MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Course notes for Week 2

1 Higher-order functions

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.

We've been talking about the fundamental principles of software engineering as being modularity and abstraction. But another important principle is laziness! Don't ever do twice what you could do only once.[1] This standardly means writing a procedure whenever you are going to do the same computation more than once. In this section, we'll explore ways of using procedures to capture common patterns in ways that might be new to you.

What if we find that we're often wanting to perform the same procedure twice on an argument? That is, we seem to keep writing `square(square(x))`. If it were always the same procedure we were applying twice, we could just write a new function

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

But what if it's different functions? The usual strategies for abstraction seem like they won't work.

In the functional programming style, we treat functions as first-class objects. So, for example, we can do:

```
>>> m = square
>>> m(7)
49
```

And so we can write a procedure that consumes a function as an argument. So, now we could write:

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

This is cool, because we can apply it to any function and argument. So, if we wanted to square twice, we could do:

[1]Okay, so the main reason behind this rule isn't laziness. It's that if you write the same thing more than once, it will make it harder to write, read, debug, and modify your code reliably.

```
>>> doTwice(square,2)
16
```

Python gives us a way to define functions without naming them. The expression `lambda y: y + y` denotes a function of a single variable; in this case it is a function that takes numbers as input, and doubles them. In Python, `lambda` doesn't require a `return` expression, and it can only be used with a single expression; you can't have `if` or `for` inside a `lambda` expression. If you need to put those in a function, you have to name that function explicitly.

Another way to apply a procedure multiple times is this:

```
def doTwiceMaker(f):
    return lambda x: f(f(x))
```

This is a procedure that *returns a procedure*! If you'd rather not use `lambda`, you could write it this way:

```
def doTwiceMaker(f):
    def twoF(x):
        return f(f(x))
    return twoF
```

Now, to use `doTwiceMaker`, we could do:

```
>>> twoSquare = doTwiceMaker(square)
>>> twoSquare(2)
16
>>> doTwiceMaker(square)(2)
16
```

A somewhat deeper example of capturing common patterns is sums. Mathematicians have invented a notation for writing sums of series, such as

$$\sum_{i=1}^{100} i \quad \text{or} \quad \sum_{i=1}^{100} i^2 \ .$$

Here's one that gives us a way to compute π:

$$\pi^2/8 = \sum_{i=1,3,5,\ldots} \frac{1}{i^2} \ .$$

It would be easy enough to write a procedure to compute any one of them. But even better is to write a higher-order procedure that allows us to compute any of them, simply:

```
def summation(low, high, f, next):
    s = 0
    x = low
    while x <= high:
        s = s + f(x)
        x = next(x)
    return s
```

This procedure takes integers specifying the lower and upper bounds of the index of the sum, the function of the index that is supposed to be added up, and a function that computes the next value of the index from the previous one. This last feature actually makes this notation more expressive than the usual mathematical notation, because you can specify any function you'd like for incrementing the index. Now, given that definition for sum, we can do all the special cases we described in mathematical notatation above:

```
def sumint(low,high):
    return summation(low, high, lambda x: x, lambda x: x+1)

def sumsquares(low,high):
    return summation(low, high, lambda x: x**2, lambda x: x+1)

def piSum(low,high):
    return summation(low, high, lambda x: 1.0/x**2, lambda x: x+2)

>>> (8 * piSum(1, 1000000))**0.5
3.1415920169700393
```

Now, we can use this to build an even cooler higher-order procedure. You've seen the idea of approximating an integral using a sum. We can express it easily in Python, using our `sum` higher-order function, as

```
def integral(f, a, b):
    dx = 0.0001
    return dx * summation(a, b, f, lambda(x): x + dx)

>>> integral(lambda x: x**3, 0, 1)
0.2500500024999337
```

We'll do one more example of a very powerful higher-order procedure. In the previous chapter, we saw an iterative procedure for computing square roots. We can see that procedure as a special case of a more general process of computing the *fixed-point* of a function. The fixed point of a function f, called f^*, is defined as the value such that $f^* = f(f^*)$. Fixed points can be computed by starting at an initial value v_0, and iteratively applying f: $f^* = f(f(f(f(f(\ldots f(v_0))))))$. A function may have many fixed points, and which one you'll get may depend on the v_0 you start with.

