Homework 4: Nonlocal, Iterators

> Adapted from cs61a of UC Berkeley.

Readings

You might find the following references useful:

- Section 2.4
- Section 4.2

Starter Files

Get your starter file by cloning the repository: https://github.com/JacyCui/sicp-hw04.git

```
git clone https://github.com/JacyCui/sicp-hw04.git
```

`hw04.zip` is the starter file you need, you might need to unzip the file to get the skeleton code.

```
unzip hw04.zip
```

`README.md` is the handout for this homework. `solution` is a probrab solution of the homework. However, I might not give my solution exactly when the homework is posted. You need to finish the task on your own first. If any problems occurs, please make use of the comment section.

Required Questions

Nonlocal

Q1: Make Bank

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 new function `make_bank`, which should create a bank account with value `balance` and should also return another function. This new function should be able to withdraw and deposit money. The second function will take in two arguments: `message` and `amount`. When the message passed in is `'deposit'`, the bank will deposit `amount` into the account. When the message passed in is `'withdraw'`, the bank will attempt to withdraw `amount` from the account. If the account does not have enough money for a withdrawal, the string `'Insufficient funds'` will be returned. If the `message` passed in is neither of the two commands, the function should return `'Invalid message'` Examples are shown in the doctests.

```
def make_bank(balance):
    """Returns a bank function with a starting balance. Supports
    withdrawals and deposits.

    >>> bank = make_bank(100)
    >>> bank('withdraw', 40)    # 100 - 40
    60
    >>> bank('hello', 500)      # Invalid message passed in
    'Invalid message'
    >>> bank('deposit', 20)     # 60 + 20
    80
    >>> bank('withdraw', 90)    # 80 - 90; not enough money
    'Insufficient funds'
    >>> bank('deposit', 100)    # 80 + 100
    180
    >>> bank('goodbye', 0)      # Invalid message passed in
    'Invalid message'
    >>> bank('withdraw', 60)    # 180 - 60
    120
    """
    def bank(message, amount):
        "*** YOUR CODE HERE ***"
    return bank
```

Use Ok to test your code:

```
python3 ok -q make_bank --local
```

Q2: Password Protected Account

Write a version of the `make_withdraw` function shown in the previous question 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 frozen. All subsequent calls to the function should return:

```
"Frozen account. Attempts: [<p1>, <p2>, <p3>]"
```

> *Hint:* You can use the `str` function to turn a list into a string. For example, for a list `s = [1, 2, 3]`, the expression `"The list s is: " + str(s)` simplifies to `"The list s is: [1, 2, 3]"`.

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')
    "Frozen account. Attempts: ['hwat', 'a', 'n00b']"
    >>> w(10, 'l33t')
    "Frozen account. Attempts: ['hwat', 'a', 'n00b']"
    >>> type(w(10, 'l33t')) == str
```

```
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_withdraw --local
```

Iterators and Generators

Q3: Repeated

Implement a function (not a generator function) that returns the first value in the iterator `t` that appears `k` times in a row. As described in lecture, iterators can provide values using either the `next(t)` function or with a for-loop. Do not worry about cases where the function reaches the end of the iterator without finding a suitable value, all lists passed in for the tests will have a value that should be returned. If you are receiving an error where the iterator has completed then the program is not identifying the correct value. Iterate through the items such that if the same iterator is passed into `repeated` twice, it continues in the second call at the point it left off in the first. An example of this behavior is shown in the doctests.

```
def repeated(t, k):
    """Return the first value in iterator T that appears K times in a row. Iterate through the items such that
    if the same iterator is passed into repeated twice, it continues in the second call at the point it left off
    in the first.

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s, 2)
    9
    >>> s2 = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s2, 3)
    8
    >>> s = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> repeated(s, 3)
    2
    >>> repeated(s, 3)
    5
    >>> s2 = iter([4, 1, 6, 6, 7, 7, 8, 8, 2, 2, 2, 5])
    >>> repeated(s2, 3)
    2
    """
    assert k > 1
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q repeated --local
```

Q4: Generate Permutations

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:* If you're having trouble getting started, see the hints video for this question for tips on how to approach this question.

```
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 ***"
```

Use Ok to test your code:

```
python3 ok -q permutations --local
```

Extra Questions

Q5: Joint Account

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')
    "Frozen account. Attempts: ['my', 'secret', 'password']"
    >>> j(5, 'secret')
    "Frozen account. Attempts: ['my', 'secret', 'password']"
    >>> w(5, 'hax0r')
    "Frozen account. Attempts: ['my', 'secret', 'password']"
    >>> make_joint(w, 'hax0r', 'hello')
    "Frozen account. Attempts: ['my', 'secret', 'password']"
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_joint --local
```

Q6: Remainder Generator

Like functions, generators can also be *higher-order*. For this problem, we will be writing `remainders_generator`, which yields a series of generator objects.

`remainders_generator` takes in an integer `m`, and yields `m` different generators. The first generator is a generator of multiples of `m`, i.e. numbers where the remainder is 0. The second is a generator of natural numbers with remainder 1 when divided by `m`. The last generator yields natural numbers with remainder `m - 1` when divided by `m`.

> *Hint*: You can call the `naturals` function to create a generator of infinite natural numbers.

> *Hint*: Consider defining an inner generator function. Each yielded generator varies only in that the elements of each generator have a particular remainder when divided by `m`. What does that tell you about the argument(s) that the inner function should take in?

```
def remainders_generator(m):
    """
    Yields m generators. The ith yielded generator yields natural numbers whose
    remainder is i when divided by m.

    >>> import types
```

```
    >>> [isinstance(gen, types.GeneratorType) for gen in remainders_generator(5)]
    [True, True, True, True, True]
    >>> remainders_four = remainders_generator(4)
    >>> for i in range(4):
    ...     print("First 3 natural numbers with remainder {0} when divided by 4:".format(i))
    ...     gen = next(remainders_four)
    ...     for _ in range(3):
    ...         print(next(gen))
    First 3 natural numbers with remainder 0 when divided by 4:
    4
    8
    12
    First 3 natural numbers with remainder 1 when divided by 4:
    1
    5
    9
    First 3 natural numbers with remainder 2 when divided by 4:
    2
    6
    10
    First 3 natural numbers with remainder 3 when divided by 4:
    3
    7
    11
    """
    "*** YOUR CODE HERE ***"
```

Note that if you have implemented this correctly, each of the generators yielded by `remainder_generator` will be *infinite* - you can keep calling `next` on them forever without running into a `StopIteration` exception.

Use Ok to test your code:

```
python3 ok -q remainders_generator --local
```

Finally, you can run doctest to check your answer again.

```
python3 -m doctest hw03.py
```