

\$Id: //depot/prj/pyfpintro/branch/2008-03-04/ukuug/index.txt#7 \$

PyCon UK 2007
Introduction to Functional Programming
David Jones
drj@ravenbrook.com
2008-04-02

== Slide 0 ==

My name is David Jones and I'm here today to give a talk on function programming in Python. First of all I'd like to thank you all for coming to PyCon UK and coming to my talk.

I'm a professional programmer; if I'm trying to give the impression that I'm more disciplined in my approach, I might call myself a software engineer. I work for a tiny little consulting company called Ravenbrook Limited.

The work I do varies but on the whole is what I would call bespoke systems software. I tend not to focus on a particular approach or language, I've done work in C, Lua, Python, even Java and C++; but I tend to do a lot of work in C. Ravenbrook very much regards Python as a core part of our toolset. We use it for company administration and for internal and client financed projects. Our use of Python goes back to about the year 2000 when we selected it as the main implementation language for the open-source p4dti. This is a defect tracking integration tool supplied by Perforce and maintained by Ravenbrook. It's about 12,000 lines of Python, and when we started we were using Python 1.5.2.

== Slide 2 ==

Functional programming is a style of programming and as such sits alongside other styles of programming, such as object oriented programming, procedural programming, and so on. Each of these styles emphasises a certain set of programming techniques.

Imperative programming emphasises changes of state; object-oriented programming emphasises encapsulation of state in structured bundles and hierarchical organisation; functional programming emphasises computing by composing functions.

There are others but these are the ones you hear being talked about today.

These styles can be used to describe programming languages, programmers, and programs, as well as describing the corresponding design methods. Of course it's usually possible to write programs in any style you like in most programming languages, you can write object oriented programs in BASIC, the issue becomes how easy it is.

Python is usually described as an object-oriented language, and it is. It provides good support for programming in the object-oriented mode. One of the things I like about Python and something that I regard as a big strength is that it supports many different programming styles without restricting you. Python has good support for all

of these programming styles and you can pick-and-mix even within a single program if that seems to be a good idea.

== Slide 3 ==

Like most attributes in the real world, whether a language is functional or not is not a black and white issue, but rather a matter of degree. There's a spectrum of possibilities.

Languages like ML and Haskell are the traditionally regarded as classic functional programming languages. It's difficult to do anything other than functional programming in those languages.

A language like C lies at the other extreme, it offers hardly any features to support functional programming; the programmer must implement many facilities; it's very difficult to do functional programming in C (the easiest approaches end up implementing a functional language in C).

Python is somewhere in the middle. It has pretty good support for functional programming, but also for other styles; functional programming is not the main idea. In terms of functional programming I'd place it somewhere between Common Lisp (which is more imperative) and Scheme (which is more functional).

== Slide 4 ==

Before I go into more detail about what functional programming is, let's have a look at something that obviously pretty important in functional programming: The Function

Here I use a definition that's borrowed from secondary school. A function is a machine that takes an input and produces an output (dependent on that input). Python, like many other programming languages, extends this notion to multiple inputs (arguments). The function result None is very much like no result.

The key here is that a function has no side-effects; no input nor output, and no modification of any state. These days we tend to call side-effect free functions /pure/ functions and accept that function includes functions with side-effects. Sometimes we'll say /functional/ in the sense of side-effect free.

== Slide 5 ==

Here's something that not a function. The square root operator from maths. This operator indicates that the square root of the number is required, but a number has 2 square roots, and if we restrict ourselves to real numbers then negative numbers have no square roots.

Of course we can usually turn something like that into a function that returns a single result. Python's sqrt function in the the math module returns the positive square root of a non-negative number.

== Slide 6 ==

So, what is functional programming? Central to functional programming is the fact that functions are data.

For a language to support functional programming well, functions have to be *first-class values*. That means that we can treat functions like any other datatype. Make sure your language has first-class functions. Java doesn't.

Of course, you can do all these things in Python.

== Slide 7 ==

Here's a very simple example. We define a function called 'hello', we store that function in a variable, 'y'. We can now call that function using either its original name, hello, or its new name, y. And we can pass our function to another function, such as the builtin function 'type'.

