==========

Nicholas Center for Corporate Finance
& Investment Banking

USING BLOOMBERG IN PYTHON

Wisconsin School of Business // September 2019

BLOOMBERG

CONTENTS >>>>>>>>>>>

USING THE BLOOMBERG API

The Bloomberg Terminal is a state-of-the-art tool that provides access to a wide array of live and historical financial data services. Integrating the Terminal into Python allows users to write scripts to systematically complete tasks that would otherwise be impossible or time-consuming.

Bloomberg API

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

An Application Programming Interface (API) is an intermediary that allows two programs to communicate with each other. Many applications, include the Bloomberg Terminal, come equipped with an API that allow outside users to incorporate the application's functionality into their own scripts and programs. One example of an integrated API can be found in Excel's VBA.

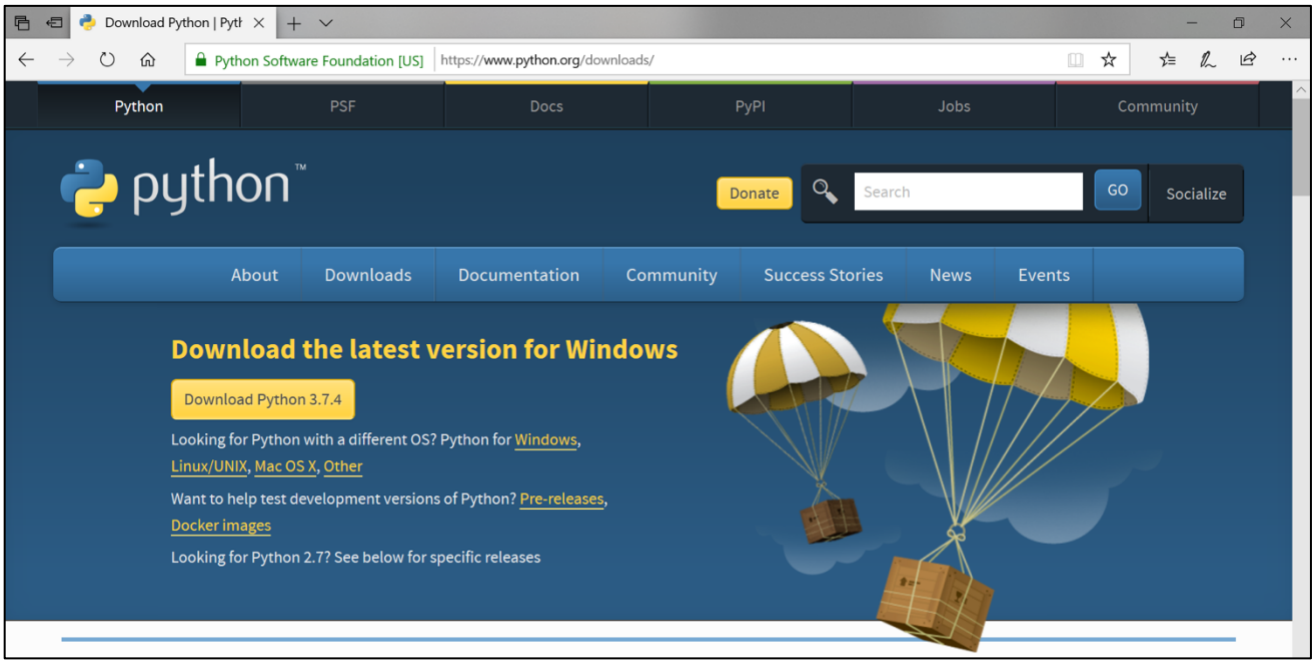
For the purposes of this guide, we will be showing how to use the Bloomberg API to connect the Bloomberg Terminal to Python. This API allows users to write scripts that will quickly complete repetitive, time-intensive tasks.

