03-Final Capstone Suggested Walkthrough

July 30, 2019

1 Final Capstone Project - Suggested Walkthrough:

This is a suggested method for handling one of the Final Capstone Projects. We start by coding out the strictest requirements, and then build out from a working baseline model. Feel free to adapt this solution, and add features you think could help. Good luck!

1.1 Bank Account Manager

Under the Classes section in the list of suggested final capstone projects is a Bank Account Manager program. The goal is to create a class called Account which will be an abstract class for three other classes called CheckingAccount, SavingsAccount and BusinessAccount. Then you should manage credits and debits from these accounts through an ATM style program.

1.1.1 Project Scope

To tackle this project, first consider what has to happen. 1. There will be three different types of bank account (Checking, Savings, Business) 2. Each account will accept deposits and withdrawals, and will need to report balances

1.1.2 Project Wishlist

We might consider additional features, like: * impose a monthly maintenance fee * waive fees for minimum combined deposit balances * each account may have additional properties unique to that account: * Checking allows unlimited transactions, and may keep track of printed checks * Savings limits the number of withdrawals per period, and may earn interest * Business may impose transaction fees * automatically transfer the "change" for debit card purchases from Checking to Savings, where "change" is the amount needed to raise a debit to the nearest whole dollar * permit savings autodraft overdraft protection

1.1.3 Let's get started!

Step 1: Establish an abstract Account class with features shared by all accounts. Note that abstract classes are never instantiated, they simply provide a base class with attributes and methods to be inherited by any derived class.

[1]:
```
class Account:
    # Define an __init__ constructor method with attributes shared by all␣
↪accounts:
```

```
    def __init__(self,acct_nbr,opening_deposit):
        self.acct_nbr = acct_nbr
        self.balance = opening_deposit

    # Define a __str__ mehthod to return a recognizable string to any print()␣
↪command
    def __str__(self):
        return f'${self.balance:.2f}'

    # Define a universal method to accept deposits
    def deposit(self,dep_amt):
        self.balance += dep_amt

    # Define a universal method to handle withdrawals
    def withdraw(self,wd_amt):
        if self.balance >= wd_amt:
            self.balance -= wd_amt
        else:
            return 'Funds Unavailable'
```

Step 2: Establish a Checking Account class that inherits from Account, and adds Checking-specific traits.

[2]:
```
class Checking(Account):
    def __init__(self,acct_nbr,opening_deposit):
        # Run the base class __init__
        super().__init__(acct_nbr,opening_deposit)

    # Define a __str__ method that returns a string specific to Checking␣
↪accounts
    def __str__(self):
        return f'Checking Account #{self.acct_nbr}\n  Balance: {Account.
↪__str__(self)}'
```

Step 3: TEST setting up a Checking Account object

[3]:
```
x = Checking(54321,654.33)
```

[4]:
```
print(x)
```

```
Checking Account #54321
  Balance: $654.33
```

[5]:
```
x.withdraw(1000)
```

[5]:
```
'Funds Unavailable'
```

[6]:
```
x.withdraw(30)
```

[7]:
```
x.balance
```

[7]:
```
624.33
```

Step 4: Set up similar Savings and Business account classes

[8]:
```
class Savings(Account):
    def __init__(self,acct_nbr,opening_deposit):
        # Run the base class __init__
        super().__init__(acct_nbr,opening_deposit)

    # Define a __str__ method that returns a string specific to Savings␣
 ↪accounts
    def __str__(self):
        return f'Savings Account #{self.acct_nbr}\n  Balance: {Account.
 ↪__str__(self)}'


class Business(Account):
    def __init__(self,acct_nbr,opening_deposit):
        # Run the base class __init__
        super().__init__(acct_nbr,opening_deposit)

    # Define a __str__ method that returns a string specific to Business␣
 ↪accounts
    def __str__(self):
        return f'Business Account #{self.acct_nbr}\n  Balance: {Account.
 ↪__str__(self)}'
```

At this point we've met the minimum requirement for the assignment. We have three different bank account classes. Each one can accept deposits, make withdrawals and report a balance, as they each inherit from an abstract Account base class.

So now the fun part - let's add some features!

Step 5: Create a Customer class For this next phase, let's set up a Customer class that holds a customer's name and PIN and can contain any number and/or combination of Account objects.

[9]:
```
class Customer:
    def __init__(self, name, PIN):
        self.name = name
        self.PIN = PIN

        # Create a dictionary of accounts, with lists to hold multiple accounts
        self.accts = {'C':[],'S':[],'B':[]}

    def __str__(self):
        return self.name
```

```
    def open_checking(self,acct_nbr,opening_deposit):
        self.accts['C'].append(Checking(acct_nbr,opening_deposit))

    def open_savings(self,acct_nbr,opening_deposit):
        self.accts['S'].append(Savings(acct_nbr,opening_deposit))

    def open_business(self,acct_nbr,opening_deposit):
        self.accts['B'].append(Business(acct_nbr,opening_deposit))

    # rather than maintain a running total of deposit balances,
    # write a method that computes a total as needed
    def get_total_deposits(self):
        total = 0
        for acct in self.accts['C']:
            print(acct)
            total += acct.balance
        for acct in self.accts['S']:
            print(acct)
            total += acct.balance
        for acct in self.accts['B']:
            print(acct)
            total += acct.balance
        print(f'Combined Deposits: ${total}')
```

