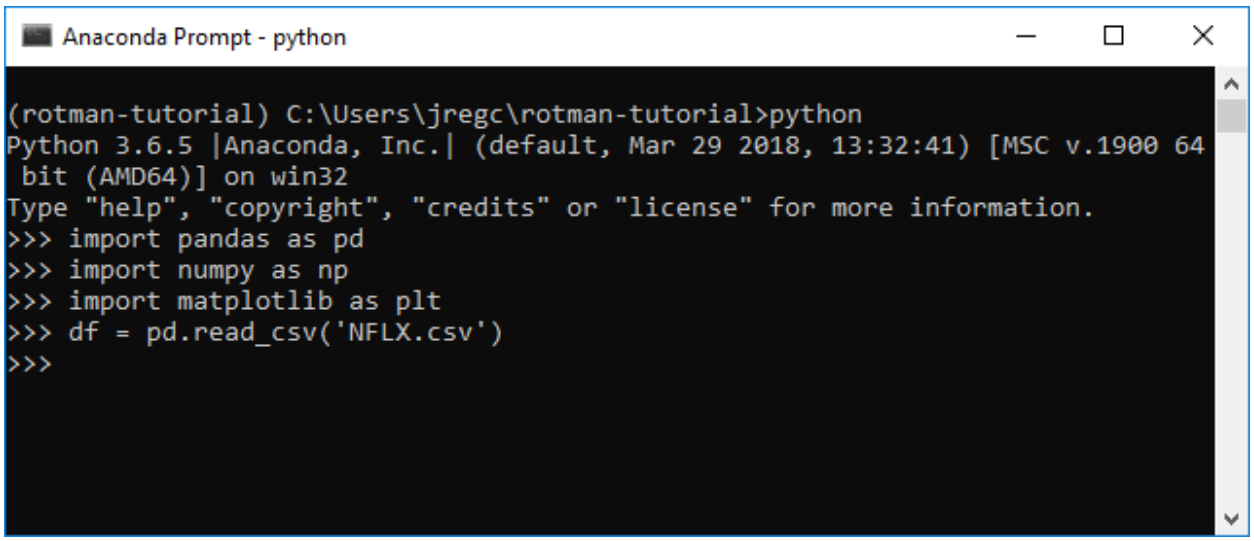
These three lines import the 'pandas', 'numpy', and 'matplotlib.pyplot' packages that we installed in the 'rotman-tutorial' virtual environment which was set up in the Python Virtual Environments section of the tutorial. Additionally, we create nicknames to reference them by ('pd', 'np', and 'plt' respectively). The next section will show how to call methods from these packages.

Reading In Data From CSV

Let's get some data in the form of a CSV file to read. Go to Yahoo Finance, query an equity ticker, and download a 1Y span of historical daily data. Save this CSV in your work directory, as set up in the Create A Work Directory section.

In this tutorial, we're using Netflix (NFLX) historical data.

```
df = pd.read_csv('NFLX.csv')
```



This command calls the read_csv() method available in the 'pandas' package, passing in the filename 'NFLX.csv' as the parameter specifying the file to open and read in the same directory. Relative paths

are also possible, for example `pd.read_csv('data/NFLX.csv')` would read a 'NFLX.csv' file located in a subdirectory named 'data'.

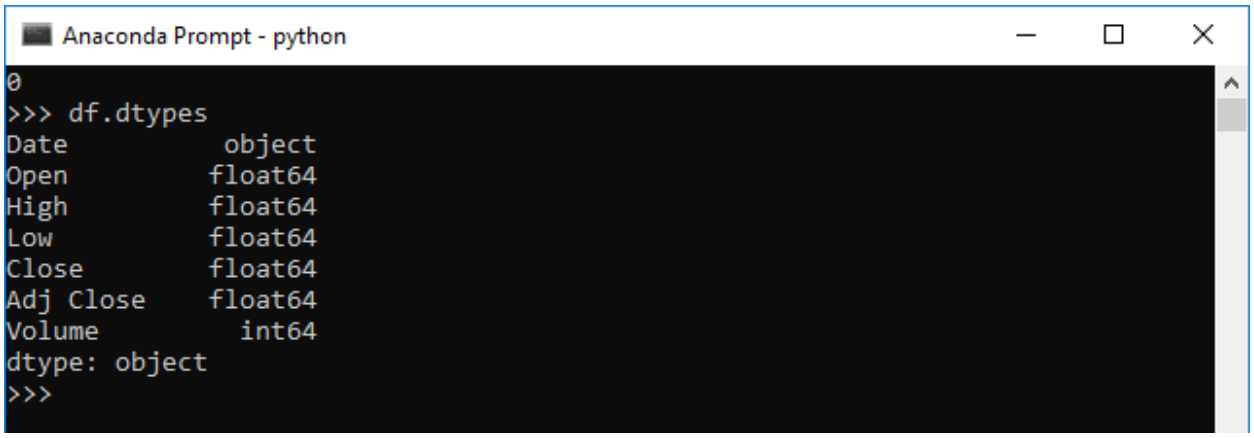
After reading the data in from the CSV file, the read_csv() method returns it as a DataFrame object, and the variable named **df** (for DataFrame) refers to that DataFrame object.

DataFrames

DataFrames are the primary data structure in Pandas, and can be thought of as two dimensional tables with labeled axes, similar to how data is laid out in a `.csv` or `.xls/.xslx` file in rows and columns.

