Lab 6 Solutions **lab06.zip (lab06.zip)**

Solution Files

Required Questions

Object-Oriented Programming

Here's a refresher on Object-Oriented Programming. It's okay to skip directly to the questions and refer back here if you get stuck.

Q1: Bank Account

Extend the `BankAccount` class to include a `transactions` attribute. This attribute should be a list that keeps track of each transaction made on the account. Whenever the `deposit` or `withdraw` method is called, a new `Transaction` instance should be created and added to the list, **even if the action is not successful**.

The `Transaction` class should have the following attributes:

- `before` : The account balance before the transaction.
- `after` : The account balance after the transaction.
- `id` : The transaction ID, which is the number of previous transactions (deposits or withdrawals) made on that account. The transaction IDs for a specific `BankAccount` instance must be unique, but this `id` does not need to be unique across all accounts. In other words, you only need to ensure that no two `Transaction` objects made by the same BankAccount have the same `id` .

In addition, the `Transaction` class should have the following methods:

- `changed()` : Returns `True` if the balance changed (i.e., `before` is different from `after`), otherwise returns `False` .

- `report()`: Returns a string describing the transaction. The string should start with the transaction ID and describe the change in balance. Take a look at the doctests for the expected output.

```
class Transaction:
    def __init__(self, id, before, after):
        self.id = id
        self.before = before
        self.after = after

    def changed(self):
        """Return whether the transaction resulted in a changed balance."""
        return self.before != self.after

    def report(self):
        """Return a string describing the transaction.

        >>> Transaction(3, 20, 10).report()
        '3: decreased 20->10'
        >>> Transaction(4, 20, 50).report()
        '4: increased 20->50'
        >>> Transaction(5, 50, 50).report()
        '5: no change'
        """
        msg = 'no change'
        if self.changed():
            if self.after < self.before:
                verb = 'decreased'
            else:
                verb = 'increased'
            msg = verb + ' ' + str(self.before) + '->' + str(self.after)
        return str(self.id) + ': ' + msg

class BankAccount:
    """A bank account that tracks its transaction history.

    >>> a = BankAccount('Eric')
    >>> a.deposit(100)    # Transaction 0 for a
    100
    >>> b = BankAccount('Erica')
    >>> a.withdraw(30)    # Transaction 1 for a
    70
    >>> a.deposit(10)     # Transaction 2 for a
    80
    >>> b.deposit(50)     # Transaction 0 for b
    50
    >>> b.withdraw(10)    # Transaction 1 for b
    40
    >>> a.withdraw(100)   # Transaction 3 for a
```

```
    'Insufficient funds'
    >>> len(a.transactions)
    4
    >>> len([t for t in a.transactions if t.changed()])
    3
    >>> for t in a.transactions:
    ...     print(t.report())
    0: increased 0->100
    1: decreased 100->70
    2: increased 70->80
    3: no change
    >>> b.withdraw(100)   # Transaction 2 for b
    'Insufficient funds'
    >>> b.withdraw(30)    # Transaction 3 for b
    10
    >>> for t in b.transactions:
    ...     print(t.report())
    0: increased 0->50
    1: decreased 50->40
    2: no change
    3: decreased 40->10
    """

    # *** YOU NEED TO MAKE CHANGES IN SEVERAL PLACES IN THIS CLASS ***
    def next_id(self):
        # There are many ways to implement this counter, such as using an instance
        # attribute to track the next ID.
        return len(self.transactions)

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
        self.transactions = []

    def deposit(self, amount):
        """Increase the account balance by amount, add the deposit
        to the transaction history, and return the new balance.
        """
        self.transactions.append(Transaction(self.next_id(), self.balance, self.balance +
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        """Decrease the account balance by amount, add the withdraw
        to the transaction history, and return the new balance.
```

```
        """
        if amount > self.balance:
            self.transactions.append(Transaction(self.next_id(), self.balance, self.balanc
            return 'Insufficient funds'
        self.transactions.append(Transaction(self.next_id(), self.balance, self.balance -
        self.balance = self.balance - amount
```

Use Ok to test your code:

```
python3 ok -q BankAccount
```

Q2: Email

An email system has three classes: `Email`, `Server`, and `Client`. A `Client` can `compose` an email, which it will `send` to the `Server`. The `Server` then delivers it to the `inbox` of another `Client`. To achieve this, a `Server` has a dictionary called `clients` that maps the name of the `Client` to the `Client` instance.

Assume that a client never changes the server that it uses, and it only composes emails using that server.

Fill in the definitions below to finish the implementation! The `Email` class has been completed for you.

> **Important**: Before you start, make sure you read the entire code snippet to understand the relationships between the classes, and pay attention to the parameter type of the methods. Think about what variables you have access to in each method and how can you use them to access the other classes and their methods.

