Charming Python: Functional programming in Python, Part 2

Wading into functional programming?

David Mertz (mertz@gnosis.cx)
Applied Metaphysician
Gnosis Software, Inc.

01 April 2001

This column continues David's introduction to functional programming (FP) in Python. Enjoy this introduction to different paradigms of program problem-solving, where David demonstrates several intermediate and advanced FP concepts.
An object is a piece of data with procedures attached to it... A closure is a procedure with a piece of data attached to it.

View more content in this series

In Part 1, my previous column on functional programming, I introduced some basic concepts of FP. This column will delve a little bit deeper into this quite rich conceptual realm. For much of our delving, Bryn Keller's "Xoltar Toolkit" will provide valuable assistance. Keller has collected many of the strengths of FP into a nice little module containing pure Python implementations of the techniques. In addition to the module `functional`, Xoltar Toolkit includes the `lazy` module, which supports structures that evaluate "only when needed." Many traditionally functional languages also have lazy evaluation, so between these components, the Xoltar Toolkit lets you do much of what you might find in a functional language like Haskell.

Bindings

Alert readers will remember a limitation that I pointed out in the functional techniques described in Part 1. Specifically, nothing in Python prevents the rebinding of names that are used to denote functional expressions. In FP, names are generally understood to be abbreviations of longer expressions, but the promise is implicit that "the same expression will always evaluate to the same result." If denotational names get rebound, the promise is broken. For example, let's say that we define some shorthand expressions that we'd like to use in our functional program, such as:

Listing 1. Python FP session with rebinding causing mischief

```
>>> car = lambda lst: lst[0]
>>> cdr = lambda lst: lst[1:]
>>> sum2 = lambda lst: car(lst)+car(cdr(lst))
>>> sum2(range(10))
1
>>> car = lambda lst: lst[2]
>>> sum2(range(10))
5
```

Unfortunately, the very same expression `sum2(range(10))` evaluates to two different things at two points in our program, even though this expression itself does not use any mutable variables in its arguments.

The module *`functional`*, fortunately, provides a class called `Bindings` (proposed to Keller by yours truly) that prevents such rebindings (at least accidentally, Python does not try to prevent a determined programmer who wants to break things). While use of `Bindings` requires a little extra syntax, it makes it difficult for accidents to happen. In his examples within the *`functional`* module, Keller names a `Bindings` instance `let` (I presume after the `let` keyword in ML-family languages). For example, we might do:

Listing 2. Python FP session with guarded rebinding

```
>>> from functional import *
>>> let = Bindings()
>>> let.car = lambda lst: lst[0]
>>> let.car = lambda lst: lst[2]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "d:\tools\functional.py", line 976, in __setattr__
    raise BindingError, "Binding '%s' cannot be modified." % name
functional.BindingError:  Binding 'car' cannot be modified.
>>> car(range(10))
0
```

Obviously, a real program would have to do something about catching these "BindingError"s, but the fact they are raised avoids a class of problems.

Along with `Bindings`, *`functional`* provides a `namespace` function to pull off a namespace (really, a dictionary) from a `Bindings` instance. This comes in handy if you want to compute an expression within a (immutable) namespace defined in a `Bindings`. The Python function `eval()` allows evaluation within a namespace. An example should clarify:

Listing 3. Python FP session using immutable namespaces

```
>>> let = Bindings()      # "Real world" function names
>>> let.r10 = range(10)
>>> let.car = lambda lst: lst[0]
>>> let.cdr = lambda lst: lst[1:]
>>> eval('car(r10)+car(cdr(r10))', namespace(let))
>>> inv = Bindings()      # "Inverted list" function names
>>> inv.r10 = let.r10
>>> inv.car = lambda lst: lst[-1]
>>> inv.cdr = lambda lst: lst[:-1]
>>> eval('car(r10)+car(cdr(r10))', namespace(inv))
17
```

Closures

One very interesting concept in FP is a *closure*. In fact, closures are sufficiently interesting to many developers that even generally non-functional languages like Perl and Ruby include closures as a feature. Moreover, Python 2.1 currently appears destined to add lexical scoping, which will provide most of the capabilities of closures.

