CS61A Lecture 15

Wednesday, October 2nd, 2019

Announcements

- Cats project is due tomorrow! Early submission bonus point if submitted today.
- Homework 4 is due next Thursday. It is worth 4 points.
- Hog Composition revisions is due next Thursday as well.

Mutable Functions

Today we're gonna talk about functions that change their behavior throughout the course of a program, which is helpful for describing certain real world phenomena.

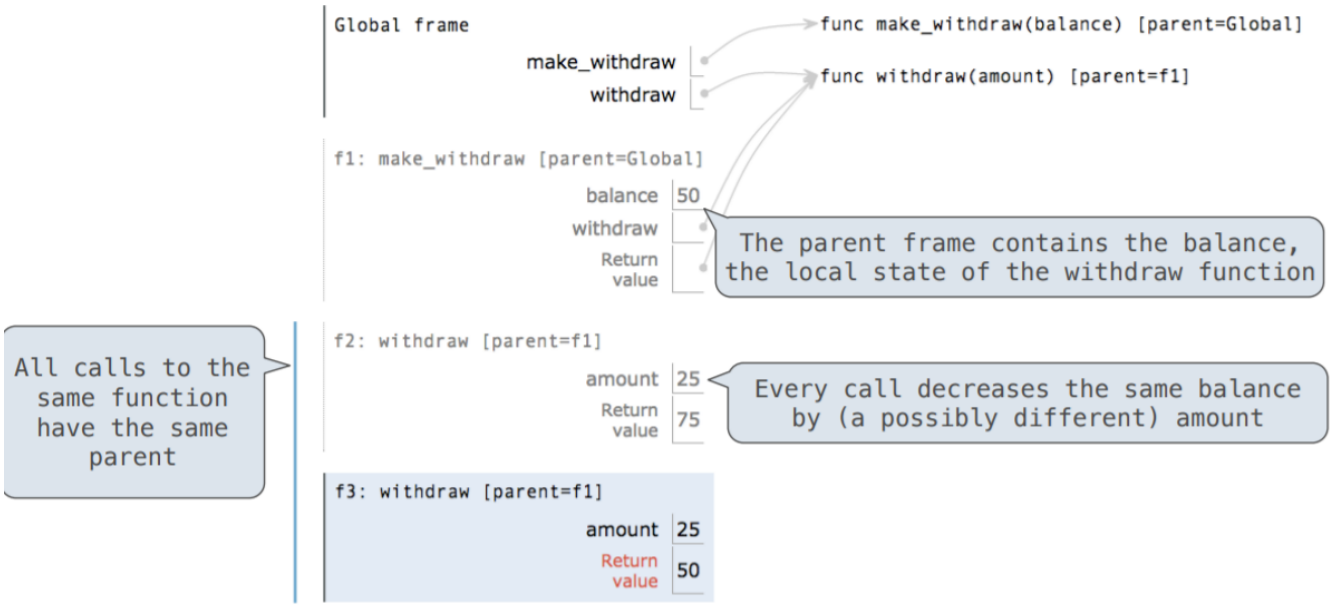
For example, let's take a bank account with a balance of $100. Let's call the function `withdraw`:

```
>>> withdraw(25)
75
>>> withdraw(25)
50
>>> withdraw(60)
'Insufficient funds!'
```

This is new! Previously, if we called a function with the same parameters, it would always return the same return value! So where's this balance stored? There are many ways to trigger this behavior, but for today, we will store some data within the function itself through a higher-order function.

```
>>> withdraw = make_withdraw(100)
```

Before we get to the code, let's see the environment diagram.



You can see the balance is never passed into the `withdraw` function itself. What's happening is every time you call `withdraw`, a balance in the parent frame, not the local frame is being updated.

All calls to the same function have the same parent! Both the first and second withdraw have the parent `make_withdraw`. We've never done this before. Assignment binds name(s) to value(s) in the first frame the current environment. But `withdraw` does something different.

Nonlocal Assignment and Persistent Local State

All frames outside the current local and global frame is called **nonlocal**.

