Lab 6: OOP **lab06.zip (lab06.zip)**

Due by 11:59pm on Wednesday, March 6.

Starter Files

Download lab06.zip (lab06.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Required Questions

Getting Started Videos

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.

Object-Oriented Programming

Object-oriented programming (OOP) is a method of organizing programs in terms of objects and classes.

Here's an example class:

```
class Car:
    num_wheels = 4

    def __init__(self, color):
        self.wheels = Car.num_wheels
        self.color = color

    def drive(self):
        if self.wheels <= Car.num_wheels:
            return self.color + ' car cannot drive!'
        return self.color + ' car goes vroom!'

    def pop_tire(self):
        if self.wheels > 0:
            self.wheels -= 1
```

Here's some terminology:

- **class**: the type of an object, which describes the behavior of its instances. The `Car` class (shown above) describes the behavior that all `Car` objects have.
- **instance**: the term "instance" means then same thing as "object"; every object is an instance of its class. In Python, new instances are created by calling a class.

  ```
  >>> ferrari = Car('red')
  ```

  Here, `ferrari` is a name bound to an instance of the `Car` class.
- **attribute**: Objects have named attributes, such as `wheels` and `color` in this example. They can be accessed using dot expressions and changed using assignment.

  ```
  >>> ferrari.color
  'red'
  >>> ferrari.wheels
  4
  >>> ferrari.color = 'green'
  >>> ferrari.color
  'green'
  ```
- **method**: Methods are functions that are attributes of a class and called using a dot expression. Methods often describe actions associated with an object, such as `drive` a car.

  ```
  >>> ferrari = Car('red')
  >>> ferrari.drive()
  'red car goes vroom!'
  ```
- **The __init__ special method**: The method called `__init__` is special because it is called automatically when a new instance of a class is created; it is where we initialize the instance attributes. The expression `Car('red')` calls the `__init__` method of the `Car` class.

- `self` : in Python, `self` is the first parameter for methods When a method is called, `self` is automatically bound to an instance of the class that was used in the dot expression that called the method. For example, in:

```
>>> ferrari = Car('red')
>>> ferrari.drive()
```

  Notice how the `drive` method takes in `self` as an argument, but it looks like we didn't pass one in! This is because the dot notation *implicitly* passes in `ferrari` as `self` for us. So in this example, `self` is bound to the object called `ferrari` in the global frame.

Q1: Bank Account

Extend the `Account` class from lecture so that each `Account` instance has a `transactions` attribute, which is a list of `Transaction` instances, one for every call made to `deposit` or `withdraw`. A `Transaction` instance records the balance `before` and `after` each call to `deposit` or `withdraw`. In addition, each `Transaction` is assigned an `id` attribute, which is the number of previous calls to `deposit` or `withdraw` on that account. The `id` is only unique within the scope of each account, rather than being a universally unique identifier across all transactions.

A `Transaction` has two methods:

- `changed` returns `True` if the balance was different before and after the transaction and `False` otherwise.
- `report` returns a string describing the transaction. It starts with the ID and then a message.

```
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."""
        "*** YOUR CODE HERE ***"

    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():
            "*** YOUR CODE HERE ***"
        return str(self.id) + ': ' + msg

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

    >>> a = Account('Eric')
    >>> a.deposit(100)    # Transaction 0 for a
    100
    >>> b = Account('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 __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        """Increase the account balance by amount, add the deposit
        to the transaction history, and return the new 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:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

Use Ok to test your code:

```
python3 ok -q Account                                                    ✂
```

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 matches the `recipient_name` in an `Email` to the `Client` object with that name.

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.

```
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."""
        ____.inbox.append(email)

    def register_client(self, client):
        """Add a client to the dictionary of clients."""
        ____[____] = ____

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(____)

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

Use Ok to test your code:

```
python3 ok -q Client                                                    ✂
```

Q3: Make Change

Implement `make_change`, which takes a positive integer `amount` and a dictionary of `coins`. The `coins` dictionary keys are positive integer denominations and its values are positive integer coin counts. For example, `{1: 4, 5: 2}` represents four pennies and two nickels. The `make_change` function returns a list of coins that sum to `amount`, where the count of any denomination `k` in the return value is at most `coins[k]`.

If there are multiple ways to make change for `amount`, prefer to use as many of the smallest coins available and place the smallest coins first in the returned list.

```
def make_change(amount, coins):
    """Return a list of coins that sum to amount, preferring the smallest coins
    available and placing the smallest coins first in the returned list.

    The coins argument is a dictionary with keys that are positive integer
    denominations and values that are positive integer coin counts.

    >>> make_change(2, {2: 1})
    [2]
    >>> make_change(2, {1: 2, 2: 1})
    [1, 1]
    >>> make_change(4, {1: 2, 2: 1})
    [1, 1, 2]
    >>> make_change(4, {2: 1}) == None
    True

    >>> coins = {2: 2, 3: 2, 4: 3, 5: 1}
    >>> make_change(4, coins)
    [2, 2]
    >>> make_change(8, coins)
    [2, 2, 4]
    >>> make_change(25, coins)
    [2, 3, 3, 4, 4, 4, 5]
    >>> coins[8] = 1
    >>> make_change(25, coins)
    [2, 2, 4, 4, 5, 8]
    """
    if not coins:
        return None
    smallest = min(coins)
    rest = remove_one(coins, smallest)
    if amount < smallest:
        return None
    "*** YOUR CODE HERE ***"
```

You should use the `remove_one` function in your implementation:

```
def remove_one(coins, coin):
    """Remove one coin from a dictionary of coins. Return a new dictionary,
    leaving the original dictionary coins unchanged.

    >>> coins = {2: 5, 3: 2, 6: 1}
    >>> remove_one(coins, 2) == {2: 4, 3: 2, 6: 1}
    True
    >>> remove_one(coins, 6) == {2: 5, 3: 2}
    True
    >>> coins == {2: 5, 3: 2, 6: 1} # Unchanged
    True
    """
    copy = dict(coins)
    count = copy.pop(coin) - 1  # The coin denomination is removed
    if count:
        copy[coin] = count      # The coin denomination is added back
    return copy
```

> Hint: Try using the smallest coin to make change. If it turns out that there is no way to make change using the smallest coin, then try making change without the smallest coin.
>
> Hint: The simplest solution does not involve defining any local functions, but you can define additional functions if you wish.

