Homework 05：Nonlocal, Generators, and OOP

1. Instructions

> Please download homework materials `hw05.zip` from our QQ group if you don't have one.

In this homework, you are required to complete the problems described in section 3. The starter code for these problems is provided in `hw05.py`, which is distributed as part of the homework materials in the `code` directory.

Submission: When you are done, submit your code to our OJ website as instructed in lab00. You may submit more than once before the deadline; only the final submission will be scored. See lab00 for more instructions on submitting assignments.

Readings: You might find the following references to the textbook useful:

- Section 2.4
- Section 2.5

2. Review

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the next section and refer back here should you get stuck.

2.1 Object-Oriented Programming

Object-oriented programming (OOP) is a style of programming that allows you to think of code in terms of "objects". Here's an example of a `Car` class:

```
class Car(object):
    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**: a blueprint for how to build a certain type of object. The `Car` class (shown above) describes the behavior and data that all `Car` objects have.
- **instance**: a particular occurrence of a class. In Python, we create instances of a class like this:

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

  `my_car` is an instance of the `Car` class.
- **attribute** or **field**: a variable that belongs to the class. Think of an attribute as a quality of the object: cars have *wheels* and *color*, so we have given our `Car` class `self.wheels` and `self.color` attributes. We can access attributes using **dot notation**:

  ```
  >>> my_car.color
  'red'
  >>> my_car.wheels
  4
  ```

- **method**: Methods are just like normal functions, except that they are tied to an instance or a class. Think of a method as a "verb" of the class: cars can *drive* and also *pop* their tires, so we have given our `Car` class the methods `drive` and `pop_tire`. We call methods using **dot notation**:

  ```
  >>> my_car = Car('red')
  >>> my_car.drive()
  'red car goes vroom!'
  ```

- **constructor**: As with data abstraction, constructors describe how to build an instance of the class. Most classes have a constructor. In Python, the constructor of the class defined as `__init__`. For example, here is the `Car` class's constructor:

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

  The constructor takes in one argument, `color`. As you can see, the constructor also creates the `self.wheels` and `self.color` attributes.
- **self**: in Python, `self` is the first parameter for many methods (in this class, we will only use methods whose first parameter is `self`). When a method is called, `self` is bound to an instance of the class. For example:

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

Notice that 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 `my_car` as `self` for us.

3. Required Problems

Nonlocal

Q1: Password Protected Account (200pts)

In lecture, we saw how to use functions to create mutable objects. Here, for example, is the function `make_withdraw` which produces a function that can withdraw money from an account:

```
def make_withdraw(balance):
    """Return a withdraw function with BALANCE as its starting balance.
    >>> withdraw = make_withdraw(1000)
    >>> withdraw(100)
    900
    >>> withdraw(100)
    800
    >>> withdraw(900)
    'Insufficient funds'
    """
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
           return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Write a version of the `make_withdraw` function that returns password-protected withdraw functions. That is, `make_withdraw` should take a password argument (a string) in addition to an initial balance. The returned function should take two arguments: an amount to withdraw and a password.

A password-protected `withdraw` function should only process withdrawals that include a password that matches the original. Upon receiving an incorrect password, the function should:

1. Store that incorrect password in a list, and
2. Return the string 'Incorrect password'.

If a withdraw function has been called three times with incorrect passwords `<p1>`, `<p2>`, and `<p3>`, then it is locked. All subsequent calls to the function should return:

```
"Your account is locked. Attempts: [<p1>, <p2>, <p3>]"
```

The incorrect passwords may be the same or different:

```
def make_withdraw(balance, password):
    """Return a password-protected withdraw function.

    >>> w = make_withdraw(100, 'hax0r')
    >>> w(25, 'hax0r')
    75
    >>> error = w(90, 'hax0r')
    >>> error
    'Insufficient funds'
    >>> error = w(25, 'hwat')
    >>> error
    'Incorrect password'
    >>> new_bal = w(25, 'hax0r')
    >>> new_bal
    50
    >>> w(75, 'a')
    'Incorrect password'
    >>> w(10, 'hax0r')
    40
    >>> w(20, 'n00b')
    'Incorrect password'
    >>> w(10, 'hax0r')
    "Your account is locked. Attempts: ['hwat', 'a', 'n00b']"
    >>> w(10, 'l33t')
    "Your account is locked. Attempts: ['hwat', 'a', 'n00b']"
    >>> type(w(10, 'l33t')) == str
    True
    """
    "*** YOUR CODE HERE ***"
```

Remember to use doctest to test your code:

```
$ python -m doctest hw05.py
```

Q2: Joint Account (100pts)

Suppose that our banking system requires the ability to make joint accounts. Define a function `make_joint` that takes three arguments.

1. A password-protected `withdraw` function,
2. The password with which that `withdraw` function was defined, and
3. A new password that can also access the original account.

If the password is incorrect or cannot be verified because the underlying account is locked, the `make_joint` should propagate the error. Otherwise, it returns a `withdraw` function that provides additional access to the original account using *either* the new or old password. Both functions draw from the same balance. Incorrect passwords provided to either function will be stored and cause the functions to be locked after three wrong attempts.

