

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.081—Introduction to EECS I
Spring Semester, 2007
Lecture 2 Notes

The 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 result 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.

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))
```

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 did it because it's a function that we don't expect to be using elsewhere; it's just here to help out in `sqrt`. 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.

Recursion

In the previous example, we defined a `sqrt` function, and now we can use it without remembering or caring about the details of how it is implemented. We sometimes say that we can treat it as a *black box*, meaning that it is unnecessary to look inside it to use it. This is crucial for maintaining sanity when building complex pieces of software. An even more interesting case is when we can think of the procedure that we're in the middle of defining as a black box. That's what we do when we write a recursive procedure.

Recursive procedures are ways of doing a lot of work. The amount of work to be done is controlled by one or more arguments to the procedure. The way we are going to do a lot of work is by calling the procedure, over and over again, from inside itself! The way we make sure this process actually terminates is by being sure that the argument that controls how much work we do gets smaller every time we call the procedure again. The argument might be a number that counts down to zero, or a string or list that gets shorter.

There are two parts to writing a recursive procedure: the base case(s) and the recursive case. The *base case* happens when the thing that's controlling how much work you do has gotten to its smallest value; usually this is 0 or the empty string or list, but it can be anything, as long as you know it's sure to happen. In the base case, you just compute the answer directly (no more calls to the recursive function!) and return it. Otherwise, you're in the *recursive* case. In the recursive case, you try to be as lazy as possible, and foist most of the work off on another call to this function, but

¹Hero 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...

with one of its arguments getting smaller. Then, when you get the answer back from the recursive call, you do some additional work and return the result.

Here's an example recursive procedure that returns a string of n 1's:

```
def bunchaOnes(n):
    if n == 0:
        return ""
    else:
        return bunchaOnes(n-1) + "1"
```

The thing that's getting smaller is n . In the base case, we just return the empty string. In the recursive case, we get someone else to figure out the answer to the question of $n-1$ ones, and then we just do a little additional work (adding one more "1" to the end of the string) and return it.

Here's another example. It's kind of a crazy way to do multiplication, but logicians love it.

```
def mult(a,b):
    if a==0:
        return 0
    else:
        return b + mult(a-1,b)
```

Trace through an example of what happens when you call `mult(3, 1)`, by adding a print statement as the first line of the function that prints out its arguments, and seeing what happens.

Here's a more interesting example of recursion. Imagine we wanted to compute the binary representation of an integer. For example, the binary representation of 145 is '10010001'. Our procedure will take an integer as input, and return a string of 1's and 0's.

```
def bin(n):
    if n == 0:
        return '0'
    elif n == 1:
        return '1'
    else:
        return bin(n/2) + bin(n%2)
```

The easy cases (base cases) are when we're down to a 1 or a 0, in which case the answer is obvious. If we don't have an easy case, we divide up our problem into two that are easier. So, if we convert $n/2$ into a string of digits, we'll have all but the last digit. And $n\%2$ is 1 or 0 depending on whether the number is even or odd, so one more call of `bin` will return a string of '0' or '1'. The other thing that's important to remember is that the `+` operation here is being used for string concatenation, not addition of numbers.

How do we know that this procedure is going to terminate? We know that the number it's operating on is getting smaller and smaller, and will eventually be either a 1 or a 0, which can be handled by the base case.

You can also do recursion on lists. Here's another way to do our old favorite `addList`:

```
def addList6(list):
    if list == []:
        return 0
    else:
        return list[0] + addList6(list[1:])
```

There's a new bit of syntax here: `list[1:]` gives all but the first element of `list`. Go read the section in the Python documentation on subscripts to see how to do more things with list subscripts.

The `addList` procedure consumed a list and produced a number. The `incrementElements1` procedure below shows how to use recursion to do something to every element of a list and make a new list containing the results.

```
def incrementElements1(elts):
    if elts == []:
        return []
    else:
        return [elts[0]+1] + incrementElements1(elts[1:])
```

Higher-order functions

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.² 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:

²Okay, 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.

```
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:

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

Another way to do this is:

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

This is a procedure that *returns a procedure*! The expression `lambda y: y + y` returns a function of a single variable; in this case that function returns a value that doubles its argument.

For reasons that defy explanation, 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 that you're going to return, you have to name that function explicitly. So, for example, 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 :

$$\pi^2/8 = \sum_{i=1,3,5,\dots}^{100} \frac{1}{i^2} .$$

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 sum(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 notation above:

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

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

def piSum(low,high):
    return sum(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 * sum(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. Early in this chapter, we saw an iterative procedure for computing square roots. We can see that 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 x , and iteratively applying f : $f^* = f(f(f(f(f(\dots f(x)))))$). A function may have many fixed points, and which one you'll get may depend on the x 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

$$x = y^2 \quad ,$$

or, equivalently,

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

We can try to solve this equation using Newton's method for finding roots, which is another instance of a fixed-point computation.³ In order to find a solution to the equation $f(x) = 0$, we can find a fixed point of a different function, g , where

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

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)

```

³Actually, this is Raphson's improvement on Newton's method, so it's often called the Newton-Raphson method.

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:
        guess=next
        next=f(guess)
        count = count + 1
    if count == 0:
        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.

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)

```

Is this better than the previous methods? Yes, if you find it clearer or more likely to get right. Otherwise, no.

List Comprehensions

Python has a very nice built-in facility for doing many of the things you can do with `map`, 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!']
>>> [div2(x) for x in [4, 5, 6]]
[2.0, 2.5, 3.0]
```

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 $x_1 \dots x_n$, and the function is f , the result will be $f(\dots, f(f(x_1, x_2), x_3), \dots, x_n)$.

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

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

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]
```