

6.01—Introduction to EECS I
Fall 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 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.

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 ourselves 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.

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 was implemented. We sometimes say that we can treat it as a *black box*, meaning that it's unnecessary to look inside it in order to know how to use it. This is

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...

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 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. It only works when the first argument is greater than 0.

```
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,² 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`, which adds up a list of numbers:

```
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. In this case, it assumes the argument `elts` is a list of numbers, and it returns a list with each of those numbers, incremented by 1.

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

In many cases, there is a straightforward way to write a recursive procedure using a `while` loop (or using a Python list comprehension, as we'll see below). But there are some cases when it is unknown, until you get the input exactly what the looping structure will be; then recursion is by far the cleanest and clearest approach. We'll show two of our favorite such procedures at the end of the following section, because they also make use of higher-order functions.

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.³ 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.

²Remember that `%` is the modulus operator.

³Okay, 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.

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:

```
>>> 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*!

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 approximate 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
    i = low
    while i < high:
        s = s + f(i)
        i = next(i)
    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 : $f^* = f(f(f(f(\dots f(x))))))$. 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)
    if closeEnough(guess, next):
        return next
    else:
        return fixedPoint(f, next)
```

Alternatively, we could have written it using explicit iteration. In Python, the iterative version is actually likely to be more efficient than the recursive version (but that isn't true in languages for which the compiler implements tail-recursion elimination, like Scheme and many versions of Lisp).

```
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.⁴ In order to find a solution to the equation $f(x) = 0$, we can find a fixed point of a different function, g , where

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

and $Df(x)$ is the derivative of f at 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
```

Note that our procedure `deriv` is a higher-order procedure, in the sense that it consumes a function as an argument and returns a function as a result.

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)
```

⁴Actually, 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(next)
        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)
```

In fact, Python already has a built-in `map` procedure, so we don't actually have to define our own.

Now that we have `map`, we can do one of our favorite really recursive problems. Imagine you are given a list of elements, and you need to return a list of all combinations of those elements. So, for example,

```
>>> combinations([1, 2, 3])
[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
```

if we ask for all combinations of the elements `[1, 2, 3]`, we get a list of 8 lists. If we knew in advance that there would be some number `n` elements in the list, then we could do this pretty easily with `n` nested loops. But since we don't know that, it is much easier to do it recursively.

Here is our solution:

```
def combinations(elts):
    if elts == []:
        return [[]]
    else:
        combinationsOfRest = combinations(elts[1:])
        firstElt = elts[0]
        return combinationsOfRest + \
            map(lambda c: [firstElt] + c, combinationsOfRest)
```

We start with the base case. If there aren't any elements in the list, then there is exactly one combination, which is the empty combination. We're required to return a list of all the combinations, so we return a list containing the empty combination.

Otherwise, we get someone (our alter ego) to do most of the work for us, and generate a list of all possible combinations of all but the first element of the list, and call it `combinationsOfRest`. Now, for each of those combinations, there should be two in our result: one with the first element of the list in it, and one without. So, we return the result of appending (with `+`), two lists: the first one is just the combinations of the rest of the elements, and the second one is each of those

combinations, but with the first element added on to the front. If this doesn't make sense, try typing it in and playing with different arguments. Making it print out some intermediate values can often be instructive, as well.

