Guide

To

Interactive Brokers API

With Python Code,

Examples, and Explanations

Corbin Balzan

Table of Contents

DEDICATION

This guide is dedicated to hobbyists, programmers, students, and learners.

Created out of a need to fill knowledge gaps and enrich the community, “Guide to Interactive Brokers API” is focused on helping others explore their interest and gain a deeper understanding of systemic trading. Thank you for your support in helping the community grow and learn together. Customers like you make the creation of this book possible.

To learn how you can contribute further, please contact Corbin at corbin@thequantacademy.com

DISCLAIMER

This guide is written by Corbin Balzan for The Quant Academy. It is not sponsored or authored by Interactive Brokers or its members. The resources are intended to compliment Interactive Brokers API, not replace it in any way. When establishing your account with Interactive Brokers, you must agree to their requirements and restrictions.

Additionally, investing should be done at your discretion. This guide is intended to assist the user in configuring a programming environment to implement their own strategies. It does not offer investment advice. By downloading or using this guide, you are expressly agreeing that neither Corbin nor The Quant Academy is liable for losses you incur when running or testing your algorithm. Please take caution in your endeavors.

Any security or technique mentioned in the guide is for example purposes only. Demonstration code should be modified before production.

This guide also walks through the use of Amazon Web Services. These services, however, are owned by Amazon and independent of Corbin or The Quant Academy. Views expressed towards this service are advised by the conditions and services available at the time of writing.

DISTRIBUTION

By downloading this guide, you agree to only utilize the guide materials for independent use and may not be re-distributed, white-labeled, or claimed. **For official copies, please refer to thequantacademy.com**. The official resources are updated regularly and are only available to original purchasers.

Additional code, not developed by Interactive Brokers or others, is the copyright of Corbin Balzan. You may use the code for your own purposes, but can not distribute or claim as your original work. If you’d like to use the guide resources for commercial use or group distribution, please reach out to corbin@thequantacademy.com. Corbin is more than happy to collaborate with your efforts and provide additional resources to ensure your success.

Chapter 1: Introduction

WHY AUTOMATE FINANCE?

In 1995, a firm called Renaissance Capital changed the way the world used mathematics and data to interact with the finance world. Founded by a famous and respected mathematician, James Simons, Renaissance Capital began breaking, then shattering Wall Street records. Between the years of 1994 to 2014, Renaissance returned an unprecedented 71.8% annual average return.

Despite the firm's internal secrecy, the word of their success spread throughout Wall Street and the world. Today, the words "quantitative finance" buzz around the internet as anyone from the solo hobbyists to high profile investors, such as Marc Andreessen and Steven Cohen, try to duplicate Renaissance's success.

Despite the widespread discussion, successful quantitative finance and systemic methods of investing introduce little shortcuts. There's no publicly available secret or promise to get rich overnight; however, by gaining proficiency in algorithms and developing strategies of your own, it's more than possible to produce systems that can deliver an upper hand over traditional investors.

Before the 1960s, the traditional pattern of trading was to wait for a notification of a news article or signal, open and read the content, make a decision, then execute the trade. Usually, this process was done by representatives on the NYSE trading floor. Purchasing a security could take up to an hour, with no guarantee that you're receiving the price you expected. The process improved through a wave of technology that enabled individual brokers to trade through a digital terminal; however until the 2000s, a substantial human aspect remained, and the process took minutes.

Now, we are on the dawn of a new age of digital intervention in the stock market. Computers are able to process an incomprehensible amount of data with unparalleled execution times. Instead of hours or minutes, algorithms can execute the same trading process within seconds or even milliseconds. Trades are delivered magnitudes quicker and leave traditional investors scrambling to find a new advantage.

Algorithms deliver convictions at the best point calculated. They rarely fall prey to the human pitfalls that hold many traders back. There is no crippling hesitation, just probability and decisions. By learning quantitative finance methods, you will be able to drastically set yourself apart from others in the field of finance. Algorithms run as an extension of yourself and your strategies. By bringing your vision to life, you can learn quicker and profit more.

WHY INTERACTIVE BROKERS?

Interactive Brokers is commonly ranked among the top online trading platform for many reasons, here are a few:

- **Low commission and free trades:** Interactive Brokers offers one of the most competitive pricing platforms. With IBKR Lite, you receive free trading. With Pro, stock commissions are down to fractions of a cent per share for bulk orders with competitive options pricing.
- **Powerful trading platform:** The TWS is considered one of the best interfaces for advanced users. It offers quick and real-time data, tools like volatility lab and heat mapping, paper trading, and custom notifications.
- **Investment Selection:** The selection of investment assets include stocks, options, bonds, ETFs, forex, warrants, and futures. These assets can be traded in 31 countries on over 125 markets.
- **Robust API:** Finally, the feature most relevant to this guide is their extensive API. The Interactive Brokers API (IB API) gives access to all the features included in the Trader Workstation. This allows unrestricted access to trades from inside a program.

WHY PYTHON?

Python is quickly becoming one of the most powerful programming languages to exist. Large companies, universities, and disciplines have embraced the combination of simplicity and power of Python, which has led to widespread adoption.

Python is an object-oriented scripting language designed to make development easier and code more readable. The syntax is clean and straightforward, lending large amounts of content to be visually digestible. This guide implements fairly simplistic symbolism, such as colons, parentheses, and basic math operators, with the intent that only a beginner's level knowledge is necessary to complete this guide.

Python is known for reading similarly to the English language. Operations can be performed with keywords that contribute to a smooth learning curve. Users of this guide come from all backgrounds; hence, little programming experience is required. Any user of this guide should be able to digest all code snippets without extensive references to outside documentation (although it can't hurt to brush up on the fundamental components).

Widespread adoption has led to comprehensive documentation and forum support. So, many common questions can be answered with a Google search. There is a community of people on the internet capable of answering many of the problems users of this guide may encounter (especially on StackOverflow).

Despite its simplicity, Python is known to be one of the most powerful languages available. The applications of Python range from computation and data manipulation to natural language processing and artificial intelligence. Hence, the skills learned in this guide can carry over to several other disciplines and projects in the future.

Critics of the role of Python in quantitative finance mention the fact that Python is an interpreted language, as opposed to alternatives such as Java, which are pre-compiled; however, for the majority of applications, the difference is an unnoticeable influence in performance (obvious exceptions include strategies that employ HFT or instantaneous market reactions). For those determined to use Java, C, or others, the concepts presented in this guide are similar and the explanations will still prove to be valuable (the official documentation can assist in the syntax conversion).

In short, the brevity, readability, and power of Python make this language an excellent pair for algorithmic trading.

WHY THIS GUIDE?

The world of systematic trading can be incredibly confusing and competitive. Creating a great algorithm requires complexity and creativity. Every strategy is different, and there is no holy grail to being successful. This book exists as the quickest path to learn about the field of quantitative finance and generate a functioning algorithm.

Many of the producers of current algorithms give little back to the community to which they belong. Those with the most knowledge rarely fill the field with educational content. When I was just starting, I noticed the community severely lacked quality and depth.

Instead of needlessly spending dozens of hours troubleshooting, experimenting, and testing on your own, this book will help you accomplish the same algorithm in one-tenth of the time. Development teams can potentially save thousands of dollars and new strategists can gain exposure to a new range of ideas.

Other platforms exist for absolute beginners to experiment with; however, they are are nothing more than a restricted playground. After developing an algorithm on a 3rd party text editor, testing on a limited set of data, and deploying on a delayed engine, your algorithm would suffer in performance and creativity. In an effort to oversimplify quantitative trading for absolute beginners, other platforms have limited their user's algorithms from ever outgrowing their box. This guide is the opposite. It intends to expose readers to the power of Interactive Brokers and enables users to run and test programs on their own, more powerful systems.

This guide not only delivers a practical algorithm by the end, but it also sets the reader up on a platform to fulfill their expectations and execute their vision. The knowledge gained in this guide will empower you to create amazing, high-performance algorithms.

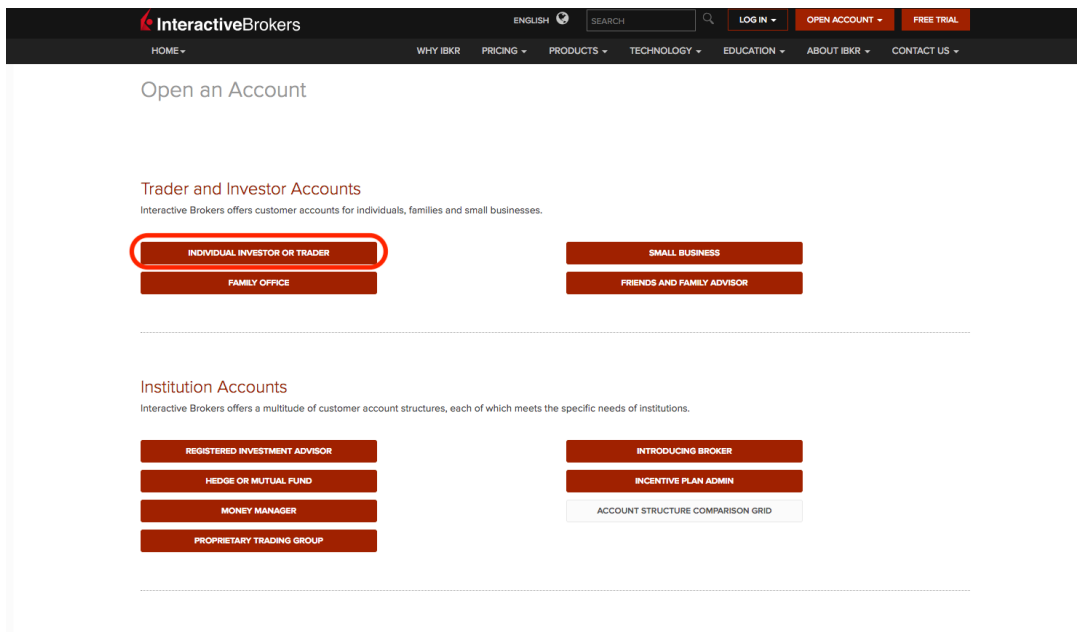
Chapter 2: Installation and Setup

ACCOUNT SETUP

Throughout the guide, the algorithm is developed in a local text editor (Sublime Text, VS Code, Textedit, etc.), run in the local terminal, then communicates directly with the Interactive Brokers servers through the Trader Workstation. The Trader Workstation is a GUI that is downloaded locally and listens for an incoming connection.

To gain access to the Interactive Brokers system, register an account online. Begin by searching "open an interactive brokers account" or by typing the address https://www.interactivebrokers.com/en/index.php?f=4695 into a web browser. You may also use a demo account at login to become comfortable with the Trade Work Station (Figure 2.3) by selecting “No username? Try The Demo”. However, the demo account will not work with the API and you will run into errors later on if you do not go through the account approval process.

On this page, titled "Open an Account", there are several options you can select depending on your situation. For the demo, I chose "Individual Investor or Trader" (Figure 2.1).



(Figure 2.1)

Selecting "Individual Investor or Trader" leads to a page with a large red button called "Start Application", where you can begin the sign-up process. After choosing an email, username, and password, you will receive a verification email where you can complete the full sign-up after logging in.

The application asks questions required for personal identification. These include questions about residency, investment experience, and income. Further, some user agreements require consent and approval before continuing to the end of the application.

Note: In this process, you should select a Pro account, instead of the Lite account. Lite does not offer access to the API, which this program operates on.

The application process is fairly lengthy and requires approval at the end that can sometimes take days. Unfortunately, this can not be circumvented and can slow down the process if you are eager to begin. However, many traders on the Interactive Brokers carry large balances and are subject to great risk. So, Interactive Brokers takes great measures to ensure users are ready.

This account is used to log in to the Trader Workstation. For the majority of the development of the application, we are using a paper trading account; however, in a later section, the account will need to be funded in order to gain permission to subscribe to live market data.

UNDERSTANDING THE DOCUMENTATION

Before we can fully begin testing, there is a short list of applications and folders that are needed from Interactive Broker. Interactive Brokers provides a list of downloads and setup instructions on their GitHub website. The documentation can be found by going to interactivebrokers.github.io/tws-api or by searching "Trader Workstation API documentation". This website contains an overview of the features included in the Interactive Brokers API software. Throughout this guide, there are many references to this page for implementation syntax. Although the documentation is excellent documentation for technical implementation details, it does not provide many critical steps and explanations covered here.

In this guide, we will extensively cover multiple sections of the documentation. These sections include, but are not limited to:

- Initial Setup
- Programming Architecture
- Connectivity
- Orders
- Account and Market Data

These are the primary components for developing a successful algorithm. Once you have gained the fundamental building blocks of a good program, the rest of the documentation will be much easier to comprehend.

DOWNLOADS OVERVIEW

Before beginning the downloads, it is essential to check system requirements for the Trader Workstation. Navigate to the Introduction portion of the documentation and familiarize yourself with some of the limitations. For most users and intents, your system and environment should be more than adequate to handle the software and make a request to the API.

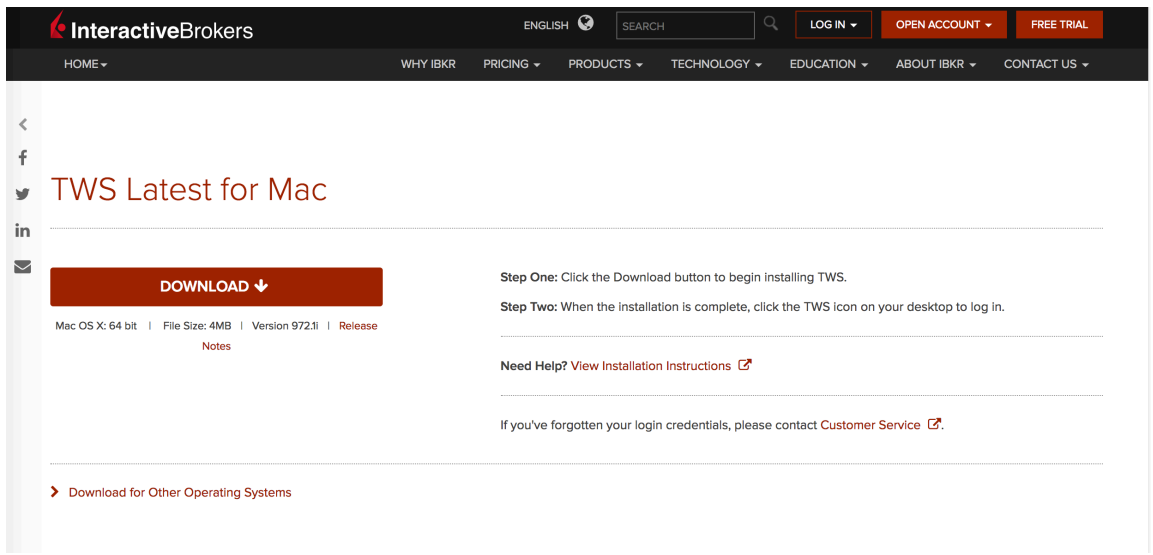
Due to the breadth of possible local system configurations and environments, this guide will not cover an exhaustive list of possible configurations you may require. We will only discuss the significant pieces of software necessary to perform commands covered in this book.

Note: In some cases, users may find difficulties or limitations with their machines that can be resolved by setting up a virtual environment elsewhere (whether it be on an AWS EC2 or Virtual Machine).

THE TRADER WORKSTATION (TWS)

To begin the download of the Trader Workstation, search "Trader workstation download" and find the page with the latest version (Figure 2.2).

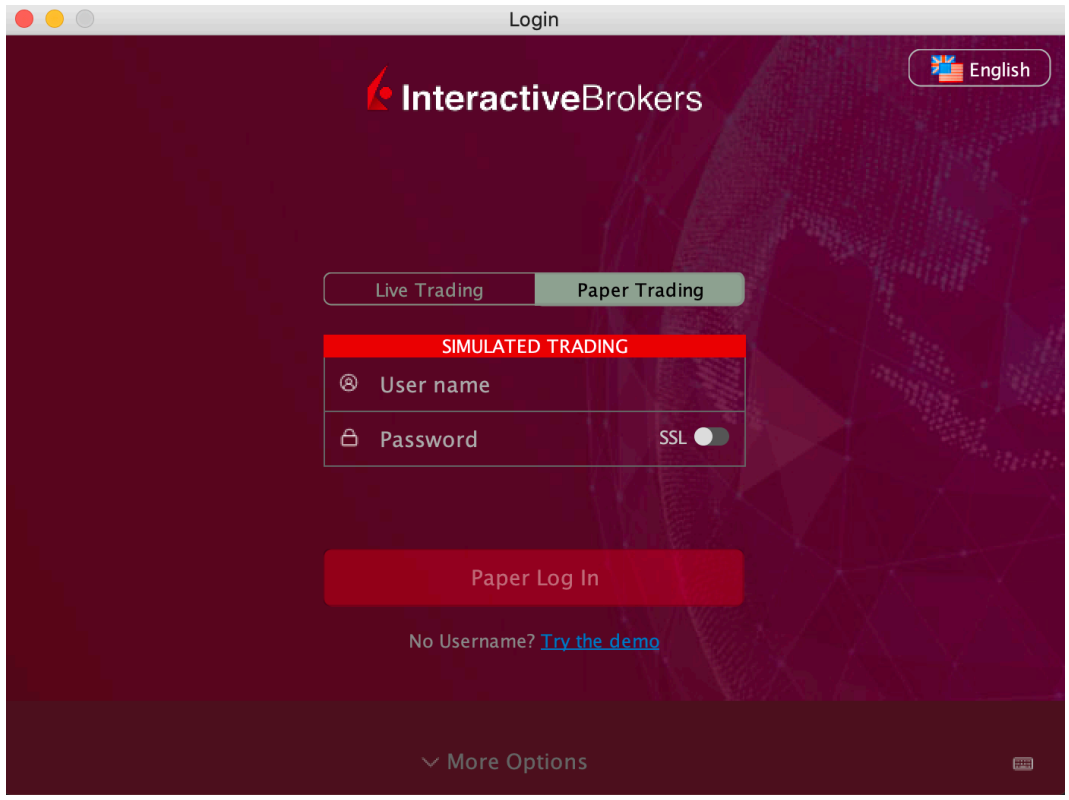
Note: you may need to click the "Download for Other Operating Systems" link if your version does not match the operating system displayed.



(Figure 2.2)

By clicking download, a .dmg (or .exe) file will be downloaded, and you can begin the installation wizard as instructed. The wizard installs the Trader Workstation on your desktop or in the applications

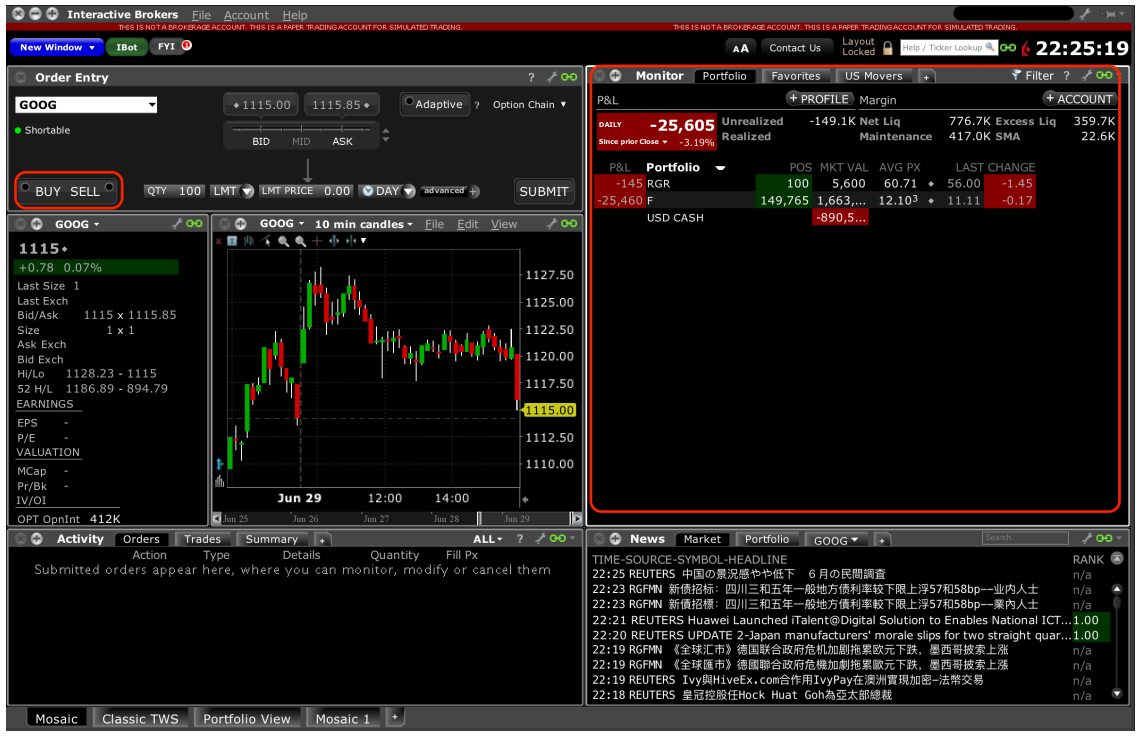
folder. Double click on the icon to start the program. You will know you are successful when the login screen appears (Figure 2.3).