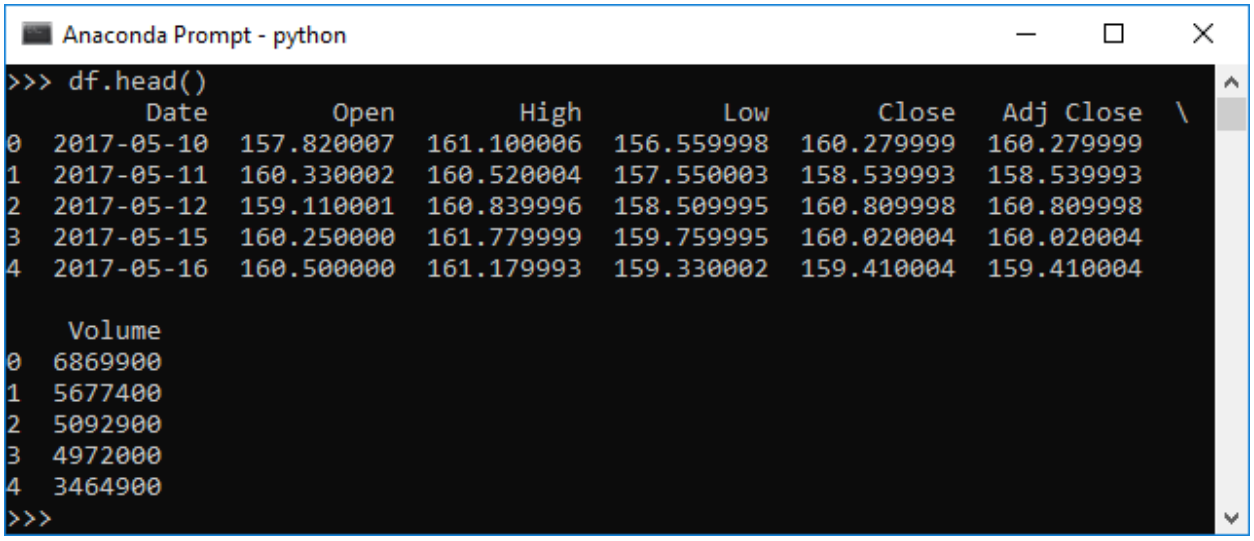
Viewing Data From DataFrames

```
df.dtypes
```



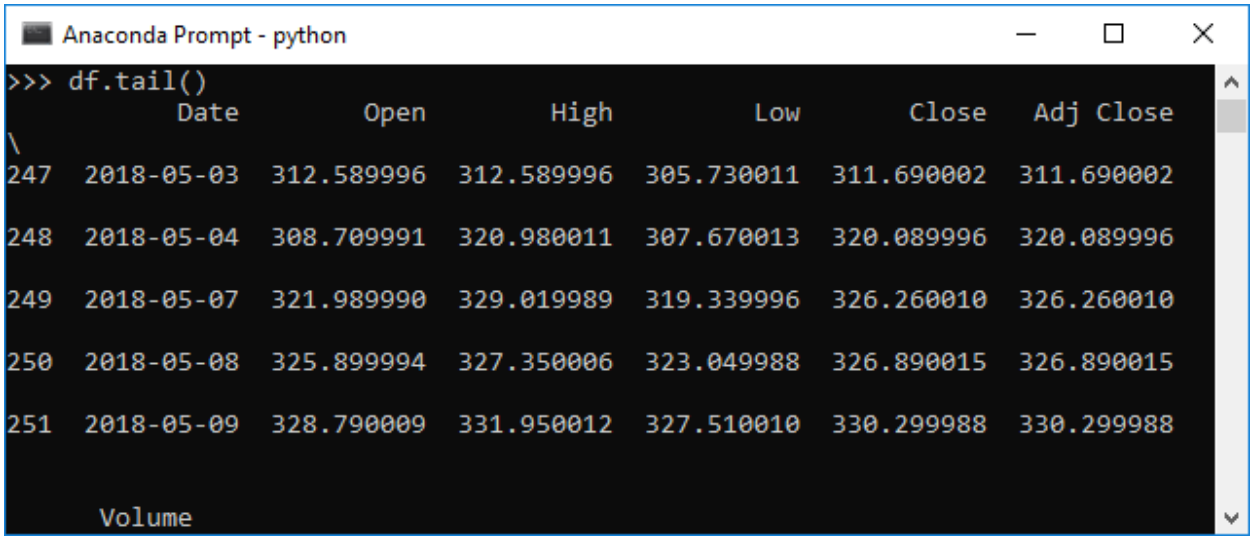
The dtypes attribute provides a list of the data types of each column.

```
df.head()
```



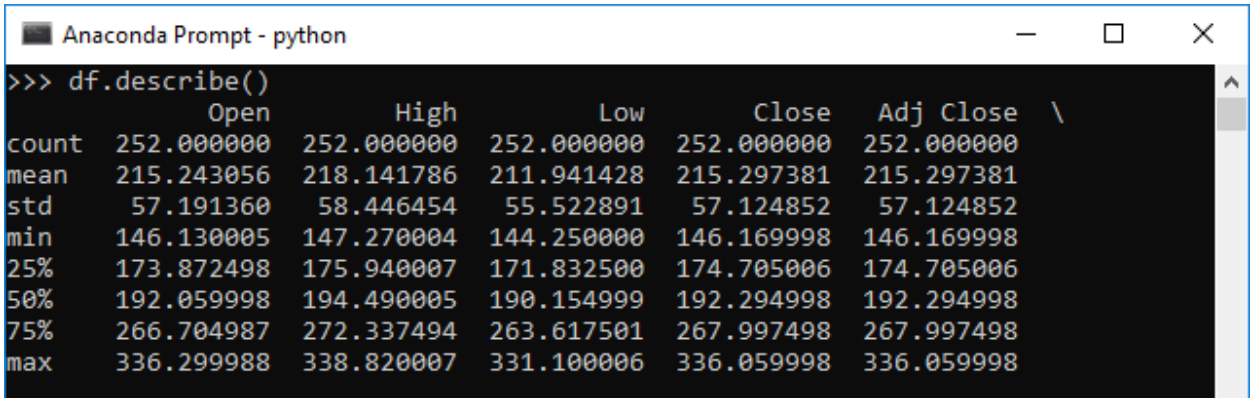
The head() method display the first 5 rows in the DataFrame. A different number of rows to display can be passed in as a parameter (for example `df.head(10)` would display the first 10 rows).

```
df.tail()
```