== Slide 9 ==

If you've been playing around with Python at all, you've probably been passing functions to functions without realising. The help function in Python is just an ordinary function. So when we ask for help, help on the function map in this case which I'll be talking about later on, we actually passing a function to the function help. The help function extracts the function's docstrings and prints them.

== Slide 11 ==

Now I'd like to talk about lists. Lots of programs consisting of doing something over a bunch of items in a list. Maybe that's the whole program or maybe it's a small part. There are imperative and functional ways of doing this.

The imperative way uses Python's for statement to loop over the list.

The functional way uses Python's builtin map function.

Let's say we wanted to create a list of square roots of the integers from 0 up to some number.

== Slide 12 ==

Here's how to do that in both imperative and functional versions. The most obvious thing you notice is that the functional version is a lot shorter. That's kind of because I deliberately chose an example where it's extremely convenient to use map.

So the imperative example works by creating an empty list for the result, iterating over our list of numbers from 0 to 9, and adding the square root to our result list in turn.

The functional example uses map. Map works by taking a function and a list, it applies the function to each element of the list in turn, and produces a list of the results. Map is a classic example of using first class functions; we wouldn't be able to have a map function if we couldn't pass functions around as arguments.

It's important to notice the difference in how `math.sqrt` is handled in the two examples. In the imperative example the body of the for loop calls `math.sqrt` to get the results. In the functional example, we don't call `math.sqrt` directly at all, we pass the function `math.sqrt`, without calling it, to the `map` function. The internals of `map` will call `math.sqrt`.

This sort of thing comes up quite a lot in functional programming, there's a world of difference between calling a function and merely passing it to another function.

One of the reasons the functional version is shorter is that there are fewer variables that need to be created. The imperative version needs a variable `r` for the result, and it also needs the variable `x` to hold each element of the list. These variables are not particularly important, and I tend to think that sometimes the mental effort of having to give names to such trivial variables gets in the way of actual programming.

What if the thing that we want to do to each element of our list is not a convenient function?

== Slide 13 ==

Here's an example adapted from a script I use to manage my blog. The `blog` variable holds an `xmlrpclib.ServerProxy` instance, and I can use that to get all my posts as some sort of Python datastructure.

The posts are returned as a list

and each post is a Python dictionary of key/values pairs. Suppose I want a list of the postids, these are useful for referring to the posts in other XML-RPC query functions. Each post's postid is stored as the value corresponding to the dictionary key called 'postid', unsurprisingly.

I can create a little function, called `postid` here, that takes a post and returns its postid. Then I can use `map` to get a list of postids. This `postid` function though doesn't really deserve a name, and once I've given it a name it clutters up my namespace.

I can create anonymous functions by using a lambda expression. The syntax is `lambda`, a list of the arguments names, and then an expression which becomes the result of the function when its called. You often use these anonymous lambda functions if you use `map`.

== Slide 014 ==

Now a different sort of operation on lists.

Consider Python's functions `sum` and `string.join`. `sum` adds up a list of numbers. `join` joins a list of strings together, with a separator in between.

They both have a similar form, they take a list and they accumulate a single result by iterating over the list.

The `reduce` function abstracts this behaviour of iterating over a list and accumulating a result.

== Slide 014a0 ==

reduce is used to accumulate lists into a single result.

In simple cases you can think of reduce as inserting an operator between the elements of a list.

Here's a way to compute the factorial of 10.

== Slide 014a1 ==

So we can see how to implement Python's sum and join functions. Well, poor man's versions of them anyway.

== Slide 014a2 ==

Checksums. So an industry standard protocol that I happen to be in the middle of implementing has a checksum. The checksum is pretty naive: Treat each byte as an unsigned 8-bit integer and add them all up. You get a 16-bit result (because all the checksummed messages have less than 256 bytes in them).

That turns out to be pretty easy to do in Python.

A couple of things to note here. map works with strings as well as lists. It iterates over the characters of a string, just like it iterates over the elements of a list. In fact it works with all sequence types. Python doesn't have a character type, so when you map a function over a string the function gets passed a string of length 1:

[point at slide]

map has a special, and not particularly elegant, feature. If you pass None as the first argument it defaults to a function that returns its arguments. I've shown that using an explicit lambda as well.

The checksum operates on the binary representation of the strings, so that's why I'm using ord here. First of all I need to convert my string, a sequence of characters, into a sequence of numbers. That's a good use of map. I'm pretty sure that ord is an homage to Pascal by the way.

Like the factorial example I use a pretty simple lambda to add all the numbers together.

Of course I could've used sum, but I'm making the reduce explicit.

== Slide 014a3 ==

The two previous examples both had simple lambda functions that did nothing more than wrap the multiply and the add operators. That's such a common use that Python provides a module that does this.

operator.add is a 2-argument function that does the same thing as the add operator. Similarly for operator.mul.

Every Python operator has a version that is a function in the operator module. It stops you having to write a lot of trivial lambda expressions.

== Slide 014a4 ==

The function passed to reduce is not always so simple. The PNG and Ethernet specifications have a proper checksum algorithm, a 32-bit polynomial residue.

All this talk about polynomial residue is just maths talk. What it comes down to is XOR which is why CRCs get used at all.

The fast version of the algorithm has at its heart the following:

```
residue = table[(residue ^ input) & 0xff] ^ (residue >> 8)
```

table is a precomputed table of partial residues used to speed up the computation.

This is perfect for reduce. The residue, the CRC, computed at one stage becomes the input for the next stage.

== Slide 014a5 ==

```
def crcbyte(residue, input) :  
    """Add a single octet of input to an incremental CRC computation."""  
    return table[(residue ^ input) & 0xff] ^ (residue >> 8)
```

```
reduce(crcbyte, 'food', 0xffffffff)
```

In this case table is a precomputed table of CRC values that speeds up the algorithm.

This crcbyte function was just large enough to warrant giving a name and documentation, rather than leaving as an anonymous lambda, I felt.

== Slide 015 ==

Consider a two-argument function like string.join. As you probably know this function takes a sequence, usually a list of words, and a separator. Maybe you want a specialised version of this function; one that always uses the same separator. '/' (slash) say. Here's a definition (called mkpath):

```
def mkpath(components) :  
    """Join components together using '/' to make a string."""  
  
    return string.join(components, sep='/')
```

That's fine as definitions go, but once you get into this, you find yourself yourself wanting this sort of thing quite a lot. A version of a multi-argument function that is specialised by fixing one or more of the arguments. A 2-argument function can become a 1-argument function by fixing one of its arguments. In general this is called Partial Evaluation. Our mkpath example above is a partially evaluated version

of `string.join`.

The `functools` module has a higher-order function called `partial` that helps you build partially evaluated functions.

Note that `functools` is only in 2.5 and above, but `functools.partial` isn't particularly hard to write in earlier versions of Python.

```
mkpath = functools.partial(string.join, sep='/')
```

So what this call to `functools.partial` is doing is asking for a version of `string.join` but where the keyword argument `sep` is always set to `'/'`.

== Slide 016 ==

So here's the pattern:

Using `string.join` with some particular argument a lot:

```
installprefix = string.join(install, '/')  
w = string.join(dirs, '/')
```

Abstract into function:

```
def mkpath(c) : return string.join(c, sep='/')
```

Definition as higher-order artefact:

```
mkpath = functools.partial(string.join, sep='/')
```

Aside: it would have been a little bit easier to use `partial` had `string.join` taken the separator argument first. But it doesn't.

== Slide 017 ==

In the case of `string.join`, and lots of similar functions, Python has a feature called `bound-methods`. They provide a sort of bridge between the functional world and the object oriented world.

You probably already knew that `'/'.join(things)` was the same as `string.join(things, '/')`. A bound method is what `'/'.join` is; it's what you get if you ask for an object's method but don't call it. It's the same as a partially evaluated function. `'/'.join` is a version of `join` where the self argument is `'/'`.

The nice thing about bound methods is that they are values in themselves. We can define `mkpath` even more directly. Do not do this in JavaScript!

== Slide 018 ==

So there's an equivalence between these 3 forms:

```
'/'.join <=> lambda x: '/'.join(x) <=> functools.partial(join, sep='/')
```

Partial function evaluation from functional programming gives a more general framework for understanding bound methods.

