NONLOCAL ASSIGNMENT, LOCAL STATE 10

COMPUTER SCIENCE 61A

July 19, 2012

1 Looking Back

Before we move into new syntax, let's review a familar topic: functions. We've previously introduced two categories of functions:

1. *Pure*: a function that, when called, produces no effects other than returning a value
2. *Non-Pure*: a function that, when called, produces some side-effect, such as printing to the screen.

During the first few weeks of this course, the functions you have written have been, for the most part, pure functions. For example, the `sum` procedure is a pure function:

```
def sum(sequence):
    total = 0
    for elem in sequence:
        total += elem
    return total
```

The only non-pure function we've defined dealt with `print`. Namely, anytime we saw the `print` statement, we immediately knew that the function was non-pure.

Let's define another attribute of pure functions: *referentially transparent*. An expression is *referentially transparent* if it can be replaced with its value, without any change in program behavior. So for example,

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

is exactly equivalent to:

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

where we replaced `square((1,2,3))` with 6. And since there is no change in program behavior (we still would get 22), `sum` is considered referentially transparent.

Earlier, we were vague when we said that pure functions had no "side-effects", but now we can more precisely state the properties of a pure function. A pure function is referentially transparent, and cannot rely on state – that is, given the same arguments, a pure function will always return the same output. In addition, pure functions cannot change state or produce observable side effects (such as printing to the screen).

We've already seen mutable data structures – structures that can change their state as a program runs. Let's now look at a different way to mutate state: the `nonlocal` keyword.

2 Nonlocal Assignment

Say I wanted to define a `make_counter` function that, given a number `k`, returns a function that repeatedly increments (and returns) `k`:

```
>>> c1 = make_counter(0)
>>> c1()
0
>>> c1()
1
```

In addition, I'd like to be able to have separate counters, each keeping track of their own state:

```
>>> c1()
2
>>> c2 = make_counter(42)
>>> c2()
42
>>> c3()
43
>>> c1()
3
```

Our first attempt could look something like this:

```
def make_counter(k):
    def counter():
        toreturn = k
        k = k + 1
        return toreturn
    return counter

>>> counter = make_counter(61)
>>> counter()
UnboundLocalError: local variable 'k' referenced before assignment
```

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

Here is what's happening – when Python executes the `counter` procedure, it sees the line

```
...
k = k + 1
return toreturn
...
```

Python does the same thing every time it sees assignment – it will create a new variable in the current frame to store whatever is on the right side. So Python will try to create a `k` variable in the frame. But it realizes something is amiss, because in the previous line, we said:

```
...
toreturn = k
k = k + 1
...
```

We attempted to access `k` in the previous line. Even though we meant to refer to the outer `k` variable, Python doesn't know that. In other words, the above error scenario is similar to:

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

When dealing with assignment, Python will only focus on the current frame (this is in contrast to how Python handles variable lookups). When Python sees an assignment statement:

1. If the variable exists in the current frame, then Python will update it.

2. Otherwise, it will create a new variable in the current frame and set it to the given value.

Unlike when we're looking up variables, Python won't normally follow the frame's parent pointers when updating a variable. Luckily, Python has a special keyword that forces Python to do this:

```
def make_counter(k):
    def counter():
        nonlocal k
        toreturn = k
        k = k + 1
        return k
    return counter
```

As Jon would say, "Awesome!"

The `nonlocal` statement tells Python that the listed variable is in some parent frame, and that assignment to a `nonlocal` variable will re-bind that variable's value (instead of creating a new variable in the current frame). `nonlocal` will follow frame after frame until it finds the first instance of the varible that needs re-binding.

There is one caveat: `nonlocal` will **not** go all the way back to the global frame. It will jump back frames until it hits the global frame, and then stop. Thus, you can not use `nonlocal` to modify a variable defined in the global frame while you're in a function's local frame[1].

2.1 Rules for `nonlocal`

1. The `nonlocal` variable must exist in the environment, but not in the global frame. The variable will be looked up in the parent frames all the way up to, but excluding, the global frame. If it cannot be found, then Python will throw an error.
2. Once a variable is declared `nonlocal` within a block, any attempt to modify that variable will trace back through frames until the variable is found, and then that variable's value will be changed.
3. A variable declared `nonlocal` cannot already exist in the current frame (either as a local variable or local parameter).

[1] We have another keyword for re-binding variables in the global frame: `global`. But we'll get to this soon.

3 A Word About State

If we look back at the `make_counter` 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 something we've talked about before. *State* is the idea that our programs can change or mutate, an idea central to the notion of Object Oriented Programming. We had objects with state, like Banks remembering the Accounts they controlled, or Pokémon remembering their names. As `make_delayed_repeater` has shown us, functions can have state too.

We can create persistent state for functions. The variables that represent the persistent state are considered "local" because only nested functions that reference the state can access them.

Let's practice a little bit with `nonlocal` and state:

1. **What Would Python Print?** For the following exercises, write down what Python would print. If an error occurs, just write 'Error', and briefly describe the error. Hint: Drawing the environment diagram might help.

   a)

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

   Solution:

```
rose
```

   b)

```
>>> def abra(age):
...     def get_name():
...         return name
...     name = 'ash'
...     def kadabra(name):
...         def alakazam(level):
...             nonlocal name
...             name = 'misty'
```

```
...             return name
...         return alakazam
...     return kadabra, get_name
>>> my_pokemon, get_name = abra(12)
>>> my_pokemon('sleepy')(15)
________ ?
>>> get_name()
________ ?
```

Solution:

```
>>> my_pokemon('sleepy')(15)
misty
>>> get_name()
ash
```

c)

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

Solution:

```
>>> ultimate_machine()
SyntaxError: no binding for nonlocal 'ultimate_answer' found.

(In fact, the program crashes right after you finish defining
ultimate_machine!)

>>> ultimate_answer
42
```

4 More Environment Diagram Practice

Draw the environment diagrams for the following questions, and use the diagrams to determine the responses of the interpreters:

1.
```
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)
________ ?
```

Solution:

```
>>> interesting(3)
5
>>> boring(3)
3
```

2.
```
def f(t):
    def g(t):
        def h():
            nonlocal t
            t = t + 1
        return h, lambda: t
    h, gt = g(0)
    return h, gt, lambda: t
```

```
>>> h, gt, ft = f(0)
>>> ft(), gt()
```

```
_________ ?
>>> h()
>>> ft(), gt()
_________ ?
```

Solution:

```
>>> h, gt, ft = f(0)
>>> ft(), gt()
0 0
>>> h()
>>> ft(), gt()
0 1
```

3. Define the `make_delayed_repeater` procedure that returns a function which always returns the previous argument it was called with. The first time you call the function, it should just return the string `'...'`.

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

def make_delayed_repeater():
```

Solution:

```
    last_phrase = '...'
    def repeater(thing):
        nonlocal last_phrase
        toreturn = last_phrase
        last_phrase = thing
        return toreturn
    return repeater
```

4. We can represent numerical sequences as a function of one argument (taking in the index). For example, here is the `nth_even` function that returns the nth even number:

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

We would like to have a function `generate_evens` that sequentially returns the even numbers, one by one:

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

Define the `make_seq_generator` function that, given a sequence function `fn`, returns a new function that returns the elements of the sequence one at a time (as in the above example).

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

Solution:

```
    cur_idx = 0
    def seq_generator():
        nonlocal cur_idx
        result = seq_fn(cur_idx)
        cur_idx += 1
        return result
    return seq_generator
```