Lab 6: Nonlocal, Mutability

> Adapted from cs61a of UC Berkeley.

Starter Files

Get your starter file by cloning the repository: https://github.com/JacyCui/sicp-lab06.git

```
git clone https://github.com/JacyCui/sicp-lab06.git
```

`lab06.zip` is the starter file you need, you might need to unzip the file to get the skeleton code.

```
unzip lab06.zip
```

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

Topics

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

Nonlocal

We say that a variable defined in a frame is *local* to that frame. A variable is **nonlocal** to a frame if it is defined in the environment that the frame belongs to but not the frame itself, i.e. in its parent or ancestor frame.

So far, we know that we can access variables in parent frames:

```python
def make_adder(x):
    """ Returns a one-argument function that returns the result of
    adding x and its argument. """
    def adder(y):
        return x + y
    return adder
```

Here, when we call `make_adder`, we create a function `adder` that is able to look up the name `x` in `make_adder`'s frame and use its value.

However, we haven't been able to *modify* variables defined in parent frames. Consider the following function:

```
def make_withdraw(balance):
    """Returns a function which can withdraw
    some amount from balance

    >>> withdraw = make_withdraw(50)
    >>> withdraw(25)
    25
    >>> withdraw(25)
    0
    """
    def withdraw(amount):
        if amount > balance:
            return "Insufficient funds"
        balance = balance - amount
        return balance
    return withdraw
```

The inner function `withdraw` attempts to update the variable `balance` in its parent frame. Running this function's doctests, we find that it causes the following error:

```
UnboundLocalError: local variable 'balance' referenced before assignment
```

Why does this happen? When we execute an assignment statement, remember that we are either creating a new binding in our current frame or we are updating an old one in the current frame. For example, the line `balance = ...` in `withdraw`, is creating the local variable `balance` inside `withdraw`'s frame. This assignment statement tells Python to expect a variable called `balance` inside `withdraw`'s frame, so Python will not look in parent frames for this variable. However, notice that we tried to compute `balance - amount` before the local variable was created! That's why we get the `UnboundLocalError`.

To avoid this problem, we introduce the `nonlocal` keyword. It allows us to update a variable in a parent frame!

> Some important things to keep in mind when using `nonlocal`
>
> - `nonlocal` cannot be used with global variables (names defined in the global frame).
> - If no nonlocal variable is found with the given name, a `SyntaxError` is raised.
> - A name that is already local to a frame cannot be declared as nonlocal.

Consider this improved example:

```
def make_withdraw(balance):
    """Returns a function which can withdraw
    some amount from balance

    >>> withdraw = make_withdraw(50)
    >>> withdraw(25)
```

```
    25
    >>> withdraw(25)
    0
    """
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return "Insufficient funds"
        balance = balance - amount
        return balance
    return withdraw
```

The line `nonlocal balance` tells Python that `balance` will not be local to this frame, so it will look for it in parent frames. Now we can update `balance` without running into problems.

Mutability

We say that an object is **mutable** if its state can change as code is executed. The process of changing an object's state is called **mutation**. Examples of mutable objects include lists and dictionaries. Examples of objects that are *not* mutable include tuples and functions.

We have seen how to use the `==` operator to check if two expressions evaluate to *equal* values. We now introduce a new comparison operator, `is`, that checks whether two expressions evaluate to the *same* values.

Wait, what's the difference? For primitive values, there is none:

```
>>> 2 + 2 == 3 + 1
True
>>> 2 + 2 is 3 + 1
True
```

This is because all primitives have the same *identity* under the hood. However, with non-primitive values, such as lists, each object has its own identity. That means you can construct two objects that may look exactly the same but have different identities.

```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = [1, 2, 3, 4]
>>> lst1 == lst2
True
>>> lst1 is lst2
False
```

Here, although the lists referred to by `lst1` and `lst2` have *equal* contents, they are not the *same* object. In other words, they are the same in terms of equality, but not in terms of identity.