We can capture this idea in Python with:

```
def closeEnough(g1,g2):
    return abs(g1-g2)<.0001
def fixedPoint(f,guess):
    next=f(guess)
    while not closeEnough(guess, next):
        guess=next
        next=f(next)
    return next
```

And now, we can use this to write the square root procedure much more compactly:

```
def sqrt(x):
    return fixedPoint(lambda g: average(g,x/g), 1.0)
```

Another way to think of square roots is that to compute the square root of x, we need to find the y that solves the equation

$$x = y^2 \ ,$$

or, equivalently,

$$y^2 - x = 0 \ .$$

We can try to solve this equation using Newton's method for finding roots, which is another instance of a fixed-point computation.[2] In order to find a solution to the equation $f(x) = 0$, we can find a fixed point of a different function, g, where

$$g(x) = x - \frac{f(x)}{Df(x)} \ .$$

The first step toward turning this into Python code is to write a procedure for computing derivatives. We'll do it by approximation here, though there are algorithms that can compute the derivative of a function analytically, for some kinds of functions (that is, they know things like that the derivative of x^3 is $3x^2$.)

```
dx = 1.e-4
def deriv(f):
    return lambda x:(f(x+dx)-f(x))/dx

>>> deriv(square)
<function <lambda> at 0x00B96AB0>

>>> deriv(square)(10)
20.000099999890608
```

Now, we can describe Newton's method as an instance of our `fixedPoint` higher-order procedure.

```
def newtonsMethod(f,firstGuess):
    return fixedPoint(lambda x: x - f(x)/deriv(f)(x), firstGuess)
```

How do we know it is going to terminate? The answer is, that we don't really. There are theorems that state that if the derivative is continuous at the root and if you start with an initial guess that's close enough, then it will converge to the root, guaranteeing that eventually the guess will be close enough to the desired value. To guard against runaway fixed point calculations, we might put in a maximum number of iterations:

```
def fixedPoint(f, guess, maxIterations = 200):
    next=f(guess)
    count = 0
    while not closeEnough(guess, next) and count < maxIterations:
```

[2] Actually, this is Raphson's improvement on Newton's method, so it's often called the Newton-Raphson method.

```
        guess=next
        next=f(next)
        count = count + 1
    if count == maxIterations:
        print "fixedPoint terminated without desired convergence"
    return next
```

The `maxIterations = 200` is a handy thing in Python: an optional argument. If you supply a third value when you call this procedure, it will be used as the value of `maxIterations`; otherwise, `200` will be used.

Now, having defined `fixedPoint` as a black box and used it to define `newtonsMethod`, we can use `newtonsMethod` as a black box and use it to define `sqrt`:

```
def sqrt(x):
    return newtonsMethod(lambda y:y**2 - x, 1.0)
```

So, now, we've shown you two different ways to compute square root as an iterative process: directly as a fixed point, and indirectly using Newton's method as a solution to an equation. Is one of these methods better? Not necessarily. They're roughly equivalent in terms of computational efficiency. But we've articulated the ideas in very general terms, which means we'll be able to re-use them in the future.

We're thinking more efficiently. We're thinking about these computations in terms of more and more general methods, and we're expressing these general methods themselves as procedures in our computer language. We're capturing common patterns as things that we can manipulate and name. The importance of this is that if you can name something, it becomes an idea you can use. You have power over it. You will often hear that there are any ways to compute the same thing. That's true, but we are emphasizing a different point: There are many ways of expressing the same computation. It's the ability to choose the appropriate mode of expression that distinguishes the master programmer from the novice.

1.1 Map

What if, instead of adding 1 to every element of a list, you wanted to divide it by 2? You could write a special-purpose procedure:

```
def halveElements(list):
    if list == []:
        return []
    else:
        return [list[0]/2.0] + halveElements(list[1:])
```

First, you might wonder why we divided by `2.0` rather than `2`. The answer is that Python, by default, given two integers does integer division. So `1/2` is equal to 0. Watch out for this, and if you don't want integer division, make sure that you divide by a float.

