Defining Functions III: Nested Definitions

The above examples demonstrate how the ability to pass functions as arguments significantly enhances the expressive power of our programming language. Each general concept or equation maps onto its own short function. One negative consequence of this approach is that the global frame becomes cluttered with names of small functions, which must all be unique. Another problem is that we are constrained by particular function signatures: the `update` argument to `improve` must take exactly one argument. Nested function definitions address both of these problems, but require us to enrich our environment model.

Let's consider a new problem: computing the square root of a number. In programming languages, "square root" is often abbreviated as `sqrt`. Repeated application of the following update converges to the square root of `x`:

```
>>> def average(x, y):
        return (x + y)/2
>>> def sqrt_update(guess, x):
        return average(guess, x/guess)
```

This two-argument update function is incompatible with `improve` (it takes two arguments, not one), and it provides only a single update, while we really care about taking square roots by repeated updates. The solution to both of these issues is to place function definitions inside the body of other definitions.

```
>>> def sqrt(x):
        def sqrt_update(guess):
            return average(guess, x/guess)
        def sqrt_close(guess):
            return approx_eq(square(guess), x)
        return improve(sqrt_update, sqrt_close)
```

Like local assignment, local `def` statements only affect the current local frame. These functions are only in scope while `sqrt` is being evaluated. Consistent with our evaluation procedure, these local `def` statements don't even get evaluated until `sqrt` is called.

Lexical scope. Locally defined functions also have access to the name bindings in the scope in which they are defined. In this example, `sqrt_update` refers to the name `x`, which is a formal parameter of its enclosing function `sqrt`. This discipline of sharing names among nested definitions is called *lexical scoping*. Critically, the inner functions have access to the names in the environment where they are defined (not where they are called).

We require two extensions to our environment model to enable lexical scoping.

1. Each user-defined function has a parent environment: the environment in which it was defined.
2. When a user-defined function is called, its local frame extends its parent environment.

Previous to `sqrt`, all functions were defined in the global environment, and so they all had the same parent: the global environment. By contrast, when Python evaluates the first two clauses of `sqrt`, it create functions that are associated with a local environment. In the call

```
>>> sqrt(256)
16.000000002151005
```

the environment first adds a local frame for `sqrt` and evaluates the `def` statements for `sqrt_update` and `sqrt_close`.

```
1  def average(x, y):
2      return (x + y)/2
3
4  def improve(update, isclose, guess=1):
```

```
    while not isclose(guess):
        guess = update(guess)
    return guess

def approx_eq(x, y, tolerance=1e-3):
    return abs(x - y) < tolerance

def sqrt(x):
    def sqrt_update(guess):
        return average(guess, x/guess)
    def sqrt_close(guess):
        return approx_eq(guess * guess, x)
    return improve(sqrt_update, sqrt_close)

result = sqrt(256)
```

The function values for `sqrt_update` and `sqrt_close` each has a new annotation: a *parent*. The parent of a function value is the first frame of the environment in which that function was defined. Functions without parent annotations were defined in the global environment. In order to annotate the parent of `sqrt_update`, we give the frame `sqrt` a label, `f1`, and refer to that label. We only label frames that we need to reference.

Subsequently, the name `sqrt_update` resolves to this newly defined function, which is passed as an argument to `improve`. Within the body of `improve`, we must apply our `update` function (bound to `sqrt_update`) to the initial `guess` of 1. This final application

creates an environment for `sqrt_update` that begins with a local frame containing only `guess`, but with the parent frame `sqrt` still containing a binding for `x`.

```
def average(x, y):
    return (x + y)/2

def improve(update, isclose, guess=1):
    while not isclose(guess):
        guess = update(guess)
    return guess

def approx_eq(x, y, tolerance=1e-3):
    return abs(x - y) < tolerance

def sqrt(x):
    def sqrt_update(guess):
        return average(guess, x/guess)
    def sqrt_close(guess):
        return approx_eq(guess * guess, x)
    return improve(sqrt_update, sqrt_close)

result = sqrt(256)
```

The most critical part of this evaluation procedure is the transfer of the parent for `sqrt_update` to the frame created by calling`sqrt_update`. This frame is also annotated with `[parent=f1]`.

Extended Environments. An environment can consist of an arbitrarily long chain of frames, which always concludes with the global frame. Previous to this `sqrt` example, environments had at most two frames: a local frame and the global frame. By calling functions that were defined within other functions, via nested `def` statements, we can create longer chains. The environment for this call to `sqrt_update` consists of three frames: the local `sqrt_update` frame, the `sqrt` frame in which `sqrt_update` was defined (labeled `f1`), and the global frame.

The return expression in the body of `sqrt_update` can resolve a value for `x` by following this chain of frames. Recall that looking up a name finds the first value bound to that name in the current environment. Python checks first in the `sqrt_update` frame -- no`x` exists. Python checks next in the parent frame, `f1`, and finds a binding for `x` to 256.

Hence, we realize two key advantages of lexical scoping in Python.

- The names of a local function do not interfere with names external to the function in which it is defined, because the local function name will be bound in the current local environment in which it was defined, rather than the global environment.
- A local function can access the environment of the enclosing function, because the body of the local function is evaluated in an environment that extends the evaluation environment in which it was defined.

The `sqrt_update` function carries with it some data: the value for `x` referenced in the environment in which it was defined. Because they "enclose" information in this way, locally defined functions are often called *closures*

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/functions.html#defining-functions-iii-nested-definitions