The `nonlocal` statement tells Python that future assignments of that variable name come from somewhere within the environment (except the global frame).

```
def make_withdraw(balance):
    """Return a withdraw function with a starting BALANCE."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds!'
        balance = balance - amount
        return balance
    return withdraw
```

You can put `nonlocal` anywhere before you refer to it within the body of the child function, but it's conventional to put it at the top of the child function. You can also do:

```
def f(x):
    def g(y):
        def h(z):
            nonlocal x
            x += 1
        return h
    return g
```

In the above example: `g` will never have access to `x`. When the nonlocal statement is executed, the interpreter looks for the name `x` first in `g`, when it doesn't find it, then moves to `f`, where `x` is defined.

The global frame is not considered a nonlocal frame. To edit a variable in the global frame, you use the `global` statement, though you will not need it in this course.

The Effect on Nonlocal Statements

Effect: Future assignments to that name change its pre-existing binding in the first non-local frame of the current environment in which that name is bound.

And rules from the official Python language reference:

- Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope.
- Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope

The Many Meanings of Assignment Statements

Status	Effect
No `nonlocal` statement and `x` is not bound locally	Create a new binding from name `x` to object 2 in the first frame of the current environment
No `nonlocal` statement and `x` is bound locally	Re-bind name `x` to object 2 in the first frame of the current environment
`nonlocal` statement and `x` is bound on a nonlocal frame	Re-bind `x` to 2 in the first non-local frame of the current environment in which `x` is bound
`nonlocal` statement and `x` is not bound in a nonlocal frame	SyntaxError: no binding for nonlocal x found
`nonlocal` statement, `x` is bound in a non-local frame, and also bound locally	SyntaxError: name x is parameter and nonlocal

Python Particulars

Almost everything in this class can be applied to other programming languages, but there are a few things that are unique to Python:

Python pre-computes which frame contains each name before executing the body of a function. That means within the body of a function, all instances of a name must refer to the same frame. You cannot, for example:

```
def make_withdraw(balance):
    """Return a withdraw function with a starting BALANCE."""
    def withdraw(amount):
        if amount > balance: #This checks the balance in the parent frame
            return 'Insufficient funds!'
        balance = balance - amount #Without nonlocal, balance would create a new variabl
        return balance
    return withdraw
```

Before running the code, Python essentially looks ahead and sees that you are going to assign `balance` locally. You cannot have both a local and a nonlocal variable of the same name within the same frame.

Mutable Values and Persistent Nonlocal State

Mutable values can be changed without nonlocal statement. Remember that mutable values like lists are only pointed to with names, not bound to the name itself. Anything you draw a line to, outside of the boxes of frame in an environment diagram, are fair game to change from any inner frame.

Python didn't use to have nonlocal statements, and people managed just fine. Here's an example of code that does the same thing, with a different implementation.

```
def make_withdraw_list(balance):
```

```
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw

withdraw = make_withdraw_list(100)
withdraw(25)
```

This never changes what `b` is, which is a pointer to a list, but it does change the list that `b` points to.

These functions are impure, because they have mutable values.

`nonlocal` never confuses names between different functions. See the following example for Professor DeNero and his evil twin Jack:

```
>>> john = make_withdraw(100)
>>> jack = make_withdraw(1000000)
>>> john(10)
90
>>> jack(1000)
999000
>>> john(30)
60
>>> jack(998940)
60
>>> john is jack
False
>>> john
<function make_withdraw...>
>>> jack
<function make_withdraw...>
>>> john == jack
False
>>> john(0) == jack(0)
True
>>> john(10)
50
>>> jack(20)
40
>>> john(0) == jack(0)
False
```

Referential Transparency, Lost

Expressions are referentially transparent if substituting an expression with its value does not change the meaning of a program.

```
mul(add(2, mul(4, 6)), add(3, 5))

mul(add(2,     24     ), add(3, 5))

mul(        26          , add(3, 5))
```

Mutation operations violate the condition of referential transparency because they do more than just return a value, which is modify the environment.

Being careful with nonlocal assignment

Look at the following code:

```
def f(x):
    x = 4
    def g(y):
        def h(z):
            nonlocal x
            x = x + 1
            return x + y + z
        return h
    return g
a = f(1)
b = a(2)
total = b(3) + b(4)
```

This code returns 22. But if we replace `b(3)` with its return value 10, the whole thing will return 21! That's because `b(3)` increased the count of `x` to 5, not 4, and skipping it means `x` is returned as 5, not 6, when `b(4)` is called.