(Figure 2.3)

At this point, you will be able to log in to your account and start experimenting with TWS's features. It's valuable to familiarize yourself with the options available on the Trader Workstation as there are many similarities to the options offered through the API.

To familiarize yourself with the TWS, you can begin by placing orders and observing the changes in the portfolio on the right-hand side (recommend logging into a paper trading account). We will be referring to the portfolio section frequently in the future, as this is where outputs from the program are most noticeable (Figure 2.4).



(Figure 2.4)

Beyond this section, there is no need for detailed coverage of the TWS interface. The Trader Workstation is a Graphical User Interface (GUI) that has roughly the same features we will use in our Python script; however, this GUI lacks the automation that makes our program special.

Note: There are other tutorials online that offer walkthroughs of the GUI if you'd like additional information.

THE API FILES

To begin diving into the API, download the necessary files by navigating to http://interactivebrokers.github.io. Read over the terms and conditions, then download the latest stable (downloading latest may cause errors) software that matches your operating system. After unzipping the download, move the API files to the directory (folder) you have created for this project. Our program files will be adjacent to the file tree to these API files (Figure 3.2 for reference).

Note: You're welcome to relocate the main Python script; however, you would then need to modify the imports.

Once the files are in a directory that is easily accessible to you, we now need to complete the following installations and download the necessary Python packages. First, it is also important to build a source distribution so the files can access the correct targets. To start these downloads, we will be accessing setup.py, a file included in the API that directs us to the correct downloads. Setup.py is under the folder called "pythonclient". To access this file from the root folder, cd through the file tree with the following command (which may differ depending on device):

```
cd twsapi_macunix/IBJts/source/pythonclient
```

In the pythonclient folder, type "`ls`" to ensure that setup.py is there (Figure 2.5).

```
[Corbin-Macbook-Pro:ibAlgo corbinbalzan$ cd twsapi_macunix/IBJts/source/pythonclient
[Corbin-Macbook-Pro:pythonclient corbinbalzan$ ls
MANIFEST.in     build           ibapi           setup.py        tox.ini
README.md       dist            ibapi.egg-info  tests
Corbin-Macbook-Pro:pythonclient corbinbalzan$
```

(Figure 2.5)

If it is, then you're in the right level of the folder. From within the pythonclient folder, run the following commands in your CLI:

To install IB API Python packages:

```
python3 setup.py install
```

To build the source distribution:

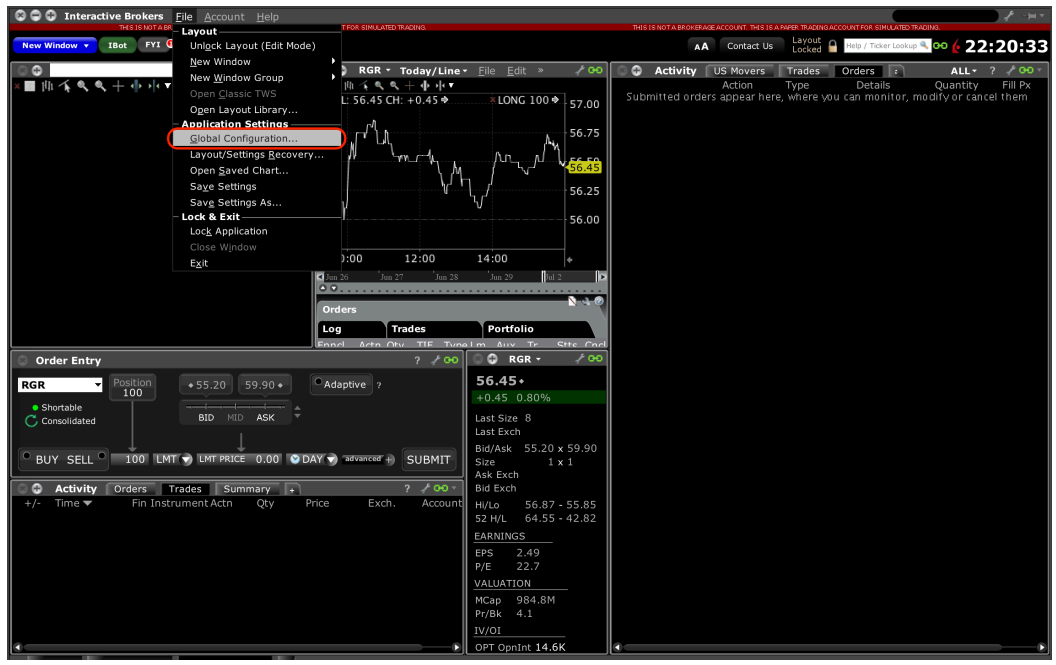
```
python3 setup.py sdist

python3 setup.py bdist_wheel

python3 -m pip install --user --upgrade dist/ibapi-9.73.7-py3-none-any.whl
```

Note: The ibapi-version number (the number after `dist/ibapi-`) may differ on this last command depending on your installation. Refer to the README inside pythonclient for the most up to date commands.

After these installs are complete, your API files are ready to send information to the TWS and Interactive Brokers servers.

PREFERENCES

The final part of the environment setup is to configure the TWS configuration settings. With the TWS logged in, navigate to global configurations (Figure 2.6).



(Figure 2.6)

Under setting configuration note several important setting options (Figure 2.7):

1) Enable ActiveX Socket Clients: This must be enabled to allow the TWS to accept API connections.

2) Read-Only API: Disable (or uncheck) this as sending orders to the TWS require write access.

3) Socket Port: Set this to 7496 for regular brokerage accounts and 7497 for paper trading accounts.

4) Logging Level: Set this to "Detail" as this records all program logs that can be used for future reference or debugging.

5) Allow Connections from localhost only: For this guide, we are using the localhost to send requests. You can change this later if you'd like to access from an external or virtual machine.

(Figure 2.7)

With the TWS downloaded and configured and the API files downloaded, it is now time to begin writing the first line of the program!

Chapter 3: Macrostructure

PROGRAM OVERVIEW

To begin programming with the Interactive Brokers API, it is first essential to understand the API information exchange. Our program is structured to make the information preparation and exchange as comprehensible as possible. Therefore, the main content of ibProgram1.py is divided into nine parts:

1) Import statements: Defines the classes we import from relative locations in the file tree.

2) Global Variables: Store and allow access to values throughout the program. These can also later be limited in scope to local variables as desired.

3) Custom classes and methods: Classes and methods that are created throughout the guide on an as-needed basis.

4) TestWrapper Class: Class to implement the EWrapper interface and handle the returns from our server requests. In other words, this is the "GET" portion of the information exchange. The wrapper is where we receive and handle information from the TWS by overriding API functions.

5) TestClient Class: Class which implements the ClientSocket and sends the request to the server. The client is the "POST" portion of the information exchange. In this section, we invoke functions to send messages to the TWS.

6) TestApp: Establish connection variables, algorithm preferences, and initialize the program.

7) Execution: Function where we start the program, call associated methods, and print messages to the console.

8) Input area: Where we connect to an input source and listen for incoming information. In this guide, scrape information from news sources and monitor for trigger words.

9) Processing area: Interprets the input source and delivers a decision to the correct Interactive Brokers object. That is, if there is a word grouping trigger on Apple, we trade that stock in *Execution*.

To begin truly understanding this architecture, create a new file named ibProgram1.py. This script is where we will add the major program components. Let's start by outlining the structure we just defined with comments (Code 3.1). With many advanced programs, organization and functions can quickly get out of hand. So, it is good practice to leave thoughtful comments along the way.

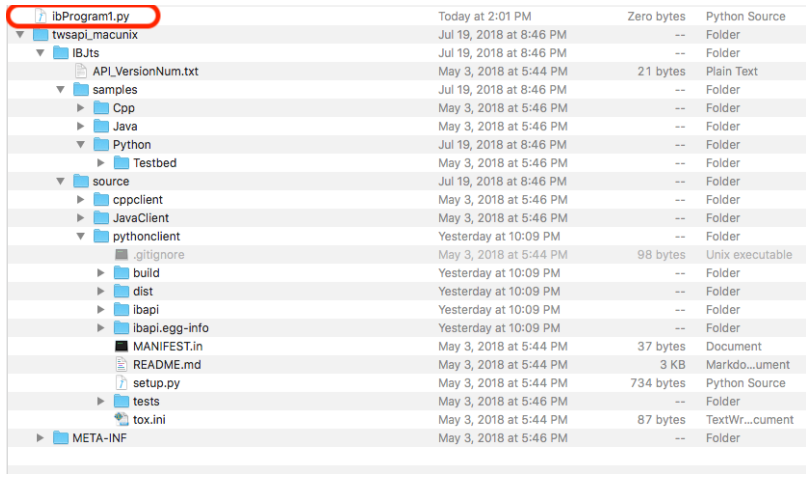
```
# Below are the import statements
# Below are the global variables
# Below are the custom classes and methods
# Below is the TestWrapper/EWrapper class
# Below is the TestClient/EClient class
# Below is the TestApp
# Below is the program execution
# Below is the import area
# Below is the logic processing area
```

(Code 3.1)

IMPORT STATEMENTS

To inherit the necessary classes and functions from the API interfaces, we need to import everything at the top of the program. To avoid redundancy and errors down the line, we use star imports to get known functions. Generally, star imports are not recommended due to their performance issues; however, for our purposes, star imports are quick enough. It's easy to revise imports later on after you discover the exact needs of your program and begin optimizing for speed.

At this point, if you are getting path errors, make sure you have the API files and main ibProgram1.py file set up as below (Figure 3.1). We are using dependent selectors to target the files in the IB API folder, so your file tree must be set up accordingly.



(Figure 3.1)

Now, inside ibProgram1.py, we import the necessary files from the "ibapi" folder. We are implementing the wrapper, client, contract, and order object from the API files. Then queue, date time, and time from the native Python package (Code 3.2).

```
# Below are the import statements

from ibapi.wrapper import *
from ibapi.client import *
from ibapi.contract import *
from ibapi.order import *
from threading import Thread
import queue
import datetime
```

```
import time
import math
```

(Code 3.2)

As we work through the program, you will understand exactly how each of these is implemented. But for now, it's helpful to gather the anticipated imports so we don't need to continually backtrack during development.

TEST WRAPPER

Now it's time to set up the Test Wrapper to handle the incoming messages from the IB server. This class is where we have the power to override the default return methods and process information in a structure that is easy to access. The methods added in TestWrapper are designed to manipulate, process, and store information returned from the IB servers (you'll see how this works more in a bit).

Our first step is to create the class signature of TestWrapper and pass the EWrapper interface (Code 3.3).

```
# Below is the TestWrapper/EWrapper class
'''Here we will override the methods found inside api files'''

class TestWrapper(EWrapper):
```

(Code 3.3)

Now, inside this class, it's time to create methods to handle returned errors. The server communicates with your code primarily through the following error code methods. Every CLI message, whether it's an error or just an update, comes through labeled as an error. The error codes can return several types of notices. The notices range from simple errors such as a broken pipeline (telling us our connection is faulty), to timeout expirations, or order completion success messages.

The first method, `init_error()`, is the first step in the process of storing messages (Code 3.4). This method initializes a Python queue that we can write to later. After the queue is created, it is stored inside the instance of the class. By using `self.my_errors_queue` , we can access the properties throughout `TestWrapper`.

```
class TestWrapper(EWrapper):
    # error handling methods
    def init_error(self):
        error_queue = queue.Queue()
        self.my_errors_queue = error_queue
```

(Code 3.4)

Next, create an `is_error` method to check the state of the queue and evaluate whether an error was returned. If `self.my_errors_queue` is not empty (meaning there is an error inside), then the `is_error` should return a *true* boolean. The value of this boolean will then be stored in the `error_exist` variable that we can access later.

```
def is_error(self):
     error_exist = not self.my_errors_queue.empty()
     return error_exist
```

(Code 3.5)

The third method, `get_error`, uses the `is_error` boolean value from the `error_exist` variable to get messages from the queue if they are returned (Code 3.6). If no errors are present, then `get_error` will return nothing.

Note: There is a timeout and try/except. If you would like to receive a message every time an exception is raised, you can add a printout or return in the exception.

```
def get_error(self, timeout=6):
     if self.is_error():
          try:
               return self.my_errors_queue.get(timeout=timeout)
            except queue.Empty:
               return None
```

(Code 3.6)

Finally, we override the error method in the API files to produce a message that is easier to read. Rewriting the returned message string can help us better interpret the value in the console and will make the debugging process a little easier. The error method creates a string from the TWS return values and pushes it onto our `my_errors_queue` (Code 3.7).

Note: This code snippet also includes all the previous error handling methods and is included for clarity.

```
class TestWrapper(EWrapper):

    # error handling methods
    def init_error(self):
        error_queue = queue.Queue()
        self.my_errors_queue = error_queue

    def is_error(self):
        error_exist = not self.my_errors_queue.empty()
        return error_exist

    def get_error(self, timeout=6):
        if self.is_error():
            try:
                return self.my_errors_queue.get(timeout=timeout)
            except queue.Empty:
```

```
                return None
        return None

    def error(self, id, errorCode, errorString):
        ## Overrides the native method
        errormessage = "IB returns an error with %d errorcode %d that says %s" % (id,
errorCode, errorString)
        self.my_errors_queue.put(errormessage)
```

(Code 3.7)

After the implementation of these four methods, your program is set up to handle errors. Errors are pushed from the server, not invoked by our application. So, we only need to add the methods here in the wrapper. Remember, the application is built on top of the Interactive Brokers API. So, many of the behind the scenes work is handled by their API files and the Trader Workstation. In the following examples in this guide, communication with the server is invoked in the client class (covered in the next section) and returned in the wrapper class (the class we just programmed in).

Now that error handling is ready, we can add methods responsible for accepting more robust messages from the server. These are received after an explicit request gets invoked in the client. The simplest example to prove we have established a connection is to ask for the server time.

Below our error methods, still inside the TestWrapper class, the time values are stored. Just like with the error messages, we begin with an init method that initializes a queue (Code 3.8). The queue is assigned to the variable `time_queue`. The variables act as a temporary holder that immediately assigns the queue to a class instance and returns the queue data structure.

```
# time handling methods
def init_time(self):
    time_queue = queue.Queue()
    self.my_time_queue = time_queue
    return time_queue
```

(Code 3.8)

Since the time queue is stored as an instance of the class, we can access the values in another wrapper method called `currentTime`. This is an overridden API wrapper method and allows us to place the value in `server_time` (our newly created queue) (Code 3.9).

```
# time handling methods
def init_time(self):
    time_queue = queue.Queue()
    self.my_time_queue = time_queue
    return time_queue

def currentTime(self, server_time):
    ## Overriden method
    self.my_time_queue.put(server_time)
```

(Code 3.9)

Now that we have a place to handle and store our errors and time, we can move to the TestClient to send the requests.

TEST CLIENT

The `TestClient` class is used by the API to send messages to the server. In this class, we do not override the methods like in the `TestWrapper`. The `TestClient` is just used to invoke messages and requests.

To begin, we must implement the wrapper as a constructor of the `TestClient`. This is necessary to start handling the returned messages. The socket must be initialized to an instance of the wrapper. An `_init_` method is used to facilitate this interaction (Code 3.10).

```
# Below is the TestClient/EClient Class

'''Here we will call our own methods, not overriding the api methods'''

class TestClient(EClient):

    def __init__(self, wrapper):
    ## Set up with a wrapper inside
        EClient.__init__(self, wrapper)
```

(Code 3.10)

After creating the constructor, declare the method to invoke a time request from the server (Code 3.11). The first line is a print message to the console, indicating that our function has run.

Note: When creating functions and testing for the first time, it is helpful to debug via print messages. Print messages are used throughout our program to communicate testing milestones and check the stored value in a variable at a certain point in time. Feel free to omit or add any print statements that are simple messages. Print statements that call methods can also be moved, but the method must still be called.

Next, the queue is initialized to handle the return before we make a request. This is done by calling the `init_time` function created in the wrapper.

The third line, `self.reqCurrentTime()`, is the most critical aspect of this `TestClient` class. The `TestClient` is responsible for sending the official request for information to the TWS. In this case, the request is officially invoked with the line `self.reqCurrentTime()`. This pattern of requests continues throughout the program, as you will see in the following sections.

```
def server_clock(self):

    print("Asking server for Unix time")
```

```
# Creates a queue to store the time
time_storage = self.wrapper.init_time()

# Sets up a request for unix time from the Eclient
self.reqCurrentTime()
```

(Code 3.11)

Following the server request, a max timeout is declared (Code 3.12). Timeouts are a good practice to follow for any request since Interactive Brokers can struggle to connect on certain occasions. In some cases, requests to the server may experience an error without any clear feedback to the tester. In this way, we watch for any lack of error messages by including the timeout on the queue's "get" request. This can be handled inside a try & except conditional (Code 3.12).

Finally, a `while` loop checks for errors stored from the `get_error` method created in the wrapper class. The loop prints any return values identified. On the case there is no error, the `server_clock` method will skip the loop and just return the value of time that we print in our execution area. At this stage, confirm your `TestClient` class looks very similar to Code 3.12. These client methods set us up to move forward to the third main IB API class, TestApp.

```
def server_clock(self):

    print("Asking server for Unix time")
# Creates a queue to store the time
time_storage = self.wrapper.init_time()

# Sets up a request for unix time from the Eclient
self.reqCurrentTime()

#Specifies a max wait time if there is no connection
max_wait_time = 10

try:
    requested_time = time_storage.get(timeout = max_wait_time)
except queue.Empty:
    print("The queue was empty or max time reached")
    requested_time = None

while self.wrapper.is_error():
    print("Error:")
    print(self.get_error(timeout=5))

return requested_time
```

(Code 3.12)

TEST APP

The `TestApp` class is where we establish the environmental variables. The first step to start the main script is to call `TestApp` to initialize all the classes. This class is run before every other method in our program, directly after the program is started. `TestApp` is primarily used to implement the `TestWrapper` and `TestClient` classes and begins the server connection.

Every execution inside the `TestApp` becomes encapsulated within an Init method (Code 3.13). The program begins by initializing the wrapper and client then starting a connection to the TWS. The connections is established with the `EClientSocket` object, `connect().Connect()` starts a TCP connection to the TWS with the specified IP address and port number of your computer. If a connection can not be opened, the TWS returns a 502 error code.

Note: This usually indicates an incorrect IP address or the TWS not being open and running on the host computer.

At program execution, essential information is exchanged to ensure a connection between the TWS server and the client. This information includes program versions, sync preferences, and security access.

Immediately after connection, execution threads are established. By default, there are two threads of execution:

1) Outbound thread from the client to send messages

2) Inbound to wrapper for adding messages to queues.

The EReader is responsible for reading and parsing data from these two threads. In the python API version, the EReader is initialized and started within `EClientSocket(connect())`, so we won't have to worry about this being called explicitly in our program.

In other languages, however, this will need to be created separately and you can refer to the documentation for further instructions. As the execution threads are stable, proper connection is completed.

As the final part of the TestApp class, we call `init_error` from the wrapper class to begin listening for any updates.

```
# Below is TestApp Class

class TestApp(TestWrapper, TestClient):
    #Intializes our main classes
    def __init__(self, ipaddress, portid, clientid):
        TestWrapper.__init__(self)
        TestClient.__init__(self, wrapper=self)
```

```
#Connects to the server with the ipaddress, portid, and clientId specified in
  the program execution area
self.connect(ipaddress, portid, clientid)

#Initializes the threading
thread = Thread(target = self.run)
thread.start()
setattr(self, "_thread", thread)

#Starts listening for errors
self.init_error()
```

(Code 3.13)

The `TestClient` and `TestWrapper` are now ready to send and receive communications to server!

PROGRAM EXECUTION

The final step is to trigger the proper functions when the program is started in the terminal. Since we are running ibProgram1.py as our main file, we want it to execute the script code only on the run of this specific module (Code 3.14), not whenever there is an import reference to it. In our case, this may not be necessary; however, as you begin to develop more advanced algorithms, it is good practice to control when you want to execute individual code blocks.

If you run `python3 ibProgram.py` in the terminal, this sets `__name__ = "__main__"` as True and executes the code inside (Code 3.14). The `TestApp` parameters specify the IP address, port number, and clientId of the TWS. In our case, we are running the TWS and program on localhost, IP 127.0.0.1. For the second parameter, the TWS is set to paper trading mode, which is port 7497. If you are using a live trading account, make sure to update this to 7496 (you can find the port in TWS preferences). The fourth parameter is clientId. This is set to zero and ignored as part of the TWS.

```
#Below is the program execution

if __name__ == '__main__':

    print("before start")

    # Specifies that we are on local host with port 7497 (paper trading port number)
    app = TestApp("127.0.0.1", 7497, 0)

    # A printout to show the program began
    print("The program has begun")

    #assigning the return from our clock method to a variable
    requested_time = app.server_clock()

    #printing the return from the server
    print("This is the current time from the server " )
    print(requested_time)
```

```
# Optional disconnect. If keeping an open connection to the input don't disconnet
# app.disconnect()
```

(Code 3.14)

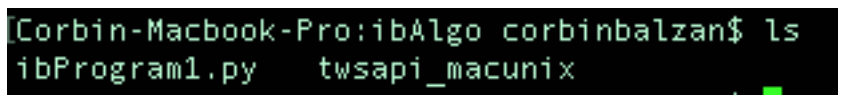
After the initialization of the program as the variable `app`, we print a start of program message then call the client method `server_clock` (Code 3.14). Server clock is referenced as an instance of `TestApp` and returns the Unix time to a variable that can be printed afterward.

Finally, we have the option to disconnect the application. This step is non-critical and is commented out as we develop an algorithm that is continuously listening for inputs.

Note: If you receive error messages relating to an inability of close type of app, then this step can be removed and you can manually end the program in the terminal (cmd+ c, or ctrl + c).

RUNNING THE PROGRAM

To start testing the program on your local machine, open the command terminal and navigate to the home folder where ibProgram1.py is saved (Figure 3.15). Open the Trader Workstation and keep that running in the background as we execute the program.

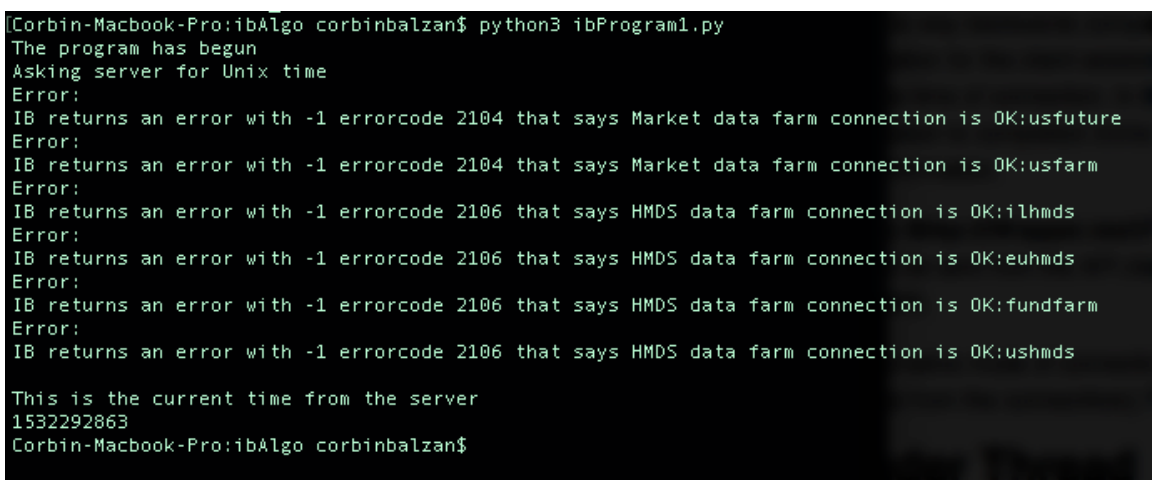


(Figure 3.15)

In the terminal, run:

```
python3 ibProgram1.py
```

then hit enter. This starts the program, and if the instructions were completed correctly, your output should look similar to below:

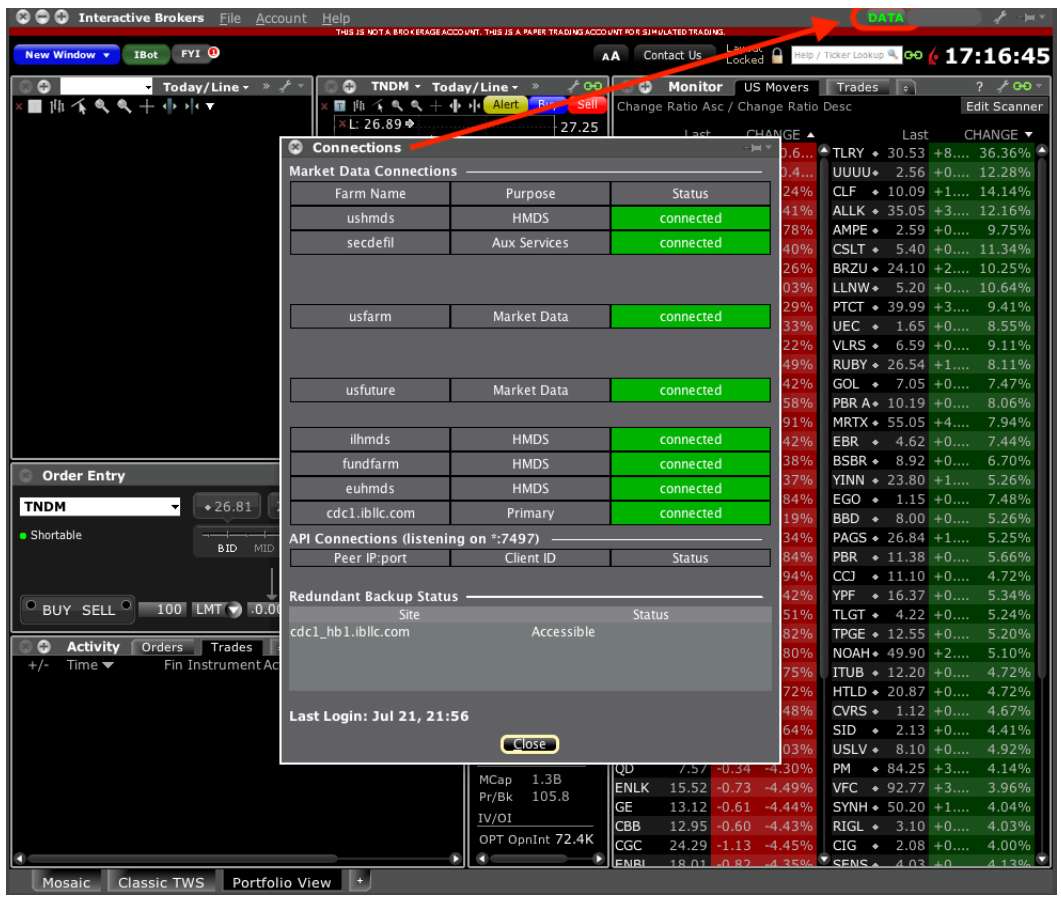


(Figure 3.16)

The program prints “The program has begun” from the program execution area, then a print status is returned from `server_clock`. Following, there is a list of messages returned from the error method in `TestWrapper`, with the time being returned shortly after.

Despite the “error” messages, this is a complete and properly functioning program. The error codes received were of type 2104 and 2106, which indicate a functioning data line. Remember, Interactive Brokers uses the error interface to communicate with the user for all messages; in this case, the “error” is indicating that we have properly connected to the data services that are part of the TWS.

If you click “Data” in the top right corner of the TWS, you will see the available pipes for market data (Figure 3.17).



(Figure 3.17)

Note: In some instances, if the program is running without a printout or returns a broken pipe error, then you may need to head over to the TWS and click a pop up to accept incoming connections. For other broken pipeline errors, check that your preferences match according to Chapter 2 and you have the correct IP address and port in the code.

Congratulations on your first Interactive Brokers program! We have created a Python script that sends a request to the server then receives a reply and prints that message. Although the return may seem simplistic, it indicates a proper connection to the TWS and establishes a great basis for a more advanced program in the following chapters.

Chapter 4: Adding Order Execution

HOW DOES ORDER EXECUTION WORK?

Orders are one of the most significant aspects of any finance application. After the program receives a flag from analysis, it is critical to deliver the resulting conviction quickly to the server and submit the order to the market. Interactive Brokers distinguishes itself in its ability to execute an order faster than its competitors. With over 35 options, there is an incredibly wide range of order methods that IB can fulfill.

For most beginning algorithms, it makes sense to begin with a simple market order. In the example algorithm, we will implement a method to buy and short stocks. As you become more familiar with the API, however, I encourage you to take a more in-depth look at the documentation to find order types that better suit your algorithm.

Logistically, orders are sent thought the TestClient class and received in the TWS. These orders are created as objects specified in the API code, filled with data from our application, then sent through the client method `placeOrder()`.

In this chapter, orders are sent with fixed information after a timed event. As we develop the algorithm further in the next several chapters, we will gather the necessary pieces to create a more dynamic algorithm.

BUILDING A CONTRACT OBJECT

The contract object is the first element needed to build a market trade. The exact object details can be found inside the "ibapi" folder with the file name contract.py (file path: *IBJts/source/pythonclient/ibapi/contract.py*). In a previous chapter, we imported this class on line 5, so it is available for use.

To understand the advanced parameters each contract type has available, read contract.py. Files within the source folder describe the API's implementation. In this way, contract.py offers insights into the complexities of each contract object; however, for examples, we refer to ContractSamples.py in *IBJts/samples/Python/Testbed*. The testbed files contain helpful methods that can be tested and implemented in your program with little modification needed. Interactive Brokers has provided this file with all the possible contract types to use as a reference. For our purposes, we use USStock (Code 4.1).

```
@staticmethod
def USStock():
    #! [stkcontract]
```

```
contract = Contract()
contract.symbol = "IBKR"
contract.secType = "STK"
contract.currency = "USD"
# In the API side, NASDAQ is always defined as ISLAND in the exchange field
contract.exchange = "ISLAND"
#! [stkcontract]
return contract
```

(Code 4.1)

This method is used and customized inside our main program, ibProgram1.py. We create this function below the import statements in the custom classes and methods section (Code 4.2). For now, the contract fields are populated with static data according to the following fields:

Symbol: The target ticker symbol of the security, now filled with Apple.

SecType: The security type, now filled with "STK" (stock).

Currency: The currency used. Since we are creating a USStock contract type, we are using US dollars.

Exchange: The desired exchange to trade this contract on. In our object we are using a "smart" exchange. Smart is a routing service offered by Interactive Brokers that continually scans markets and routes your order to the optimal exchange for price and availability.

PrimaryExch: Smart routing is recommended by default; however, if there is an issue with the smart system or you desire one exchange for your program, then you can override with a primary exchange (commented out in the example).

```
# Below are the custom classes and methods

def contractCreate():
    # Fills out the contract object
    contract1 = Contract()  # Creates a contract object from the import
    contract1.symbol = "AAPL"  # Sets the ticker symbol
    contract1.secType = "STK"   # Defines the security type as stock
    contract1.currency = "USD"  # Currency is US dollars
    # In the API side, NASDAQ is always defined as ISLAND in the exchange field
    contract1.exchange = "SMART"
    # contract1.PrimaryExch = "NYSE"
    return contract1    # Returns the contract object
```

(Code 4.2)

After creating the contract and populating the fields, we are ready to begin constructing the order object.

BUILDING AN ORDER OBJECT

The order object is the next step needed to build a market purchase. This object is found inside the "ibapi" folder with the file name object.py. In a previous chapter, we imported the class from this file on line 6, so like the contract object, it is ready to be used.

The template order object can be found inside OrderSamples.py in *IBJts/samples/Python/Testbed* (Code 4.3).

```
@staticmethod
def MarketOrder():
    #! [market]
    order = Order()
    order.action = action
    order.orderType = "MKT"
    order.totalQuantity = quantity
    #! [market]
    return order
```

(Code 4.3)

Similar to the contract object, we build the order object inside our main program, below the `contractCreate()` method (Code 4.4).

```
def orderCreate():
    # Fills out the order object
    order1 = Order()     # Creates an order object from the import
    order1.action = "BUY"    # Sets the order action to buy
    order1.orderType = "MKT"     # Sets order type to market buy
    order1.transmit = True
    order1.totalQuantity = 10 # Setting a static quantity of 10
    return order1    # Returns the order object
```

(Code 4.4)

Action: This specifies the market action. The example shows a purchase of stock; however, the same object can be filled with a "SELL" action. If a sell action precedes a buy action, that would be equivalent to a short order.

OrderType: Set to a market order for our demo, one of the simplest execution types. As stated before, this can be set to dozens of other types as seen inside OrderSamples.py.

T**ransmit:** This field ensures the IB server receives the order and passes through the Trader Workstation. In almost every case this should be set to "True".

T**otalQuantity:** The amount of shares we are purchasing with this order. Later on in this guide, the value will dynamically respond to the contents of the account.

Now that we have the contract and order object in place, we are ready to piece the two together and send an order through to the Trader Workstation.

SENDING THE ORDER

Since we have created methods to build the contract and order object, we are ready to instantiate those in variables and pass them into `placeOrder()`. The order is submitted in a custom method called `orderExecution` (Code 4.5).

```
    # Below are the custom classes and methods

    def contractCreate():
        # Fills out the contract object
        contract1 = Contract()  # Creates a contract object from the import
        contract1.symbol = "AAPL"  # Sets the ticker symbol
        contract1.secType = "STK"   # Defines the security type as stock
        contract1.currency = "USD"  # Currency is US dollars
        # In the API side, NASDAQ is always defined as ISLAND in the exchange field
        contract1.exchange = "SMART"
        # contract1.PrimaryExch = "NYSE"
        return contract1    # Returns the contract object

    def orderCreate():
        # Fills out the order object
        order1 = Order()    # Creates an order object from the import
        order1.action = "BUY"   # Sets the order action to buy
        order1.orderType = "MKT"    # Sets order type to market buy
        order1.transmit = True
        order1.totalQuantity = 10 # Setting a static quantity of 10
        return order1   # Returns the order object

def orderExecution():

    # Places the order with the returned contract and order objects
    contractObject = contractCreate()
    orderObject = orderCreate()
    nextID = 101
    app.placeOrder(nextID, contractObject, orderObject)
    print("Order was placed")
```

(Code 4.5)

The key to the execution is a method called `placeOrder`. `PlaceOrder` accepts the contract and order object as the second and third parameters. The first parameter is `orderId`, which is an identifier from a sequence used to keep track of individual orders by number. The `orderId` identifier needs to be unique for each order until cleared inside the settings. This means that there can not exist two orders with the same id of 1. There needs to be one order with an id of 1, the next with 2, and so on. If you attempt to submit an order with a duplicate id, the execution will not complete and there will be no error returned in the terminal. Each orderId needs to be higher than all the previous.

For example:

1, 2, 3 - valid sequence

2, 1, 3 - not valid, fails on 1

1, 20, 39 - valid, all values increasing

For our first order, we test with an id of 101. This value can be used one time before it has to be incremented to 102 or greater for successive tests

Note: At the end of this section, we will create a function that automatically increments by asking the TWS for the previous valid orderId and adding one.

After the order is placed, we can finish with a print statement to show that the program has executed up to that point.

To run the program, we add a method call at the end of the program for `orderExecution`. A short wait of 3 seconds should be added ahead of `orderExecution` to give the TWS time to process initial returns (Code 4.6). This value can be lengthened or shorted to accommodate for the client machine's internet speeds (or replaced with listeners later on). An error may occur in the absence of a short timeout if there is a request made before the TWS connection has been established.

```
time.sleep(3)    # Wait three seconds to gather initial information
orderExecution()
```

(Code 4.6)

Now that the code is all set up, the next section will review how to run the script and check the output.

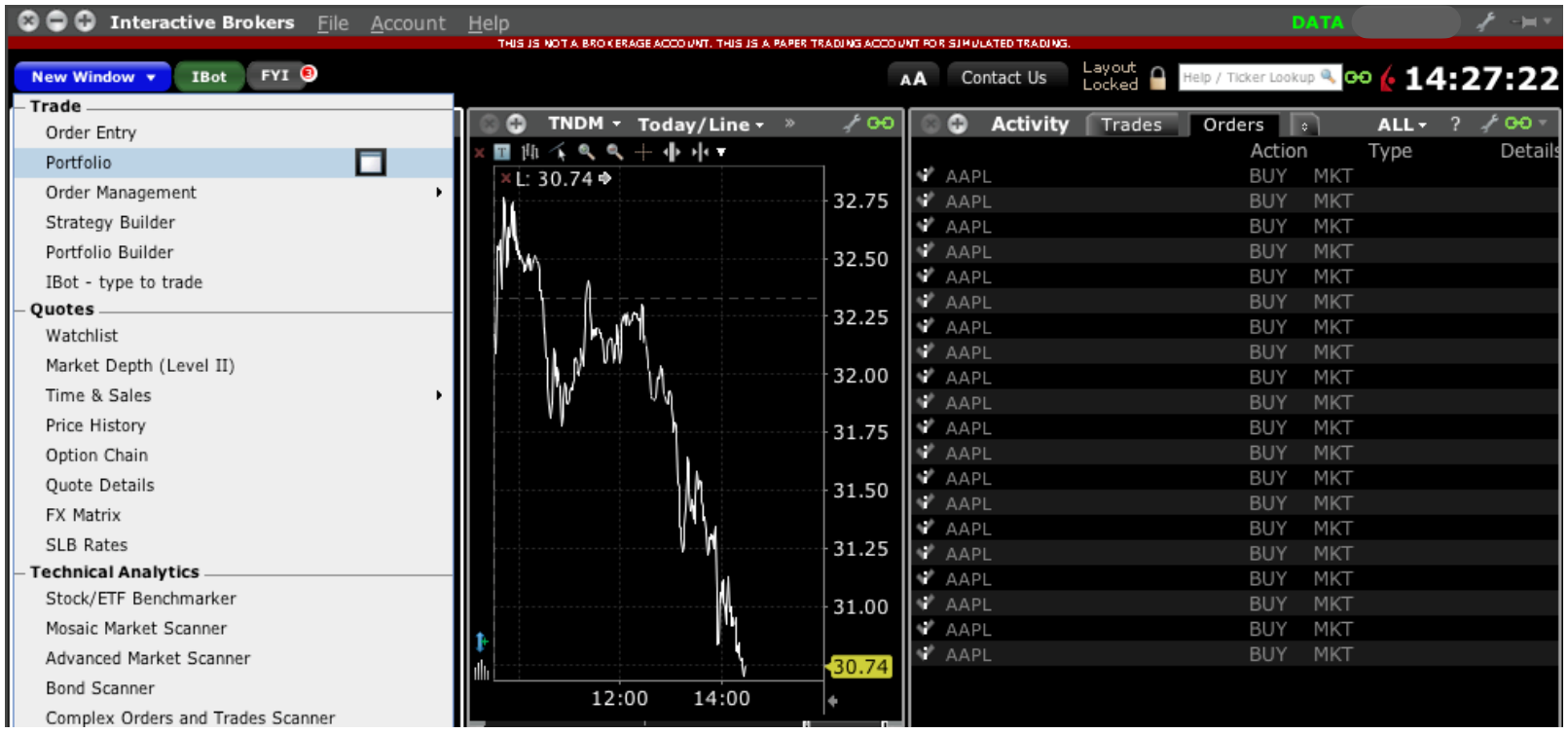
CHECKING OUTPUT IN THE TWS

To test the script, navigate to the parent directory of your program and run the program with python3 (Figure 4.7). The return will look similar to the output we received after our time return method in the second chapter. Then, after a three second wait, there will be a printout that says "order was placed" in the terminal.

```
[Corbin-Macbook-Pro:ibalgo corbinbalzan$ python3 ibProgram1.py
before start
The program has begun
Asking server for Unix time
Error:
IB returns an error with -1 errorcode 2104 that says Market data farm connection is OK:usfuture
Error:
IB returns an error with -1 errorcode 2104 that says Market data farm connection is OK:usfarm.us
Error:
IB returns an error with -1 errorcode 2104 that says Market data farm connection is OK:usfarm
Error:
IB returns an error with -1 errorcode 2106 that says HMDS data farm connection is OK:ilhmds

This is the current time from the server
1533665753
order was placed
```

(Figure 4.7)

In the TWS, you may receive a popup box or toaster message at the bottom of your screen with the details of your order. Then there will be an order registered in the "Activities" section of the TWS. If you are testing during market day, this will also register an action in the trade section (after the order has been fulfilled). Additionally, you can see the entirety of your portfolio after clicking "New Window" and selecting "Portfolio" in the dropdown (Figure 4.8).



(Figure 4.8)

UPDATING WITH ORDER ID

The final crucial element to developing a repeatable execution method is the ability to autogenerate orderIds in sequence. With every order placed, the orderId will automatically generate the next acceptable value. This will be the first parameter in the `placeOrder` function and replace the statically coded "101" that was used in the previous section.

Below `currentTime` in the wrapper, we implement two methods from the API. `NextValidId` will get the next valid ID from the TWS then act as a temporary storage until the value is accessed in `nextOrderId` and incremented by one (Code 4.9).

```
    def currentTime(self, server_time):
        ## Overriden method
        self.my_time_queue.put(server_time)

    # ID handling methods
    def nextValidId(self, orderId: int):
        super().nextValidId(orderId)
        logging.debug("setting nextValidOrderId: %d", orderId)
        self.nextValidOrderId = orderId

    def nextOrderId(self):
        oid = self.nextValidOrderId
        self.nextValidOrderId += 1
        return oid
```

(Code 4.9)

To access this value stored in `self.nextValidId`, we call the `nextOrderId` function and store the return value in the variable `nextID` (Code 4.10). So, whenever we need a newID, all that we have to do is call app.nextOrderId().

```
    # This function executes our order
    def orderExecution():
        contractObject = contractCreate()
        orderObject = orderCreate()
        nextID = app.nextOrderId()

        # Print statement to confirm correct values
        print("The next valid id is - " + str(nextID)

        # Place order
        app.placeOrder(nextID, contractObject, orderObject)
        print("order was placed")
```

(Code 4.10)

After implementing the methods in the wrapper and calling in orderExecution, you are ready to make continuous orders without intervention. Take some time to experiment with multiple orders (Code 4.11) or different order types.

```
# Calls the order execution function at the end of the program

time.sleep(3)
for x in range(3):
     orderExecution()
```

(Code 4.11)

We're now in a great spot to send messages to the TWS. Next section will focus on receiving information from the TWS.

