DeVry University

College of Engineering and Information Sciences

1. Python Stock Tracking Project

Module 4

# Background

1. The project will provide students with experience creating applications in Python. Students will use object-oriented techniques to develop a stock tracking application. By processing the historical stock data, profit/loss reports can be generated. Data storage will allow users to save and retrieve stock data. The system will use the Python libraries to create charts.

**In this part of the project, we will cover inheritance.**

# Objectives – Module 4

1. Implement inheritance in the stock program
2. Create three classes
3. Run Unit Test

# Steps

1. **Always test as you go!**

Inheritance is a key part of object oriented programming. We will implement inheritance in our stock program by creating two types of retirement accounts. When investing, you can choose a traditional retirement account or an automatic account. The traditional account will allow you to choose to invest in specific stocks. The automatic account (also known as a robo account) will simply invest the account in a fund for a certain number of years.

## 2. Create Classes

Create a new python program named **account\_class.py** in the same directory as **stock\_menu.py** and **stock\_class.py**.

Add your name and date in the comments area.

Review the following UML diagram.



Write the code for your Retirement\_Account class, Traditional, and Robo accounts. Use the stock\_class code as a guide.

Check your code comments. Comments should describe each major section of code.

Add the following code as the first line in your program:

from stock\_class import Stock

**Hints**

* Each class should start with a constructor definition using the **\_\_init\_\_()** method.
* To create a class that inherits the functionality of another class, add the parameter of the parent class when creating the child class and add a call to the parent’s init function:

class Animal:

def \_\_init\_\_(self, age, name):

self.age = age

self.name = name

class Dog(Animal):

def \_\_init\_\_(self, age, name, breed):

Parent.\_\_init\_\_(self, age, name)#parent constructor

self.breed = breed

* Always use **self** as the first parameter in any method in a class.
* Always use **self** in front of any variables in the class.
* Review the stock\_class.py code for examples on how to create a class
* The investment\_return code for the Robo account is the following:

def investment\_return(self):

return (self.years\*self.balance\*1.05)

## 3. Run Unit Test

Developers use unit testing to test the code they write. Unit tests are typically done on small sections of code that can act independently of other code in the program.

When you run the unit tests it will attempt to create a new stock and add price history to the stock. It will also test changing the stock symbol (not allowed so a successful test would fail to change the symbol, change the name, change shares, etc. The unit test will produce a report showing which tests passed or failed, then provide a summary list of errors that need to be fixed.

Once your class module passes all the unit testing, you are finished and ready to move on to the next module. If your classes fail any test, be sure to fix before moving on.

1. Run the unit test by running the account\_class.py file.

2. Make note of any errors and go back and fix them.

3. Repeat until there are no errors.

Copy and paste the following code to create unit tests below your class definitions:

# Unit Test - Do Not Change Code Below This Line \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\*

# main() is used for unit testing only. It will run when stock\_class.py is run.

# Run this to test your class code. Once you have eliminated all errors, you are

# ready to continue with the next part of the project.

def main():

error\_count = 0

error\_list = []

print("Unit Testing Starting---")

# Test Add Traditional

stock\_list =[]

testStock = Stock("TEST", "Testcompany", 200)

stock\_list.append(testStock)

print("Testing Add Retirement Account...",end="")

try:

testRetire = Retirement\_Account(200,"12345")

print("Successful!")

except:

print("\*\*\*Adding retirement Account Failed!")

error\_count = error\_count+1

error\_list.append("Retirement Constructor Error")

print("Testing Add Traditional Account...",end="")

try:

testTraditional = Traditional(200,"12345")

testTraditional.add\_stock(stock\_list)

print("Successful!")

except:

print("\*\*\*Adding Traditional Account Failed!")

error\_count = error\_count+1

error\_list.append("Traditional Constructor Error")

# Test Change Balance

print("Test Change Balance...",end="")

try:

testTraditional.balance = 1000

if testTraditional.balance == 1000:

print("Successful!")

else:

print("\*\*\*ERROR! Balance change unsuccessful.")

error\_count = error\_count+1

error\_list.append("Balance Change Error")

except:

print("\*\*\*ERROR! Balance change failed.")

