CS61A Lecture 5

Friday, September 6th, 2019

Announcements

- Midterm will be in various locations, at 7 to 9 PM on Monday, September 16th.
- Homework 2 due on Thursday, shorter than HW 1.

Environments for Higher-Order Functions

The reason we talk about environment diagrams is to keep track of what happens in higher-order functions. Environments enable higher-order functions

Functions are first-class objects: they can be assigned, called or returned. Higher-order functions are functions that either take a function as an argument value or return a function as its return value.

For example, this function takes a function `f` and applies it twice to `x`:

```
def apply_twice(f,x):
    return f(f(x))
```

When you define a new user-defined function, you create a new frame. In the `apply_twice` function above, you will draw the environment diagram first by creating a frame for `apply_twice`, then when you call `f`, you create another frame.

The parent of `f` will still be global, because the function was defined within the global frame. The parent of a frame is the frame in which it was defined, not where it is called.

There will be two frames for `f`, one which takes `x` as its input values and returns a return value, which is taken in as the input value `x` for the second `f` and returns a new return value.

Names can be bound to functional arguments. Any names sharing the same variable in the global frame, then the local frame, will be overwritten by the assignment in the local frame.

In other words, look in the global frame only when the name you're looking for is not in the local frame. Every time you call a function, you create a new frame with its local variables. The old frame is not overwritten.

Nesting `def` statements

When you execute a `def` statement that has a `def` statement, then you haven't called the inner def statement at all. The body only **defines** a new function, not computes it.

The parent of a `def` statement within another function definition is the frame in which it was defined.

You can look up variables from the parent frame from within the inner frame, but you cannot modify it without using the `nonlocal` statement, which we will not discuss for another couple weeks.

There will always be a new environment for every frame in the diagram, the question is how many environments are within that environment. For example:

```
n = 2
def new_function(n):
    def old_function(n):
        return n
    return n
```

There are three environments above:

1. Global
2. F1-Global
3. F2-F1-Global

Local names are not visible to other non-nested functions.

How to Draw an Environment Diagram

We can keep track with all this with an environment diagram, much like the one you see on Python Tutor. You can draw your own environment diagrams too! There are separate rules for when functions are defined, and when they are called.

When a function is defined:

- Create a function value.
- Its parent is the current frame.
- Bind `<name>` to the function value in the current frame.

When a function is called:

- Add a local frame, titled with the `<name>` of the function being called:
- Copy the parent of the frame to the local frame: `[parent=<label>]`
- Bind the `<formal parameters>` to the arguments in the local frame.
- Execute the body of the function in the environment that starts with the local frame.

Lambda Expressions

`lambda` expressions are not common in Python, but important in other languages.

A `lambda` expression is a function which has a one-line body, and the value of that line is always the return value. They cannot contain statements at all in Python.

For example:

```
multiply_by_y = lambda x: x * y
```

The name immediately after the `lambda` is the formal parameter of the function. The function will then look for the value of `y` in its environment.

Lambda Expressions versus Def Statements

Both will create a function with the same domain, range and behavior, and both can be bound to a name (with an assignment statement in `lambda` 's case.)

However, only the def statement gives the function an intrinsic name, which won't matter much unless the function is printed, or in an environment diagram:

```
>>> def square(x):
...     return x*x
>>> square(5)
25
>>> square
<function square ...>
>>> square = lambda x: x * x
>>> square(5)
25
>>> square
<function lambda ...>
```