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!']
>>> [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]
```

Reduce

Another cool thing you can do with higher-order programming is to use the `reduce` function. Reduce takes a two-argument 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 $f(\dots, f(f(x_1, x_2), x_3), \dots, x_n)$.

One favorite binary function is addition. We know how to use it in Python in expressions such as `3 + 2`, but the `+` symbol is treated specially in Python, and can't be used to denote the addition

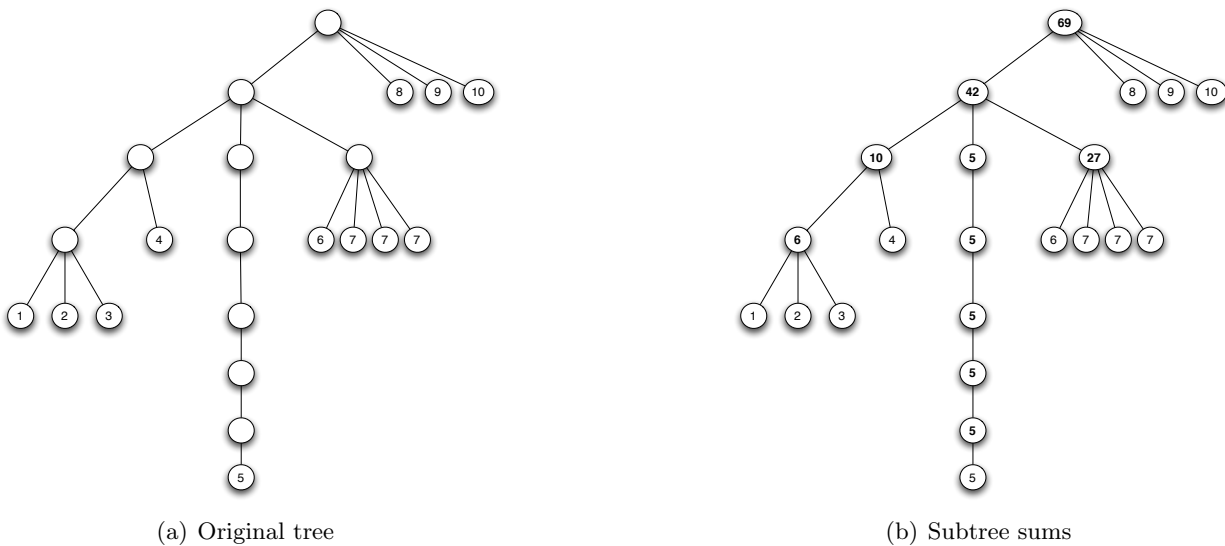


Figure 1: Tree corresponding to the structure `[[[1, 2, 3], 4], [[[[5]]]], [6, 7, 7, 7]], 8, 9, 10]`.

function. Instead we need to import a package called `operator`, and use the function `operator.add`. Doing this would let us write another version of `addList`:

```
import operator
def addList7(list):
    return reduce(operator.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(operator.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]
```

Here's another one of our favorite recursive functions, which is very hard to write iteratively, and shows off `map` and `reduce`. Imagine that we had an arbitrarily nested list of lists of lists of numbers, and we wanted to sum them up. You could think of that structure as a tree (with a branch for each element of a list at each level), and the job of a procedure `addTree` as computing the sum of the numbers in each subtree of the tree. In Python, we'd have:

```
>>> addTree([[[[1, 2, 3], 4], [[[[5]]]], [6, 7, 7, 7]], 8, 9, 10])
69
```

Figure 1(a) shows that structure as a tree; figure 1(b) has the sum of each subtree shown in the internal nodes.

The code for computing this is very compact, but can be a little bit tricky to understand:

```
def isNumber(x):
    return isinstance(x, int) or isinstance(x, float)
```

```
def addTree(tree):
    if isNumber(tree):
        return tree
    else:
        return reduce(operator.add, map(addTree, tree))
```

We start by defining a procedure that will return `True` if its argument is a number. Now, we write our recursive procedure. The base case is when the “tree” it is supposed to add up is a number. If so, then our job is easy, and we just return that number.

Otherwise, we have to recursively add up the values in each of our subtrees, which we do by mapping `addTree` over the tree. Now we have a list of numbers, one for each subtree. We combine them by reducing them with `add`.

We could rewrite this in a way that makes use the Python features of `sum` and list comprehensions as:

```
def addTree(tree):
    if isNumber(tree):
        return tree
    else:
        return sum([addTree(s) for s in tree])
```

Here is the trace we get, when we print out the result of each recursive call before returning it:

```
Sum of [1, 2, 3] is: 6
Sum of [[1, 2, 3], 4] is: 10
Sum of [5] is: 5
Sum of [[5]] is: 5
Sum of [[[5]]] is: 5
Sum of [[[[5]]]] is: 5
Sum of [[[[[5]]]]] is: 5
Sum of [6, 7, 7, 7] is: 27
Sum of [[[1, 2, 3], 4], [[[[[5]]]]], [6, 7, 7, 7]] is: 42
Sum of [[[[1, 2, 3], 4], [[[[[5]]]]], [6, 7, 7, 7]], 8, 9, 10] is: 69
```

The ideas of map and reduce form the basis of a library that Google uses to perform huge computations in parallel on their vast collection of computers. It is much easier to parallelize computations described in functional form than by loops, because they don’t make any commitment to the order in which various computations will actually be done. So, if you’re mapping a function over a huge database of items, those computations can be done in any order, by many different computers; similarly, aggregating the results with reduce (for some standard symmetric operators) can be done in parallel (add up first half of the list, add up second half of the list, add the results); and in fact the reduce computation can be started before the entire list that is being reduced has been computed. This is another example of a powerful abstraction, in this case, away from the order of operations being carried out.

Shared structure

When we work with lists, it is important to know when we are working with multiple references to the same list. You can think of a list as a container with some elements in it. It is possible to have two different containers with exactly the same elements. Let’s explore these ideas a bit.

In the listing below, `a` and `b` are names that both refer to the same list. So, if we change `a`, we change `b`.