error\_count = error\_count+1

error\_list.append("Balance Change Failure")

# Test Change Number

print("Test Change Number...",end="")

try:

testTraditional.number = "99999"

if testTraditional.number == "99999":

print("Successful!")

else:

print("\*\*\*ERROR! Number change unsuccessful.")

error\_count = error\_count+1

error\_list.append("Number Change Error")

except:

print("\*\*\*ERROR! Number change failed.")

error\_count = error\_count+1

error\_list.append("Number Change Failure")

print("Testing Add Robo Account...",end="")

try:

testRobo = Robo(200,"12345",5)

print("Successful!")

except:

print("\*\*\*Adding Robo Account Failed!")

error\_count = error\_count+1

error\_list.append("Robo Constructor Error")

# Test Change years

print("Test Change Balance...",end="")

try:

testRobo.years = 1000

if testRobo.years == 1000:

print("Successful!")

else:

print("\*\*\*ERROR! Years change unsuccessful.")

error\_count = error\_count+1

error\_list.append("Years Change Error")

except:

print("\*\*\*ERROR! Years change failed.")

error\_count = error\_count+1

error\_list.append("Years Change Failure")

# Test investment return

print("Test investment return...",end="")

try:

testRobo.years = 1000

testRobo.balance = 1

if testRobo.investment\_return() == 1050:

print("Successful!")

else:

print("\*\*\*ERROR!Investment return unsuccessful.")

error\_count = error\_count+1

error\_list.append("investment return Error")

except:

print("\*\*\*ERROR! investment return failed.")

error\_count = error\_count+1

error\_list.append("Investment Return Failure")

if (error\_count) == 0:

print("Congratulations - All Tests Passed")

else:

print("-=== Problem List - Please Fix ===-")

for em in error\_list:

print(em)

print("Goodbye")

# Program Starts Here

if \_\_name\_\_ == "\_\_main\_\_":

# run unit testing only if run as a stand-alone script

main()

## 4. Add code to stock\_menu

1. Open the **stock\_menu.py** file in your IDE.
2. **At the top of your code, uncomment the from account\_class import Traditional, Robo line.**
3. *Code Explanation*
4. The investment\_type method creates a retirement account for the user based on two different types. First, the user inputs the balance and account number. Then they choose whether it is a traditional or robo account.
5. If it is a robo account the program will calculate the investment based on a number of years with no stock purchases.

If the user chooses a traditional account. The user will choose which stocks they want to purchase and how many shares.

1. Replace the stub code for the **investment\_type** method in **stock\_menu**.**py** with the following code to handle both types of investors.

print("Investment Account ---")

balance = float(input("What is your initial balance: "))

number = input("What is your account number: ")

acct= input("Do you want a Traditional (t) or Robo (r) account: ")

if acct.lower() == "r":

years = float(input("How many years until retirement: "))

robo\_acct = Robo(balance, number, years)

print("Your investment return is ",robo\_acct.investment\_return())

print("\n\n")

elif acct.lower() == "t":

trad\_acct = Traditional(balance, number)

temp\_list=[]

print("Choose stocks from the list below: ")

while True:

print("Stock List: [",end="")

for stock in stock\_list:

print(stock.symbol," ",end="")

print("]")

symbol = input("Which stock do you want to purchase, 0 to quit: ").upper()

if symbol =="0":

break

shares = float(input("How many shares do you want to buy?: "))

found = False

for stock in stock\_list:

if stock.symbol == symbol:

found = True

current\_stock = stock

if found == True:

current\_stock.shares += shares

temp\_list.append(current\_stock)

print("Bought ",shares,"of",symbol)

else:

print("Symbol Not Found \*\*\*")

trad\_acct.add\_stock(temp\_list)