So what *is* a closure, anyway? Steve Majewski has recently provided a nice characterization of the concept on the Python newsgroup:

That is, a closure is something like FP's Jekyll to OOP's Hyde (or perhaps the roles are the other way around). A closure, like an object instance, is a way of carrying around a bundle of data and functionality, wrapped up together.

Let's step back just a bit to see what problem both objects and closures solve, and also to see how the problem can be solved without either. The result returned by a function is usually determined by the context used in its calculation. The most common -- and perhaps the most obvious -- way of specifying this context is to pass some arguments to the function that tell it what values it should operate on. But sometimes also, there is a natural distinction between "background" and "foreground" arguments -- between what the function is doing this particular time, and the way the function is "configured" for multiple potential calls.

There are a number of ways to handle background, while focussing on foreground. One way is to simply "bite the bullet" and, at every invocation, pass every argument a function needs. This often amounts to passing a number of values (or a structure with multiple slots) up and down a call chain, on the possibility the values will be needed somewhere in the chain. A trivial example might look like:

Listing 4. Python session showing cargo variable

```
>>> def a(n):
...     add7 = b(n)
...     return add7
...
>>> def b(n):
...     i = 7
...     j = c(i,n)
...     return j
...
>>> def c(i,n):
...     return i+n
...
>>> a(10)     # Pass cargo value for use downstream
17
```

In the cargo example, within `b()`, `n` has no purpose other than being available to pass on to `c()`. Another option is to use global variables:

Listing 5. Python session showing global variable

```
>>> N = 10
>>> def addN(i):
...     global N
...     return i+N
...
>>> addN(7)   # Add global N to argument
17
>>> N = 20
>>> addN(6)   # Add global N to argument
26
```

The global `N` is simply available whenever you want to call `addN()`, but there is no need to pass the global background "context" explicitly. A somewhat more Pythonic technique is to "freeze" a variable into a function using a default argument at definition time:

Listing 6. Python session showing frozen variable

```
>>> N = 10
>>> def addN(i, n=N):
...     return i+n
...
>>> addN(5)   # Add 10
15
>>> N = 20
>>> addN(6)   # Add 10 (current N doesn't matter)
16
```

Our frozen variable is essentially a closure. Some data is "attached" to the `addN()` function. For a complete closure, all the data present when `addN()` was defined would be available at invocation. However, in this example (and many more robust ones), it is simple to make *enough* available with default arguments. Variables that are never used by `addN()` thereby make no difference to its calculation.

Let's look next at an OOP approach to a slightly more realistic problem. The time of year has prompted my thoughts about those "interview" style tax programs that collect various bits of data -- not necessarily in a particular order -- then eventually use them all for a calculation. Let's create a simplistic version of this:

Listing 7. Python-style tax calculation class/instance

```
class TaxCalc:
    deftaxdue(self):return (self.income-self.deduct)*self.rate
taxclass = TaxCalc()
taxclass.income = 50000
taxclass.rate = 0.30
taxclass.deduct = 10000
print"Pythonic OOP taxes due =", taxclass.taxdue()
```

In our `TaxCalc` class (or rather, in its instance), we can collect some data -- in whatever order we like -- and once we have all the elements needed, we can call a method of this object to perform a calculation on the bundle of data. Everything stays together within the instance, and further, a different instance can carry a different bundle of data. The possibility of creating multiple instances, differing only their data is something that was not possible in the "global variable" or "frozen

variable" approaches. The "cargo" approach can handle this, but for the expanded example, we can see it might become necessary to start passing around numerous values. While we are here, it is interesting to note how a message-passing OOP style might approach this (Smalltalk or Self are similar to this, and so are several OOP xBase variants I have used):

Listing 8. Smalltalk-style (Python) tax calculation

```
class TaxCalc:
    deftaxdue(self):return (self.income-self.deduct)*self.rate
    def setIncome(self,income):
        self.income = income
        return self
    def setDeduct(self,deduct):
        self.deduct = deduct
        return self
    def setRate(self,rate):
        self.rate = rate
        return self
print"Smalltalk-style taxes due =", \
      TaxCalc().setIncome(50000).setRate(0.30).setDeduct(10000).taxdue()
```

Returning `self` with each "setter" allows us to treat the "current" thing as a result of every method application. This will have some interesting similarities to the FP closure approach.