```
a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[1] = 100
>>> a
[1, 100, 3]
>>> b
[1, 100, 3]
```

Now, let's make a new list with the same elements, and see what happens when we make various comparisons:

```
>>> c = [1, 100, 3]
>>> a == b
True
>>> a == c
True
>>> a is b
True
>>> a is c
False
```

There are really two different equality tests in Python. The `==` test, when applied to lists, tests to see if the contents of the container are the same. So `a == c` is true. But `is` tests to see if the actual containers are the same. In our case, `a` is the same as `b`, but not the same as `c`.

It's important to keep track of which operations on lists actually change the existing list and which make a copy of it. We can see from the following transcript that when we use `*` to append several copies of a list, it makes all new lists structures:

```
>>> d = 2*a
>>> d
[1, 100, 3, 1, 100, 3]
>>> a[1] = 9
>>> a
[1, 9, 3]
>>> d
[1, 100, 3, 1, 100, 3]
```

But if we make a list containing `a`, twice, the list has the same container in it twice, and it doesn't copy the list structure.

```
>>> e = [a, a]
>>> e
[[1, 9, 3], [1, 9, 3]]
>>> a[1] = 200
>>> e
[[1, 200, 3], [1, 200, 3]]
```

Note that `2*[a]` is the same as `[a,a]`.

You can experiment with other list-manipulation procedures in Python, to see whether they make copies of lists or just refer to the containers. You will find that **append**, **sort** and **reverse** actually change the structure of the lists they operate on. However, slicing (getting a sub-list) gives you a whole new list structure. So, changing one of *y* or *x* in the example below won't change the other one.

```
>>> x = [1, 2, 3]
>>> y = x[1:]
>>> y
[2, 3]
>>> x[2] = 100
>>> x
[1, 2, 100]
>>> y
[2, 3]
```

This leads us to an important idea: if you have a list named *z* and you want to make a copy of it (because, for example, you're about to change it, but you want the original around, too), then you can just do *z[:]*, and you'll get a completely new container with the same elements in it.

```
z = [1, 2, 3, 4]
>>> zcopy = z[:]
>>> zcopy
[1, 2, 3, 4]
>>> z[0] = 'ha'
>>> z
['ha', 2, 3, 4]
>>> zcopy
[1, 2, 3, 4]
```

One more thing to know about and watch out for is procedures that actually change their arguments. In *pure* functional programming, procedures never change anything; they always return a newly made structure or part of an old structure. In a pure functional program, in fact, it doesn't matter whether your structures are shared. Any computation that can be done by a computer can be done by a pure functional program with no assignment statements or changes to structures, but it can be *wildly* inefficient, in many cases, to do so.

So, we may end up writing procedures that change the values of parameters passed into them. In fact, you cannot change the entire value of a parameter; so, for instance, this **changer** procedure doesn't actually change anything (as we'll see next week, it makes a new internal variable called *x*).

```
>>> a = 6
>>> def changer(x):
...     x = 4
>>> a
6
>>> changer(a)
>>> a
6
```

However, we can change values contained in a list structure that is passed in as a parameter:

```
>>> def changeFirstEntry(x):
```

```

...     x[0] = 'nya nya'
>>> b = [1, 2, 3]
>>> changeFirstEntry(b)
>>> b
['nya nya', 2, 3]

```

There is nothing wrong with doing this; in fact it's often a good way to structure programs. But whenever you write a procedure that changes the contents of one of its arguments, you should take care to document that fact as obviously as possible.

Execution model

You all have an informal understanding of what happens when your programs are executed. Usually, the informal understanding is enough to let you write effective programs. But it's useful to really understand what's going on, both because the principles are interesting and important and because it will help you predict what your program will do if it's more complicated or differently structured than usual.

We'll explore exactly what happens when a Python program is executed, in the sections below.

Environments

The first thing we have to understand is the idea of *binding environments* (we'll often just call them *environments*; they are also called *namespaces* and *scopes*). An environment is a stored mapping between names and entities in a program. The entities can be all kinds of things: numbers, strings, lists, procedures, objects, etc. In Python, the names are strings and environments are actually dictionaries, which we've already experimented with.

Environments are used to determine values associated with the names in your program. There are two operations you can do to an environment: add a binding, and look up a name. You do these things implicitly all the time in programs you write. Consider a file containing