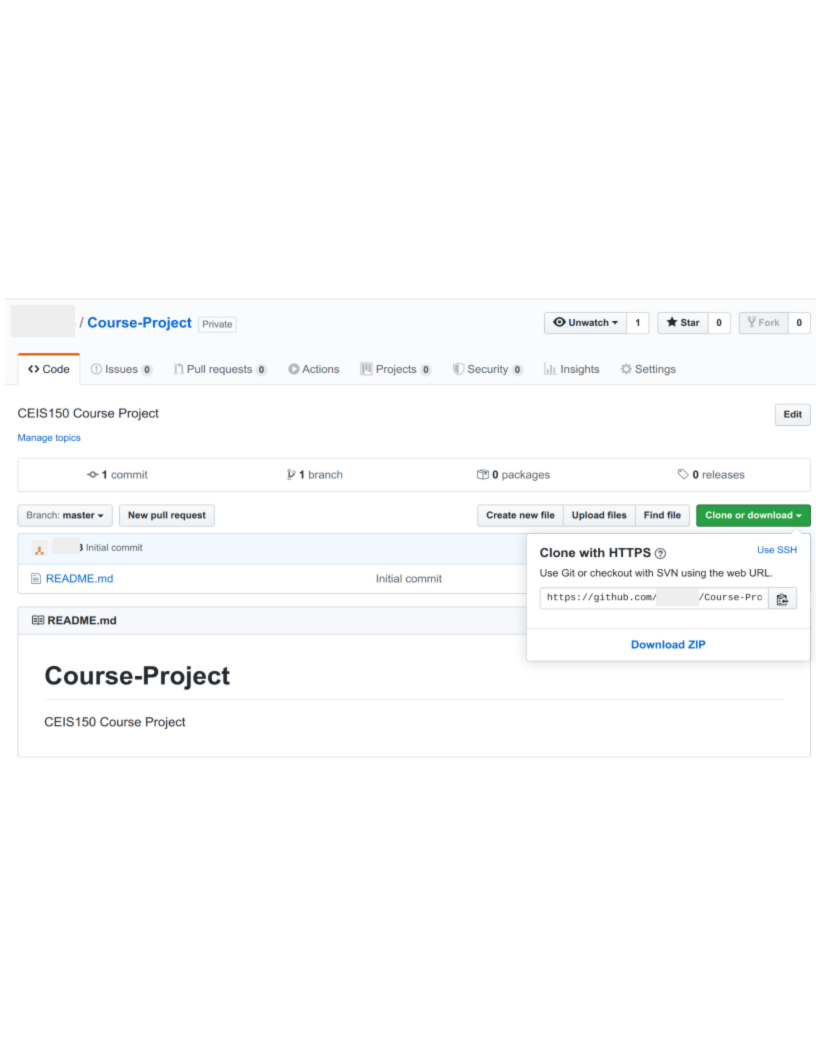
# Test your classes

1. Run your program and add at least two stocks. In the main menu, choose investor type. Add in a balance and account number. Test the traditional account. Choose a stock to buy shares and test buying shares of that stock. Next, test the robo account and add in a number of years to see your investment return.

# Deliverables

1. **To submit this part of the project:**
2. 1. Take a screen shot of your completed account\_class.py file in your IDE. It is not necessary to show all the code but you should include as much of the top of your code including your name as possible.
3. 2. Take a screen shot of the completed unit test showing no errors.
4. 3. Take a screenshot of testing your classes in the main menu.
5. 3. Paste the screen shots into the appropriate slide in the PowerPoint template file provided in the Project area for this module.
6. 4. Submit the file in the Assignments area.

# Optional

1. Push your updated project files to GitHub. This will provide a backup for your project should anything happen to your local files. It will also allow you to go back through your files to see a change history. If you accidentally modify or overwrite code, you can always find an earlier version of the file.
2. ****
3. Find your repository url on GitHub. Then navigate to your project folder and use the following commands. Replace X with the week/module number you are submitting. Replace {your url} with the address of your GitHub repository.
4. At the **Anaconda Prompt**, the following commands will: stage the changes, commit the changes, and push the updates to GitHub.
5. **git add --all**
6. **git commit -m "Module X"** ←Change X to the Module you are submitting.
7. **git push {your url}** ←Change {your url} to the url for your repository on GitHub.
8. **Note: VS Code users can use the Source Control tab on the left to stage, commit, and push updates to GitHub.**
9. **Note: Use the Snipping Tool built into Windows to take screen shots (click Start and search for Snipping Tool)**