For more detailed documentation on the Bloomberg API, and how to use it with C++ and Java, visit https://www.bloomberg.com/professional/support/api-library/

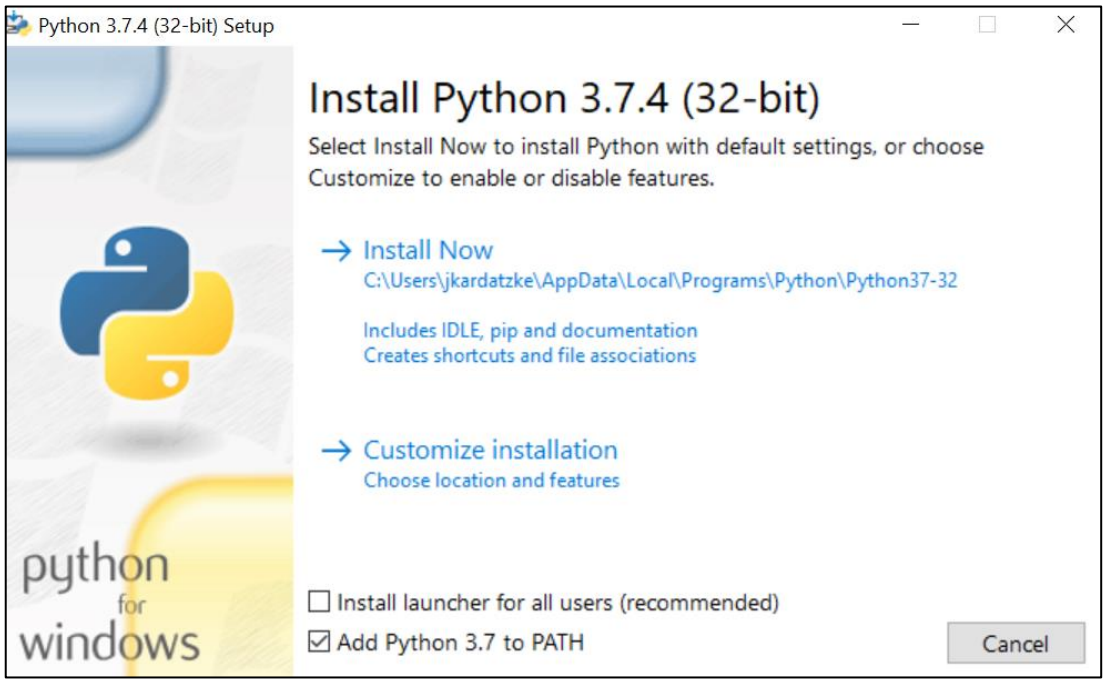
Setting Up Python

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Python (v3) should be downloaded and installed on your computer prior to completing any of the next steps. This can be done by visiting https://www.python.org/downloads/, choosing the latest stable release, and running the executable installer that coincides with your operating system.



While running the installer, make sure to check the option that asks whether you would like to add Python to your system PATH.



We recommend also installing an IDE, such as Visual Studio (https://visualstudio.microsoft.com/downloads/), which provides basic tools to read, write, and test Python code.

Installing API Packages

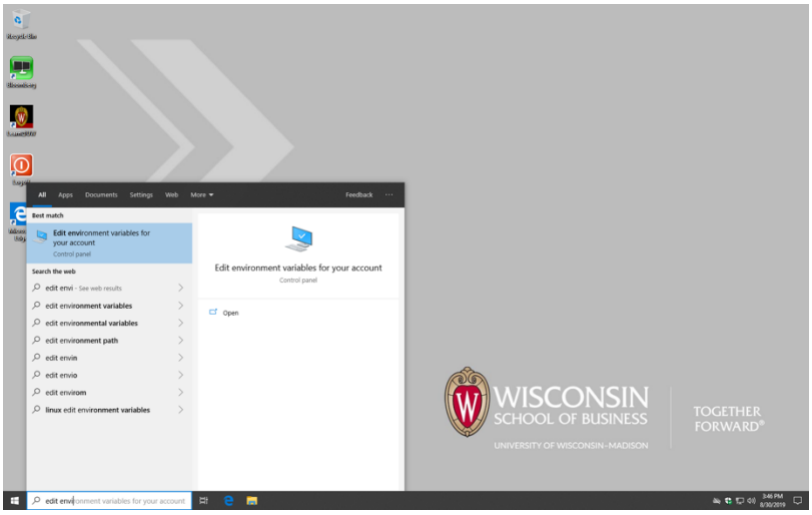
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

The Bloomberg-Python-API (BLPAPI) package must also be installed.