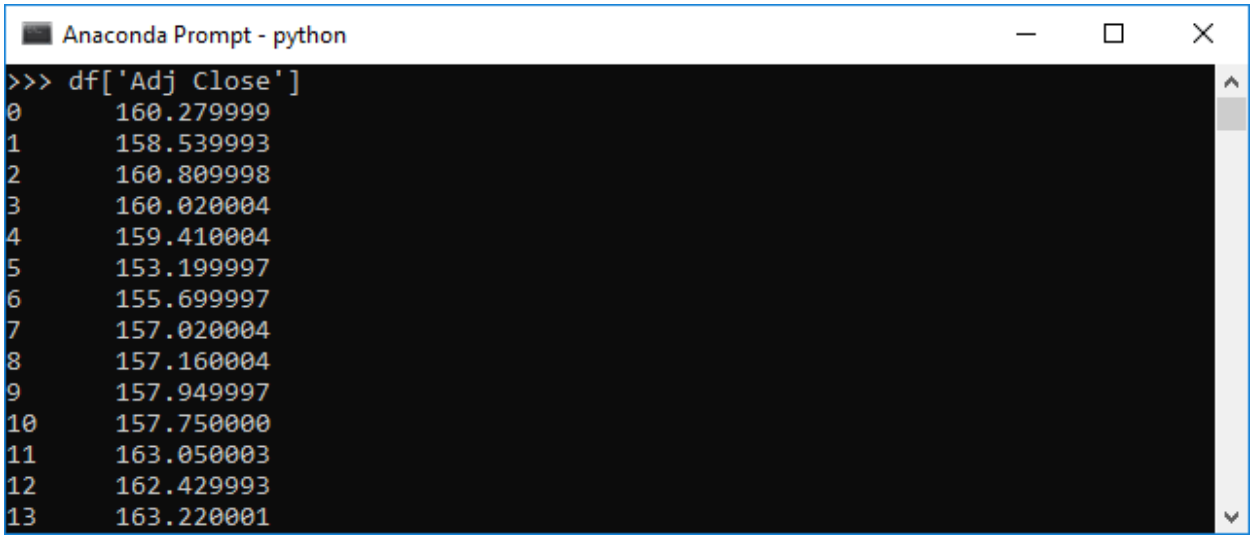
The tail() method displays the last 5 rows in the DataFrame. A different number of rows to display can be passed in as a parameter (for example `df.tail(20)` would display the last 20 rows).

```
df.describe()
```



The describe() method calculates and displays some common sample statistics for the DataFrame's columns, including the count, mean, standard deviation, min/max values, and quartiles. It skips NA values.

```
df['Adj Close']
```

Display a column from the DataFrame, selected by label.

Manipulating Data In DataFrames

It's also possible to add new columns to a DataFrame and perform other calculations:

```
df['Daily Return'] = df['Adj Close'].pct_change()
df['10DMA'] = df['Adj Close'].rolling(window=10, center=False).mean()
df['30DMA'] = df['Adj Close'].rolling(window=30, center=False).mean()
summary = df.describe()
```

From the commands above, three additional columns ('Daily Return', '10DMA', and '30DMA') are added to the DataFrame. The 'Daily Return' column is calculated by calling the pct_change() method, which calculates the percentage change between each row in the 'Adj Close' column. The '10DMA' and '30DMA' columns are calculated by creating rolling 10-day or 30-day windows on the rows in the 'Adj Close' column, and then calculating the mean on those windows.

Then the 'summary' variable is defined as 'df.describe()'. As demonstrated in the previous section, this method will display some common sample statistics whenever a user types 'summary' and hits enter in the prompt.

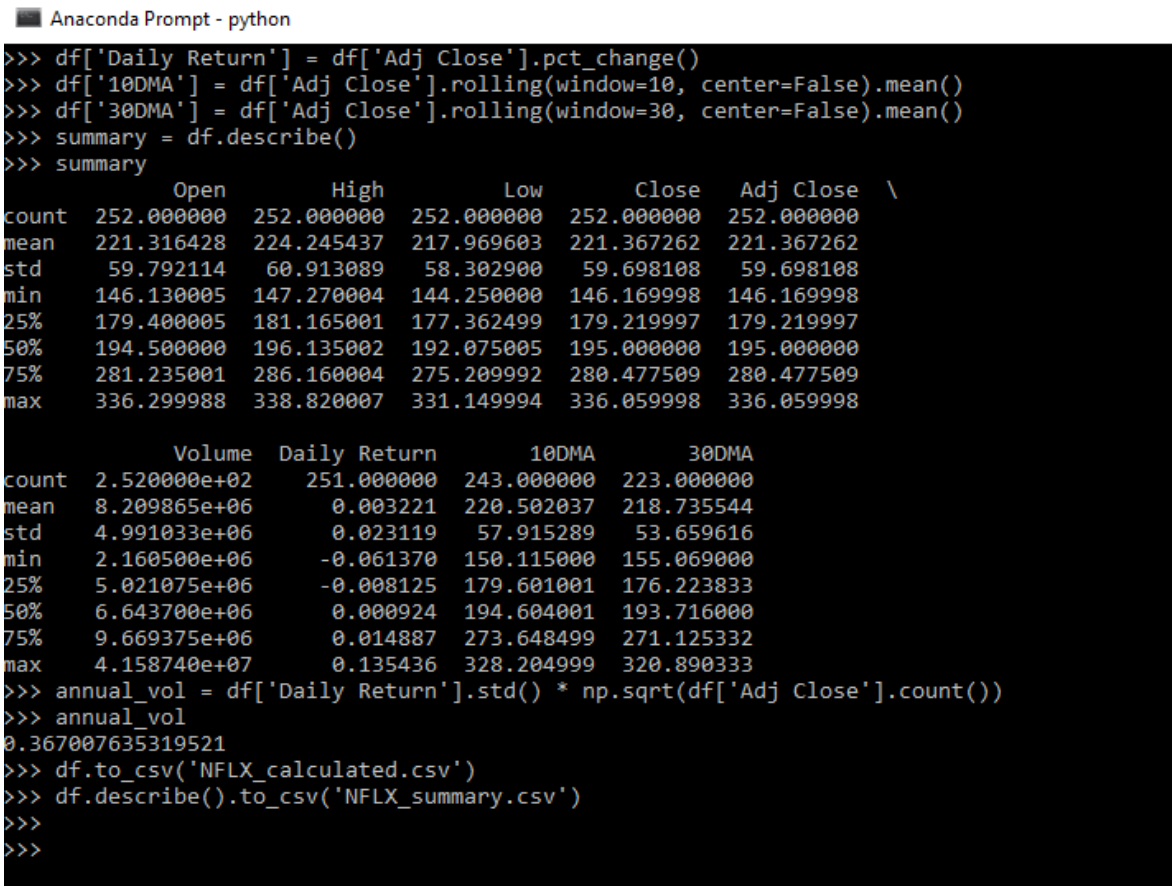
Using the daily return values, a user can also calculate an annualized volatility. From the command below, a standard deviation of the daily returns is first calculated and multiplied by the square root of the number of trading days in a year.

```
annual_vol = df['Daily Return'].std() * np.sqrt(df['Adj Close'].count())
```

