CS61A Notes – Week 6: Nonlocal Assignment, Local State, and More Environments!!

This week, we're going to focus on functions that keep local state, and examine the interesting consequences. But first, let's review what functions are.

Review: Functions

Definitions:

- Pure Function -- a function that, when called, produces no effects other than returning a value.
- Non-pure Function -- a function that, when called, produces some side-effect, such as changing the environment, generating output to a screen, etc.

During the first five weeks of this course, the functions that you have written have been (for the most part) pure functions. For instance, the `sum` procedure (from discussion) is a pure function:

```
def sum(tuple):
   result = 0
   for elem in tuple:
       result += elem
   return result
```

An interesting property of pure functions is that they are referentially transparent:

> Referentially Transparent -- an expression is referentially transparent if it can be replaced with its value, without any change in program behavior.

For instance, the expression

```
add(sum((1, 2, 3)), square(4))
```

is exactly equivalent to replacing `sum((1, 2, 3))` with its value, `6`:

```
add(6, square(4))
```

Up until now, we haven't (officially) described how to write non-pure functions (other than using `print` or `random`) - so, let's look at a Python keyword: `nonlocal`.

The `nonlocal` statement

Say we are writing a function make_`delayed_repeater` that returns a function that returns the last thing it received (the first time it's called, it returns '`...`'), like:

```
>>> goo = make_delayed_repeater()
>>> goo('hi there')
...
>>> goo('i like chocolate milk')
hi there
>>> goo('stop repeating what i say')
i like chocolate milk
```

Our first attempt might look something like:

```
>>> def make_delayed_repeater():
...     my_phrase = '...'
...     def repeater(phrase):
...         to_return = my_phrase
...         my_phrase = phrase
...         return to_return
...     return repeater

>>> goo = make_delayed_repeater()
>>> goo('hi there')
UnboundLocalError: local variable 'my_phrase' referenced before assignment
```

Wait, what? That's an extremely cryptic error message.

Here is what is happening – when Python executes the repeater procedure, it notices the line

```
…
my_phrase = phrase
return to_return
```

So, Python thinks that we're creating a new local variable called `my_phrase` in the current frame. However, in the previous line:

```
to_return = my_phrase
my_phrase = phrase
return to_return
```

A reference is made to `my_phrase`. Even though we (the programmer) know that we meant to refer to the outer `my_phrase` variable, Python insists that we are trying to refer to a variable before it is assigned, like:

```
def foo():
    x = y + 4          # y hasn't been defined yet!
    y = 7
    return x + y
```

When Python sees an assignment statement, it does the following:

- First, check to see if the variable name currently exists in the current frame.
  - If it does, then do re-bind the variable to the new value.
  - Otherwise, create a new variable in this frame, and set it to the given value.

Notice that, unlike variable lookups, Python won't normally follow the frame 'parent pointers' for assignments. Luckily, we can tell Python to behave differently, by using the `nonlocal` keyword:

```
>>> def make_delayed_repeater():
...     my_phrase = '...'
...     def repeater(phrase):
...         nonlocal my_phrase
...         to_return = my_phrase
...         my_phrase = phrase
...         return to_return
...     return repeater
```

```
>>> goo = make_delayed_repeater()
>>> goo('hi there')
...
>>> goo('i like chocolate milk')
hi there
>>> goo('stop repeating what i say')
i like chocolate milk
```

Success!

The `nonlocal` statement tells Python that the listed variable is in some parent scope, and that an assignment to a `nonlocal` variable will re-bind the variable's value (instead of creating a new variable in the current scope). The only tricky case is that `nonlocal` will stop before the global frame - in other words, it will not find variables in the global frame.

For shorthand, you can list multiple nonlocal variables by separating each name with a comma, like:

```
nonlocal foo, bar, garply
```

Note: The `nonlocal` keyword was first introduced in Python **3.0**, and is not available for Python **2.x**!

Rules for `nonlocal`:

1.) The `nonlocal` variable must exist in a parent frame (that isn't the global environment). If a declared nonlocal variable doesn't exist in some parent frame, then Python will signal an error.
2.) Once a variable is declared `nonlocal`, any attempt to modify that variable will trace back through frames until the variable is found, and then that found variable will be changed.
3.) A variable declared `nonlocal` can't already exist in the current frame.

Examples:

```
>>> def f():
...     nonlocal x
...     return 42
...
SyntaxError: no binding for nonlocal 'x' found

>>> foo = 53
>>> def g():
...     nonlocal foo
...     foo = 42
SyntaxError: no binding for nonlocal 'foo' found

>>> def f():
...     foo = 2
...     def g(x):
...         nonlocal x
...
SyntaxError: name 'x' is parameter and nonlocal
```

Functions with Local State

If we look back at the `make_delayed_repeater` function, there's something pretty awesome going on - the function is keeping track of something (in this case, the last phrase it was passed).
This is totally new - in the past, your functions never 'remembered' anything. They would return the same value for the same arguments every time.