Chapter 5: Retrieving Account Data From the Server

WHAT IS OFFERED?

In this section, we will pull certain pieces of our account information from the TWS to gain some realtime insights on our account status. The main request will be for “account summary” and “portfolio,” which hold crucial information regarding the buying, selling, and monitoring of our orders.

When creating the order object, we passed a value of 10 into the contract quantity; however, in the effort to create a dynamic model, this will be updated to a quantity relative to the total funds available. For this application, we request the available funds and buying power to determine a relative percentage (10 percent) to be used as the trade quantity.

Most beginner applications mainly rely on the available funds' value to calculate the quantity. But, for algorithms that incorporate margin trading, buying power is available as well. In addition, the algorithm shown here requests a current cash value of the account as part of the account summary information request. Cash value will not be utilized in our version of the program but is included at parts to demonstrate the handling of a request with an array of information.

REQUESTING ACCOUNT SUMMARY

Requesting the account summary gives us the ability to isolate specific values as parameters that we wish to receive from the TWS. We will establish a subscription connection to the account and receive updated information every 3 minutes (IB's default), or as needed. The first step to utilizing this information and ensuring it is readable at all points of the application is to create a global variable to hold the target values.

Note: The variables’ information can be controlled locally inside a class but for ease of use and consistency, we will assign it globally.

The global variables are assigned after the wrapper receives the return from the server. The default values will be overridden, so the zeros serve as a placeholder for now (Code 5.1).

```
# Below are the global variables

availableFunds = 0
buyingPower = 0
```

(Code 5.1)

Now, similar to before, it's time to set up the wrapper to receive incoming messages. The IB API determines the method used to receive the account status as `accountSummary`. By default, this method is configured to accept then print the account values to the command terminal; however, we override the default method to include an assignment to the global variables that were just initialized (Code 5.2).

When information is received, there is no unique identifier specifying the value types except for something IB calls the "tag". So, to isolate the values, it is easiest to filter tags via a conditional statement and assign the value to the corresponding global variable.

`AccountSummary` is followed by the `accountSummaryEnd` method to alert that the accessing of information has been completed (Code 5.2). This method is optional but can help debug to show that the request has completed successfully.

```
# Account details handling methods
def accountSummary(self, reqId: int, account: str, tag: str, value: str,
currency:str):
     super().accountSummary(reqId, account, tag, value, currency)
     print("Acct Summary. ReqId:", reqId, "Acct:", account, "Tag: ", tag, "Value:",
value, "Currency:", currency)
     if tag == "AvailableFunds":
          global availableFunds
          availableFunds = value
     if tag == "BuyingPower":
          global buyingPower
          buyingPower = value

def accountSummaryEnd(self, reqId: int):
     super().accountSummaryEnd(reqId)
     print("AccountSummaryEnd. Req Id: ", reqId)
```

(Code 5.2)

Now that the wrapper is ready to receive the account information, the request is sent through the client. The easiest way to control the message is to wrap the request line in a callable method named `account_update` (Code 5.3). We will ultimately be making the account information request multiple times throughout our program, so making the account summary callable is very useful in the future.

```
def account_update(self):
     self.reqAccountSummary(9001, "All", "TotalCashValue, BuyingPower,
AvailableFunds")
```

(Code 5.3)

The chain of events can now be triggered at the end of the program by calling the client method. This request should be placed before order execution to ensure the request has been completed and the global variables have the correct values (Code 5.4).

```
app.account_update()     # Call this whenever you need to start accounting data

time.sleep(3)
for x in range(3):
     orderExecution()
```

(Code 5.4)

Now before we run, it is helpful to add print statements in `orderExecution` to confirm the accuracy of the assignments (Code 5.5).

```
def orderExecution():

    # Places the order with the returned contract and order objects
    contractObject = contractCreate()
    orderObject = orderCreate()
    nextID = app.nextOrderId()
    print("Next valid id: " + str(nextID))
    print("Buying Power: " + str(buyingPower))
    print("Available Funds: " + str(availableFunds))
    app.placeOrder(nextID, contractObject, orderObject)
    print("Order was placed")
```

(Code 5.5)

There is now global access to account buying power and available funds. The `accountSummary` can be modified to filter any of the tags listed in Figure 5.6 (A screenshot of the official Interactive Brokers documentation). These additional values can be requested by adding the tag as a parameter in `reqAccountSummary`. After requesting values through `reqAccountSummary`, the assignment and access to these variables can be formatted similarly to `accountSummary`.

- AccountType — Identifies the IB account structure
- NetLiquidation — The basis for determining the price of the assets in your account. Total cash value + stock value + options value + bond value
- TotalCashValue — Total cash balance recognized at the time of trade + futures PNL
- SettledCash — Cash recognized at the time of settlement - purchases at the time of trade - commissions - taxes - fees
- AccruedCash — Total accrued cash value of stock, commodities and securities
- BuyingPower — Buying power serves as a measurement of the dollar value of securities that one may purchase in a securities account without depositing additional funds
- EquityWithLoanValue — Forms the basis for determining whether a client has the necessary assets to either initiate or maintain security positions. Cash + stocks + bonds + mutual funds
- PreviousEquityWithLoanValue — Marginable Equity with Loan value as of 16:00 ET the previous day
- GrossPositionValue — The sum of the absolute value of all stock and equity option positions
- RegTEquity — Regulation T equity for universal account
- RegTMargin — Regulation T margin for universal account
- SMA — Special Memorandum Account: Line of credit created when the market value of securities in a Regulation T account increase in value
- InitMarginReq — Initial Margin requirement of whole portfolio
- MaintMarginReq — Maintenance Margin requirement of whole portfolio
- AvailableFunds — This value tells what you have available for trading
- ExcessLiquidity — This value shows your margin cushion, before liquidation
- Cushion — Excess liquidity as a percentage of net liquidation value
- FullInitMarginReq — Initial Margin of whole portfolio with no discounts or intraday credits
- FullMaintMarginReq — Maintenance Margin of whole portfolio with no discounts or intraday credits
- FullAvailableFunds — Available funds of whole portfolio with no discounts or intraday credits
- FullExcessLiquidity — Excess liquidity of whole portfolio with no discounts or intraday credits
- LookAheadNextChange — Time when look-ahead values take effect
- LookAheadInitMarginReq — Initial Margin requirement of whole portfolio as of next period's margin change
- LookAheadMaintMarginReq — Maintenance Margin requirement of whole portfolio as of next period's margin change
- LookAheadAvailableFunds — This value reflects your available funds at the next margin change
- LookAheadExcessLiquidity — This value reflects your excess liquidity at the next margin change
- HighestSeverity — A measure of how close the account is to liquidation
- DayTradesRemaining — The Number of Open/Close trades a user could put on before Pattern Day Trading is detected. A value of "-1" means that the user can put on unlimited day trades.
- Leverage — GrossPositionValue / NetLiquidation

(Figure 5.6)

After running the program in your local environment, you should expect to see a similar output to the printouts in Figure 5.7. There is a printout from the wrapper function as well as the printouts from within orderExecution. These printouts show the program is able to access crucial account information.

```
Corbin-Macbook-Pro:ibAlgo corbinbalzan$ python3 ibProgram1.py
before start
The program has begun
Asking server for Unix time
Error:
IB returns an error with -1 errorcode 2104 that says Market data farm connection is OK:usfuture
Error:
IB returns an error with -1 errorcode 2104 that says Market data farm connection is OK:usfarm.us
Error:
IB returns an error with -1 errorcode 2104 that says Market data farm connection is OK:usfarm
Error:
IB returns an error with -1 errorcode 2106 that says HMDS data farm connection is OK:ilhmds
Error:
IB returns an error with -1 errorcode 2106 that says HMDS data farm connection is OK:euhmds
Error:
IB returns an error with -1 errorcode 2106 that says HMDS data farm connection is OK:fundfarm
Error:
IB returns an error with -1 errorcode 2106 that says HMDS data farm connection is OK:ushmds

This is the current time from the server
1535828244
Acct Summary. ReqId: 9001 Acct: DU928356 Tag:  AvailableFunds Value: 502273.60 Currency: USD
Acct Summary. ReqId: 9001 Acct: DU928356 Tag:  BuyingPower Value: 2009094.42 Currency: USD
Acct Summary. ReqId: 9001 Acct: DU928356 Tag:  TotalCashValue Value: 33008.77 Currency: USD
AccountSummaryEnd. Req Id:  9001
The next valid id is - 107
Buying power 2009094.42
Available Funds 502273.60
order was placed
```

(Figure 5.7)

REQUESTING PORTFOLIO HOLDINGS

To monitor our account regularly and ensure we can sell our positions when necessary, it is crucial to keep track of the portfolio holdings. This request is very similar to the process detailed in accounting information from the previous section.

The easiest way to store and manipulate the list of positions and their content is to create a dictionary. This new dictionary is initialized alongside the global variables so we can read and manipulate the items throughout the entire script (Code 5.8).

```
# Below are the global variables

availableFunds = 0
buyingPower = 0
positionsDict = {}
```

(Code 5.8)

Next, we create a method that sets up the wrapper to receive return information from the TWS. The `position` method below is an override of an API method and loads the returned position values into our global dictionary. This dictionary is structured as below (Code 5.9):

- **Key:** the stock symbol
- **Value:** Another dictionary of the positions and average cost of that stock.

Note: At the moment, the information printout is commented out. For debugging, it can be helpful to uncomment and see the return information.

```
# Position handling methods
def position(self, account: str, contract: Contract, position: float, avgCost: float):
     super().position(account, contract, position, avgCost)
     positionsDict[contract.symbol] = {'positions' : position, 'avgCost' : avgCost}
     # print("Position.", account, "Symbol:", contract.symbol, "SecType:",
contract.secType, "Currency:", contract.currency,"Position:", position, "Avg cost:",
avgCost)
```

(Code 5.9)

The next step is to invoke a method call within the client. This process is very similar to the request of the `accountSummary` where we create a method that can be triggered to request the server (Code 5.10).

```
def position_update(self):
     self.reqPositions()
```

(Code 5.10)

Finally, the request chain (`position_update`) is triggered at the end of the program next to `account_update` (Code 5.11). It helps to populate this dictionary before the program begins running so we can ensure there are values inside.

```
app.account_update()     # Call this whenever you need to start accounting data
app.position_update()    # Call for current position

time.sleep(3)    # Wait three seconds to gather initial information
```

(Code 5.11)

The request and output are now capable of receiving positions from the Trader Workstation. To make this a bit easier, however, we are going to iterate over each value in the dictionary and assign the symbol, position quantity, and average cost to readily available variables (Code 5.12). This will be especially helpful in later sections as we use the portfolio values to cover our positions.

```
orderExecution()
print(positionsDict)

# Iterates over everything in the positions list and reverses the quantity (covers
position)
for key, value in positionsDict.items():
     parsedSymbol = key
     parsedQuantity = value['positions']
     parsedCost = value['avgCost']
     print(str(parsedSymbol) + " " + str(parsedQuantity) + " " + str(parsedCost))
```

(Code 5.12)

Chapter 6: Subscribing to Market Information

GETTING MARKET SUBSCRIPTION APPROVAL

This next section shows the process of accessing market data from Interactive Brokers. This tool can be a powerful way to gain insights into the current state of the market. Information lines include, to name a few:

- U.S. Securities and Futures
- Bond Ratings
- OTC Markets
- U.S. Mutual Funds

By default, Interactive Brokers provides free delayed market information. You can subscribe to real time information (which we will be using) for a small fee as long as you meet some baseline requirements. You must:

- Establish an account with over two thousand dollars in value
- Maintain an account value above 500 dollars at all times

If you meet these requirements, you should have access to live data. Additionally, you have to pay a small fee depending on the market data bundle you subscribe to. The most up to date information regarding prices can be found at https://www.interactivebrokers.com/en/index.php?f=14193 or by contacting the Interactive Brokers Support.

Note: It is worth contacting their general support number in many cases if you run into any errors along the way. The general support personnel are usually helpful in this facet of the API

MARKET DATA FORMAT

The process of receiving market information is very similar to the method we defined in the account summary sections of chapter 5. A subscription is made to the server and a continuous stream of information gets delivered to the wrapper.

After a subscription, the information is returned as a "tick". The tick is a way of identifying the various parts of return information and is identified with a unique ID (Figure 6.1). The main tick types are bid price, ask price, bid size, and ask size. But, from the wrapper methods, we can receive even more information. The list included in this manual is not exhaustive; it only contains the most relevant components (Figure 6.1). The complete list can be found by navigating the side menu and looking under Streaming Market Data -> Top Market Data (Level 1) -> Requesting Watchlist Data -> Available Tick Types on the official Github documentation.

Tick Name	Tick Id	Description	Delivery Method
Bid Size	0	Number of contracts or lots offered at the bid price.	IBApi.EWrapper.tickSize
Bid Price	1	Highest priced bid for the contract.	IBApi.EWrapper.tickPrice
Ask Price	2	Lowest price offer on the contract.	IBApi.EWrapper.tickPrice
Ask Size	3	Number of contracts or lots offered at the ask price.	IBApi.EWrapper.tickSize
Last Price	4	Last price at which the contract traded (does not include some trades in RTVolume).	IBApi.EWrapper.tickPrice
Last Size	5	Number of contracts or lots traded at the last price.	IBApi.EWrapper.tickSize
High	6	High price for the day.	IBApi.EWrapper.tickPrice
Low	7	Low price for the day.	IBApi.EWrapper.tickPrice
Volume	8	Trading volume for the day for the selected contract (US Stocks: multiplier 100).	IBApi.EWrapper.tickSize
Close Price	9	The last available closing price for the *previous* day. For US Equities, we use corporate action processing to get the closing price, so the close price is adjusted to reflect forward and reverse splits and cash and stock dividends.	IBApi.EWrapper.tickPrice
Bid Option Computation	10	Computed Greeks and implied volatility based on the underlying stock price and the option bid price. See Option Greeks	IBApi.EWrapper.tickOptionComputation
Ask Option Computation	11	Computed Greeks and implied volatility based on the underlying stock price and the option ask price. See Option Greeks	IBApi.EWrapper.tickOptionComputation
Last Option Computation	12	Computed Greeks and implied volatility based on the underlying stock price and the option last traded price. See Option Greeks	IBApi.EWrapper.tickOptionComputation
Model Option Computation	13	Computed Greeks and implied volatility based on the underlying stock price and the option model price. Correspond to greeks shown in TWS. See Option Greeks	IBApi.EWrapper.tickOptionComputation
Open Tick	14	Current session's opening price. Before open will refer to previous day. The official opening price requires a market data subscription to the native exchange of the instrument.	IBApi.EWrapper.tickPrice

(Figure 6.1)

This chart shows the various traded security's information that can be received. Refer back to this chart for clarification as it may be helpful while filtering the return information by tick Id in the upcoming section.

Four crucial wrapper methods handle all the necessary information regarding contract information:

tickPrice - Returns all the price related tick information. IDs: 1,2,4,6,7,9,14

tickSize - Monitors size related information. IDs: 0,3,5,8

tickString - Returns information about the type of tick received. Most likely, this will return an ID of 45, which shows the Unix time of the request. ID: 45

tickGeneric - Also returns information about the tick. Commonly returns an ID of 49 and value of 0.0, which indicates the halted state on a tradable security. ID: 49

ACCESSING THE MARKET DATA PIPELINE

As per the usual procedure of requesting information, we will receive the tick values in the wrapper and invoke a request to the server from the client. In this request, however, the request is contract dependent. This means we will have to send the specific contract object along with the request from the client.

The first step is to initialize a variable to store the information from the data pipeline. This variable will hold the price of the requested security. The initialized value is set to one hundred million (Later, this will be changed to infinity when we import the math library). The large number is designed as a placeholder and is overridden after the request finishes processing. If the request does not return as expected when the script attempts to process an order through the TWS, the stock value will be interpreted as too high. This forces a quantity of zero to be traded. In this way, the large number acts as a final fail-safe.

```
# Below are the global variables
availableFunds = 0
buyingPower = 0
positionsDict = {}
stockPrice = 1000000
```

(Code 6.2)

Next, we move on to the implementation of the four main wrapper methods that are going to assign a value to `stockPrice`. The four methods, `tickPrice`, `tickSize`, `tickString`, and `tickGeneric`, all contain relevant information regarding the specific contract; however, for our purposes, we are most interested in receiving the most up to date price on a specific stock. Therefore, `tickPrice` is initially the most valuable to our program.

To store the price in a usable format, we will make the assignment to the `stockPrice` global variable from the returned tickPrice information. The most effective way of storing the price is to assign the price of the correct tick number to the `stockPrice` global variable. Since `tickPrice` is a method inside the wrapper class, we have to make the variable global by calling the global modifier in front of the variable name (Code 6.3).

To avoid an incorrect assignment to `stockPrice`, we use a conditional statement to confirm the correct tick information is available (Code 6.3). When the program runs, it may return a `tickPrice` for

any of the IDs, 1,2,4,6,7,9,14 (Refer to the chart in the previous section for all the possible return values). The "if" statement compares a `tickType` of 4, the last price of a stock, and assigns that price to the global variable `stockPrice`; however, if the algorithm is tested after market hours, then we catch tickType 9, the close price.

```
# Market Price handling methods
def tickPrice(self, reqId: TickerId, tickType: TickType, price: float, attrib: TickAttrib):
      super().tickPrice(reqId, tickType, price, attrib)
      # print("Tick Price. Ticker Id:", reqId, "tickType:",
      #      tickType, "Price:", price, "CanAutoExecute:", attrib.canAutoExecute,
      #      "PastLimit:", attrib.pastLimit, end=' ')

      global stockPrice  # Declares that we want stockPrice to be treated as a global
      global stockPriceBool    # A boolean flag that signals if the price has been updated

      # Use tickType 4 (Last Price) if you are running during the market day
      if tickType == 4:
            print("\nParsed Tick Price: " + str(price))
            stockPrice = price
            stockPriceBool = True

         # Uses tickType 9 (Close Price) if after market hours
      elif tickType == 9:
            print("\nParsed Tick Price: " + str(price))
            stockPrice = price
            stockPriceBool = True

def tickSize(self, reqId: TickerId, tickType: TickType, size: int):
      super().tickSize(reqId, tickType, size)
      # print("Tick Size. Ticker Id:", reqId, "tickType:", tickType, "Size:", size)

def tickString(self, reqId: TickerId, tickType: TickType, value: str):
      super().tickString(reqId, tickType, value)
      # print("Tick string. Ticker Id:", reqId, "Typxe:", tickType, "Value:", value)

def tickGeneric(self, reqId: TickerId, tickType: TickType, value: float):
      super().tickGeneric(reqId, tickType, value)
      # print("Tick Generic. Ticker Id:", reqId, "tickType:", tickType, "Value:", value)
```

(Code 6.3)

Now, whenever the wrapper receives a response from the server regarding tick information, the global price is updated and the value can be used throughout the program.

The next step is to send the request in the client. We create a method to execute the request in a process similar to the `account_update` and `position_update` functions (Code 6.4). In this case, however, the market data request requires information about which security for which it is receiving information. So, we have to pass the contract object of a specific security and a unique ID as a parameter into the API request. This request, `reqMktData`, requires six total parameters:

1) Ticker ID - This is used to identify the request. Uses next id, just like `placeOrder`.

2) Contract - Pass the contract object for the specific security.

3) Generic Tick List - Specify the comma separated numbers if you'd like to request additional information.

4) Snapshot - Boolean to specify the return frequency of the data. A True value specifies a one time return (or snapshot), and a false value returns a stream of updated data.

5) Regulatory Snapshot - A boolean value of the same system as the snapshot parameter but for users with a US Securities Snapshot Bundle.

6) Market Data Options - An array of additional market data options that can be specified for advanced users.

```
def price_update(self, Contract, tickerid):
     self.reqMktData(tickerid, Contract, "", False, False, [])
     return tickerid
```

(Code 6.4)

This function can be invoked any time as necessary as long as the contract object and next valid ID are passed as arguments. Since we build the trade details in the `orderExecution` method, it makes sense to call `price_update` directly after we make the contract and order object. As you can see, the first `price_update` argument is filled with the newly created contract object, and we ask for a new unique ID in the second argument (Code 6.5).