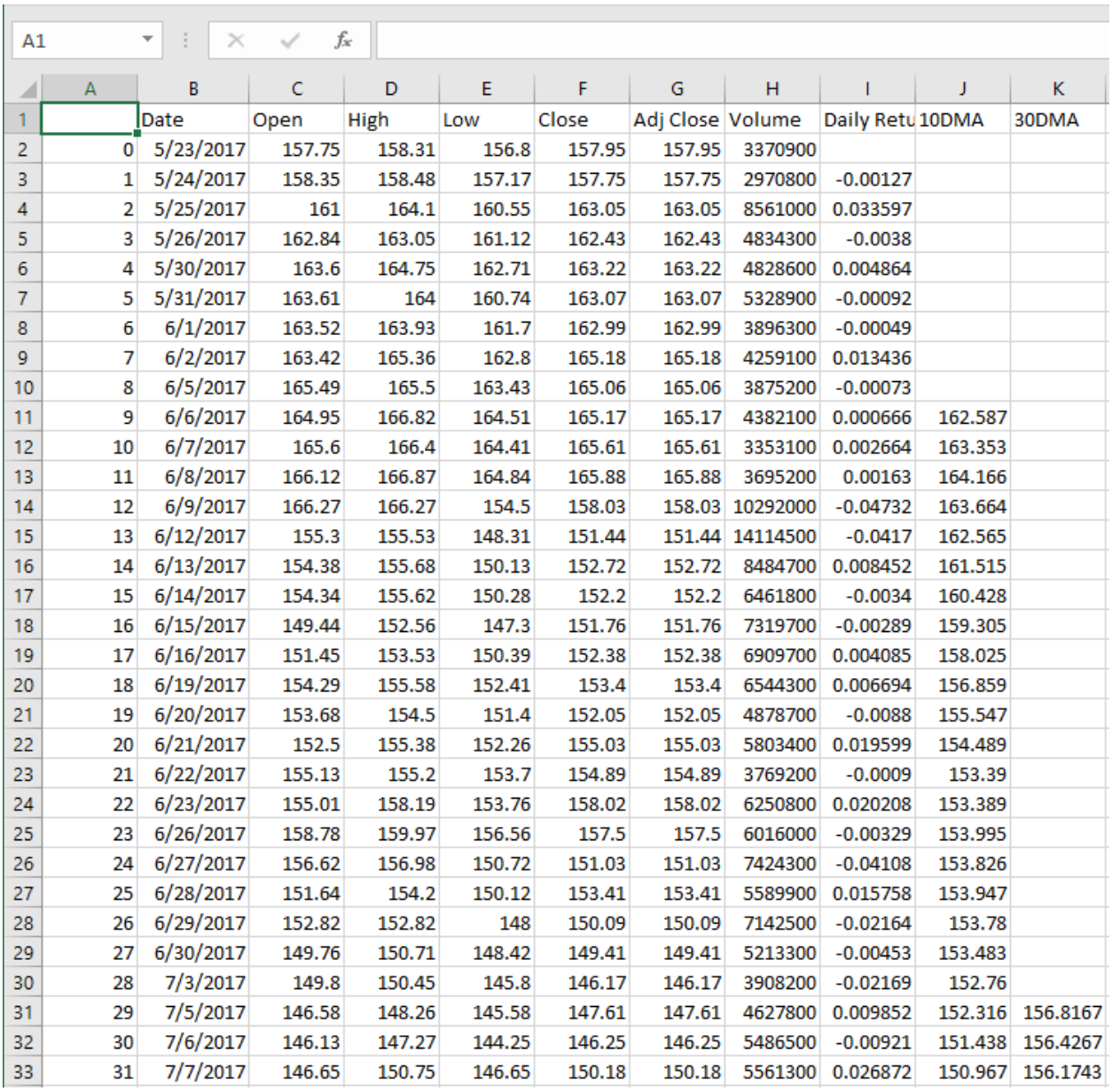
A user can simply type in 'annual_vol' and hit enter in the prompt to query the calculated annualized volatility.

