# MORE MUTATION, NONLOCAL, AND MORE ENVIRONEMENTS 5

COMPUTER SCIENCE 61A

Oct. 1, 2012

## 1 More Mutation (is vs. ==)

Now that we have lists and data objects, it's important to talk about the difference between == (value equality) and is (object/reference equality). == checks if two things evaluate to the same value, while "is" checks if two variables literally point to the same object in an environment. Your TA will go over this more.

**Example:**

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
>>> x == y
True
>>> x is y
False
>>> z is x
True
>>> x is z
True
```

### 1.1 What would Python print? (it might help to draw the box/pointer diagram for this)

CS61A Fall 2012: John Denero, with
Akihiro Matsukawa, Hamilton Nguyen, Phillip Carpenter, Steven Tang, Varun Pai, Joy Jeng, Keegan Mann, Allen Nguyen, Stephen Martinis, Andrew Nguyen, Albert Wu, Julia Oh, Shu Zhong

```
>>> ls = [1, 2, 3, 4]
>>> list = [1, 2, 3, 4]
>>> ls == list

---------

>>> ls is list

---------

>>> x = [100, 101]
>>> ls[2] = x
>>> list[3] = ls[2]
>>> list[3] is x

---------
>>> x = list[1]
>>> ls[2] is x

---------

>>> x = ls
>>> ls[0] = x
>>> ls is ls[0]

--------

>>>  ls == ls[0]

---------
```

## 2 Nonlocal Assignment (Modeling State)

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 delayed_repeater that returns a function that returns the last thing it received (the first time it's called, it returns '...'), like:

```
>>> goo = delayed_repeater()
>>> goo('hi there')
…
>>> goo('i like chocolate milk')
hi there
>>> goo('stop repeating what i say')
i like chocolate milk
>>> goo('grr')
stop repeating what i say
```

Our first attempt might look something like:

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

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

Hey, what gives? It seems as if the my_phrase variable isn't being updated, despite the assignment statement to_return = my_phrase.

The reasoning behind this behavior is: 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 value.
Notice that, unlike variable lookups, Python won't follow the environment 'parent pointers' for assignment.
Luckily, we can tell Python to behave differently, by using the nonlocal keyword:

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

```
>>> goo = 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 a parent scope, and to assign to do a re-bind when assigning to it (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
```

**Rules for nonlocal:**

1.) The variable must exist in a parent scope (that isn't the global environment).
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.) Python signals an error upon the declaration of a nonlocal variable if it cannot be found.

Example:

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

**The variable can't exist in the global frame**

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

**The variable can't already exist in the current scope**
Example:
```
>>> def f():
...   foo = 2
...   def g(x):
...     nonlocal x
...
SyntaxError: name 'x' is parameter and nonlocal
```

## 2.1 What would Python print? (it might help to draw the environment diagram for this)

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.)

```
>>> 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()
______ ?
```

## 2.2 More Environments!!

Draw the environment diagram for the following, and write return values where prompted:

1.)

```
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)   ----------------------
```

2.)

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

>>> stephen = make_person('stephen')
>>> stephen('aki-ify')
>>> stephen('name')
```

## 3 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, it would return the same value for the same arguments every time.

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's balance separate:

```
>>> 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.

### 3.1 Nonlocal State Function with an Environment

1) Now let's wrap up by seeing how nonlocal programs and environment programs come together. Write a procedure make_counter that returns a dispatch procedure that behaves in the following way. Then also draw the environment diagram for this interaction. Do you see the connection? Try and see if you can fit the two together at the same time.

```
>>> counter1 = make_counter(0)

>>> counter1('inc')
>>> counter1('count')
1

>>> counter1(‘inc’)
>>> counter1(‘inc’)

>>> counter1(‘count’)

3

>>> counter2 = make_counter(42)
>>> counter1('inc')
>>> counter2('inc')
>>> counter2('count')
43
>>> counter1('reset')
>>> counter1('inc')
>>> counter2(‘count’)

43

>>> counter1(‘count’)

 1
```