Hint: The solution is short (less than 10 lines) and contains no string literals! The key is to call withdraw with the right password and amount, then interpret the result. You may assume that all failed attempts to withdraw will return some string (for incorrect passwords, locked accounts, or insufficient funds), while successful withdrawals will return a number.

Use `type(value) == str` to test if some value is a string:

```
def make_joint(withdraw, old_pass, new_pass):
    """Return a password-protected withdraw function that has joint access to
    the balance of withdraw.

    >>> w = make_withdraw(100, 'hax0r')
    >>> w(25, 'hax0r')
    75
    >>> make_joint(w, 'my', 'secret')
    'Incorrect password'
    >>> j = make_joint(w, 'hax0r', 'secret')
    >>> w(25, 'secret')
    'Incorrect password'
    >>> j(25, 'secret')
    50
    >>> j(25, 'hax0r')
    25
    >>> j(100, 'secret')
    'Insufficient funds'

    >>> j2 = make_joint(j, 'secret', 'code')
    >>> j2(5, 'code')
    20
    >>> j2(5, 'secret')
    15
    >>> j2(5, 'hax0r')
    10

    >>> j2(25, 'password')
    'Incorrect password'
    >>> j2(5, 'secret')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    >>> j(5, 'secret')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    >>> w(5, 'hax0r')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    >>> make_joint(w, 'hax0r', 'hello')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    """
    "*** YOUR CODE HERE ***"
```

Generator

Q3: Generate Permutations (300pts)

Given a sequence of unique elements, a *permutation* of the sequence is a list containing the elements of the sequence in some arbitrary order. For example, `[2, 1, 3]`, `[1, 3, 2]`, and `[3, 2, 1]` are some of the permutations of the sequence `[1, 2, 3]`.

Implement `permutations`, a generator function that takes in a sequence `seq` and returns a generator that yields all permutations of `seq`.

Permutations may be yielded in any order. Note that the doctests test whether you are yielding all possible permutations, but not in any particular order. The built-in `sorted` function takes in an iterable object and returns a list containing the elements of the iterable in non-decreasing order.

> Hint: If you had the permutations of all the elements in `seq` not including the first element, how could you use that to generate the permutations of the full seq?
>
> Hint: There is no need to use some advanced non-recursive algorithms, which you may find on the Internet. Combining `yield` and recursion.

```
def permutations(seq):
    """Generates all permutations of the given sequence. Each permutation is a
    list of the elements in SEQ in a different order. The permutations may be
    yielded in any order.

    >>> perms = permutations([100])
    >>> type(perms)
    <class 'generator'>
    >>> next(perms)
    [100]
    >>> try: #this piece of code prints "No more permutations!" if calling next
would cause an error
    ...     next(perms)
    ... except StopIteration:
    ...     print('No more permutations!')
    No more permutations!
    >>> sorted(permutations([1, 2, 3])) # Returns a sorted list containing
elements of the generator
    [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
    >>> sorted(permutations((10, 20, 30)))
    [[10, 20, 30], [10, 30, 20], [20, 10, 30], [20, 30, 10], [30, 10, 20], [30,
20, 10]]
    >>> sorted(permutations("ab"))
    [['a', 'b'], ['b', 'a']]
    """
    "*** YOUR CODE HERE ***"
```

OOP

Q4: Vending Machine (400pts)

Create a class called `VendingMachine` that represents a vending machine for some product. A `VendingMachine` object returns strings describing its interactions.
Fill in the `VendingMachine` class, adding attributes and methods as appropriate, such that its behavior matches the following doctests:

```
class VendingMachine:
    """A vending machine that vends some product for some price.

    >>> v = VendingMachine('candy', 10)
    >>> v.vend()
    'Machine is out of stock.'
    >>> v.add_funds(15)
    'Machine is out of stock. Here is your $15.'
    >>> v.restock(2)
    'Current candy stock: 2'
    >>> v.vend()
    'You must add $10 more funds.'
    >>> v.add_funds(7)
    'Current balance: $7'
    >>> v.vend()
    'You must add $3 more funds.'
    >>> v.add_funds(5)
    'Current balance: $12'
    >>> v.vend()
    'Here is your candy and $2 change.'
    >>> v.add_funds(10)
    'Current balance: $10'
    >>> v.vend()
    'Here is your candy.'
    >>> v.add_funds(15)
    'Machine is out of stock. Here is your $15.'

    >>> w = VendingMachine('soda', 2)
    >>> w.restock(3)
    'Current soda stock: 3'
    >>> w.restock(3)
    'Current soda stock: 6'
    >>> w.add_funds(2)
    'Current balance: $2'
    >>> w.vend()
    'Here is your soda.'
    """
    "*** YOUR CODE HERE ***"
```

> You may find Python string formatting syntax useful. A quick example:
>
> ```
> >>> ten, twenty, thirty = 10, 'twenty', [30]
> >>> '{0} plus {1} is {2}'.format(ten, twenty, thirty)
> '10 plus twenty is [30]'
> ```