Note:

- The `sender` parameter from the `__init__(self, msg, sender, recipient_name)` method in the `Email` class is a `Client` instance.
- The `client` parameter from the `register_client(self, client)` method in the `Server` class is a `Client` instance.
- The `email` parameter from the `send(self, email)` method in the `Server` class is an `Email` instance.

```
class Email:
    """An email has the following instance attributes:

        msg (str): the contents of the message
        sender (Client): the client that sent the email
        recipient_name (str): the name of the recipient (another client)
    """
    def __init__(self, msg, sender, recipient_name):
        self.msg = msg
        self.sender = sender
        self.recipient_name = recipient_name

class Server:
    """Each Server has one instance attribute called clients that is a
    dictionary from client names to client objects.
    """
    def __init__(self):
        self.clients = {}

    def send(self, email):
        """Append the email to the inbox of the client it is addressed to.
            email is an instance of the Email class.
        """
        self.clients[email.recipient_name].inbox.append(email)

    def register_client(self, client):
        """Add a client to the clients mapping (which is a
        dictionary from client names to client instances).
            client is an instance of the Client class.
        """
        self.clients[client.name] = client

class Client:
    """A client has a server, a name (str), and an inbox (list).

    >>> s = Server()
    >>> a = Client(s, 'Alice')
    >>> b = Client(s, 'Bob')
    >>> a.compose('Hello, World!', 'Bob')
    >>> b.inbox[0].msg
    'Hello, World!'
    >>> a.compose('CS 61A Rocks!', 'Bob')
    >>> len(b.inbox)
    2
    >>> b.inbox[1].msg
```

```
    'CS 61A Rocks!'
    >>> b.inbox[1].sender.name
    'Alice'
    """
    def __init__(self, server, name):
        self.inbox = []
        self.server = server
        self.name = name
        server.register_client(self)

    def compose(self, message, recipient_name):
        """Send an email with the given message to the recipient."""
        email = Email(message, self, recipient_name)
        self.server.send(email)
```

Use Ok to test your code:

```
python3 ok -q Client
```

Q3: Mint

A mint is a place where coins are made. In this question, you'll implement a `Mint` class that can output a `Coin` with the correct year and worth.

- Each `Mint` instance has a `year` stamp. The `update` method sets the `year` stamp of the instance to the `present_year` class attribute of the `Mint` *class*.
- The `create` method takes a subclass of `Coin` (*not* an instance!), then creates and returns an instance of that subclass stamped with the `Mint` 's year (which may be different from `Mint.present_year` if it has not been updated.)
- A `Coin` 's `worth` method returns the `cents` value of the coin plus one extra cent for each year of age beyond 50. A coin's age can be determined by subtracting the coin's year from the `present_year` class attribute of the `Mint` class.

```
class Mint:
    """A mint creates coins by stamping on years.

    The update method sets the mint's stamp to Mint.present_year.

    >>> mint = Mint()
    >>> mint.year
    2024
    >>> dime = mint.create(Dime)
    >>> dime.year
    2024
    >>> Mint.present_year = 2104  # Time passes
    >>> nickel = mint.create(Nickel)
    >>> nickel.year     # The mint has not updated its stamp yet
    2024
    >>> nickel.worth()  # 5 cents + (80 - 50 years)
    35
    >>> mint.update()   # The mint's year is updated to 2104
    >>> Mint.present_year = 2179     # More time passes
    >>> mint.create(Dime).worth()    # 10 cents + (75 - 50 years)
    35
    >>> Mint().create(Dime).worth()  # A new mint has the current year
    10
    >>> dime.worth()     # 10 cents + (155 - 50 years)
    115
    >>> Dime.cents = 20  # Upgrade all dimes!
    >>> dime.worth()     # 20 cents + (155 - 50 years)
    125
    """
    present_year = 2024

    def __init__(self):
        self.update()

    def create(self, coin):
        return coin(self.year)

    def update(self):
        self.year = Mint.present_year

class Coin:
    cents = None # will be provided by subclasses, but not by Coin itself

    def __init__(self, year):
        self.year = year
```

```
    def worth(self):
        return self.cents + max(0, Mint.present_year - self.year - 50)

class Nickel(Coin):
    cents = 5

class Dime(Coin):
    cents = 10
```

Use Ok to test your code:

```
python3 ok -q Mint
```

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit Assignment

If you are in a regular section of CS 61A, fill out this lab attendance and feedback form (https://forms.gle/dHxj8gttNWRY6Ptm9). (If you are in the mega section, you don't need to fill out the form.)

Then, submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 (../lab00/#submit-with-gradescope) has detailed instructions.