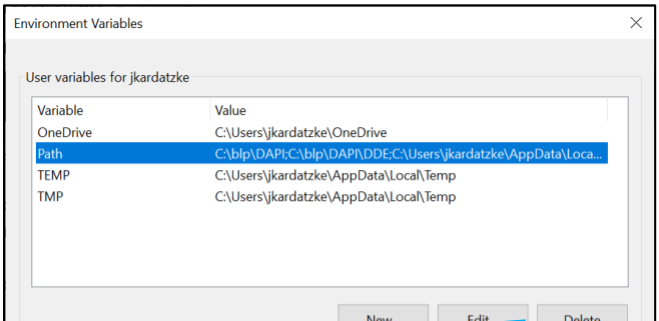
To do so, open up command prompt and enter “python -m pip install --index-url=https://bloomberg.bintray.com/pip/simple blpapi”.

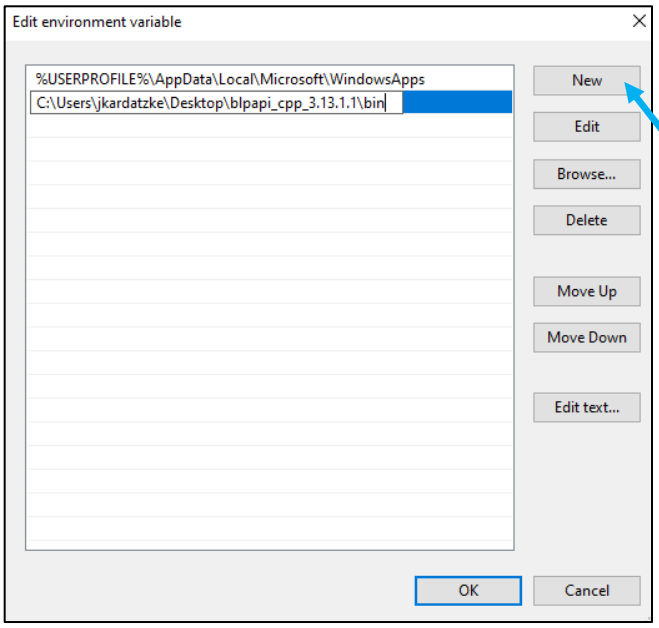
Finally, you must download the Bloomberg C++ SDK and add it to your account environment PATH. The SDK can found at https://www.bloomberg.com/professional/support/api-library/. If you are using a Windows computer, you should download the file titled "C/C++ Experimental Release" under the API Windows heading. After the download is finished, move the folder to a permanent location on your C Drive (for example, your Desktop or your Downloads folder). Copy the absolute location of the bin folder.

Then, complete the steps on the following page to add this folder to your environment PATH:



Select "Edit environment variables for your account" from the Windows menu





Choose "Path" and then click on "Edit…"

Click on "New" and paste the absolute location of the BLPAPI bin folder, then click "Ok"

Example Problem: Testing an Investment Thesis

To demonstrate how the Bloomberg API can be used with Python, we will go step-by-step through how to test an example investment thesis.

Investment thesis: Recent institutional investor selling activity is positively correlated with a stock's future performance.

First, open your IDE and save a new file titled "SimpleExample.py" to your desktop. Then, go to https://raw.githubusercontent.com/UW-Nicholas-Center/Adage/master/Other/history.py and copy and paste all of the code on the page into your file. Think of this file as a template for making historical data requests through the Bloomberg API. Many of the lines in this code will remained unchanged throughout different use cases of the API.

This code contains three mains sections[1]. First, a Bloomberg session is started by passing server data to the API. Next the data request is created, which specifies the time period, securities, and data fields. Finally, the data request is sent, and the results are parsed and interpreted. You do not need to understand how the first section works, but we will walk through the code for the other two sections and show how they can be altered to gather data and test our investment hypothesis.

The Data Request

We will go line-by-line over this section to describe its function. It runs from line 53 to line 65 in your example file.

```
refDataService = session.getService("//blp/refdata")
```

Your request uses Bloomberg's "reference data" service, which means that you are only accessing static data points, not a stream of updates.

```
request = refDataService.createRequest("HistoricalDataRequest")
```

It is also a historical data request, meaning that you are not limited to current data points.

```
request.getElement("securities").appendValue("IBM US Equity")
request.getElement("securities").appendValue("MSFT US Equity")
```

[1] See Appendix 1 to reference which code belongs to which section

These lines specify that you are requesting data from IBM and Microsoft. You are able to request data from any of the stocks, bonds, or other securities that are included in Bloomberg Professional. For the request to work correctly, the name of the security must match precisely with its counterpart in Bloomberg. Use the Terminal's command line to search for securities.

```
request.getElement("fields").appendValue("PX_LAST")
request.getElement("fields").appendValue("OPEN")
```