```

a = 5
print a

```

The first statement, `a = 5`, creates a binding of the name `a` to the value 5. The second statement prints something. First, to decide that it needs to print, it looks up `print` and finds an associated built-in procedure. Then, to decide what to print, it evaluates the associated expression. In this case, the expression is a name, and it is evaluated by looking up the name in the environment and returning the value it is bound to (or generating an error if the name is not bound).

In Python, there are environments associated with each module (file) and one called `__builtin__` that has all the procedures that are built into Python. If you do

```

>>> import __builtin__
>>> dir(__builtin__)

```

you'll see a long list of names of things (like `'sum'`), which are built into Python, and whose names are defined in the builtin module. You don't have to type `import __builtin__`; as we'll see below, you always get access to those bindings. You can try importing `math` and looking to see what names are bound there.

Another operation that creates a new environment is a function call. In this example,

```
def f(x):
    print x
>>> f(7)
```

when the function `f` is called with argument 7, a new *local* environment is constructed, in which the name `x` is bound to the value 7.

So, what happens when Python actually tries to evaluate `print x`? It takes the symbol `x` and has to try to figure out what it means. It starts by looking in the *local* environment, which is the one defined by the innermost function call. So, in the case above, it would look it up and find the value 7 and return it.

Now, consider this case:

```
def f(a):
    def g(x):
        print x, a
        return x + a
    return g(7)
>>> f(6)
```

What happens when it's time to evaluate `print x, a`? First, we have to think of the environments. The first call, `f(6)` establishes an environment in which `a` is bound to 6. Then the call `g(7)` establishes another environment in which `x` is bound to 7. So, when needs to print `x` it looks in the local environment and finds that it has value 7. Now, it looks for `a`, but doesn't find it in the local environment. So, it looks to see if it has it available in an *enclosing environment*; an environment that was enclosing this procedure *when it was defined*. In this case, the environment associated with the call to `f` is enclosing, and it has a binding for `a`, so it prints 6 for `a`. So, what does `f(6)` return? 13.

You can think of every environment actually consisting of two things: (1) a dictionary that maps names to values and (2) an enclosing environment.

If you write a file that looks like this:

```
b = 3
def f(a):
    def g(x):
        print x, a, b
        return x + a + b
    return g(7)

>>> f(6)
7 6 3
16
```

When you evaluate `b`, it won't be able to find it in the local environment, or in an enclosing environment created by another procedure definition. So, it will look in the *global environment*. The name `global` is a little bit misleading; it means the environment associated with the file. So, it will find a binding for `b` there, and use the value 2.

One way to remember how Python looks up names is the LEGB rule: it looks, in order, in the *Local*, then the *Enclosing*, then the *Global*, then the *Builtin* environments, to find a value for a name. As soon as it succeeds in finding a binding, it returns that one.

We have seen two operations that cause bindings to be created: assignments, and function calls. While we’re talking about assignments, here’s a nice trick, based on the packing and unpacking of *tuples*.

```
>>> [a, b, c] = [1, 2, 3]
>>> a
1
>>> b
2
>>> c
3
```

When you have a list (or a tuple) on the left-hand side of an assignment statement, you have to have a list (or tuple) of matching structure on the right-hand side. Then Python will “unpack” them both, and assign to the individual components of the structure on the left hand side. You can get fancier with this method:

```
thing = [8, 9, [1, 2], 'John', [33.3, 44.4]]
>>> [a, b, c, d, [e1, e2]] = thing
>>> c
[1, 2]
>>> e1
33.299999999999997
```

Bindings are also created when you execute an `import` statement. If you execute

```
import math
```

Then the `math` module is loaded and the name `math` is bound, in the current environment, to the `math` module. No other names are added to the current environment, and if you want to refer to internal names in that module, you have to qualify them, as in `math.sqrt`. If you execute

```
from math import sqrt
```

then, the `math` module is loaded, and the name `sqrt` is bound, in the current environment, to whatever the the name `sqrt` is bound to in the `math` module. But note that if you do this, the name `math` isn’t bound to anything, and you can’t access any other procedures in the `math` module.

Another thing that creates a binding is the definition of a function: that creates a binding of the function’s name, in the environment in which it was created, to the actual function.

Finally, bindings are created by `for` statements and list comprehensions; so, for example,

```
for element in listOfThings:
    print element
```

creates successive bindings for the name `element` to the elements in `listOfThings`.

Figure 3 shows the state of the environments when the `print` statement in the example code shown in figure 2 is executed. Names are first looked for in the local scope, then its enclosing scope (if there were more than one enclosing scope, it would continue looking up the chain of enclosing scopes), and then in the global scope.