The to_csv() method allows a user to export the DataFrame to a csv file. Using the following sample commands, a user can export the entire DataFrame as a csv file with a file name 'NFLX_calculated.csv' or just the summary part with a file name 'NFLX_summary.csv'. The exported files will be made available in the same directory.

```
df.to_csv('NFLX_calculated.csv')
df.describe().to_csv('NFLX_summary.csv')
```

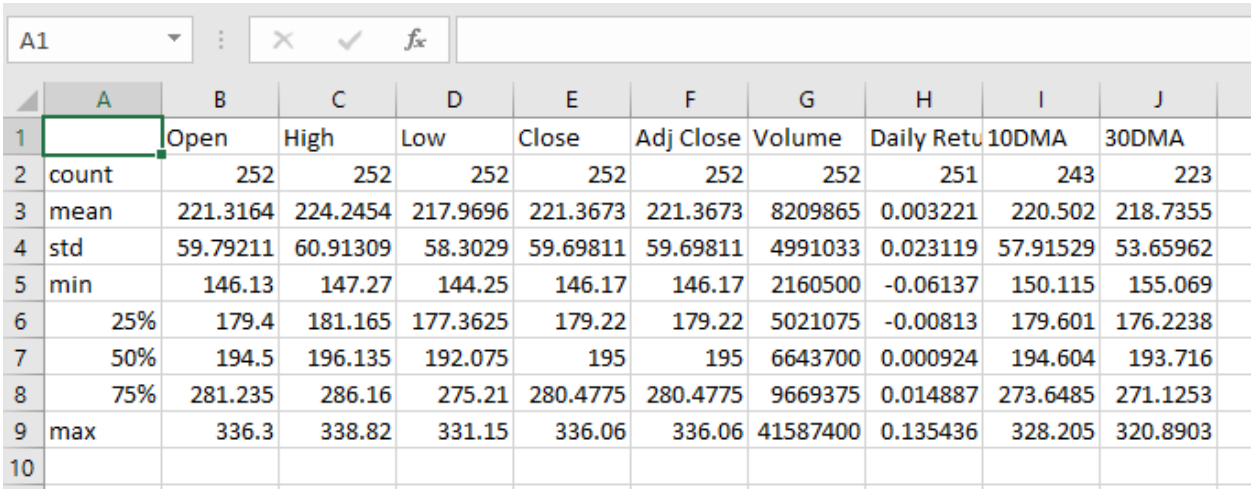


NFLX_calculated.csv



	A	B	C	D	E	F	G	H	I	J	K
1		Date	Open	High	Low	Close	Adj Close	Volume	Daily Retu	10DMA	30DMA
2	0	5/23/2017	157.75	158.31	156.8	157.95	157.95	3370900			
3	1	5/24/2017	158.35	158.48	157.17	157.75	157.75	2970800	-0.00127		
4	2	5/25/2017	161	164.1	160.55	163.05	163.05	8561000	0.033597		
5	3	5/26/2017	162.84	163.05	161.12	162.43	162.43	4834300	-0.0038		
6	4	5/30/2017	163.6	164.75	162.71	163.22	163.22	4828600	0.004864		
7	5	5/31/2017	163.61	164	160.74	163.07	163.07	5328900	-0.00092		
8	6	6/1/2017	163.52	163.93	161.7	162.99	162.99	3896300	-0.00049		
9	7	6/2/2017	163.42	165.36	162.8	165.18	165.18	4259100	0.013436		
10	8	6/5/2017	165.49	165.5	163.43	165.06	165.06	3875200	-0.00073		
11	9	6/6/2017	164.95	166.82	164.51	165.17	165.17	4382100	0.000666	162.587	
12	10	6/7/2017	165.6	166.4	164.41	165.61	165.61	3353100	0.002664	163.353	
13	11	6/8/2017	166.12	166.87	164.84	165.88	165.88	3695200	0.00163	164.166	
14	12	6/9/2017	166.27	166.27	154.5	158.03	158.03	10292000	-0.04732	163.664	
15	13	6/12/2017	155.3	155.53	148.31	151.44	151.44	14114500	-0.0417	162.565	
16	14	6/13/2017	154.38	155.68	150.13	152.72	152.72	8484700	0.008452	161.515	
17	15	6/14/2017	154.34	155.62	150.28	152.2	152.2	6461800	-0.0034	160.428	
18	16	6/15/2017	149.44	152.56	147.3	151.76	151.76	7319700	-0.00289	159.305	
19	17	6/16/2017	151.45	153.53	150.39	152.38	152.38	6909700	0.004085	158.025	
20	18	6/19/2017	154.29	155.58	152.41	153.4	153.4	6544300	0.006694	156.859	
21	19	6/20/2017	153.68	154.5	151.4	152.05	152.05	4878700	-0.0088	155.547	
22	20	6/21/2017	152.5	155.38	152.26	155.03	155.03	5803400	0.019599	154.489	
23	21	6/22/2017	155.13	155.2	153.7	154.89	154.89	3769200	-0.0009	153.39	
24	22	6/23/2017	155.01	158.19	153.76	158.02	158.02	6250800	0.020208	153.389	
25	23	6/26/2017	158.78	159.97	156.56	157.5	157.5	6016000	-0.00329	153.995	
26	24	6/27/2017	156.62	156.98	150.72	151.03	151.03	7424300	-0.04108	153.826	
27	25	6/28/2017	151.64	154.2	150.12	153.41	153.41	5589900	0.015758	153.947	
28	26	6/29/2017	152.82	152.82	148	150.09	150.09	7142500	-0.02164	153.78	
29	27	6/30/2017	149.76	150.71	148.42	149.41	149.41	5213300	-0.00453	153.483	
30	28	7/3/2017	149.8	150.45	145.8	146.17	146.17	3908200	-0.02169	152.76	
31	29	7/5/2017	146.58	148.26	145.58	147.61	147.61	4627800	0.009852	152.316	156.8167
32	30	7/6/2017	146.13	147.27	144.25	146.25	146.25	5486500	-0.00921	151.438	156.4267
33	31	7/7/2017	146.65	150.75	146.65	150.18	150.18	5561300	0.026872	150.967	156.1743