Definitely try to solve this without reading the walkthrough, but if you're really stuck then read the walkthrough.

Walkthrough

The code for `make_change(amount, coins)` should do the following:

1. Check if `amount == smallest`, in which case return a one-element list containing just `smallest`.
2. Otherwise, call `make_change(amount-smallest, rest)`, which returns either `None` or a list of numbers.
3. If the call in Step 2 returned a list, then return a longer list that includes `smallest` at the front.
4. If the call in Step 2 returned `None`, then there's no way to use the `smallest` coin, so just `make_change(amount, rest)`

Use Ok to test your code:

```
python3 ok -q make_change                                                    ✂
```

Q4: Change Machine

Complete the `change` method of the `ChangeMachine` class. A `ChangeMachine` instance holds some `coins`, which are initially all pennies. The `change` method takes a positive integer `coin`, adds that coin to its `coins`, and then returns a list that sums to `coin`. The machine prefers to return as many of the smallest coins available, ordered from smallest to largest. The coins returned by `change` are removed from the machine's `coins`.

```
class ChangeMachine:
    """A change machine holds a certain number of coins, initially all pennies.
    The change method adds a single coin of some denomination X and returns a
    list of coins that sums to X. The machine prefers to return the smallest
    coins available. The total value in the machine never changes, and it can
    always make change for any coin (perhaps by returning the coin passed in).

    The coins attribute is a dictionary with keys that are positive integer
    denominations and values that are positive integer coin counts.

    >>> m = ChangeMachine(2)
    >>> m.coins == {1: 2}
    True
    >>> m.change(2)
    [1, 1]
    >>> m.coins == {2: 1}
    True
    >>> m.change(2)
    [2]
    >>> m.coins == {2: 1}
    True
    >>> m.change(3)
    [3]
    >>> m.coins == {2: 1}
    True

    >>> m = ChangeMachine(10) # 10 pennies
    >>> m.coins == {1: 10}
    True
    >>> m.change(5) # takes a nickel & returns 5 pennies
    [1, 1, 1, 1, 1]
    >>> m.coins == {1: 5, 5: 1} # 5 pennies & a nickel remain
    True
    >>> m.change(3)
    [1, 1, 1]
    >>> m.coins == {1: 2, 3: 1, 5: 1}
    True
    >>> m.change(2)
    [1, 1]
    >>> m.change(2) # not enough 1's remaining; return a 2
    [2]
    >>> m.coins == {2: 1, 3: 1, 5: 1}
    True
    >>> m.change(8) # cannot use the 2 to make 8, so use 3 & 5
    [3, 5]
    >>> m.coins == {2: 1, 8: 1}
    True
    >>> m.change(1) # return the penny passed in (it's the smallest)
```

```
    [1]
    >>> m.change(9) # return the 9 passed in (no change possible)
    [9]
    >>> m.coins == {2: 1, 8: 1}
    True
    >>> m.change(10)
    [2, 8]
    >>> m.coins == {10: 1}
    True

    >>> m = ChangeMachine(9)
    >>> [m.change(k) for k in [2, 2, 3]]
    [[1, 1], [1, 1], [1, 1, 1]]
    >>> m.coins == {1: 2, 2: 2, 3: 1}
    True
    >>> m.change(5) # Prefers [1, 1, 3] to [1, 2, 2] (more pennies)
    [1, 1, 3]
    >>> m.change(7)
    [2, 5]
    >>> m.coins == {2: 1, 7: 1}
    True
    """
    def __init__(self, pennies):
        self.coins = {1: pennies}

    def change(self, coin):
        """Return change for coin, removing the result from self.coins."""
        "*** YOUR CODE HERE ***"
```

> Hint: Call the `make_change` function in order to compute the result of `change`, but update `self.coins` before returning that result.

Definitely try to solve this without reading the walkthrough, but if you're really stuck then read the walkthrough.

Walkthrough

The code for `change(self, coin)` should do the following:

1. Add the `coin` to the machine. This way, you can just make change and you'll always get some result, although it might just be that coin back. The simplest way is to `get` the count of that `coin` (defaulting to 0) and add 1 to it: `self.coins[coin] = 1 + self.coins.get(coin, 0)`
2. Call `make_change(coin, self.coins)` and assign the result to a name (such as `result`). You'll return this at the end.
3. Before returning, reduce the count of each coin you are returning. One way is to repeatedly call `remove_one(self.coins, c)` for each coin `c` in the result of calling `make_change` in Step 2.

4. Return the result of calling `make_change` in Step 2.

Use Ok to test your code:

```
python3 ok -q ChangeMachine                                    ✂
```

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

Submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 (https://cs61a.org/lab/lab00/#submit-with-gradescope) has detailed instructions.

In addition, all students who are **not** in the mega lab must complete this attendance form (https://go.cs61a.org/lab-att). Submit this form each week, whether you attend lab or missed it for a good reason. The attendance form is not required for mega section students.