With the Xoltar toolkit, we can create full closures that have our desired property of combining data with a function, and also allowing multiple closure (nee objects) to contain different bundles:

Listing 9. Python Functional-style tax calculations

```
from functional import *

taxdue        = lambda: (income-deduct)*rate
incomeClosure = lambda income,taxdue: closure(taxdue)
deductClosure = lambda deduct,taxdue: closure(taxdue)
rateClosure   = lambda rate,taxdue: closure(taxdue)

taxFP = taxdue
taxFP = incomeClosure(50000,taxFP)
taxFP = rateClosure(0.30,taxFP)
taxFP = deductClosure(10000,taxFP)
print"Functional taxes due =",taxFP()

print"Lisp-style taxes due =", \
      incomeClosure(50000,
          rateClosure(0.30,
              deductClosure(10000, taxdue)))()
```

Each closure function we have defined takes any values defined within the function scope, and binds those values into the global scope of the function object. However, what appears as the function's global scope is not necessarily the same as the true module global scope, nor identical to a different closure's "global" scope. The closure simply "carries the data" with it.

In our example, we utilize a few particular functions to put specific bindings within a closure's scope (income, deduct, rate). It would be simple enough to modify the design to put any arbitrary

binding into scope. We also -- just for the fun of it -- use two slightly different functional styles in the example. The first successively binds additional values into closure scope; by allowing `taxFP` to be mutable, these "add to closure" lines can appear in any order. However, if we were to use immutable names like `tax_with_Income`, we would have to arrange the binding lines in a specific order, and pass the earlier bindings to the next ones. In any case, once everything necessary is bound into closure scope, we can call the "seeded" function.

The second style looks a bit more like Lisp, to my eyes (the parentheses mostly). Beyond the aesthetic, two interesting things happen in the second style. The first is that name binding is avoided altogether. This second style is a single expression, with no statements used (see Part 1 for a discussion of why this matters).

The other interesting thing about the "Lisp-style" use of the closures is how much it resembles the "Smalltalk-style" message-passing methods given above. Both essentially accumulate values along the way to calling the `taxdue()` function/method (both will raise errors in these crude versions if the right data is not available). The "Smalltalk-style" passes an object between each step, while the "Lisp-style" passes a continuation. But deep down, functional and object-oriented programming amount to much the same thing.

Tail recursion

In this installment, we have knocked off a bit more of the domain of functional programming. What remains is less (and provably simpler?) than before (the title of the section is a minor joke; unfortunately, its concept is not explained herein). Reading the *`functional`* module's source is an excellent way to continue exploring a number of FP concepts. The module is very well commented, and provides examples for most of its functions/classes. Not covered in this column are a number of simplifying meta-functions that make the combinations and interaction of other functions simpler to handle. These are definitely worth examining for a Python programmer seeking to continue the exploration of functional paradigms.

Resources

Learn

- Read all three parts in this series.
- Read more installments of *Charming Python*.
- Bryn Keller's "xoltar toolkit", which includes the module `functional`, adds a large number of useful FP extensions to Python. Since the `functional` module is itself written entirely in Python, what it does was already possible in Python itself. But Keller has figured out a very nicely integrated set of extensions, with a lot of power in compact definitions.
- Peter Norvig has written an interesting article, *Python for Lisp Programmers*. While his focus is somewhat the reverse of my column, it provides very good general comparisons between Python and Lisp.
- A good starting point for functional programming is the Frequently Asked Questions for comp.lang.functional.
- I've found it much easier to get a grasp of functional programming in the language Haskell than in Lisp/Scheme (even though the latter is probably more widely used, if only in Emacs). Other Python programmers might similarly have an easier time without quite so many parentheses and prefix (Polish) operators.
- An excellent introductory book is *Haskell: The Craft of Functional Programming (2nd Edition)*, Simon Thompson (Addison-Wesley, 1999).
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools, as well as IT industry trends.
- Watch developerWorks on-demand demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.
- Follow developerWorks on Twitter, or subscribe to a feed of Linux tweets on developerWorks.

Get products and technologies

- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

David Mertz

Since conceptions without intuitions are empty, and intuitions without conceptions, blind, David Mertz wants a cast sculpture of Milton for his office. Start planning for his birthday. David may be reached at mertz@gnosis.cx; his life pored over at http://gnosis.cx/dW/. Suggestions and recommendations on this, past, or future, columns are welcomed.