This is important in our discussion of mutability because when we mutate an object, we simply change its state, *not* its identity.

```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = lst1
>>> lst1.append(5)
>>> lst2
[1, 2, 3, 4, 5]
>>> lst1 is lst2
True
```

Required Questions

Nonlocal Codewriting

Q1: Make Adder Increasing

Write a function which takes in an integer `a` and returns a one-argument function. This function should take in some value `b` and return `a + b` the first time it is called, similar to `make_adder`. The second time it is called, however, it should return `a + b + 1`, then `a + b + 2` the third time, and so on.

```
def make_adder_inc(a):
    """
    >>> adder1 = make_adder_inc(5)
    >>> adder2 = make_adder_inc(6)
    >>> adder1(2)
    7
    >>> adder1(2) # 5 + 2 + 1
    8
    >>> adder1(10) # 5 + 10 + 2
    17
    >>> [adder1(x) for x in [1, 2, 3]]
    [9, 11, 13]
    >>> adder2(5)
    11
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_adder_inc --local
```

Q2: Next Fibonacci

Write a function `make_fib` that returns a function that returns the next Fibonacci number each time it is called. (The Fibonacci sequence begins with 0 and then 1, after which each element is the sum of the preceding two.) Use a `nonlocal` statement! In addition, do not use python lists to solve this problem.

```
def make_fib():
    """Returns a function that returns the next Fibonacci number
    every time it is called.

    >>> fib = make_fib()
    >>> fib()
    0
    >>> fib()
    1
    >>> fib()
    1
    >>> fib()
    2
    >>> fib()
    3
    >>> fib2 = make_fib()
    >>> fib() + sum([fib2() for _ in range(5)])
    12
    >>> from construct_check import check
    >>> # Do not use lists in your implementation
    >>> check(this_file, 'make_fib', ['List'])
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_fib --local
```

Mutability

Q3: List-Mutation

> Test your understaninding of list mutation with the following questions. What would Python display? Type it in the intepreter if you're stuck!
>
> ```
> python3 ok -q list-mutation -u --local
> ```
>
> Note: if nothing would be output by Python, type `Nothing`.

```
>>> lst = [5, 6, 7, 8]
>>> lst.append(6)
______

>>> lst
______

>>> lst.insert(0, 9)
>>> lst
______

>>> x = lst.pop(2)
>>> lst
______

>>> lst.remove(x)
>>> lst
______

>>> a, b = lst, lst[:]
>>> a is lst
______

>>> b == lst
______

>>> b is lst
______
```

Q4: Insert Items

Write a function which takes in a list `lst`, an argument `entry`, and another argument `elem`. This function will check through each item present in `lst` to see if it is equivalent with `entry`. Upon finding an equivalent entry, the function should modify the list by placing `elem` into the list right after the found entry. At the end of the function, the modified list should be returned. See the doctests for examples on how this function is utilized. Use list mutation to modify the original list, no new lists should be created or returned.

Be careful in situations where the values passed into `entry` and `elem` are equivalent, so as not to create an infinitely long list while iterating through it. If you find that your code is taking more than a few seconds to run, it is most likely that the function is in a loop of inserting new values.

```
def insert_items(lst, entry, elem):
    """
    >>> test_lst = [1, 5, 8, 5, 2, 3]
    >>> new_lst = insert_items(test_lst, 5, 7)
    >>> new_lst
    [1, 5, 7, 8, 5, 7, 2, 3]
```

```
    >>> large_lst = [1, 4, 8]
    >>> large_lst2 = insert_items(large_lst, 4, 4)
    >>> large_lst2
    [1, 4, 4, 8]
    >>> large_lst3 = insert_items(large_lst2, 4, 6)
    >>> large_lst3
    [1, 4, 6, 4, 6, 8]
    >>> large_lst3 is large_lst
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q insert_items --local
```

Congratulations! You've finished all problems of the lab. Feel free to run doctest to verify your answer again.

```
python3 -m doctest lab06.py
```