Step 6: TEST setting up a Customer, adding accounts, and checking balances

[10]:
```
bob = Customer('Bob',1)
```

[11]:
```
bob.open_checking(321,555.55)
```

[12]:
```
bob.get_total_deposits()
```

```
Checking Account #321
  Balance: $555.55
Combined Deposits: $555.55
```

[13]:
```
bob.open_savings(564,444.66)
```

[14]:
```
bob.get_total_deposits()
```

```
Checking Account #321
  Balance: $555.55
Savings Account #564
  Balance: $444.66
Combined Deposits: $1000.21
```

[15]:
```
nancy = Customer('Nancy',2)
```

[16]:
```
nancy.open_business(2018,8900)
```

[17]:
```
nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900
```

Wait! Why don't Nancy's combined deposits show a decimal? This is easily fixed in the class definition (mostly copied from above, with a change made to the last line of code):

[18]:
```
class Customer:
    def __init__(self, name, PIN):
        self.name = name
        self.PIN = PIN
        self.accts = {'C':[],'S':[],'B':[]}

    def __str__(self):
        return self.name

    def open_checking(self,acct_nbr,opening_deposit):
        self.accts['C'].append(Checking(acct_nbr,opening_deposit))

    def open_savings(self,acct_nbr,opening_deposit):
        self.accts['S'].append(Savings(acct_nbr,opening_deposit))

    def open_business(self,acct_nbr,opening_deposit):
        self.accts['B'].append(Business(acct_nbr,opening_deposit))

    def get_total_deposits(self):
        total = 0
        for acct in self.accts['C']:
            print(acct)
            total += acct.balance
        for acct in self.accts['S']:
            print(acct)
            total += acct.balance
        for acct in self.accts['B']:
            print(acct)
            total += acct.balance
        print(f'Combined Deposits: ${total:.2f}') # added precision formatting␣
↪here
```

So it's fixed, right?

[19]:
```
nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900
```

Nope! Changes made to the class definition do *not* affect objects created under different sets of instructions.To fix Nancy's account, we have to build her record from scratch.

[20]:
```
nancy = Customer('Nancy',2)
nancy.open_business(2018,8900)
nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900.00
```

This is why testing is so important!

Step 7: Let's write some functions for making deposits and withdrawals. Be sure to include a docstring that explains what's expected by the function!

[21]:
```
def make_dep(cust,acct_type,acct_num,dep_amt):
    """
    make_dep(cust, acct_type, acct_num, dep_amt)
    cust      = variable name (Customer record/ID)
    acct_type = string 'C' 'S' or 'B'
    acct_num  = integer
    dep_amt   = integer
    """
    for acct in cust.accts[acct_type]:
        if acct.acct_nbr == acct_num:
            acct.deposit(dep_amt)
```

[22]:
```
make_dep(nancy,'B',2018,67.45)
```

[23]:
```
nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8967.45
Combined Deposits: $8967.45
```

[24]:
```
def make_wd(cust,acct_type,acct_num,wd_amt):
    """
    make_dep(cust, acct_type, acct_num, wd_amt)
    cust      = variable name (Customer record/ID)
    acct_type = string 'C' 'S' or 'B'
    acct_num  = integer
    wd_amt    = integer
    """
    for acct in cust.accts[acct_type]:
        if acct.acct_nbr == acct_num:
            acct.withdraw(wd_amt)
```

[25]:
```
make_wd(nancy,'B',2018,1000000)
```

[26]:
```
nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8967.45
Combined Deposits: $8967.45
```

What happened?? We seemed to successfully make a withdrawal, but nothing changed!This is because, at the very beginning, we had our Account class *return* the string 'Funds Unavailable' instead of print it. If we change that here, we'll have to also run the derived class definitions, and Nancy's creation, but *not* the Customer class definition. Watch:

[27]:
```
class Account:
    def __init__(self,acct_nbr,opening_deposit):
        self.acct_nbr = acct_nbr
        self.balance = opening_deposit

    def __str__(self):
        return f'${self.balance:.2f}'

    def deposit(self,dep_amt):
        self.balance += dep_amt

    def withdraw(self,wd_amt):
        if self.balance >= wd_amt:
            self.balance -= wd_amt
        else:
            print('Funds Unavailable')  # changed "return" to "print"
```

[30]:
```
class Checking(Account):
    def __init__(self,acct_nbr,opening_deposit):
        super().__init__(acct_nbr,opening_deposit)

    def __str__(self):
        return f'Checking Account #{self.acct_nbr}\n  Balance: {Account.
 ↪__str__(self)}'


class Savings(Account):
    def __init__(self,acct_nbr,opening_deposit):
        super().__init__(acct_nbr,opening_deposit)

    def __str__(self):
        return f'Savings Account #{self.acct_nbr}\n  Balance: {Account.
 ↪__str__(self)}'


class Business(Account):
    def __init__(self,acct_nbr,opening_deposit):
        super().__init__(acct_nbr,opening_deposit)
```

```
    def __str__(self):
        return f'Business Account #{self.acct_nbr}\n  Balance: {Account.
↪__str__(self)}'
```

[31]:
```
nancy = Customer('Nancy',2)
nancy.open_business(2018,8900)
nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900.00
```

[32]:
```
make_wd(nancy,'B',2018,1000000)
```

```
Funds Unavailable
```

[33]:
```
nancy.get_total_deposits()
```

```
Business Account #2018
  Balance: $8900.00
Combined Deposits: $8900.00
```

1.2 Good job!