5 Functional Style

In the functional programming style, one tries to use a very small set of primitives and means of combination. We'll see that recursion is a very powerful primitive, which could allow us to dispense with all other looping constructs (`while` and `for`) and results in code with a certain beauty.

Another element of functional programming style is the idea of functions as *first-class objects*. That means that we can treat functions or procedures in much the same way we treat numbers or strings in our programs: we can pass them as arguments to other procedures and return them as the results from procedures. This will let us capture important common patterns of abstraction and will also be an important element of object-oriented programming.

5.1 Basics

But let's begin at the beginning. The primitive elements in the functional programming style are basic functions. They're combined via function composition: so, for instance `f(x, g(x, 2))` is a composition of functions. To abstract away from the details of an implementation, we can define a function

```
def square(x):
    return x*x
```

Or

```
def average(a,b):
    return (a+b)/2.0
```

And now, having defined those functions, we can use them exactly as if they were primitives. So, for example, we could compute the mean square of two values as

```
average(square(7),square(11))
```

We can also construct functions "anonymously" using the `lambda` constructor:

```
>>> f = lambda x: x*x
>>> f
<function <lambda> at 0x4ecf0>
>>> f(4)
16
```

The word lambda followed by a sequence of variables separated by commas, then a colon, then a single expression using those variables, defines a function. It doesn't need to have an explicit `return`; the value of the expression is always returned. A single expression can only be one line of Python, such as you could put on the right hand side of an assignment statement. Here are some other examples of functions defined using `lambda`.

This one uses two arguments.

```
>>> g = lambda x,y : x * y
>>> g(3, 4)
12
```

You don't need to name a function to use it. Here we construct a function and apply it all in one line:

```
>>> (lambda x,y : x * y) (3, 4)
12
```

Here's a more interesting example. What if you wanted to compute the square root of a number? Here's a procedure that will do it:

```
def sqrt(x):
    def goodEnough(guess):
        return abs(x-square(guess)) < .0001
    guess = 1.0
    while not goodEnough(guess):
        guess=average(guess,x/guess)
    return guess

>>>sqrt(2)
1.4142156862745097
```

This is an ancient algorithm, due to Hero of Alexandria.[1] It is an *iterative* algorithm: it starts with a guess about the square root, and repeatedly asks whether the guess is good enough. It's good enough if, when we square the guess, we are close to the number, `x`, that we're trying to take the square root of. If the guess isn't good enough, we need to try to improve it. We do that by making a new guess, which is the average of the original guess and `x / guess`.

How do we know that this procedure is ever going to finish? We have to convince ourself that, eventually, the guess will be good enough. The mathematical details of how to make such an argument are more than we want to get into here, but you should at least convince yourself that the new guess is closer to `x` than the previous one was. This style of computation is generally quite useful. We'll see later in this chapter how to capture this *common pattern* of function definition and re-use it easily.

Note also that we've defined a function, `goodEnough`, inside the definition of another function. We defined it internally because it's a function that we don't expect to be using elsewhere; but we think naming it makes our program for `sqrt` easier to understand. We'll study this style of definition in detail in the next chapter. For now, it's important to notice that the variable `x` inside `goodEnough` is the same `x` that was passed into the `sqrt` function.

5.2 List Comprehensions

Python has a very nice built-in facility for doing many interative operations called *list comprehensions*. The general template is
[*expr* `for` *var* `in` *list*]
where *var* is a single variable, *list* is an expression that results in a list, and *expr* is an expression that uses the variable *var*. The result is a list, of the same length as *list*, which is obtained by successively letting *var* take values from *list*, and then evaluating *expr*, and collecting the results into a list.

Whew. It's probably easier to understand it by example.

```
>>> [x/2.0 for x in [4, 5, 6]]
[2.0, 2.5, 3.0]
>>> [y**2 + 3 for y in [1, 10, 1000]]
[4, 103, 1000003]
>>> [a[0] for a in [['Hal', 'Abelson'],['Jacob','White'],
                    ['Leslie','Kaelbling']]]
['Hal', 'Jacob', 'Leslie']
>>> [a[0]+'!' for a in [['Hal', 'Abelson'],['Jacob','White'],
```

```
                      ['Leslie','Kaelbling']]]
['Hal!', 'Jacob!', 'Leslie!']
>>> [x/2.0 for x in [4, 5, 6]]
[2.0, 2.5, 3.0]
```

Another useful feature of list comprehensions is that they implement a common functional-programming capability of *filtering*. Imagine that you have a list of numbers and you want to construct a list containing just the ones that are odd. You might write

```
>>> nums = [1, 2, 5, 6, 88, 99, 101, 10000, 100, 37, 101]
>>> [x for x in nums if x%2==1]
[1, 5, 99, 101, 37, 101]
```

And, of course, you can combine this with the other abilities of list comprehensions, to, for example, return the squares of the odd numbers:

```
>>> [x*x for x in nums if x%2==1]
[1, 25, 9801, 10201, 1369, 10201]
```

Footnotes:

[1] Hero of Alexandria is involved in this course in multiple ways. We'll be studying feedback later on in the course, and Hero was the inventor of several devices that use feedback, including a self-filling wine goblet. Maybe we'll assign that as a final project...

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011