Following the nextID assignment, a short time.sleep is called to ensure the `tickPrice` has enough time to return information. The API connection works asynchronously, so without a short wait, there may be a server return too late (after other parts of the program that depend on the value have finished executing). Later we will remove this static time.sleep quantity of 2 seconds with a conditional wait period that checks for a change for the assignment of `stockPrice`.

```
# This function executes our order
def orderExecution(symbolEntered):

    # Call client methods to gather most recent information
    contractObject = contractCreate(symbolEntered)
    orderObject = orderCreate()
    app.price_update(contractObject, app.nextOrderId())
    nextID = app.nextOrderId()
    time.sleep(2)
    print("Global Tick Price " + str(stockPrice))

    # Print statement to confirm correct values
    print("Next valid id: " + str(nextID))
    print("Buying Power: " + str(buyingPower))
    print("Available Funds: " + str(availableFunds))

    # Place order
    app.placeOrder(nextID, contractObject, orderObject)
    print("order was placed")
```

(Code 6.5)

We now have a value stored in a global variable "stockPrice" that can determine the quantity of a security to purchase. The exact quantity we can purchase at any time is incredibly vital as the size of account grows. In chapter 8, we will utilize the current stock price we found to make more effective and active decisions.

Chapter 7: Gathering and Processing Input

STRATEGY AND TECHNIQUE OVERVIEW

By creating an automated algorithm that runs a python script in the command line, we have given ourselves the flexibility to deal with a wide array of input options. Several dominant strategies rely on avenues where data can be readily accessible and found in large quantities through APIs, open-source datasets, or web pages.

In this chapter, we establish an input source that can easily be adapted to fit a multitude of online strategies through web scraping. With web scraping, you have access to the internet's information without the need for a website to provide an official API. Additionally, web scraping can be utilized to generate large, formatted data sets. These data sets can then be loaded into your program for backtesting and analysis.

Our web scraping approach will focus on tracking breaking news articles published to major media sites. We gather specific URLs and ping the webpage at an interval to receive the most up to date information. This section covers several major media sites, all with small variants in their type of reports:

- **CNN:** Mainly focuses on politics
- **The Economist:** can be used for tracking economic policy
- **Reuters:** Has a wide variety of breaking news
- **Seeking Alpha:** Breaking company news and announcements.

This algorithm operates under the assumption that breaking news on earnings, dividends, and administrative changes are reported first on major news sites. In the real world, this is not always the case. So, depending on the strategy of your algorithm, you may want to substitute these websites for other sources to fit your strategy. Nevertheless, as an example, our four websites serve as excellent examples to begin practicing.

This scraper uses the Beautiful Soup library to make the parsing of our page requests easier.

Note: To learn more advanced methods or customize your traversal of the DOM, it is helpful to read the official documentation. There are also other robust scraper options such as Selenium or Scrapy.

CREATING A SIMPLE WEB SCRAPER

To get comfortable with web scraping and test strategies from the documentation, we create a separate file for testing before implementation. We create this file inside the same folder as ibProgram1.py and name it scraper.py. The implementation in this guide is geared towards beginner to intermediate scrapping strategies. So, by following along, you can gain sufficient knowledge to customize the scraper for your own websites and applications.

Note: If you run into problems with Beautiful Soup, refer to chapter 9 or the official documentation for help troubleshooting. Many of these website update their HTML quickly and without notice. If you find a website is not returning headlines as intended, refer to Chapter 9 for help identifying new HTML tags to target.

To begin using Beautiful Soup, download the Beautiful Soup and "requests" library into your folder. To do this, run these commands in the terminal:

1) Use this or a similar command to navigate to your algorithm's folder:

```
cd Desktop/ibAlgo
```

2) This downloads the request library that will be necessary to fetch web page content:

```
pip install requests
```

3) This downloads Beautiful Soup which will parse the returned content:

```
pip install beautifulsoup4
```

Next, we import Beautiful Soup and the request library into a Python script so we can set up the scraping methods. In addition, we will be importing time from Python to use as a crawl delay. The imports should be included at the top of the newly created scraper.py script (Code 7.1).

```
from bs4 import BeautifulSoup
import requests
import time
```

(Code 7.1)

Now, we initialize the appropriate global variables. The variable `textContent` is an array that will hold all the headlines parsed from the page. `CycleCount` stores the frequency at which we have called the crawl method, marking each request to the web server. (Code 7.2)

```
# Global variables for the scraper

textContent = []     # Holds the headlines in an array
cycleCount = 0       # Stores the frequency of requests made to the server
```

(Code 7.2)

Since the global variables are ready, we can move on to creating the primary crawl method. For the first example, we will traverse the DOM of The Economist. This method is designed to grab the page, parse the content, then iterate over the elements flagging the exact elements that match our identifiers.

After declaring the method, we create the first variable, `page_link` (Code 7.3). The `page_link` variable stores the URL of the website we are scraping. In this case, we pass in the path to the home page of The Economist.

```
# Below is the input area

def economistSearch():
    page_link = 'https://www.economist.com/'# Page Url to point request where to crawl
    page_response = requests.get(page_link, timeout=20)
    page_content = BeautifulSoup(page_response.content, "html.parser")
```

(Code 7.3)

The next variable, page_response, is temporary to store the response returned from our GET request of the page.

Note: Request.get is part of the request library that was installed and imported previously.

After passing the target URL as the first parameter of the GET, we establish a timeout for the request. In this example, the timeout is set to 20 seconds. This can be lengthened or shortened, depending on the client connection. If you run into a timeout error in your terminal, it is likely because your connection needs to be longer.

On line 17, after the response is received, it's time to use Beautiful Soup to parse the content. The first parameter accepts the content of the webpage, and the second parameter identifies the type of parser to use (Beautiful Soup also offers an XML parser).

After the content is parsed and readily manipulatable, it is time to gather the specific information needed for the analysis. The easiest way to gather specific elements is to iterate over all elements that match an identifier. Beautiful Soup offers a find_all method to mark HTML elements identified in the first parameter that match the class passed in the second parameter (Code 7.4).

```
for link in page_content.find_all("span",class_="flytitle-and-title__title",limit =
3):   # Finds all the spans with the class flytitle-and-title__title
     if link.text not in textContent:
          # print(link.text)  # Prints the title so we can verify correct operation
          textContent.append(link.text)    # Appends the headline to our main array
          tempHeadlineHolder.append(link.text)
```

(Code 7.4)

To confirm you are targeting the correct elements it is helpful to open the developer tool in your web browser and read the DOM elements. The exact location of the developer tools pane is found in

varying locations depending on the browser. It is helpful to research it for your specific browser and become comfortable with how to locate an element's class and tag.

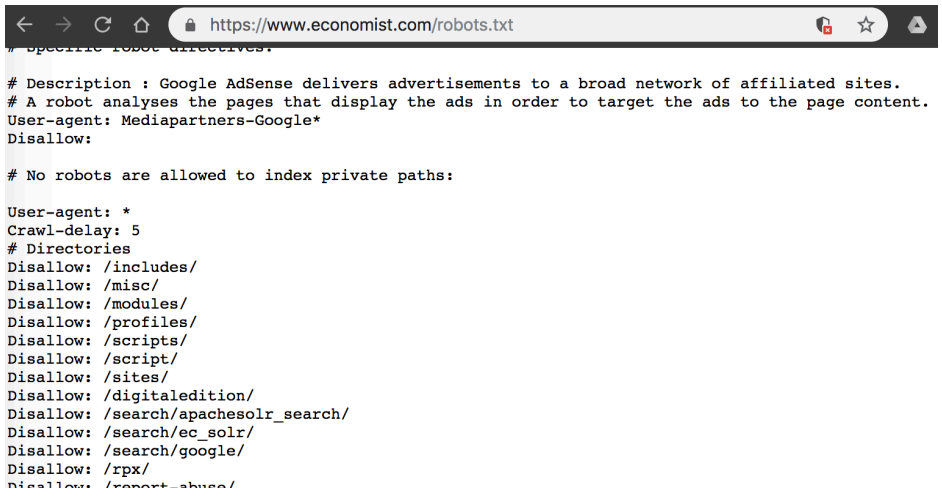
Since we can specifically target the span elements with their class, we can strip the text content inside with `.text`. For testing purposes, it is helpful to print the content and confirm it is in an acceptable format. Then, to store the headlines, the text is appended to the `textContent` array for later manipulation (Code 7.5).

Finally, there is a `time.sleep` to ensure we are respecting the restrictions of the website being scraped (Code 7.5).

```
# Below is the input area

def economistSearch():
    page_link = 'https://www.economist.com/'# Page Url to point request where to crawl
    page_response = requests.get(page_link, timeout=20)
    page_content = BeautifulSoup(page_response.content, "html.parser")

    for link in page_content.find_all("span", class_="flytitle-and-title__title",
limit = 3):   # Finds all the spans with the class flytitle-and-title__title
        if link.text not in textContent:
            # print(link.text)  # Prints the title so we can verify correct operation
            textContent.append(link.text)    # Appends the headline to our main array
            tempHeadlineHolder.append(link.text)

    print("Economist Done")
    time.sleep(5) # Creates a crawl delay of 5 seconds (which the Economist requires
in their robots.txt file)
```

(Code 7.5)

When scraping websites, it is important to visit the robots.txt file of each website you plan on using. This file can be found by adding a slash and "robots.txt" at the end of a website URL: Like *www.example.com/robots.txt*. Many websites specify specific scraping parameters for each of their URL paths. It is important to respect their wishes regarding bots (Figure 7.6). Part of being a good web citizen means following these guidelines, and in some cases, a violation of scraping policy can result in a ban of use.



(Figure 7.6)

For the Economist, the website restricts scraping of private URL paths. Confirm with each of your websites that none of the paths begin with anything that is disallowed. In addition, websites may specify a crawl delay. The Economist has a crawl delay of 5 seconds that we honor by adding the `time.sleep()` of 5 seconds after the get request.

Finally, since we want the program to monitor the headlines being displayed on websites continuously, we create an infinite loop that only terminates on a keyboard interrupt (Code 7.7). The loop calls the main search method, increments our `cycleCount` variable, then prints a line to the terminal signaling it has completed another loop. Lastly, `textContent` is reset to an empty array when we are done using it. The empty array reset is in place since we will be populating the array with the most up to date information every method call.

```
while True:

    economistSearch()    # Calls our main scraping method
    cycleCount += 1   # Tracks the method calls
    print("Search Done, Cycle: + " + str(cycleCount))

    textContent = [] # Reset the headline holder after we have searched the content to
avoid repeats
```

(Code 7.7)

This simple script is a solid foundation for setting up more advanced scrapers. In the next section, we will update the input source to track a more advanced webpage and deliver content to a dynamic array to eliminate the possibility of repetitive events.

ADVANCING THE ALGORITHM

Now, to test more advanced scraping, we are going to put the Economist scraping method aside and modify the method to crawl another financial news website, Seeking Alpha. Another array is going to be used to eliminate redundancies we might have by placing orders based on the textContent array. In our previous array, headlines were re-added even if they had already been seen. Now, a new array called tempHeadlineHolder will only contain unique headlines. To begin, initialize this temporary array inside the global variable section (Code 7.8).

```
# Global variables for the scraper

textContent = []     # Holds the headlines in an array
cycleCount = 0       # Stores the frequency of requests made to the server
tempHeadlineHolder = []    # Array to hold the most recent headline before it has been
analyzed
```

(Code 7.8)

Redesigning the scraping method to work on Seeking Alpha is very similar in overall structure, but differs slightly in how we parse specific headlines from the page content. On this page, each headline does not have a unique identifier class (Code 7.9). This means we must be more creative with the pointers.

Note: Knowing the Beautiful Soup documentation and syntax structure is very helpful at this juncture.

In our case, we target the parent div of each of the headlines, then look at the immediate child div, grab the link below that, and gather the text content of the link. This is all done with a descendant selector in the dot syntax (Code 7.9).

```
def AlphaSearch():
    page_link = 'https://seekingalpha.com/market-news/all'
    page_response = requests.get(page_link, timeout=20)
    page_content = BeautifulSoup(page_response.content, "html.parser")

    for link in page_content.find_all("div", class_="media-body", limit = 3):
        if link.div.a.text not in textContent:
            textContent.append(link.div.a.text)
            tempHeadlineHolder.append(link.div.a.text)
    print("Seeking Alpha Done")
```

(Code 7.9)

Now, as you may notice, we are applying the headline information in a slightly more advanced way than our Economist method before (Code 7.9). Inside each iteration of the 'for' loop, we are checking if the headline has already been added to the `textContent` array. The `textContent` array now holds a history of all the headlines we have seen since the program began. So, if the headline is in `textContent`, meaning we have seen it before, then we do not want to take a repeat action on it. If the headline has not been seen before (it is not in the `textContent` array), then we take two actions inside the conditional statement:

1) Append the headline to the `textContent` array, thereby adding it to our history.

2) Append the headline to `tempHeadlineHolder`. This array holds the unique headlines of the most recent scrape cycle and makes it simple to ensure analysis only happens once.

To finish implementing the scraper, we call the scraping method, just like with our Economist method (Code 7.10). However this time, we also set up the process for analyzing each unique headline. This process is accomplished by iterating over all the unique headlines in tempHeadlineHolder. At this stage, the 'for' loop should just print out the headlines to confirm there are no repeats.

```
while True: # Keeps the program constantly monitoring content

    AlphaSearch()
    cycleCount += 1  # Tracks the method calls
    print("Search Done, Cycle: + " + str(cycleCount))
```

```
# A loop that cycles through our temporary headline holder
for headline in tempHeadlineHolder:
    print(headline)

tempHeadlineHolder = [] # Reset the headline holder to avoid repeats
```

(Code 7.10)

Once the iteration through headlines in tempHeadlineHolder has completed, it is time to destroy their status in the temporary array. This eliminates the possibility of duplicate actions on the same headline. A duplicate action could be costly if multiple money orders are placed through Interactive Brokers when only when is desired. The repeat orders could quickly get out of control as more sources and execution speed are added to the program in the following sections.

MULTIPLE SOURCES AND OPTIMIZATION

Many algorithms may rely on multiple input sources to feed their analysis. We have covered a simple page parse with the Economist and a more complicated Seeking Alpha parse in the previous sections. Now, it's time to combine the two along with other sources to process a broader set of input data.

The basic infrastructure of processing multiple pages remains similar to the previous two sections. The only change here is the technique of adding multiple sources (Code 7.11).

Note: This is an excellent opportunity to play around with media sites that are of interest to you. Each site publishes content geared towards a different crowd and may fit a particular strategy or algorithm better. Also, continually monitoring the accuracy of your program as you build is incredibly helpful for debugging. It is significantly easier to fix a short list of bugs along that way than it is to catch a long set of issues at the final runtime.

Constantly monitoring the accuracy of your program as you build is incredibly helpful for debugging. It is significantly easier to fix a short list bugs along that way than it is to catch a long list at final runtime and trace back through the program.

Multiple sources have been programmed into the example code without explanation (Code 7.11 & Code 7.12). The excess explanation for each method was removed to spare redundancy. It's worth noting that all methods are created to run without custom parameters. This structure makes it easy to trigger each method individually inside the main 'while' loop. No other changes are needed at this point for the arrays to function correctly.

```
while True: # Keeps the program constantly monitoring content

    economistSearch()    # Calls our main scraping method
```

```
    CNNSearch()
    ReutersSearch()
    AlphaSearch()
    cycleCount += 1  # Tracks the method calls
    print("Search Done, Cycle: + " + str(cycleCount))

    # A loop that cycles through our temporary headline holder
    for headline in tempHeadlineHolder:
        print(headline)

    tempHeadlineHolder = []
```

(Code 7.11)

```
# Below is the input area

def economistSearch():
    page_link = 'https://www.economist.com/'
    page_response = requests.get(page_link, timeout=20)
    page_content = BeautifulSoup(page_response.content, "html.parser")

    for link in page_content.find_all("span", class_="flytitle-and-title__title", limit = 3):
        if link.text not in textContent:
            textContent.append(link.text)
            tempHeadlineHolder.append(link.text)
    print("Economist Done")
    time.sleep(5)

def CNNSearch():
    page_link = 'https://www.cnn.com/specials/last-50-stories'
    page_response = requests.get(page_link, timeout=20)
    page_content = BeautifulSoup(page_response.content, "html.parser")

    for link in page_content.find_all("span", class_="cd__headline-text", limit = 3):
        if link.text not in textContent:
            textContent.append(link.text)
            tempHeadlineHolder.append(link.text)
    print("CNN Done")

def ReutersSearch():
    page_link = 'https://www.reuters.com/'
    page_response = requests.get(page_link, timeout=20)
    page_content = BeautifulSoup(page_response.content, "html.parser")

    for link in page_content.find_all("h3", class_="article-heading", limit = 3):
        if link.text not in textContent:
            textContent.append(link.text)
            tempHeadlineHolder.append(link.text)
    print("Reuters Done")

def AlphaSearch():
    page_link = 'https://seekingalpha.com/market-news/all'
    page_response = requests.get(page_link, timeout=20)
    page_content = BeautifulSoup(page_response.content, "html.parser")

    for link in page_content.find_all("div", class_="media-body", limit = 3):
        if link.div.a.text not in textContent:
            textContent.append(link.div.a.text)
            tempHeadlineHolder.append(link.div.a.text)
    print("Seeking Alpha Done")
```

(Code 7.12)

Now that multiple sources are involved, the process of scraping all the websites may take longer. To adjust for this, there is a simple optimization to be made at the `find_all` step of each scraping method. When `find_all` is called without limitation, Beautiful Soup scans the entire document for matches. If you limit this search to a more manageable number, then Beautiful Soup can process the document much quicker. Since we are only concerned with the most recent unique headlines, we can limit this search to three headlines per page by passing `limit=3` as a parameter in `find_all` (Code 7.12).

This seemingly small optimization can make a big difference in runtime on pages with long or repetitive content. At this point, we have created a script that scans media sources and collects the most recent headlines for use. With this information at our disposal, it's time to begin analyzing it to set our algorithm up for action.

ANALYZING THE INPUT

With all the headlines at our disposal, it is time to run a simple analysis on the content and deliver a conviction as to whether we should trade a particular stock. This section also shows how to store more accurate information on cycles and implement a simple tactic to trick some traditional scraper blocking methods.

Begin by importing two more packages, randint and datetime (Code 7.13). These are used later to document the exact time a cycle has completed and add a randomized wait time after each completion.

```
import datetime
import time
from bs4 import BeautifulSoup
import requests
from random import randint
```

(Code 7.13)

To analyze the headline content, we will create a method to iterate over the words in the headline then use the iterator to compare specific trigger words. This section of the script is where you have the most flexibility to implement your specific strategy. This algorithm will implement a simple comparison and match technique to test for the appearance of a specific headline; however, this should merely be a starting point to build bigger and better strategies.

`HeadlineAnalysis` is a new method that receives a headline as input and breaks that sentence string down to individual words (Code 7.14). The words in the sentence are appended to a new array stored in the variable `words`. Within the `words` array, the analysis looks for matches and increments a point for each match found. To keep track of the match frequency, we create a new variable called `matchScore` and initialize it to zero.

The conditional "if" statement ignores case sensitivity by transforming each word to lower case and evaluating it against our statically defined trigger words. This program is looking to trade Apple the moment a headline states that "Apple exceeds investor's expectations".

A sentence containing three or more trigger words will print "Trigger Order Execution". When we work this into our main ibProgram1.py script in the next chapter, we replace the printout with a direct call to the orderExecution.

```
# Below is the logic processing area

def headlineAnalysis(headline):

    # Splits the headline so we can look for individual word matches
    words = headline.split()

    # A simple count to track if the headline contains our keywords
    matchScore = 0

    # Iterates over the words in the headline and looks for word matches
    for individualWord in words:
        if individualWord.lower() == "apple":
            matchScore += 1
        if individualWord.lower() == "exceeds" or individualWord.lower() == "beats":
            matchScore += 1
        if individualWord.lower() == "expectations":
            matchScore += 1

    if matchScore == 3:
        print("-- Trigger Order Execution --") # Eventually call orderExecution
```

(Code 7.14)

Now that the text analyzation function is ready, it's time for implementation inside the main "while" loop.

Note: A couple more modifications have been made since the last time we visited the main loop. Since we have imported datetime, the cycle print out now includes a report on the current time the cycle has finished (Code 7.15). This can be especially helpful when reflecting on a day worth of logs.

Inside the while loop, a test case has been added to trigger the analyzation function (Code 7.15). This can be especially helpful for testing edge cases and theories you may have. A conditional statement waits for cycle 12 then adds the test headline to trigger the `OrderExecution` function (At this moment there is just a printout of “- - Trigger Order Execution- -”).

Most importantly, the analyzation function is called for each headline in our iteration through tempHeadlineHolder. Instead of just printing the unique headlines to the console, the program will check for the word matches specified above (Code 7.15).

```
while True: # Keeps the program constantly monitoring content

    economistSearch()    # Calls our main scraping method
    CNNSearch()
    ReutersSearch()
    AlphaSearch()
    cycleCount += 1  # Tracks the method calls
    print("Search Done, Cycle: + " + str(cycleCount) + " at: " +
str(datetime.datetime.now()))

    # You can append text here to test the algorithm's response to certain cases
    if cycleCount == 12:
        tempHeadlineHolder.append("Apple exceeds expectations in the latest quarter")

    # A loop that cycles through our temporary headline holder
    for headline in tempHeadlineHolder:
        print(headline)
        headlineAnalysis(headline)

    tempHeadlineHolder = []
```

(Code 7.15)

Finally, an additional optional wait time is added at the end of the loop to trick some simple websites that may be monitoring for web scraping bots. The random integer simulates irregular page request behavior that may be more characteristic of a human user; however, the sources we've used do not require this directly so it can be removed to optimize performance further (Code 7.17).

```
# A loop that cycles through our temporary headline holder
    for headline in tempHeadlineHolder:
        print(headline)
        headlineAnalysis(headline)

    tempHeadlineHolder = []

    # Optional wait time -may be necessary for websites with a crawl delay
     time.sleep(randint(0,5))
```

(Code 7.16)

Scraper.py is now a fully functioning web scraper using Beautiful Soup! The scraper runs independently and currently only flags trades that should be made. It is time to bring the code into our main program so we can complete the input, analyzation, output chain. In the next section, we are almost set up to produce a dynamic and autonomous algorithm.

WORKING THE SCRAPER INTO THE MAIN PROGRAM

This section will focus on the careful relocation and integration of code we have written in all the previous sections. This section sets us up nicely to build the final dynamic and autonomous version we will complete in chapter eight. The merge will happen in five major steps:

1) Carry over the necessary import statements. Which includes the BeautifulSoup, requests, and randint packages.

```
# Below are the import statements

from ibapi.wrapper import *
from ibapi.client import *
from ibapi.contract import *
from ibapi.order import *
from threading import Thread
import queue
import datetime
import time
from bs4 import BeautifulSoup
import requests
from random import randint
```

2) Bring over the global variables for the scraper script. These include the two arrays and `cycleCount` variables.

```
# Below are the global variables

availableFunds = 0
buyingPower = 0
positionsDict = {}
stockPrice = 1000000

# Global variables for the scraper

textContent = []     # Holds the headlines in an array
cycleCount = 0       # Stores the frequency of requests made to the server
tempHeadlineHolder = [] # Array to hold the most recent headline
```

3) Copy the web scraping functions into the input area.

```
# Below is the input area

def economistSearch():
    page_link = 'https://www.economist.com/'     # Page Url to point request where to crawl
    page_response = requests.get(page_link, timeout=20) # Get request to ask for page content
    page_content = BeautifulSoup(page_response.content, "html.parser")

    for link in page_content.find_all("span", class_="flytitle-and-title__title", limit = 3):
        if link.text not in textContent:
            # print(link.text)  # Prints the title so we can verify correct operation
            textContent.append(link.text)    # Appends the headline to our main array
            tempHeadlineHolder.append(link.text)
    print("Economist Done")
    time.sleep(3)

def CNNSearch():
    page_link = 'https://www.cnn.com/specials/last-50-stories'
    page_response = requests.get(page_link, timeout=20) # Get request to ask for page content
    page_content = BeautifulSoup(page_response.content, "html.parser")
    for link in page_content.find_all("span", class_="cd__headline-text", limit = 3):
        if link.text not in textContent:
            # print(link.text)  # Prints the title so we can verify correct operation
            textContent.append(link.text)    # Appends the headline to our main array
            tempHeadlineHolder.append(link.text)
    print("CNN Done")

def ReutersSearch():
    page_link = 'https://www.reuters.com/'  # Page Url to point request where to crawl
```

```
    page_response = requests.get(page_link, timeout=20) # Get request to ask for page content
    page_content = BeautifulSoup(page_response.content, "html.parser")
    for link in page_content.find_all("h3", class_="article-heading", limit = 3):
        if link.text not in textContent:
            # print(link.text)  # Prints the title so we can verify correct operation
            textContent.append(link.text)    # Appends the headline to our main array
            tempHeadlineHolder.append(link.text)
    print("Reuters Done")

def AlphaSearch():
    page_link = 'https://seekingalpha.com/market-news/all'  # Page Url to point request where to crawl
    page_response = requests.get(page_link, timeout=20)     # Get request to ask for page content
    page_content = BeautifulSoup(page_response.content, "html.parser")
    for link in page_content.find_all("div", class_="media-body", limit = 3):
        if link.div.a.text not in textContent:
            textContent.append(link.div.a.text)
            tempHeadlineHolder.append(link.div.a.text)
    print("Seeking Alpha Done”)
```

4) Transfer headline analysis into the logic processing area.

```
#Below is the logic processing area

def headlineAnalysis(headline):
    # Splits the headline so we can look for individual word matches
    words = headline.split()

    # A simple count to track if the headline contains our keywords
    matchScore = 0

    # Iterates over the words in the headline and looks for word matches
    for individualWord in words:
        if individualWord.lower() == "apple":
            matchScore += 1
        if individualWord.lower() == "exceeds" or individualWord.lower() == "beats":
            matchScore += 1
        if individualWord.lower() == "expectations":
            matchScore += 1

    if matchScore == 3:
        print("-- Trigger Order Execution --")
        orderExecution("AAPL")    # Completes an actual execution
```

5) Start the while loop at the end of the program. This starts after Interactive Brokers has gathered the preliminary information.

```
# Calls the order execution function at the end of the program

app.account_update()    # Call this whenever you need to start accounting data
app.position_update()   # Call for current position
time.sleep(3)   # Wait three seconds to gather initial information
orderExecution()

while True: # Keeps the program constantly monitoring content

    economistSearch()   # Calls our main scraping method
    CNNSearch()
    ReutersSearch()
    AlphaSearch()
    cycleCount += 1  # Tracks the method calls
    print("Search Done, Cycle: + " + str(cycleCount) + " at: " + str(datetime.datetime.now()))

    # You can append text here to test the algorithm's response to certain cases
    if cycleCount == 12:
        tempHeadlineHolder.append("Apple exceeds expectations in the latest quarter")
```

```
# A loop that cycles through our temporary headline holder
for headline in tempHeadlineHolder:
    print(headline)
    headlineAnalysis(headline)

tempHeadlineHolder = []

# (Optional wait time that may be necessary for websites with a crawl delay or bot monitors)
# time.sleep(randint(0,5))
```

In the final section of this chapter, new code and features were not added. This process was a walkthrough of the snippets to merge between scraper and ibProgram1. As programs grow, it may make sense to keep these two files separate; however, for simplicity's sake, we can maintain the code in one script.

At the current state, the Interactive Brokers functions run first then the web scraper begins (without communicating with Interactive Brokers). The connection to the TWS remains open the whole time, with no IB events triggered. The next chapter covers how to combine all the functionality and complete our front to back algorithm.

Chapter 8: Producing a Dynamic Model

OVERVIEW

At this point, we have developed a method for receiving a constant flow of information through a web scraper pointed at four major news sources. We've created a simple way to control duplicate headlines and analyze the information stream for word matches. And, most importantly, we have used the power of Interactive Brokers to execute trades and handle the decisions delivered by the analyzation of incoming content.

All the pieces for a front-to-back financial algorithm are in place, so it is now time to bring everything together and start testing. The majority of edits in this section are focused on handling an added layer of complexity inside the custom methods created in earlier chapters.

ORDER EXECUTION

Last time we had visited the `OrderExecution` function, we created a straightforward way of buying shares of Apple stock. `OrderExecution` formed simple contract and order objects, requested the next ID, printed a couple of values, then executed a trade for ten shares. With plans to create a more dynamic algorithm, statically coded values for share quantity and ticker symbols are too restrictive.

Our goal now is to have `OrderExecution` receive a ticker symbol, calculate the amount to trade, then place the appropriate order. The first step is simply to pass an argument into the `orderExecution` function call within our scraper method.

```
# Below is the logic processing area

def headlineAnalysis(headline):

    # Splits the headline so we can look for individual word matches
    words = headline.split()

    # A simple count to track if the headline contains our keywords
    matchScore = 0

    # Iterates over the words in the headline and looks for word matches
    for individualWord in words:
        if individualWord.lower() == "apple":
            matchScore += 1
        if individualWord.lower() == "exceeds" or individualWord.lower() == "beats":
            matchScore += 1
        if individualWord.lower() == "expectations":
            matchScore += 1
```

```
        if matchScore == 3:
            print("-- Trigger Order Execution --")
            orderExecution("AAPL")     # Completes an actual execution
```

(Code 8.1)

Next, we modify `orderExecution` to accept a ticker symbol as a parameter. This parameter is called `symbolEntered`. The value of `SymbolEntered` is placed in the contract object, where we specify the contract parameters.

```
# This function executes our order
def orderExecution(symbolEntered):

    # Call client methods to gather most recent information
    contractObject = contractCreate(symbolEntered)
    orderObject = orderCreate()
    app.price_update(contractObject, app.nextOrderId())
    nextID = app.nextOrderId()
```

(Code 8.2)

`SymbolEntered` then needs to be specified as a parameter in the contract object since the ticker value gets passed as an argument inside `orderExecution`. The ticker symbol string is assigned to the symbol field of the contract object.

Note: If your algorithm requires a dynamic currency change, specific exchanges (remember: SMART means Interactive Brokers will choose the exchange with the quickest execution time), or a different security type, then those can also be added as parameters in the contractCreate function.

```
# Below are the custom classes and methods

def contractCreate(symbolEntered):
    # Fills out the contract object
    contract1 = Contract()  # Creates a contract object from the import
    contract1.symbol = symbolEntered   # Sets the ticker symbol
    contract1.secType = "STK"   # Defines the security type as stock
    contract1.currency = "USD"  # Currency is US dollars
    # In the API side, NASDAQ is always defined as ISLAND in the exchange field
    contract1.exchange = "SMART"
    # contract1.PrimaryExch = "NYSE"
    return contract1    # Returns the contract object
```

(Code 8.3)

Now, working our way down the orderExecution function, we can make some performance improvements to our price return wait time. In general, it's best to stay away from time.sleep methods when possible in order to optimize the runtime of your program. We still need to wait for a price return before we can make a decision on the quantity to buy; however, now we will set up a boolean flag to signal that there has been a successful return. This allows us to wait the minimum amount of time possible before we proceed.

Create a global variable called “stockPriceBool” and set that to a value of False.

```
# Below are the global variables

availableFunds = 0
buyingPower = 0
positionsDict = {}
stockPrice = 1000000
stockPriceBool = False
```

(Code 8.4)

This value will be set to True when the correct tick price has been returned and set back to False when the most recent price has been used. This boolean will allow us to loop on a conditional wait time that ensures we proceed only with a valid return value. On average, this run time will be much quicker than the 2 seconds that we had in place before. The loop only runs when the value of `stockPriceBool` is False, indicating the information from `app.price_update()` has not yet been returned.

```
# This function executes our order
def orderExecution(symbolEntered):

    # Call client methods to gather most recent information
    contractObject = contractCreate(symbolEntered)
    orderObject = orderCreate()
    app.price_update(contractObject, app.nextOrderId())
    nextID = app.nextOrderId()

    # Waits for the price_update request to finish
    global stockPriceBool
    while (stockPriceBool != True):
        time.sleep(.2)
```

(Code 8.5)

To ensure the loop doesn't run endlessly and to update the boolean value properly, we change the value to True when there has been an assignment to the "stockPrice" global variable. This assignment happens inside the `tickPrice` method inside the wrapper class.

```
# Market Price handling methods
def tickPrice(self, reqId: TickerId, tickType: TickType, price: float, attrib:
TickAttrib):
     super().tickPrice(reqId, tickType, price, attrib)
     global stockPrice# Declares stockPrice as a global variable
     global stockPriceBool # A boolean flag that signals if the price has been updated

     # Use tickType 4 (Last Price) if you are running during the market day
     if tickType == 4:
          print("\nParsed Tick Price: " + str(price))
          stockPrice = price
          stockPriceBool = True

     # Uses tickType 9 (Close Price) if after market hours
     elif tickType == 9:
          print("\nParsed Tick Price: " + str(price))
          stockPrice = price
```

(Code 8.6)

So far, the variable `stockPriceBool` becomes initialized as False then set to True when there is a value for `stockPrice`. The final step is to set the value back to False as soon as we have used `stockPrice`. Right after we place the order, the most recent stock price no longer becomes useful. Consequently, we should reset it.

```
# Place order
app.placeOrder(nextID, contractObject, orderObject)
print("order was placed")
stockPriceBool = False   #Reset the flag now that the price has been used in the order
```

(Code 8.7)

At this point, orderExecution can adapt to trade a wide range of ticker symbols and has a more efficient way of tracking price returns. The final step in the dynamic order fulfillment process is the ability to adapt to trade percentage or adaptive share values.

Create a new function called `quantityCalc`. The purpose of this function is to read the global values of `buyingPower` and `stockPrice` to calculate the number of shares we can purchase with the money in the account. The shares we can buy are computed by dividing the total buying power by the price per share (`buyingPower/stockPrice`). This will return the number of shares we can buy with 100 percent of the available buying power. To test, however, we assume that we only want to allocate 1 percent of the total buying power to the purchase of new shares. So, the total quantity can be multiplied by .01 and rounded down or up to the nearest whole number.

```
def quantityCalc():

    # View the buying power and price of stock
    print("Buying power: " + str(buyingPower))
    print("Stock Price: " + str(stockPrice))

    # Divide the buying power by price of the share
    possibleShares = float(buyingPower) / stockPrice

    # Weight the value to be 1 percent of our buying power (this is an easy value to
use for testing)
    sharesToBuy = math.floor(possibleShares * 0.01)
    return sharesToBuy  # return the shares to buy so we can use it in orderExecution
```

(Code 8.8)

The return value, `sharesToBuy`, is fed back into `orderExecution` and used to alter the quantity value in the order object. When the object was initialized above, we used a default value of 10. This value will is now replaced with the return from `sharesToBuy`.

```
# Calculates the quantity of the shares we want to trade
# and Update the quantity of the order to be a portion of the portfolio
quantityOfTrade = quantityCalc()
orderObject.totalQuantity = quantityOfTrade

# Place order
```

```
app.placeOrder(nextID, contractObject, orderObject)
print("order was placed”)
```

(Code 8.9)

OrderExecution is now taking the shape of a more adaptive algorithm and can be called to trade any valid ticker symbol. At this point, orderExecution should look very similar to the overview below:

```
# This function executes our order
def orderExecution(symbolEntered):

    # Call client methods to gather most recent information
    contractObject = contractCreate(symbolEntered)
    orderObject = orderCreate()
    app.price_update(contractObject, app.nextOrderId())
    nextID = app.nextOrderId()

    # Waits for the price_update request to finish
    global stockPriceBool
    while (stockPriceBool != True):
        time.sleep(.2)
    # time.sleep(2)

    # Print statement to confirm correct values
    print("The next valid id is - " + str(nextID))

    # Calculates the quantity of the shares we want to trade
    # and Update the quantity of the order to be a portion of the portfolio
    quantityOfTrade = quantityCalc()
    orderObject.totalQuantity = quantityOfTrade

    # Place order
    app.placeOrder(nextID, contractObject, orderObject)
    print("order was placed")

    stockPriceBool = False    #Reset the flag now that the price has been used in the
order
```

(Code 8.10)

With orderExecution functioning as intended, it's time to handle our orders past their first trade. A good algorithm will not only be able to trade securities when necessary, but it will also know how to sell them.

HANDLING ORDERS AFTER EXECUTION

Your particular choice in strategy can lend to several different approaches for handling orders after they have been placed. Some stock exit strategies rely on specific price points, others on a shift in the market, and some after a period of time. The best way to encapsulate several possible strategies is to create a function that normalizes the order.

In this book, normalizing an order refers to covering the positions and bringing the total shares in our portfolio to 0. For example, if a security has been purchased, we normalize by selling the same quantity of shares bought. If a security has been shorted, normalizing refers to covering the short and repurchasing the shares.

Our normalization function is called at the end of orderExecution. After we purchase the shares, our program will stop looking for new securities to purchase and will wait a set period to sell the shares that were just purchased. This simple trade logic assumes a stock will appreciate immediately after our purchase.

Note: This strategy for holding shares operates on a fundamental assumption that our trigger was correct. Naturally, it would be beneficial to introduce some added complexity of your own.

```
# This function executes our order
def orderExecution(symbolEntered):

    # Call client methods to gather most recent information
    contractObject = contractCreate(symbolEntered)
    orderObject = orderCreate()
    app.price_update(contractObject, app.nextOrderId())
    nextID = app.nextOrderId()

    # Waits for the price_update request to finish
    global stockPriceBool
    while (stockPriceBool != True):
        time.sleep(.2)

    # Print statement to confirm correct values
    print("The next valid id is - " + str(nextID))

    # Calculates the quantity of the shares we want to trade
    # and Update the quantity of the order to be a portion of the portfolio
    quantityOfTrade = quantityCalc()
    orderObject.totalQuantity = quantityOfTrade

    # Place order
    app.placeOrder(nextID, contractObject, orderObject)
    print("order was placed")

    stockPriceBool = False   #Reset the flag now that the price has been used in the
order

    # This sells the order after a set amount of time
    normalizeOrder()
```

(Code 8.11)

Now it's time to write the normalization function. The first section of this function will execute the strategy of waiting 20 minutes before selling the shares (this can be reduced for testing). Towards the end of the wait, we will request all the currently held positions from TWS. This request, along with the iteration over positions in the dictionary, is adapted from the previous code. Instead of the iteration and request occupying the main program body, it is now inside the `normalizeOrder` function.

`NormalizeOrder` follows a very similar procedure as we've seen before. However, in this case, there is a new `orderExecution` function called `orderExecutionNormalize`. The goal is to cover each position in `positionsDict`. So for each position that is found in the "for" loop, we pass the symbol and the inverse of the quantity in a new order.

```
# This function is designed to either sell a holding or cover a short
def normalizeOrder():

    print("Waiting to sell the order...")

    # A time to wait after we make the orderExecution
    time.sleep(1195)  # Gives the stock 20 minutes to change

    # Get the positions from the server
    app.position_update()
    time.sleep(5)

    # Iterates over everything in the positions list and reverses the quantity
    for key, value in positionsDict.items():
        parsedSymbol = key
        parsedQuantity = value['positions']
        parsedCost = value['avgCost']
        print(str(parsedSymbol) + " " + str(parsedQuantity) + " " + str(parsedCost))

     # This reverses the quantity in the positions list to set the final quantity to 0
        orderExecutionNormalize(parsedSymbol, -1 * parsedQuantity)

    print(positionsDict)
    print("All positions have been sold”)
```

(Code 8.12)

`OrderExecutionNormalize` will function very similar to `orderExecution`. The only difference is the simplification of the original `orderExecution` and the passage of a `quantityEntered` parameter into the `orderObject`. The contract object is created from the parsed symbol in `positionsDict` and order object trades the opposite of the quantity that we own, as drawn from `positionsDict`.

```
# This order is called to normalize our positions
def orderExecutionNormalize(symbolEntered, quantityEntered):

    # Remakes the contract and order object
    contractObject = contractCreate(symbolEntered)
    orderObject = orderCreate(quantityEntered)
    nextID = app.nextOrderId()

    # Place order
    app.placeOrder(nextID, contractObject, orderObject)
    print("Positions were normalized”)
```

(Code 8.13)

By passing `quantityEntered` into the `orderCreate` function, we are introducing a new argument that the method is not designed for. To account for this change, we are going to handle this adaptability by adding conditional statements into `orderCreate`.

```
def orderCreate(quantityEntered=10):  # Defaults the quantity to 10 but is overridden
later in orderExecution
    # Fills out the order object
    order1 = Order()     # Creates an order object from the import

    # If the quantity is positive then we want to buy at that quantity, and sell if it
is negative
    if quantityEntered > 0:
        order1.action = "BUY"    # Sets the order action to buy
        order1.totalQuantity = int(quantityEntered)    # Uses the quantity passed in
orderExecution
    else:
        order1.action = "SELL"
        order1.totalQuantity = abs(int(quantityEntered))    # Uses the quantity passed
in orderExecution

    order1.orderType = "MKT"     # Sets order type to market buy
    order1.transmit = True
    return order1    # Returns the order object
```

(Code 8.14)

First, the parameter `quantityEntered` needs to be added. However, when `orderCreate` is called in `orderExecution`, there is no argument passed. To account for this variation, we add the default value of ten in for quantity. If no argument is supplied, ten is assigned. If there is an argument, then the ten is ignored and `quantityEntered` is dependent on the argument.

Next, the conditional statement will handle the added complexity when the method is called inside the normalization procedure. In our method, a negative quantity specifies that we'd like to sell. Whereas a positive share quantity indicates a buy order; however, since Interactive Brokers requires the total quantity to be positive for a trade, we just conditionally set the action type based on the value of quantity.

The `orderType` and `transmit` values remain the same as before. Then this method is finished by returning the order object with the new quantity values.

The normalization procedure is now ready to trade the inverse of all the positions in your portfolio, effectively selling all outstanding shares. The simple strategy implemented here may ultimately differ slightly depending on your particular approach. The `normalizeOrder` function can easily be manipulated to sell only certain stocks at a defined quantity. Nevertheless, the process described here lends a lot of flexibility and potential for creative adaptations.