NFLX_summary.csv



	A	B	C	D	E	F	G	H	I	J
1		Open	High	Low	Close	Adj Close	Volume	Daily Retu	10DMA	30DMA
2	count	252	252	252	252	252	252	251	243	223
3	mean	221.3164	224.2454	217.9696	221.3673	221.3673	8209865	0.003221	220.502	218.7355
4	std	59.79211	60.91309	58.3029	59.69811	59.69811	4991033	0.023119	57.91529	53.65962
5	min	146.13	147.27	144.25	146.17	146.17	2160500	-0.06137	150.115	155.069
6	25%	179.4	181.165	177.3625	179.22	179.22	5021075	-0.00813	179.601	176.2238
7	50%	194.5	196.135	192.075	195	195	6643700	0.000924	194.604	193.716
8	75%	281.235	286.16	275.21	280.4775	280.4775	9669375	0.014887	273.6485	271.1253
9	max	336.3	338.82	331.15	336.06	336.06	41587400	0.135436	328.205	320.8903
10										

Summary

This concludes a basic introduction to the use of the Pandas package for data analysis, similar to the basic data analysis and manipulations one would perform in Microsoft Office Excel. For more information about other methods to view and manipulate data in Pandas, please refer to the current documentation.

Introduction to the RIT REST API

REST APIs are a way of interacting with an application by sending HTTP requests (like those made by a web browser) to pre-defined URL endpoints (essentially web addresses) to request information or perform certain actions. Because the requests are made to URL endpoints, it's possible to use most programming languages to interact with a REST API, rather than being constrained to the use of only one language (for example via the VBA or MATLAB specific APIs).

The RIT Client provides a simple REST API to request information about the currently running case, as well as to submit/cancel trades and accept/decline tender offers. Detailed documentation about all the available functionality is available, but this tutorial will provide a brief introduction to interacting with the REST API via Python.

Setting Up Python

The 'requests' package in Python provides a set of methods to make and interact with HTTP requests, greatly simplifying the process. If you did not download and install it as part of the virtual environment set up, then run `conda install -n <ENV NAME> requests` or `pip install requests` to download and install it.

Similarly to the Introduction to Python section, the code examples can be saved into `.py` files in the working directory and run by entering `python <FILE NAME>.py` into the prompt.

Basic Use

The basic steps to use the 'requests' package to interact with the RIT REST API are as follows:

1. Import the 'requests' package.
2. Save your API key for easy access.
3. Create a Session object to manage connections and requests to the RIT client.
4. Add the API key to the Session to authenticate with **every request**.
5. Make a request to the appropriate URL endpoint, usually using the get() or post() methods.
    - In general, the base URL is `http://localhost:9999/v1/` followed by a method name and potentially some parameters.
    - For example, the /case endpoint would look like `http://localhost:9999/v1/case`
    - Or the /orders endpoint would look like `http://localhost:9999/v1/orders&ticker=CRZY&type=MARKET&quantity=100&action=BUY, where &ticker=CRZY&type=MARKET&quantity=100&action=BUY` are query parameters specifying a market buy order for 1000 shares of 'CRZY'.
6. Check that the response is as expected.
7. Parse the returned data (if applicable) by calling the json() method.
8. Do something with the parsed data.

For example, consider the following example to get the current case status and print out the tick number (time elapsed in the case). The inline comments match the lines of code with the steps above:


```python
import requests # step 1
API_KEY = {'X-API-key': 'YOUR API KEY HERE'} # step 2

def main():
    with requests.Session() as s: # step 3
        s.headers.update(API_KEY) # step 4
        resp = s.get('http://localhost:9999/v1/case') # step 5
        if resp.ok: # step 6
            case = resp.json() # step 7
            tick = case['tick'] # accessing the 'tick' value that was returned
            print('The case is on tick', tick) # step 8

if __name__ == '__main__':
    main()
```

Important Notes

The port in the URL endpoint (9999 in the examples above) may be different, as noted in the documentation. Users can check **what port and API key to use by clicking on the `API` icon on the bottom status bar in the RIT client**.

Additionally, users can authenticate during an HTTP request by either submitting a header (as in the examples throughout this tutorial, where the session.headers dictionary is updated to include the API key), or the API key can be passed directly into the URL as another query parameter via `&key=YOURAPIKEYHERE`).

Submitting Orders

Orders can be submitted to the RIT Client by submitting a `POST` request to `http://localhost:9999/v1/orders`, with the following query parameters:

Parameter	Possible Values
ticker*	Tickers representing securities in the case
type*	'MARKET' or 'LIMIT'
quantity*	A number; quantity to trade
action*	'BUY' or 'SELL'
price	A number; required for 'LIMIT' orders

Note that parameters with an asterisk are required.

```
import requests
API_KEY = {'X-API-key': 'YOUR API KEY HERE'}

def main():
    with requests.Session() as s:
        s.headers.update(API_KEY)
        mkt_buy_params = {'ticker': 'CRZY', 'type': 'MARKET', 'quantity': 1000,
'action': 'BUY'}
        resp = s.post('http://localhost:9999/v1/orders', params=mkt_buy_params)
        if resp.ok:
            mkt_order = resp.json()
            id = mkt_order['order_id']
            print('The market buy order was submitted and has ID', id)
        else:
            print('The order was not successfully submitted!')

if __name__ == '__main__':
    main()
```

The example above shows the steps to submit a market buy order for 1000 shares of 'CRZY'. The order parameters are first saved into a dictionary, and then passed into the post request using `params=mkt_buy_params`.

In this example, we also check the response that is returned, to determine whether the order was successfully submitted (HTTP status code 200) or not, and then parse and return information about the order if successful.

```
import requests
API_KEY = {'X-API-key': 'YOUR API KEY HERE'}

def main():
    with requests.Session() as s:
        s.headers.update(API_KEY)
        lmt_sell_params = {'ticker': 'CRZY', 'type': 'LIMIT', 'quantity': 2000,
'price': 10.00, 'action': 'SELL'}
        resp = s.post('http://localhost:9999/v1/orders', params=lmt_sell_params)
        if resp.ok:
            lmt_order = resp.json()
            id = lmt_order['order_id']
            print('The limit sell order was submitted and has ID', id)
        else:
            print('The order was not successfully submitted!')

if __name__ == '__main__':
    main()
```

The example above shows the steps to submit a limit sell order for 2000 shares of 'CRZY' at a price of 10.00. The order parameters are first saved into a dictionary, and then passed into the post request using `params=lmt_sell_params`.