These lines specify which data fields should be returned by the request. "PX_LAST" will return the closing price of a security on a given date, while "OPEN" will return its opening price. To find out which data fields can be referenced for a given security, use the "Field Finder" function in the Bloomberg Terminal. Note that Bloomberg does not store historical data for some fields.

```
request.set("periodicityAdjustment", "ACTUAL")
```

If periodicityAdjustment is set to "REAL", all dollar-demarcated data fields will be adjusted for inflation.

```
request.set("periodicitySelection", "MONTHLY")
request.set("startDate", "20060101")
request.set("endDate", "20061231")
```

This request will return data from monthly intervals between January to December of 2006.

```
request.set("maxDataPoints", 100)
```

*This request will be stopped if more than 100 data points are returned. In this case, this line won't do anything because we are only requesting 48 data points (2 securities * 2 data fields * 12 months). However, for larger data requests it can be useful to set a limit if you are worried about runtime.*

Parsing the Response

This section runs from line 66 to line 80 in your example file.

```
session.sendRequest(request)
```

This line sends the request which was created in the previous section, which prompts Bloomberg to send the requested data points to your session.

```
while(True):
    ev = session.nextEvent(500)
```

session.nextEvent() is the function for receiving the next set of data points from the Bloomberg session. For a historical data request, each set contains one day of requested data. For our example, that means there will be 12 separate events. The "500" within the parentheses tells this program to wait 500 milliseconds after one event is processed before moving onto the next one. This makes it easier to cut off the program in case of an error, which can be done by pressing Ctrl+C on your keyboard.

```
if ev.eventType() == blpapi.Event.RESPONSE or ev.eventType() ==
blpapi.Event.PARTIAL_RESPONSE:
    for msg in ev:
        comp_buys = []
        comp_sells = []
        comp_px_change = []
        sd = msg.getElement('securityData').getElement('fieldData').values()
```

```
        for date in sd:
```

The first two lines contain an IF statement that verifies that the API returned our data without any errors. If that is the case, it then iterates through every company that was returned as part of our request. We assign this company data to the variable "msg". We also create empty arrays for each company that will record the number of institutional buys and sells each day, as well as the price change over the following five days. We then assign the field data points to the variable sd, and begin to iterate through every date for the given company.

```
if(date.hasElement('EQY_INST_BUYS') and date.hasElement('EQY_INST_SELLS') and
date.hasElement('CHG_PCT_5D')):
   comp_buys.append(date.getElement('EQY_INST_BUYS').getValueAsFloat())
   comp_sells.append(date.getElement('EQY_INST_SELLS').getValueAsFloat())
   comp_px_change.append(date.getElement('CHG_PCT_5D').getValueAsFloat())
else:
      print("missing field")
```

For each date, we must ensure that we have data for each of the three fields that we included in our request. We do that with this IF statement, which will return an error message if it fails. We can then simply append values for each of the three fields to the arrays that we created earlier.

```
buys.append(comp_buys[:len(comp_buys)-1])
sells.append(comp_sells[:len(comp_sells)-1])
px_change.append(comp_px_change[1:])

if ev.eventType() == blpapi.Event.RESPONSE:
      break
```

Finally, we must make sure that the data from each company we iterate through is appended to a master array that will store data across every company and date.

Interpreting the Results

This section runs from line 115 to line 120 in your example file.

```
inst_change = []
px_change_flat = []
  for i in range(0, len(buys)):
      for j in range(0, len(buys[i])):
          inst_change.append(buys[i][j]/(buys[i][j]+sells[i][j]))
          px_change_flat.append(px_change[i][j])
print(np.corrcoef(inst_change, px_change_flat))
plt.scatter(inst_change, px_change_flat, alpha=0.5)
plt.xlabel('Institutional Buying Index')
plt.ylabel('% Change in Share Price (following week)')
plt.show()
```

This section first iterates through every date for every company in the data that was pulled, and calculates a ratio between institutional buys and institutional sells on that day. These ratios are appended into an array, which can then be used to find whether there is any correlation between institutional buying activity and next five-day price change. We do this in two ways, first by finding the correlation coefficient between the two variables and then by creating a scatter plot. Both of these outputs show there to be very little correlation, showing that this very basic measure of institutional buying activity is not an effective predictor of stock movement.