Thus, at this point we're diverging from functional programming. We can now start to view functions as *objects* that can change over time.

If we return to the `withdraw` example from lecture:

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance             # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount   # Re-bind the existing balance name
        return balance
    return withdraw
```

Each invocation of `make_withdraw` creates a withdraw object that remembers its own balance. So, if I create two different `make_withdraw` instances, `withdraw1` and `withdraw2`, then `withdraw1` and `withdraw2` each will have separate balances:

```
>>> withdraw1 = make_withdraw(0)
>>> withdraw2 = make_withdraw(42)
>>> withdraw1(10)
Insufficient funds
>>> withdraw2(10)
32
```

This is a precursor to a programming paradigm called Object-Oriented Programming (OOP), a popular style of programming that's aimed at making programs easier to reason about.

QUESTIONS

1.) What Would Python Print (WWPP)
For the following exercises, write down what Python would print. If an error occurs, just write 'Error', and briefly describe the error.

a.)

```
>>> name = 'rose'
>>> def my_func():
...     name = 'martha'
...     return None
>>> my_func()
>>> name
______ ?
```

b.)

```
>>> name = 'ash'
>>> def abra(age):
...     def kadabra(name):
...         def alakazam(level):
...             nonlocal name
...             name = 'misty'
...             return name
...         return alakazam
...     return kadabra
>>> abra(12)('sleepy')(15)
______ ?
>>> name
______ ?
```

c.)

```
>>> ultimate_answer = 42
>>> def ultimate_machine():
...     nonlocal ultimate_answer
...     ultimate_answer = 'nope!'
...     return ultimate_answer
>>> ultimate_machine()
______ ?
>>> ultimate_answer
______ ?
```

d.)

```
>>> def f(t=0):
...     def g(t=0):
...         def h():
...             nonlocal t
...             t = t + 1
...         return h, lambda: t
...     h, gt = g()
...     return h, gt, lambda: t

>>> h, gt, ft = f()
>>> ft(), gt()
______ ?
>>> h()
>>> ft(), gt()
______ ?
```

More Environment Diagrams!

1.) Draw the environment diagram for each of the following, and write return values where prompted:

a.)

```
x = 3

def boring(x):
    def why(y):
        x = y
    why(5)
    return x

def interesting(x):
    def because(y):
        nonlocal x
        x = y
    because(5)
    return x

interesting(3)
_________

boring(3)
_________
```

b.)

```
def make_person(name):
    def dispatch(msg):
        nonlocal name
        if msg == 'name':
            return name
        elif msg == 'tom-ify':
            name = 'tom'
        else:
            print("wat")
    return dispatch

>>> phillip = make_person('phillip')
>>> phillip('tom-ify')
>>> phillip('name')
```

2.) Recall that, earlier in the semester, we represented numerical sequences as a function of one argument (taking in the index). If we want to process the elements of such a sequence, it'd be nice to have a function that 'remembers' where in the sequence it is in.

For instance, consider the `evens` function:

```
def evens(n):
    """ Return the n-th even number. """
    return 2 * n
```

I'd like to have a function `generate_evens` that spits out the even numbers, one by one:

```
>>> generate_evens = make_seq_generator(evens)
>>> for i in range(4):
...     print(generate_evens())
0
2
4
6
```

Write a function `make_seq_generator` that, given a function `fn`, returns a new function that returns the elements of the sequence one by one (like in the above example):

```
def make_seq_generator(seq_fn):
```

3.) Let's implement counters, in the dispatch-procedure style!

a.) Write a procedure `make_counter` that returns a function that behaves in the following way:

```
>>> counter1 = make_counter(4)
>>> counter2 = make_counter(42)
>>> counter1('count')
5
>>> counter1('count')
6
>>> counter2('count')
43
>>> counter2('reset')
0
>>> counter1('count')
7
```

To help jog your memory, here's the skeleton of `make_counter`:

```
def make_counter(start_val):
    def dispatch(msg):
        if msg == ...
        ...
    return dispatch
```

b.) Modify your answer to (a) to include support for a new message, `'clone'`, that returns a copy of the current counter. The clone should be independent of the original:

```
>>> counter = make_counter(3)
>>> counter('count')
4
>>> clone = counter('clone')
>>> clone('count')
5
>>> counter('reset')
0
>>> clone('count')
6
```