```

a = 1
b = 2
c = 3
def d(a):
    c = 5
    from math import pi
    def e(x):
        for i in range(b):
            print a, b, c, x, i, pi, d, e
    e(100)

>>> d(1000)
1000 2 5 100 0 3.14159265359 <function d at 0x4e3f0> <function e at 0x4ecf0>
1000 2 5 100 1 3.14159265359 <function d at 0x4e3f0> <function e at 0x4ecf0>

```

Figure 2: Code and what it prints

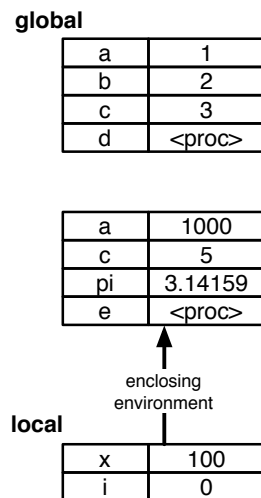


Figure 3: Binding environments that were in place when the first print statement in figure 2 was executed.

Local versus global references There is an important subtlety in the way names are handled in the environment created by a procedure call. When a name that is not bound in the local environment is referred to, then it is looked up in the enclosing, global, and built-in environments. So, as we've seen, it is fine to have

```
a = 2
def b():
    print a
```

When a name is assigned in a local environment, a new binding is created for it. So, it is fine to have

```
a = 2
def b():
    a = 3
    c = 4
    print a, c
```

Both assignments cause new bindings to be made in the local environment, and it is those bindings that are used to supply values in the print statement.

But here is a code fragment that causes trouble:

```
a = 3
def b():
    a = a + 1
    print a
```

It seems completely reasonable, and you might expect it to print 4. But, instead, it generates an error if we try to call `b`. What's going on?? It all has to do with when Python decides to add a binding to the local environment. When it sees this procedure definition, it sees that the name `a` is assigned to, and so, at the very beginning, it puts an entry for `a` in the local environment. Now, when it's time to execute the statement

```
a = a + 1
```

it starts by evaluating the expression on the right hand side: `a + 1`. When it tries to look up the name `a` in the local environment, it finds that it has been added to the environment, but hasn't yet had a value specified. So it generates an error.

We can still write code to do what we intended (write a procedure that increments a number named in the global environment), by using the `global` declaration:

```
a = 3
def b():
    global a
    a = a + 1
    print a
>>> b()
4
>>> b()
5
```

The statement `global a` asks that a new binding for `a` *not* be made in the local environment. Now, all references to `a` are to the binding in the global environment, and it works as expected. In Python, we can only make assignments to names in the local scope or in the global scope, but not to names in an enclosing scope. So, for example,

```
def outer():
    def inner():
        a = a + 1
    a = 0
    inner()
```

In this example, we get an error, because Python has made a new binding for `a` in the environment for the call to `inner`. We'd really like for `inner` to be able to see and modify the `a` that belongs to the environment for `outer`, but there's no way to arrange this.

Evaluating expressions

Now, we're ready to understand what Python does when it evaluates an expression, such as `f(3, g(a, 5))`. The evaluator takes an expression and an environment and returns a value; the value can be a number, string, dictionary, list, procedure, etc. Evaluation in Python can be nicely described as a recursive process.

There are two base cases:

1. If the expression is a constant (number or string), then the value is just that constant value;
2. If the expression is a name, then the value is the result of looking the name up in the local, enclosing, global, and built-in environments until the first binding is found, and the associated value is returned.

In the recursive case, we have the application of a function to some arguments.

- We start by evaluating the expressions that describe the function and each of the arguments (by making a recursive call to the evaluator). At this point, we have our hands on an actual function and a set of values that should be passed in as values to the function.
- Next, we establish a new environment, in which the formal parameters (the argument names in the definition) of the function are bound to the values we just computed.
- Now, we evaluate the body of the procedure in that new environment (by making a recursive call to the evaluator)

This is a beautiful process that is actually quite easy to implement for simple functional languages.

In Python, our expressions often look more like

```
3 + (a*4)
```

The infix operators `+` and `*` are just special syntax, and that expression gets converted internally by Python into something like

```
operator.plus(3, operator.times(a, 4))
```

before it is evaluated in the way described above.

Special forms

Of course, Python is more than expression evaluation and assignment. In order to have a truly general-purpose programming language, we need at least one more construct: `if`. The important thing about `if` is that it either evaluates one set of statements or another depending on whether the condition is true. You can't make that happen simply by evaluating functions using the model above.

Python supplies all sorts of other handy things, like loops, and list comprehensions. We could precisely specify the execution model for each of these statements (and such a language specification exists for Python), and then it's a medium-sized but well-defined job to write an interpreter for Python that does what the shell does: reads in Python programs and does what they specify.