Python Variable Scoping and Closures

In Lab 8, we are implementing the Carlae/Scheme variable scoping semantics:

- `(define <var> <val>)` creates a binding in the current scope
- `(set! <var> <val>)` finds the first scope in which `<var>` exists, and changes that binding.
- `(lambda (...) ...)` creates a function object that remembers the environment it is created in
- function application creates a new environment whose enclosing environment is that from the function, binds the formal parameters to the actual arguments in the new environment, and evaluates the function body with respect to the new environment

How do these (especially `define` and `set!`) compare to Python scoping?

Ref: https://stackoverflow.com/questions/1261875/python-nonlocal-statement

1. Python local variables shadow variables in surrounding scope

```python
In [1]:
x = 0
def outer():
    x = 1
    def inner():
        x = 2
        print("inner:", x)
    inner()
    print("outer:", x)
outer()
print("global:", x)
inner: 2
outer: 1
global: 0
```

2. How does python know a variable is local?

- If you `assign` a variable in a local scope, it's local and shadows the variable in the surrounding scope.
- If you `access` a variable before it is assigned in the local scope, the surrounding variable is used. Note, however, that once you've accessed the variable in the surrounding scope, you can't assign to it!

```python
In [3]:
x = 0
def outer():
    x = 1
    def inner():
        print("inner:", x)  # CHANGE
        #x = 2
        #print("inner:", x)
    inner()
    print("outer:", x)
outer()
print("global:", x)
inner: 1
outer: 1
global: 0
```
3. What if we want to assign a variable in the surrounding scope?

- Python `nonlocal` declaration tells python to use the variable in the nearest surrounding scope (except global scope).
- This is similar (but not identical) to `set!` in Scheme.

```python
In [4]:
x = 0
def outer():
    x = 1
def inner():
        nonlocal x  # CHANGE
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)
inner: 2
outer: 2
global: 0
```

- Note that a `nonlocal` variable needs to be present in a surrounding scope, not including the global scope.
- In Scheme, `set!` can find and change the variable in the global scope also.

```python
In [5]:
x = 0
def outer():
    # x = 1 # CHANGE
def inner():
        nonlocal x  # CHANGE
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)
```

File "<ipython-input-5-9e08b875e506>", line 5
nonlocal x  # CHANGE
SyntaxError: no binding for nonlocal 'x' found

4. What if we want to assign a variable in the global scope?

- Python `global` declaration tells python to use the variable in the global scope (skipping over any intervening nonlocal/surrounding scoped variables).
Python Closures Example

The idea of a "closure" is that a function definition remembers the environment in which it was defined, so that later when the function is called the function has access to the variables in the enclosing environment.

Closures enable object-oriented programming styles

In Python, we have classes, methods, instances, etc., so those are usually most convenient. However, we could build something similar using the power of closures and local state. One such style is often referred to as message passing:
And just for fun

In python, functions are first class objects. As with any object instance, we can attach instance variables to a function. These are different than local variables; they obey class-instance scoping rules. But a simple extension of the above message passing example can use function instance variables to provide access to the "methods", thus giving us usage syntax that looks very similar to our familiar python object-oriented program (OOP) syntax: