$Id: //depot/prj/pyfpintro/master/index.txt#7 $

PyCon UK 2007
Introduction to Functional Programming
David Jones
drj@ravenbrook.com
2007-09-09

== 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. We're sponsoring the coffee today, which I feel is important part of being a software developer, so I hope you enjoy the coffee!

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 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. I don't think we support that version of Python any more.

[Slide 0a shows the coffee beans and was a late insertion]

== Slide 1 ==

It would be good if I could have some idea of people's level of knowledge. Could I ask for a show of hands please; how many people would consider themselves a Python:
beginner
journeyman
master?

And now what about the same question for functional programming?
beginner
journeyman

master?

Myself, I'd consider myself somewhere between journeyman and master for Python, but probably only a journeyman for functional programming.

If people have any questions then they should obsolutely feel free to interrupt me and ask. Someone else is bound to wanting the answer to the same question.

== 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 style 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 white 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 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 possiblities.

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, and tuples can be used as a way of returning multiple values.

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.

On the left we see the machine that implements a double function, and on its right are two applications of it. 5 doubles to 10 and the string 'foo' doubles to the string 'foofoo'. That the double function operatores on strings might surprise you, but it's natural and Pythonic that it does.

== Slide 5 ==

Double is an almost completely trivial function, so simple that you probably wouldn't expect to see it defined in any real program. But here's a definition for our purposes.

The fact double works on strings arises naturally from its

implementation. This sort of thing happens all the time in Python and is related to what the OO crowd call duck typing. Just in case you didn't know, in Python the star operator used for multiplying numbers can also be used for string repetition; that's why our double function can double strings.

Not requiring double to operate on numbers is a good thing. It means I implicitly allow any type of value to be operated on by double. Python is a very extensible language, so someone else can come up with a new type and if it's sensible for double to be applied to objects of that type then they can be, and I haven't arbitrarily stopped them from using the double function for that.

That's a little aside that puts a little bit of emphasis on the Python in Functional Programming in Python.

== 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.

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 8 ==

We need to be a little bit careful here about the difference, in Python, between a variable and a value. What's actually happening when we go 'y = hello' is that the variable y is made to refer to the same thing as in variable hello. We're not creating a copy of hello and placing that in y; y and hello both refer to the same copy. This isn't so important for a function, because you don't often modify a function, but it becomes more impotant to remember later on when we're dealing with lists and objects that can be modified.

== Slide 9 ==

In fact when we defined hello using def hello this was in many ways just like a variable assignment. The function that we defined isn't intrinsically bound to the variable hello as this example illustrates. We can delete the hello variable, or we could've assigned some other value to it, and the function continues to exist, and the variable y continues to refer to it.

Using def is a little different from a straight variable assignment; for example, the help function still knows that our function was originally defined using the name hello, even though hello no longer works.

The builtin function help, by the way, is another simple example of a function that accepts other functions as an argument.

== Slide 10 ==

There's a certain tradition in functional programming of using the factorial function as an example of recursion. Probably many of you have seen this kind of example before. One thing this illustrates is something that I was talking about earlier, Python often allows you to choose your own programming style. In the case of factorial I would probably say that both examples were reasonable Python style.

There's a certain awkwardness in the imperative version using the range function, but you quickly get used to it in Python.

The first version, the imperative one, reads more like a written procedure: first take the number 1, then multiply it by all the numbers up to N in turn. The answer you get is the factorial of N.

The second one is perhaps more mathematical. It reads like a definition of factorial rather than a procedure to compute it: The factorial of 0 or 1 is 1; the factorial of any other number N is N multiplied by the factorial of the predecessor of N.

Where a function refers to itself like this, and calls itself, we call this recursion. It comes up a lot in functional programming.

== 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 ale 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 variables hold an xmlrpclib.ServerProxy instance, and I can use that to get all my posts in Python. 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 corresponing 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 14 ==

Another popular example function that crops up the fibonacci function. Here's imperative and recursive versions of this function. The fibonacci sequence is generated by starting with the two values 0 and 1, stored in variables s and t in the imperative version, and generating the next value, n, by summing the previous two. In the iterative version s and t always maintain the two most recent fibonacci numbers and we contine until we generated the required number. This line in the middle here, simultaneously assigns two variables and is tidier than using a temprary variable.

The functional version has a more direct implementation. Fibonacci of N is equal to the sum of the two previous Fibonacci numbers.

Both these definitions compute the same mathematical function. Unfortunately their running times are radically different. The iterative one runs in O(n) the functional one runs in exponential time. In practice on my laptop there is a small delay in computing fibf(30). Larger values, such as fibf(100) would be totally infeasible to compute whereas fibi(100) has no noticeable delay at all.

== Slide 15 ==

Because this is a talk about functional programming here I give a function time function. This is an example of a higher-order function, that is a function that returns a function. The timeit function takes a function as an argument and returns a new function that has the same

effect put also prints its running time when called. We can use this to time our two version of the Fibonacci function.

== Slide 16 ==

Here we see timeit in use. For each of fibi and fibf we create timed versions of those functions and store them in two new variables.
So we we see that these new functions perform just as the old functions but in addition report time spent using print.

A master functional programmer can show you ways to transform fibf into a version that is still functional but runs in linear time, making it about as efficient as the iterative version.

I'd like to show you something a bit less functional but more Pythonic.

== Slide 17 ==

I'm going to improve fibf by using the observation that the reason fibf is slow is that it computes previous Fibonacci numbers many times over.
If we could remember the result of computing previous Fibonacci numbers then we could avoid wasted computation.

We can generalise this idea and abstract it into a function. memo is another higher order function. memo takes a function as an argument and produces a new version of the function that acts just like the old version but uses a cache to remember the results of previous calls.
Depending on what the argument function is, this may or may not improve matters. In the case of fibf it should improve matters greatly since in computing fibf(30) we will only need to compute each of the 29 previous Fibonacci numbers once.

memo works by maintain a cache, which is a Python dictionary; the cache stores previous argument/result pairs as key/value pairs in the dictionary.

== Slide 18 ==

Here we see memo being used. Just like our timeit function we apply memo to fibf to get a new function and this time store our new function in fibf, effectively overwriting the old version.

We can see from the timed version that we created that our new fibf goes essentially as fast as the iterative version. One application of memo has speeded up our functional Fibonacci function by about 100,000 times.
It's quite rare to achieve such dramatic speed improvements!

Don't forget that the source code to fibf hasn't changed at all. We speeded it up by using the higher-order memo function.

== Slide 19 ==

Anoher popular function similar to map is filter. This function is also builtin to Python, like map. Like map is 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 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 combersome 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 were 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]

== Closing ==

I'd like to thank the conference organisers, the chair Simon in particular for keeping us all to schedule, and I hope you enjoy the rest of the conference.

HOMEWORK

Stuff that I cut from my original plan, but perhaps you can investigate on your own time:

reduce
  note: string.join(list, s) is reduce(lambda l,r : l + s + r, list)
factorial again via reduce
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
Strings are sequences: map(ord, 'Hello')

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.