So, back to the main topic: `halveElements` works just fine, but it's really only a tiny variation on `incrementElements1`. We'd rather be lazy, and apply the principles of modularity and abstraction so we don't have to do the work more than once. So, instead, we write a generic procedure, called `map`, that will take a procedure as an argument, apply it to every element of a list, and return a list made up of the results of each application of the procedure.

```
def map(func, list):
    if list == []:
        return []
    else:
        return [func(list[0])] + map(func, list[1:])
```

Now, we can use it to define `halveElements`:

```
def halveElements(list):
    return map(lambda x: x/2.0, list)
```

It's generally considered more in the Python style (or "Pythonic") to use list comprehensions than map, but it's good to know we could have implemented this if we had needed it. For completeness here's how to do the same thing with a list comprehension:

```
def halveElements1(list):
    return [x/2.0 for x in list]
```

1.2 Reduce

Another cool thing you can do with higher-order programming is to use the `reduce` function. Reduce takes a binary function and a list, and returns a single value, which is obtained by repeatedly applying the binary function to pairs of elements in the list. So, if the list contains elements $x_1 \ldots x_n$, and the function is f, the result will be $f(\ldots, f(f(x_1, x_2), x_3), \ldots, x_n)$.

This would let us write another version of `addList`:

```
def addList7(list):
    return reduce(add, list)
```

Note that we have to define the `add` function, as

```
def add(x, y):  return x + y
```

or import it from the `operator` package, by doing

```
from operator import add
```

You can also use `reduce` to concatenate a list of lists. Remembering that the addition operation on two lists concatenates them, we have this possibly suprising result:

```
>>> reduce(add, [[1, 2, 3],[4, 5],[6],[]])
[1, 2, 3, 4, 5, 6]

>>>addList7([[1, 2, 3],[4, 5],[6],[]])
[1, 2, 3, 4, 5, 6]
```

1.2.1 Filter

Yet another handy higher-order-function is `filter`. Given a list and a function of a single argument, it returns a new list containing all of the elements of the original list for which the function returns True. So, for example,

```
>>> filter(lambda x: x % 3 == 0, range(100))
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51,
54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

returns all of the multiples of 3 less than 100.

This is also done nicely with a list comprehension:

```
>>> [x for x in range(100) if x%3 == 0]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51,
54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

1.3 Conclusion

The combination of `map`, `reduce`, and `filter` is a very powerful computational paradigm, which lets you say what you want to compute without specifying the details of the order in which it needs to be done. Google uses this paradigm for a lot of their computations, which allows them to be easily distributed over large numbers of CPUs.

We'll find that it's useful to use these and other patterns involving higher-order functions in many situations, especially in building up new PCAP systems of our own.

2 On Style

Software engineering courses often provide very rigid guidelines on the style of programming, specifying the appropriate amount of indentation, or what to capitalize, or whether to use underscores in variable names. Those things can be useful for uniformity and readability of code, specially when a lot of people are working on a project. But they are mostly arbitrary: a style is chosen for consistency and according to some person's aesthetic preferences.

There are other matters of style that seem, to us, to be more fundamental, because they directly affect the readability or efficiency of the code.

- **Avoid recalculation of the same value.** You should compute it once and assign it to a variable instead; otherwise, if you have a bug in the calculation (or you want to change the program), you will have to change it multiple times. It is also inefficient.
- **Avoid repetition of a pattern of computation.** You should use a function instead, again to avoid having to change or debug the same basic code multiple times.
- **Avoid numeric indexing.** You should use destructuring if possible, since it is much easier to read the code and therefore easier to get right and to modify later.

- **Avoid excessive numeric constants.** You should name the constants, since it is much easier to read the code and therefore easier to get right and to modify later.

Here are some examples of simple procedures that exhibit various flaws. We'll talk about what makes them problematic.

Normalize a vector Let's imagine we want to normalize a vector of three values; that is to compute a new vector of three values, such that its length is 1. Here is our first attempt; it is a procedure that takes as input a list of three numbers, and returns a list of three numbers:

```
def normalize3(v):
    return [v[0]/math.sqrt(v[0]**2+v[1]**2+v[2]**2),
            v[1]/math.sqrt(v[0]**2+v[1]**2+v[2]**2),
            v[2]/math.sqrt(v[0]**2+v[1]**2+v[2]**2)]
```

This is correct, but it looks pretty complicated. Let's start by noticing that we're recalculating the denominator three times, and instead save the value in a variable.

```
def normalize3(v):
    magv = math.sqrt(v[0]**2+v[1]**2+v[2]**2)
    return [v[0]/magv,v[2]/magv,v[3]/magv]
```