A lot of the time you can use this bound method feature instead of using a lambda or `functools.partial`. It's a sort of stealth functional programming feature when used like this.

== Slide 19 ==

Another popular function similar to `map` is `filter`. This function is also builtin to Python, like `map`. Like `map` it takes a function and a list as arguments, and like `map` it applies the function to each element of the list. What it returns however is a list of elements for which the function returns a true value. The function is effectively treated as a predicate, so expect to return a true/false value, and `filter` returns all the elements that match the predicate.

Again, anonymous lambdas are useful. The first example selects all even numbers from a range and isn't very useful because the range function could have done that.

In the second example the predicate tests that the square root of a number is an integer. It does this by squaring the integer part of the square root; if we don't get the original number then the square root must have had a fractional part. Of course this is not a very efficient way to generate square numbers.

In one of the scripts I mentioned earlier that processes posts on my blog I use `filter` to remove draft posts. The example is a little cumbersome to show here.

== Slide 20 ==

Suppose I had a list of strings and I wanted the list of those strings that matched a particular regular expression. I can compile the regular expression into an object and then use that object's `search` method to test each string. Using `filter` and a very small lambda function I can extract the strings. Effectively the little lambda function is remembering which object's method we are calling.

It turns out, in what is quite a nice feature of Python, that if we refer to an object method such as `r.search` without calling it then the value we get is a function that when called invokes the method on the object. These functions are called bound methods. It turns out that the unbound methods are the sort you get by fetching the method from the class.

Bound methods let us dispense with the lambda and are very useful in mixing object oriented and functional code, as the last example shows. Effectively we're using the compiled regular expression object as a predicate to our `filter` function.

[This is a sort of built in currying, but I can't say that until I've introduced currying]

== Slide 097 ==

I'm tempted to say here that you can tell that Python isn't a proper functional programming language because it doesn't have a compose operator that's just one character.

== Closing ==

I'd like to thank the conference organisers, my chair in particular for keeping us all to schedule.

HOMEWORK

Go and read my 2007 "Introduction to Functional Programming in Python!"
<http://drj11.wordpress.com/2007/09/10/introduction-to-functional-programming-in-python-slides/>

sort versus (in 2.5) sorted
trees
compiler module
mapping operators to two-argument functions (useful for reduce)
curryl curryr uncurry
foldl foldr (foldl is reduce in python. reducer does not exist)
constantly
before/after/around

FURTHER READING

[FPHOWTO] "Functional Programming HOWTO"; A. M. Kuchling.
<http://docs.python.org/dev/howto/functional.html>

Very thorough overview of functional programming paradigms in Python 2.5. Including all the whizzy generators, iterators, comprehensions and stuff like that.

[MERTZ] "Functional programming in Python"; David Mertz.
<http://www.ibm.com/developerworks/library/l-prog.html>
<http://www.ibm.com/developerworks/library/l-prog2.html>
<http://www.ibm.com/developerworks/library/l-prog3.html>

Well written. Kind of like a sketch map for Haskell programmers marooned on the island of Python. In other words it takes a much more purely functional style than I have adopted here. It much more closely takes each classical functional programming idiom and translates that into Python.

[MLWP] "ML for the Working Programmer"; Lawrence C. Paulson. Of course I have a certain fondness for this book as it was the course text when I did Dr Paulson's course. ML is an embodiment of the (strict) functional programming paradigm. It encourages you to think functionally and without updates.

[SICP] "Structure and Interpretation of Computer Programs"; Harold Abelson and Gerald Jay Sussman, with Julie Sussman. Classic,

infamous, loved by all right thinking people. Every reading brings new insights. It has a nice weight to it and a big lambda on the front, so you can literally browbeat people with the power of lambda. Available online for the benefit of starving students.

[WFPM] "Why Functional Programming Matters"; John Hughes.

<http://www.cs.chalmers.se/~rjmh/Papers/whyfp.html>

Some good solid examples of programming using the lazy functional programming language, Miranda. The examples are mostly mathematical -- integration, alpha-beta pruning -- and this makes a refreshing change from automated proof systems.