In this example, we also check the response that is returned, to determine whether the order was successfully submitted (HTTP status code 200) or not, and then parse and return information about the order if successful.

Cancelling Orders

Specific orders can be cancelled by order ID, or bulk cancelled by query string to match orders.

```
import requests
API_KEY = {'X-API-key': 'YOUR API KEY HERE'}

def main():
    with requests.Session() as s:
        s.headers.update(API_KEY)
        order_id = 100 # assuming the order to cancel has ID 100
        resp = s.delete('http://localhost:9999/v1/orders/{}'.format(order_id))
        if resp.ok:
            status = resp.json()
            success = status['success']
            print('The order was successfully cancelled?', success)

if __name__ == '__main__':
    main()
```

The example above shows how to cancel a specific order by submitting a `DELETE` request. Notice that instead of passing a parameter into the request, the order ID has to be added to the end of the URL, where the `{}` curly braces are located, by using the format() method.

After the response is returned, it is parsed to check if the order cancellation was successful or not, as indicated by the value of `status['success'].`

```
import requests
API_KEY = {'X-API-key': 'YOUR API KEY HERE'}

def main():
    with requests.Session() as s:
        s.headers.update(API_KEY)
        cancel_params = {'all': 0, 'query': 'Price>20.10 AND Volume<0'} # cancel all
open sell orders with a price over 20.10
        resp = s.post('http://localhost:9999/v1/commands/cancel',
params=cancel_params)
        if resp.ok:
            status = resp.json()
            cancelled = status['cancelled_order_ids']
            print('These orders were cancelled:', cancelled)

if __name__ == '__main__':
    main()
```

Orders can also be bulk cancelled via a `POST` request to `http://localhost:9999/v1/commands/cancel`. In the example above, the query for 'Price>20.10 AND Volume<0' would select all open orders with a price above 20.10 and volume less than 0 (i.e. sell orders).

The response returned will be a list of order IDs for those orders that were cancelled.

Other possible query parameters are as follows:

Parameter	Possible Values
all	0 or 1; set to 1 to cancel all open orders
ticker	Tickers representing securities in the case; cancels all open orders for the given ticker
ids	Order ids separated by commas
query	A query string to cancel orders that fulfil the given criteria

Algorithmic Trading Example – Arbitrage

This example assumes that users are building the arbitrage Python code while connected to the RIT Client with the ALGO1 case running. By default, the case runs for 300 seconds and there is one security that is traded on two different exchanges - CRZY_A and CRZY_M.

Before starting, please ensure that the 'requests' package has been installed in your Python virtual environment, as described in the Setting Up Python section above. Then, create a new `.py` file in your working directory (e.g. algo1.py).

Basic Setup

Similar to the example in the Basic Use section, we will first import the 'requests' package as well as the 'signal' and 'time' packages in order to create some helpful boilerplate code to handle exceptions and CTRL+C commands to stop the algorithm. Then, we'll also save the API key for easy access.

```python
import signal
import requests
from time import sleep

# this class definition allows us to print error messages and stop the program when needed
class ApiException(Exception):
    pass

# this signal handler allows for a graceful shutdown when CTRL+C is pressed
def signal_handler(signum, frame):
    global shutdown
    signal.signal(signal.SIGINT, signal.SIG_DFL)
    shutdown = True

API_KEY = {'X-API-Key': 'YOUR API KEY HERE'}
shutdown = False

```

While there are many other ways to switch on/off the arbitrage algorithm, we will use the current time (or 'tick') of the simulation case to signal when the algorithm should run. Therefore, we then need a method to get the current case status and return the current time (or 'tick'). So we create a helper method to send a `GET` request to `http://localhost:9999/v1/case`.

```python
# this helper method returns the current 'tick' of the running case
def get_tick(session):
    resp = session.get('http://localhost:9999/v1/case')
    if resp.ok:
        case = resp.json()
        return case['tick']
    raise ApiException('Authorization error. Please check API key.')

```

We also need a way to get the current bid and ask prices for a given security from the case.

```python
# this helper method returns the bid and ask for a given security
def ticker_bid_ask(session, ticker):
    payload = {'ticker': ticker}
    resp = session.get('http://localhost:9999/v1/securities/book', params=payload)
    if resp.ok:
        book = resp.json()
        return book['bids'][0]['price'], book['asks'][0]['price']
    raise ApiException('Authorization error. Please check API key.')

```

We can get the market book for a security by submitting a `GET` request to `http://localhost:9999/v1/securities/book`, with a query parameter of `ticker` equal to the ticker. After checking that the response is 'OK', we then parse the response. Finally, we return the price of the first bid and price of the first ask as a tuple, as they are sorted in order of competitive price.

We'll now set up the basic set up of a main() method as shown below.

```python
def main():
    with requests.Session() as s:
        s.headers.update(API_KEY)
        tick = get_tick(s)
        while tick > 5 and tick < 295 and not shutdown:
            crzy_m_bid, crzy_m_ask = ticker_bid_ask(s, 'CRZY_M')
            crzy_a_bid, crzy_a_ask = ticker_bid_ask(s, 'CRZY_A')

            # IMPORTANT to update the tick at the end of the loop to check that the algorithm
            should still run or not
            tick = get_tick(s)

if __name__ == '__main__':
    # register the custom signal handler for graceful shutdowns
    signal.signal(signal.SIGINT, signal_handler)
    main()
```

Operationally, when the file is run with `python <FILENAME>.py`, the get_tick(session) method will be called to return the current time of the case, and while (a) the time is greater than 5 seconds into the case and less than 295 seconds into the case, and (b) the 'shutdown' flag is false, the code in the while-loop will run. As the inline comment notes, it's important to update the tick variable at the end of the loop, so that the algorithm knows whether to continue running the while-loop or not.