Now, we can see a repeated pattern, of going through and dividing each element by `magv`. Also, we observe that the computation of the magnitude of a vector is a useful and understandable operation in its own right, and should probably be put in its own procedure. That leads to this procedure:

```
def mag(v):
    return math.sqrt(sum([vi**2 for vi in v]))

def normalize3(v):
    return [vi/mag(v) for vi in v]
```

This is especially nice, because now, in fact, it applies not just to vectors of length three. So, it's both shorter and more general than what we started with. But, one of our original problems has snuck back in: we're recomputing `mag(v)` once for each element of the vector. So, finally, here's a version that we're very happy with:

```
def mag(v):
    return math.sqrt(sum([vi**2 for vi in v]))

def normalize3(v):
    magv = mag(v)
    return [vi/magv for vi in v]
```

Perimeter of a polygon Now, let's consider the problem of computing the perimeter of a polygon. The input is a list of vertices, encoded as a list of lists of two numbers (such as `[[1, 2], [3.4, 7.6], [-4.4, 3]]`). Here is our first attempt:

```
def perim(vertices):
    result = 0
    for i in range(len(vertices)-1):
        result = result + math.sqrt((vertices[i][0]-vertices[i+1][0])**2 + \
                                       (vertices[i][1]-vertices[i+1][1])**2)
    return result + math.sqrt((vertices[-1][0]-vertices[0][0])**2 + \
                                       (vertices[-1][1]-vertices[0][1])**2)
```

Again, this works, but it ain't pretty. The main problem is that someone reading the code doesn't immediately see what all that subtraction and squaring is about. We can fix this by defining another procedure:

```
def perim(vertices):
    result = 0
    for i in range(len(vertices)-1):
        result = result + pointDist(vertices[i],vertices[i+1])
    return result + pointDist(vertices[-1],vertices[0])

def pointDist(p1,p2):
    return math.sqrt(sum([(p1[i] - p2[i])**2 for i in range(len(p1))]))
```

Now, we've defined a new procedure `pointDist`, which computes the Euclidean distance between two points. And, in fact, we've written it generally enough to work on points of any dimension (not just two). Just for fun, here's another way to compute the distance, which some people would prefer and others would not.

```
def pointDist(p1,p2):
    return math.sqrt(sum([(c1 - c2)**2 for (c1, c2) in zip(p1, p2)]))
```

For this to make sense, you have to understand `zip`. Here's an example of how it works:

```
> zip([1, 2, 3],[4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
```

Bank transfer What if we have two values, representing bank accounts, and want to transfer an amount of money `amt` between them? Assume that a bank account is represented as a list of values, such as `['Alyssa', 8300343.03, 0.05]`, meaning that `'Alyssa'` has a bank balance of $8,300,343.03, and has to pay a 5-cent fee for every bank transaction. We might write this procedure as follows. It moves the amount from one balance to the other, and subtracts the transaction fee from each account.

```
def transfer(a1, a2, amt):
    a1[1] = a1[1] - amt - a1[2]
    a2[1] = a2[1] + amt - a2[2]
```

To understand what it's doing, you really have to read the code at a detailed level. Furthermore, it's easy to get the variable names and subscripts wrong.

Here's another version that abstracts away the common idea of a deposit (which can be positive or negative) into a procedure, and uses it twice:

```
def transfer(a1, a2, amt):
    deposit(a1, -amt)
    deposit(a2, amt)

def deposit(a, amt):
    a[1] = a[1] + amt - a[2]
```

Now, `transfer` looks pretty clear, but `deposit` could still use some work. In particular, the use of numeric indices to get the components out of the bank account definition is a bit cryptic (and easy to get wrong).[3]

```
def deposit(a, amt):
    (name, balance, fee) = a
    a[1] = balance + amt - fee
```

Here, we've used a destructuring assignment statement to give names to the components of the account. Unfortunately, when we want to change an element of the list representing the account, we still have to index it explicitly. Given that we have to use explicit indices, this approach in which we name them might be better.

```
acctName = 0
acctBalance = 1
acctFee = 2
def deposit(a, amt):
    a[acctBalance] = a[acctBalance] + amt - a[acctFee]
```

Strive, in your programming, to make your code as simple, clear, and direct as possible. Occasionally, the simple and clear approach will be too inefficient, and you'll have to do something more complicated. In such cases, you should still start with something clear and simple, and in the end, you can use it as documentation.

[3]We'll see other approaches to this when we start to look at object-oriented programming. But it's important to apply basic principles of naming and clarity no matter whether you're using assembly language or Java.