Chapter 9: AWS DynamoDB Connection

INTRODUCTION

In the previous parts of this guide, simple arrays and dictionaries were used to store program information. However, as the complexity of your algorithm grows, more advanced tracking and storage may be needed. Programmers may look to a cloud platform for three main reasons:

1) Persistence: Data stored in arrays and dictionaries is ephemeral. The program only holds a memory of the array and dictionaries during the program's runtime. When the execution is terminated, the system will lose the memory. Cloud computing allows you to permanently store the information which may be accessed in the case of an untimed interruption.

2) Size: Simple program arrays are not designed to store massive amounts of information. In the cases where blocks of historical data or granular information such as detailed tick information needs to be saved, array complexity can quickly outpace the ability of your machine.

3) Connections: Cloud databases are capable of handling multiple, simultaneous database connections. If you are operating in a team where users may be remote or working on several machines, a consistent storage solution may be necessary.

In this section, the guide will explore a connection to one of the largest cloud providers: Amazon Web Services (AWS). This chapter will walk through the setup, connection, storage, and retrieval of information from DynamoDB. DynamoDB is a non-relational (NoSQL) database that is highly scalable, flexible, and affordable. DynamoDB has quickly become a leader in the cloud database solutions environment. Their prevalence is for a couple of main reasons:

1) Fully Managed: DynamoDB is a fully managed Infrastructure as a Services solution (IaaS). This distinction means that Amazon will maintain the physical integrity of the systems, enable a stable baseline of security, and provide a relatively easy configuration environment. The system is also readily able to expand to accommodate increasing amounts of data without additional configuration.

2) Simple Integrations: Database systems often work closely with other services such as storage (S3), Logging and monitoring (CloudWatch), advanced permissions management (IAM), or computing (EC2). AWS makes it easy to integrate with other solutions within their online platform. The services can communicate throughout Amazon’s Virtual Private Cloud (VPC).

3) Ease of Use & Flexibility: Within the cloud computing environment, Amazon is known to have one of the easiest database setup processes. Other providers may have a simpler user interface, but AWS's combination of features and ease of use is well balanced.

The flexibility of DynamoDB allows users to store several types of information and expand attributes when necessary. In contrast, if you're looking for a rigid table structure, Amazon's Relational Database Service may suit your needs (RDS).

4) Affordable Pricing: True to Amazon's original mission when starting their marketplace, AWS services also strive to be the most competitive and affordable in the market. They include a generous free tier on many services or default configurations for a couple of dollars a month.

I encourage you to do independent research into the competitive advantages of AWS over other competing cloud computing resources. Additionally, if you are more familiar with SQL, you can look into RDS for more structured data. RDS can be queried slightly faster and maintain complex table relationships. However, DynamoDB is unbeatable for the flexibility you may need when initially testing information storage and strategies.

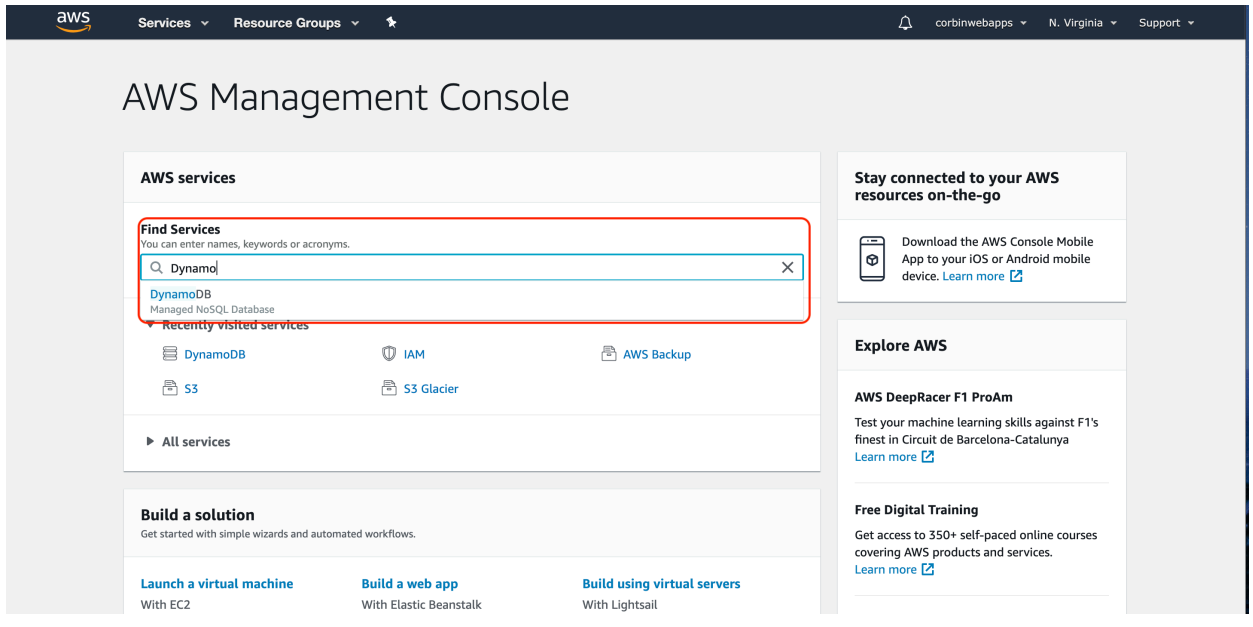
TABLE CREATION

Begin by creating an AWS account. You can do so at **https://aws.amazon.com/**. If you already have an Amazon account (yes, even one for just shopping), you can log in with that. The account setup should be relatively straightforward and will not be covered in detail in this guide. If you'd like to learn more or need assistance, refer to **https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/**.

Once you're set up with an account, it's time to create a DB table. AWS offers full table creation though Python code or from the CLI. However, we will create the table in the AWS console. The console is a great tool to become familiar with and can help monitor your items' status in a simple GUI. These next steps will walk you through the process of table creation with the default values.

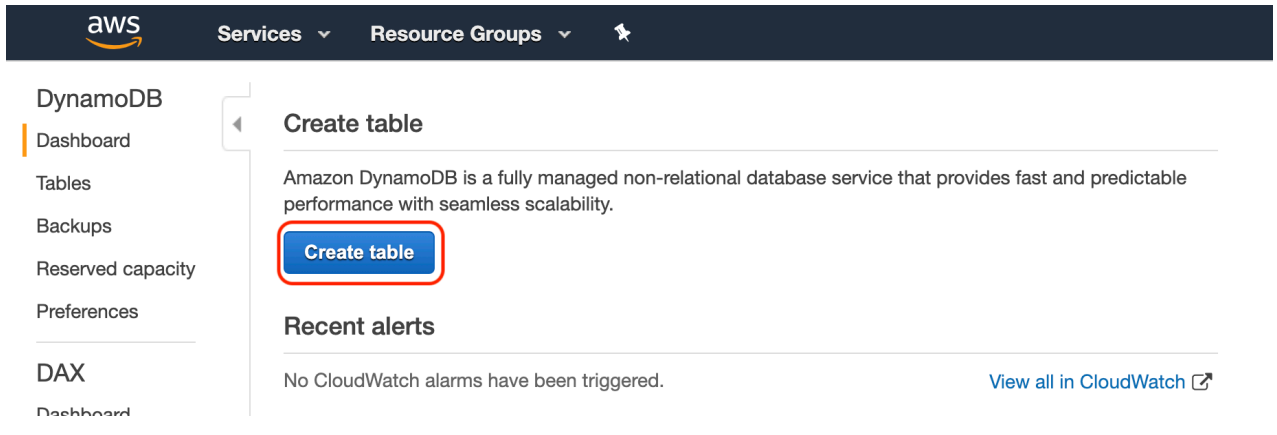
Note: The interface may change through time, but the general steps will likely remain similar. If you need an updated walkthrough, you can find AWS's version on their tutorials website.

1) Open the AWS console and search "DynamoDB" into the search bar (Figure 9.1).



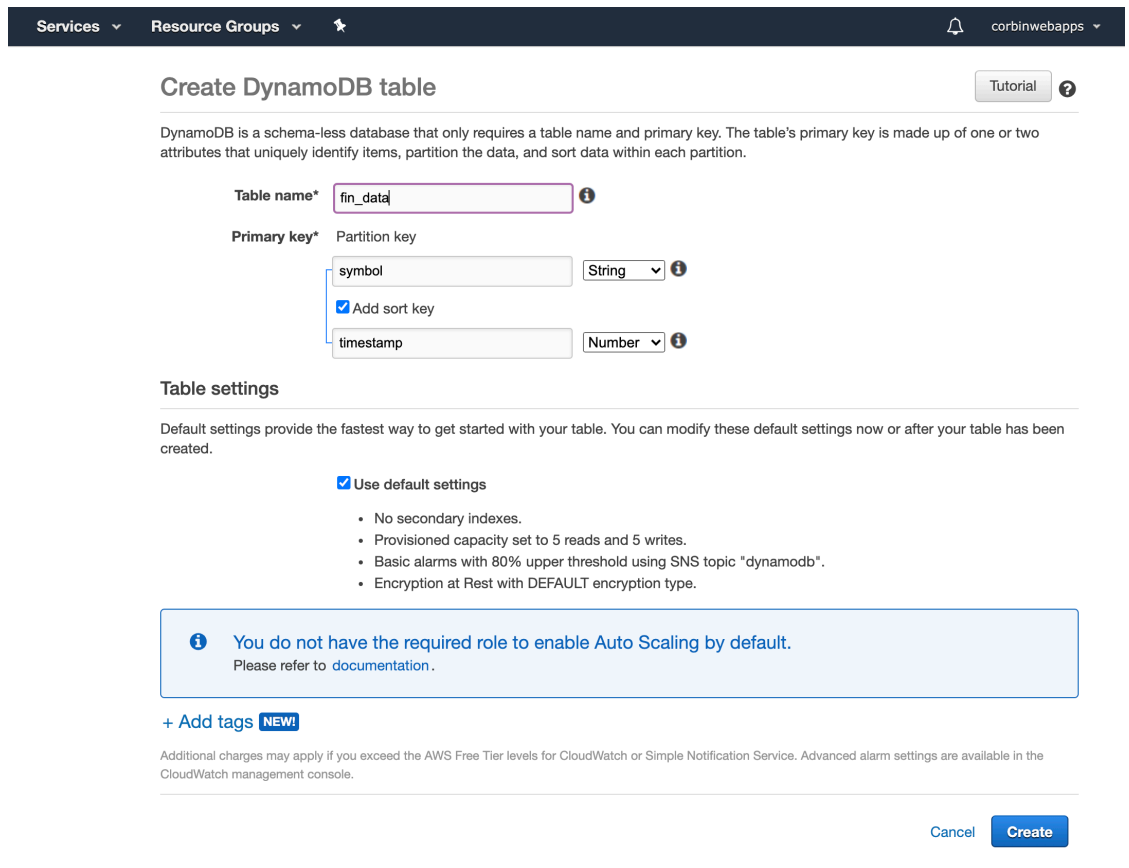
(Figure 9.1)

2) You will see a button to create a new table (Figure 9.2).



(Figure 9.2)

3) Now, configure the table accordingly. For this tutorial, we will call our table 'fin_data'. The partition key is 'symbol' with a sort key of 'timestamp' (Figure 9.3).



(Figure 9.3)

In DynamoDB, the primary key works as follows:

1) Partition Key: Used to determine the location of data storage within a hash.

2) Sort Key: Determines the order of the entries in that specific hash location.

The primary key is considered a composite key if both values are present. Items within a table may share a partition key (you can buy the same stock) but must have unique sort keys (you can not buy the same stock within the same second).

Note: If you anticipate a need to store multiple orders within the same second, you can adjust the granularity of timestamp by removing the parse to integer.

The table has been initialized with default settings, as there's no need for enhanced performance at this point. Nevertheless, I encourage you to uncheck the box and explore some of the options available to you. You can also view the impact that adjusting the read/write limits have on monthly costs. Finish the setup by selecting the “Create” button. The table will take a couple moments to initialize.

You can see the summary information on the ‘Overview’ tab. After writing an item, you can view the items and the database table by selecting the “Items” tab in the top bar. Here, you can also experiment by manually creating a new item.

The table is now ready for us to connect via code. In the next section, we will configure the python tools needed to facilitate that connection.

CONFIGURATION

In this section, we will be establishing the environment and primary code to communicate with AWS services. The topics from this section are based off the Boto3 library and similar to the quickstart tutorial (https://boto3.amazonaws.com/v1/documentation/api/latest/guide/quickstart.html). Boto3 is a library to facilitate the communication with AWS services.

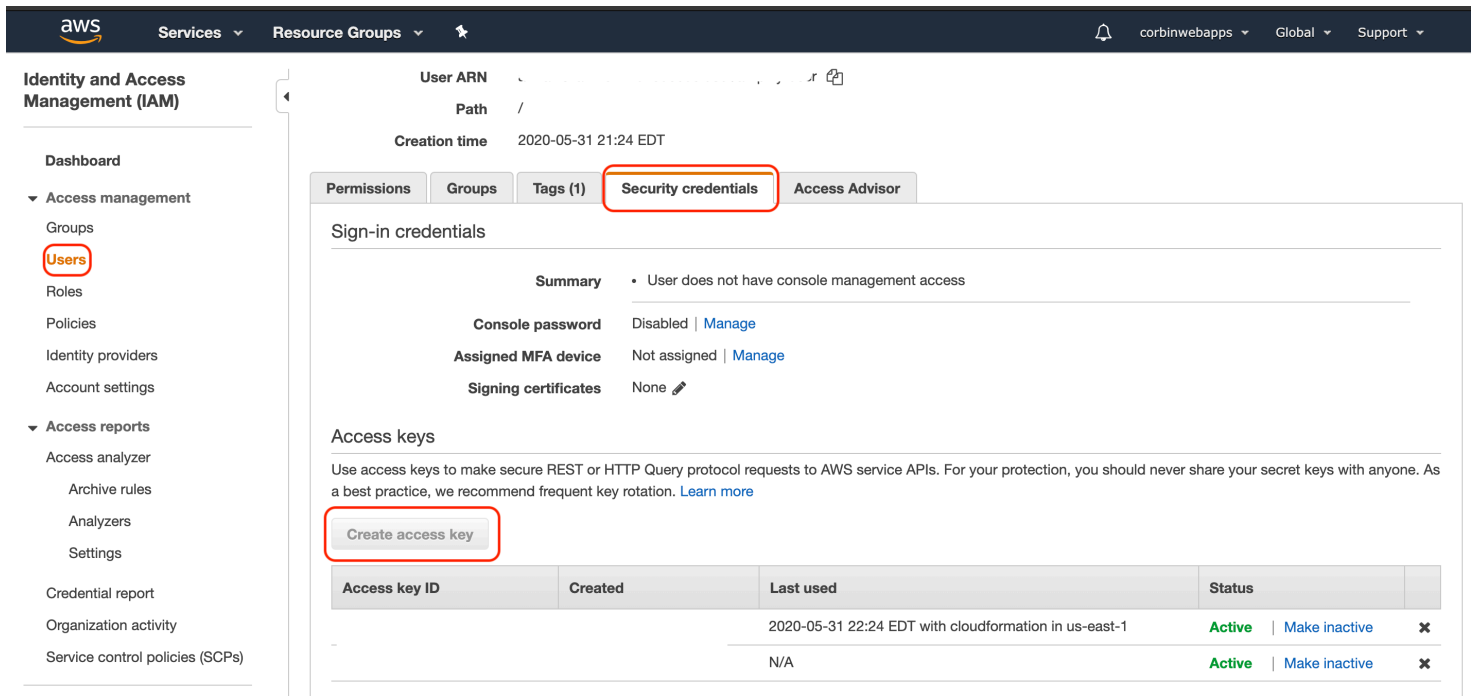
1) Install Boto3 to your computer

```
pip3 install boto3
```

2) Find your AWS credentials

Go to the IAM(Identity and Access Management) console in AWS by searching “IAM" in the products search. This is where you can find API credentials and set permissions for yourself and other users that have access to the AWS services.

Under the access management dropdown on the left panel, click on “Users”, then select the tab “Security Credentials”. Here there should be a button to “Create access key” (Figure 9.4). Here you will have the option to save the access and secret keys that we will use in the next step.



(Figure 9.4)

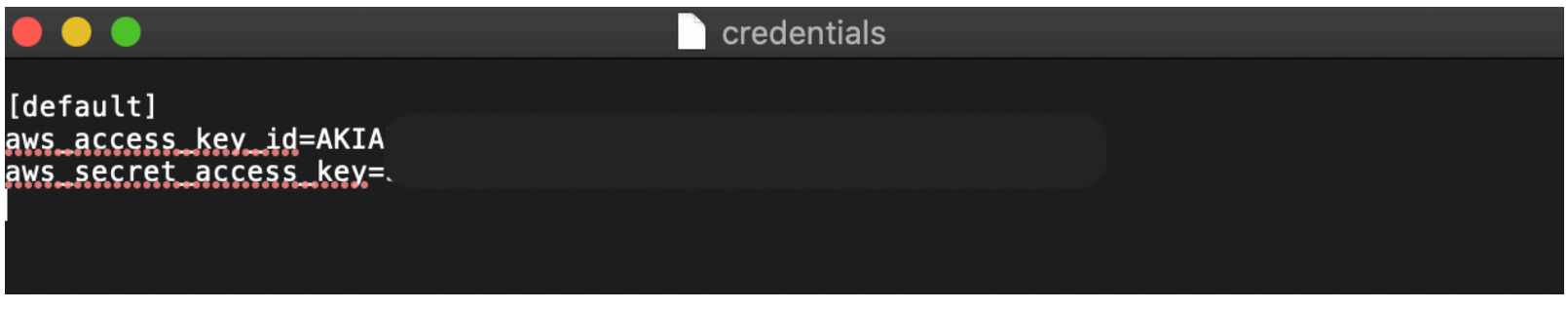
3) Configure the AWS credentials

Use the configuration terminal command:

```
aws configure
```

Or use the manual approach (helps if your aws command doesn’t work well, like mine):

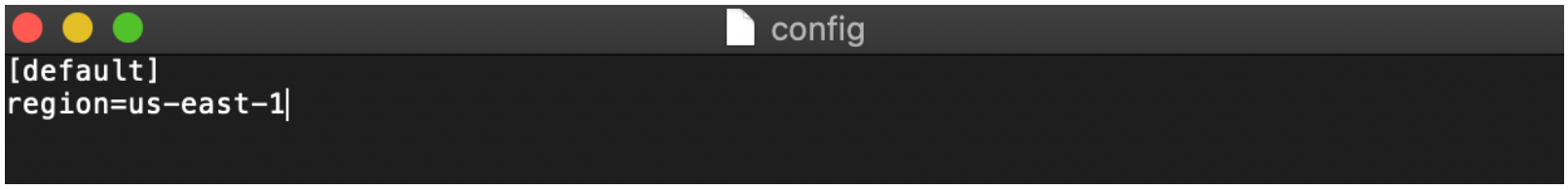
This way involves editing the config and credentials source files. Open the file at `~/.aws/credentials` (if you’re on a mac, go to finder -> Go -> Go to Folder and type that into the search). You can then manually specify your keys after the “=“ sign (Figure 9.5).



(Figure 9.5)

You may get an error if you don’t specify your preferred hosting zone. So, open `~/.aws/config` and include a zone similar to below (Figure 9.6).

Note: If you’d like to target certain users with specific credentials, their name can be specified with different values after the curly brackets (Ex. [userX]). However, In most cases, just specifying the default should be sufficient.



(Figure 9.6)

Boto3 is now finished with configuration, and you are ready to begin programming the read/write operations in Python.

CONNECTION CODE

The purpose of our code will be to write the stocks we've purchased and retrieve those to make sure we don't double trade a security within too short of a time frame. For example, if we have confirmed a trade for Apple, then "AAPL" will be stored in the DB with the quantity and time. So, if we're looking to make another trade, the program can scan the DB and deny the trade if a similar symbol has been traded within the last ten minutes.

Within the scope of this guide, the code will remain independent of ibProgram1.py. The purpose of this code is to provide modular functionality for more advanced users. The code for the AWS connection will live within an independent file called db_store.py. The system will also be simplistic, but highly flexible. As mentioned before, the non-relational nature of DynamoDB allows entries to be flexible to many data types.

Before we begin with the code, **let's review the macrostructure**:

1) Imports: The imports section is a straightforward definition of the external resources the program will use.

2) OrderExecution: This section is defined to emulate the interaction of a purchase order from ibProgram1.py. In this section, however, we are focusing on the AWS connection so no orders will be sent to Interactive Brokers.

3) Put_data: Put_data holds the code to write information to our DynamoDB instance. Here we will insert an item into the table individually by defining the symbol, timestamp, and quantity.

4) Query_data: Query is similar to a simple read operation but executes in scale. This means we can search through the whole database for particular criteria and return only those that match the definition. This can save read operations as it does not return the entire table. In this example. we will query by filter and timestamp.

5) Main Method: The main method calls the above functions and defines specific parameters for running. Inside ibProgram1.py, this will likely change the most and live inside of a new function, possibly called `db_connection()`.