Arbitrage Logic

Now that we have the helper methods to request information from the case, we just need to program the trading logic to check for arbitrage opportunities and submit the appropriate trades. We will write the trading logic under the 'while' command from the main() method to ensure that it only runs when the case is running.

```
def main():
    with requests.Session() as s:
        s.headers.update(API_KEY)
        tick = get_tick(s)
        while tick > 5 and tick < 295 and not shutdown:
            crzy_m_bid, crzy_m_ask = ticker_bid_ask(s, 'CRZY_M')
            crzy_a_bid, crzy_a_ask = ticker_bid_ask(s, 'CRZY_A')

            if crzy_m_bid > crzy_a_ask:


            if crzy_a_bid > crzy_m_ask:


            # IMPORTANT to update the tick at the end of the loop to check that the algorithm
            should still run or not
            tick = get_tick(s)

if __name__ == '__main__':
    # register the custom signal handler for graceful shutdowns
    signal.signal(signal.SIGINT, signal_handler)
    main()
```

Since ticker_bid_ask() returns both a bid price and an ask price for a particular security, we'll define the bid and ask prices for each security using the method.

The two arbitrage opportunities that exist are if the ask price of CRZY_A is less than the bid price of CRZY_M, or if the ask price of CRZY_M is less than the bid price of CRZY_A. Therefore, we'll write an if statement to check the two prices.

If the two prices are 'crossed' (i.e. once the if statement condition is satisfied), we'll submit a pair of market orders to buy one security at the ask price and to sell the other security at the bid price in order to capture the arbitrage profit. The corresponding commands are shown below.

```
            if crzy_m_bid > crzy_a_ask:
                s.post('http://localhost:9999/v1/orders', params={'ticker': 'CRZY_A', 'type':
                'MARKET', 'quantity': 1000, 'action': 'BUY'})
                s.post('http://localhost:9999/v1/orders', params={'ticker': 'CRZY_M', 'type':
                'MARKET', 'quantity': 1000, 'action': 'SELL'})
                sleep(1)

            if crzy_a_bid > crzy_m_ask:
                s.post('http://localhost:9999/v1/orders', params={'ticker': 'CRZY_M', 'type':
                'MARKET', 'quantity': 1000, 'action': 'BUY'})
                s.post('http://localhost:9999/v1/orders', params={'ticker': 'CRZY_A', 'type':
                'MARKET', 'quantity': 1000, 'action': 'SELL'})
                sleep(1)
```

In the first case, the algorithm should submit a market order to buy CRZY_A and a market order to sell CRZY_M. In the second case, the algorithm should submit a market order to buy CRZY_M and a market order to sell CRZY_A. A sleep() method was implemented after each pair order submission to ensure a stable execution of orders.

Running the Algorithm

Here's how the complete algorithmic command should look like:

```python
import signal
import requests
from time import sleep

# this class definition allows us to print error messages and stop the program when needed
class ApiException(Exception):
    pass

# this signal handler allows for a graceful shutdown when CTRL+C is pressed
def signal_handler(signum, frame):
    global shutdown
    signal.signal(signal.SIGINT, signal.SIG_DFL)
    shutdown = True

API_KEY = {'X-API-Key': 'YOUR API KEY HERE'}
shutdown = False

# this helper method returns the current 'tick' of the running case
def get_tick(session):
    resp = session.get('http://localhost:9999/v1/case')
    if resp.ok:
        case = resp.json()
        return case['tick']
    raise ApiException('Authorization error. Please check API key.')

# this helper method returns the bid and ask for a given security
def ticker_bid_ask(session, ticker):
    payload = {'ticker': ticker}
    resp = session.get('http://localhost:9999/v1/securities/book', params=payload)
    if resp.ok:
        book = resp.json()
        return book['bids'][0]['price'], book['asks'][0]['price']
    raise ApiException('Authorization error. Please check API key.')

def main():
    with requests.Session() as s:
        s.headers.update(API_KEY)
        tick = get_tick(s)
        while tick > 5 and tick < 295 and not shutdown:
            crzy_m_bid, crzy_m_ask = ticker_bid_ask(s, 'CRZY_M')
            crzy_a_bid, crzy_a_ask = ticker_bid_ask(s, 'CRZY_A')

            if crzy_m_bid > crzy_a_ask:
                s.post('http://localhost:9999/v1/orders', params={'ticker': 'CRZY_A', 'type':
                'MARKET', 'quantity': 1000, 'action': 'BUY'})
                s.post('http://localhost:9999/v1/orders', params={'ticker': 'CRZY_M', 'type':
                'MARKET', 'quantity': 1000, 'action': 'SELL'})
                sleep(1)

            if crzy_a_bid > crzy_m_ask:
                s.post('http://localhost:9999/v1/orders', params={'ticker': 'CRZY_M', 'type':
                'MARKET', 'quantity': 1000, 'action': 'BUY'})
                s.post('http://localhost:9999/v1/orders', params={'ticker': 'CRZY_A', 'type':
                'MARKET', 'quantity': 1000, 'action': 'SELL'})
                sleep(1)

            # IMPORTANT to update the tick at the end of the loop to check that the algorithm should
            still run or not
            tick = get_tick(s)

if __name__ == '__main__':
    # register the custom signal handler for graceful shutdowns
    signal.signal(signal.SIGINT, signal_handler)
    main()
```

In order to run the algorithm, ensure that the RIT client is connected and the REST API is enabled. Then, from the working directory, enter `python <FILENAME>.py` into the prompt. To stop the algorithm before the case is finished, press `CTRL+C`. If the file name has any space in it, please enter `python "<FILENAME>.py"`

Note: if students make changes to the algorithm's code while it is running in the prompt, those changes will not be reflected in what is running. Students will have to stop and restart the algorithm.