Similar to before, let's start with importing the modules we need (Code 9.7).

```
# Imports
import boto3
import time
from pprint import pprint
from boto3.dynamodb.conditions import Key
```

(Code 9.7)

Notice we are importing:

1) **Boto3:** The library used to connect to AWS.
2) **Time:** Used to capture the current time and write it to the sort key (timestamp).
3) **pprint:** Called PrettyPrint, for printing data in a readable format.
4) **Boto keys:** We use these later to query the entries by key

Directly underneath the import statements, we can place the target connection (Code 9.8). In this case, we are specifying the resource as 'dynamodb'.

Note: Boto3 extends its capabilities to other AWS services. So, the resource can easily be replaced with another AWS service such as 'S3'.

```
# Imports
import boto3
import time
from pprint import pprint
from boto3.dynamodb.conditions import Key

# Specify the target
dynamodb = boto3.resource('dynamodb')
```

(Code 9.8)

Boto3 handles a lot behind the scene, so that simple line facilitates the connection we need to DynamoDB. Next, we're ready to send data to our connected instance.

We create a new function called `put_data` with symbol and quantity parameters. First, within `put_data`, the fin_data table we created in a previous section is targeted and stored in `table`. And, time is stored inside `current_time`.

At this point, we would have symbol, quantity, and time ready to write to the database. So, it's easy to create the response object accordingly. The put item is defined in the structure below, with table

attributes on the left corresponding to program variables on the right side of the Colon. Finally, on the last line, the response is returned.

```
# Function to place a recently purchased stock into the database
def put_data(symbol, quantity):

    # Define the financial data table we will write to
    table = dynamodb.Table('fin_data')

    # We will later sort by time, so prepare in variable
    current_time = int(time.time())

    # Define the item values. These are drawn from the parameters and time variable
    response = table.put_item(
       Item={
            'symbol': symbol,
            'timestamp': current_time,
            'quantity': quantity,
        }
    )

    # Return response for printing
    return response
```

(Code 9.9)

Now, the put_data function is ready for use. In the theme of ibProgram1.py, we will call this function inside `order_execution` (Code 9.10). Since data_store.py is designed to be modular, the function call is relatively straightforward. However, commented in the code is what the function call would likely look like inside ibProgram1.py.

```
def orderExecution():
    # If placing in order Execution in ibProgram1.py, use the following line:
        # After this line: app.placeOrder(nextID, contractObject, orderObject)
        # Place this line: put_data(contractObject.stock, orderObject.totalQuantity)

    # Simple write. In ibProgram1.py, this would likely be removed by the above lines
    response = put_data("AAPL", 101)
    return response
```

(Code 9.10)

At the end of the program, it's time to call `orderExection` and print the response (Code 9.11). The response message should include a 200 HTTP code (indicating success) alongside other identifying elements.

Note: Don't worry if the output seems confusing. In the next section, the query will return a more readable format.

```
if __name__ == '__main__':

    # Let's write first
    response = orderExecution()
    pprint(response, sort_dicts=False)
```

(Code 9.11)

Your program is now capable of writing to the DynamoDB table. In the next section, we will cover how to query data conditionally.

QUERY THE DATABASE

This section will cover the steps to receive information from the table based on the ticker symbol and timestamp. We will create a new method, `query_data`, and use the main method to define the parameters for the time interval we want to query within.

First, define the function `query_data` below the `put_data` method. The ticker symbol and time interval will be passed as arguments when we call the function. Similar to before, the fin_data table is targeted, and time is stored inside a local variable.

However, in place of the put item, we now identify our response type as a query. The query returns items when the symbol in the table matches the symbol passed as an argument. Additionally, the timestamp must be between a defined range. The query will only return items that satisfy both conditions, the correct symbol and range (Code 9.12).

```
# Function to search Database. In this example, we are searching by symbol
def query_data(symbol, time_range):

    # Define target table and save time
    table = dynamodb.Table('fin_data')
    current_time = int(time.time())

    # Query by the key (which we defined when creating the table as symbol) and a
specific time range passed as an argument
    response = table.query(
        KeyConditionExpression=Key('symbol').eq(symbol) &
Key('timestamp').between(time_range[0], time_range[1])
    )

    # Return an array that we will iterate over later
    return response['Items']
```

(Code 9.12)

The last significant change will come inside the main method. You'll notice the introduction of a `time_limit` and `current_time` store. For this guide, we will use a range between the current time and 10 minutes prior. Since the past in Unix time is merely the current time minus the duration we'd like to travel into the past, we can easily calculate the far side of the range.

Note: time.time() returns Unix time. Unix time is the seconds that have passed since January 1st in 1970. So, we defined time_limit in seconds and must convert that to seconds by multiplying by 60.

```
if __name__ == '__main__':

    # Let's write first
    response = orderExecution()

    # Let's define a limit in minutes and store unix time
    time_limit = 10
    current_time = int(time.time())

    # End time will then be the current time minus the limit in seconds
    # Which would be the quantity of *time_limit seconds* into the past
    end_time = current_time - (time_limit*60)
    time_range = (end_time, current_time)

    # Query by ticker passed as an argument
    symbols = query_data("AAPL", time_range)

    # Iterate over the response and print all information returned
    for symbol in symbols:
        print(symbol['symbol'], ":", symbol['quantity'])

    # Print the output
    pprint(symbols, sort_dicts=False)
```

(Code 9.13)

After time_range is defined, we can call `query_data` with the symbol and range. The returned information (an array) is stored in symbols so we can iterate and print over the items. This 'for' loop structure will likely be highly applicable for most strategies in ibProgram1.py.

All the pieces are now in place to run to the program effectively. Navigate to the code directory and run with:

```
python data_store.py
```

Congratulations on integrating a cloud database solution into your project. This is an excellent starting point to explore more capabilities of AWS. At this point, the project code is growing in complexity, so it can be beneficial to modularize your code and organize it in a way that makes sense to you. More on that in the following section.

Chapter 10: Testing, Errors, and Code Cleanup

FURTHERING ORGANIZATION

There are still several structural improvements that can be made to handle complexity as the code base grows. This guide was designed for the code to be easily comprehensible for small to medium size algorithms without introducing over-engineering of the code that may confuse those less similar with Python. The current program organization works well for our example, which spanned approximately 450 lines; however, those interested in expanding their programs further may want to take additional actions towards coding best practices.

Organizing code is vital for the effective communication between you and the other developers potentially reading your program. The example script is designed to communicate through detailed comments and descriptive variable names. Although this is an essential aspect of beautiful code, comments and variables may not always be enough.

When programs become too complex to navigate, a useful tool is the divide and conquer method. This method aims to break logical groups of code into distinct sections. In our program, the Interactive Broker API methods were referenced inside classes: TestApp, TestClient, TestWrapper. These main classes organize the more significant components of communication with the TWS. However, the custom methods introduced throughout this guide can be divided into classes and files of their own.

When in distinct classes, you have the ability to allocate each class into a unique file. For example, you could make a main.py program that imports and implements classes for the wrapper, client, app, scraper, and the order handling methods. The rest of the custom methods can go into a file called custom.py. The distributed organization is generally recommended for programs in excess of 1000 lines.

Another major organizational improvement can come from eliminating as many global variables as possible. In our program, the global variables gave us the flexibility to access the most recent value at any part of the program. By design, this enables users to become creative with how they incorporate the values throughout their program. Nevertheless, the benefit of global variables can also be their downfall. The ability to update values from any part of the program can lead to incorrect or false values.

Further, global variables can make lengthy program traces and unit testing much more challenging to manage. So, users with more advanced Python skills should choose to keep the variables as local as possible.

Proper organization is not something to ignore. As programs expand, it becomes increasingly helpful to navigate the code efficiently. This guide does not cover the specifics of Python code organization as the internet has an array of valuable resources for those interested in learning more. Nevertheless, this program should continue to evolve further with every new skill you gain.

TESTING

Throughout the design of our program, we have incorporated many touchpoints for feedback, primarily through the use of print statements. Print statements were added to communicate the start of certain functions, wait periods, variable value confirmation, and common errors returned from the TWS. The method of utilizing printouts for user feedback is a simple and effective way of keeping track of significant execution milestones. By uncommenting the printouts that remain, or creating your own, you can start to develop a better understanding of the code, which is especially helpful if the execution goes astray.

Components of this program can be removed or added so long as the major classes and connections to the TWS remain. The construction of the algorithm incorporated unit testing along the way. I suggest, time allowing, that your program be written and tested along with the guide, instead of solely running the final product, which has several potential points of failure.

As time progresses and Interactive Brokers updates their software, certain features may be deprecated or changed. It is in the nature of software to adapt and improve. By understanding the step-by-step construction and fundamentals, you are in a much better spot to adjust to the possibility of change. Python debuggers or more careful monitoring of TWS signals can help solve even the toughest situations.

If all else fails, the Interactive Brokers' support can help alleviate some possible issues. Support generally does not provide help for specific code; however, they are likely to assist with some connection, permissions, or possible configuration issues.

Finally, your depth of algorithm testing can make or break a strategy and indicate possible success. Although you can never understand the true performance of an algorithm until runtime, backtesting can be a helpful indication of future performance. Running the example, or any algorithm, on a large sum of money should never occur until you are confident in the algorithm's ability to perform.

HANDLING ERRORS

Errors are common when creating new programs and branching into uncharted territory. Interactive Brokers is an incredibly complicated system that can be prone to error when the software does not explicitly adhere to the specification. For complicated errors that stem from the API, it is helpful to check the log files and contact Interactive Brokers' support. However, many errors are common and more easily solvable. This section will cover some of the most common errors I had approached when writing the program.

In addition, much of the returned code can be deciphered with the official error handling guide in the API: https://interactivebrokers.github.io/tws-api/message_codes.html

BrokenPipeError: [Errno 32] Broken pipe:

This error occurs when you are not properly connected to the Trader Workstation. The first step is to make sure you have the Trader Workstation opened when you run your Python code. The Trader Workstation must remain active in the background for a connection to be established. You may also try quitting and restarting the program (an ages-old tactic that can sometimes do the trick).

If the TWS appears to be working, then make sure the settings are set up correctly to receive incoming connections. Refer to Chapter Two for the proper configurations. In some cases, the TWS may reset to a "read-only" connection mode.

The final common error is to ensure you are using the correct IP and port inside the TestApp connection (these can be compared against the TWS preferences). If switching between paper and live trading, the code also needs to be altered accordingly. It's worth noting that a VPN alters the state of your IP and should be disabled unless the program is configured to accept it.

AttributeError: 'TestApp' object has no attribute 'my_errors_queue':

This specific error is a common byproduct of another system failure. The program is attempting to put an error onto the queue after the program has failed. However, the put is unsuccessful since the program has crashed. If you look at Figure 10.1 below,

```
Corbin's-MacBook-Pro:code corbinbalzan2$ python ibProgram1.py
before start
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/ibapi-9.76.1-py3.8.egg/i
bapi/client.py", line 149, in connect
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/ibapi-9.76.1-py3.8.egg/i
bapi/connection.py", line 80, in sendMsg
BrokenPipeError: [Errno 32] Broken pipe

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "ibProgram1.py", line 328, in <module>
    app = TestApp("127.0.0.1", 7497, 0)
  File "ibProgram1.py", line 311, in __init__
    self.connect(ipaddress, portid, clientid)
  File "/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/ibapi-9.76.1-py3.8.egg/i
bapi/client.py", line 183, in connect
  File "ibProgram1.py", line 175, in error
    self.my_errors_queue.put(errormessage)
AttributeError: 'TestApp' object has no attribute 'my_errors_queue'
```

(Figure 10.1)

You can see the original issue stems from a failure in the connection to the Trader Workstation (BrokenPipeError). In this specific case, I did not have the TWS open and running on my computer. So, by running and logging into my IB account, the error is resolved and the program can run.

Cannot import module:

This can occur when there is a failure of Python to build the source distribution correctly. Your terminal may return an import error that says one of the files is not found. In this case, build the source distribution with the following commands:

1) Build a source distribution (this will need to be run inside IBJts/source/pythonclient):

```
python3 setup.py sdist
```

2) Build a wheel:

```
python3 setup.py bdist_wheel
```

3) Install the wheel:

```
python3 -m pip install --user --upgrade dist/ibapi-9.73.7-py3-none-any.whl
```

*Note you may need to swap the ibapi-9.73.7 with the correct version you are using (this can be found in the printout after building the wheel). These commands can be found inside the README in IBJts/source/pythonclient.

ImportError: No module named 'bs4'

This error often arises if Beautiful Soup is not downloaded correctly through Pip. The first step is to make sure you have BS4 downloaded according to the documentation: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

If downloaded already, you may also want to make sure you have downloaded it to the right path. Sometimes BS4 defaults to Python versions 2.7+ and not Python versions 3+ if Pip has installed to the wrong location.

If Pip is too difficult (it can be buggy sometimes), you can download Beautiful Soup as a tarball, without Pip installing (see bottom of "Installing Beautiful Soup" section in documentation). Although it is worthwhile to have Pip operating in proper form, this can be a temporary solution.

Identifying new Tags to Target in Beautiful Soup:

Since websites regularly update their HTML, you may have to adapt by selecting new target tags for the headlines. This can be rather straightforward if you understand the targets as identified in Chapter 7 and know how to read HTML.

In your web browser, open developer tools for the website that needs readjustment (or right-click on the element and select "inspect"). This pane is where you can identify common classes shared between the headlines you are looking for.

For example, if you see all headlines have a class called “main_headline”, then you’d want to change the class within the iterator accordingly.

Unicode Encode Error:

This happens when you are scraping a website and the website is using a font style that has symbols that can not be converted into a Python format. The error will say: “Can not encode character ____ in position ___. If you count over the number of positions in string, you should find the character giving you trouble.

This can be solved by converting using ("string").encode("utf-8") or using a website that contains more readable font formats. In some cases, the Economist gives this error. If this is true, then comment out the Economist code and try other reliable news sources.

ImportError: No module named 'ibapi':

This shows an error when ibapi is not correctly installed. If you have the API folders set up according to the instructions in chapter two and are still receiving this error, make sure you run the setup process:

1) Build a source distribution (this will need to be run inside IBJts/source/pythonclient):

```
python3 setup.py sdist
```

[Errno 13] Permission denied:

You may receive this error when you are attempting to install the required packages for this program. This is a common issue on Mac when users do not have root access to the proper folders for install. A simple fix is to prepend `sudo` before the install command. Example:

```
sudo pip3 install requests
```

However, sudo is not without risk. In most cases, the sudo command is fine for trusted packages that are known to not conflict with your operating system. But, it's important to consider the install location and packages as sudo may override other system functions.

Difficulty Downloading Package/Module Not Found (after pip install):

Occasionally you may run into errors where Pip installs a package that does not appear to download. There are multiple possible solutions and explanations:

1) Pip may install for use with Python2, where our program requires Python3. In this case, you can use the following command to install to the correct location.

```
pip3 install packagename
```

or (if you need to trigger the main file):

```
python -m pip3 install packagename
```

2) If that doesn't fix the problem, you may want to update Pip and Python packages to the latest stable version. Sometimes, there are Pip bugs that are fixed with an updated release.

3) If Pip seems to be buggy and you've given up on it, you can use an alternative install method (for mac users, this is easy_install).

4) For more advanced users, it is also worth checking your bash profile. This can be found in various locations based on the specific operating system (give it a google: "where is the bash_profile on …). Sometimes Python needs to be re-aliased if your operating system is juggling multiple Python versions.

Waiting on Stock Price:

The program may run, read the headlines, say it's executing order execution, then run in an infinite loop. This loop can result in no orders sending to the TWS. The problem here is most likely `stockPriceBool` is not set to True, as the stock price is not stored. A failure to return can result from a faulty or invalid data line connection. Check two things:

1) Make sure the market data lines are active (click data in the top right corner of the Trader Workstation)

2) Ensure you have purchased a data subscription. Log into your IB account online and check which information you have access to. Commonly, lines can be purchased for 1.50 dollars per month.

If you're frustrated with market data lines (trust me, I get it. They can be tricky). You can still send orders by removing the wait for the boolean and hard coding in quantity or other calculation without waiting for tick price returns.

Chapter 11: Popular Strategies and Next Steps

POPULAR STRATEGIES

This section discusses possible strategies for the future of your algorithm. The analysis does little to provide in-depth explanations as previously guided in this book. This section serves to merely introduce new traders to popular strategies the algorithm can employ. If you are interested in any strategy, there is an abundance of online resources and accounts dedicated to the math behind the execution.

In trading, there is no gain without the potential of loss. When implementing a new strategy, it is imperative to test thoroughly to ensure that you have the best chance of making a profit. Consider the potential for upside and loss with each new risk you decide to take.

News Cardinality Analysis:

Significant increases in news volumes correlate to significant movements in stock price. You can create a program to monitor any upticks or downticks in news or keyword volume and predict the price of the relevant stock. This is a similar strategy to what we accomplished with the web scraper in Chapters 7 and 8.

Pessimism Factor Analysis/Relative Frequency of Negative Words:

A high level of pessimism, as measured by calculating the deviance of pessimistic words in the news from a norm, can predict downward pressure on stock prices. Unusually extraordinarily high or low pessimism can indicate a higher volume. The release of pessimistic articles can create a longer-term effect that can stretch from hours to an indeterminate period.

This follows the ideology that negative sentiment is more potent and memorable than positive news and will have a more significant impact on a stock price. Tracking the adverse reports on an individual company can be used to predict earnings or suggest future analyst ratings. This strategy can be calculated by measuring the proportion of negative words over total words (with or without a weighted word distribution).

Emerging Markets and Uncertainty:

Political and economic news has a more significant effect on prices when coupled with a period of uncertainty. The dissemination of information with little previously disclosed public predictive analysis

can have a high impact on volatility. For example, a federal funds rate hike can have larger impacts when the expert consensus was mixed. An outcome that matches expectations shares a smaller impact.

Dependency Relations:

The interconnectedness of trade or collaboration between two countries or companies can have hidden and unforeseen impacts on related stock prices. Negative global news in a developed country can have significant impacts on dependent emerging markets. In addition, companies that share manufacturing techniques can fall victim to the same vulnerabilities and events. For example, if the U.S. is a large consumer of clothing from Malaysia, and American retail forecast gets stunted, then Malaysian clothing producers will likely see a related fall in demand.

Natural Language Processing:

Language can be categorized and interpreted by natural language processing engines to deliver a probabilistic conviction on factors such as tone, sentiment, and syntax. The combination of AI with linguistic can interpret news articles and human speech with no human interaction. Cloud computing solutions, like AWS, have special packages to interpret text.

Alternative Data:

By being extra resourceful and creative, you can obtain alternative data that is not directly related to market information. There are several free and pay to access websites and APIs that allow you to pipe data into your program. A great example of alternative data is satellite imagery and image recognition. A famous Stanford University article called *Understanding Satellite-Imagery-Based Crop Yield Predictions* tested a theory that satellites could correctly predict crop yields. In this case, satellite information was used to predict crop futures.

Other strategists have predicted the traffic flow to retail locations to predict earnings, or the impact of weather on specific regions. Alternative data allows for a comprehensive interpretation of the correlation between possibly unknown factors. Calculating your own regressions and testing for your ability to make money in theoretical positions can be a very worthwhile exercise.

High Frequency/Arbitrage:

Generally, these strategies are only profitable if you are able to obtain a unique technological edge or insight known to very few. Large companies spend billions a year to compete in the high-risk HFT market, often for small slivers of a percent in profit. These strategies can be lucrative in high volume but extremely difficult to perfect. Impressive math and technological infrastructure go into these strategies. So, it is a worthwhile research topic, at the very least.

Machine Learning:

Since this algorithm was built in Python, it lends excellent flexibility in importing machine learning/deep learning libraries. Machine learning can be broad, so when researching, identify a potential industry (energy, precious resources, a particular company, etc.) and determine exactly how you'd like to train a model to react given information.

Machine learning is not a godsend that can pick the best stock automatically. Instead, machine learning is intended to train on a dataset and adapt to incoming data. In many cases, the model is only as strong as the dataset or teacher. So, invest the extra effort to learn the basis of knowledge you are trying to instill and consider if you require an adaptive algorithm to accomplish it.

Statistical Arbitrage/Prediction:

Successful quantitative finance is deep-rooted in the understanding of statistics. Statistical arbitrage is a group of strategies that rely on statistics to predict pricing errors within pairs of securities. This strategy relies on the market returning to rationality eventually.

The strategy may look at two companies that are usually correlated, such as Google and Facebook, during a regular market day. If the price of Google skyrockets, while Facebook takes a dip, then the algorithm will seek to find a center between the pair. In this case, Google would be shorted and Facebook would be purchased. The inference assumes that if they were correlated before, and fell out of pattern, then they will eventually return.

MOVING FORWARD...

Congratulations on making it through the guide and developing your own quantitative finance algorithm! Only an elite group of determined and talented individuals can make it this far. The good news is, now that you are equipped with the fundamental skills of creating finance algorithms, you are ready to improve and discover even more about this exciting field.

The field of quantitative finance and systematic trading is continuously evolving and is an incredibly exciting discipline to study. You have set yourself apart from all the other traders by welding technology as a powerful tool. Good luck with your endeavors in the future as you continue to learn and grow!