developerWorks®

Charming Python: Functional programming in Python, Part 3

Currying and other higher-order functions

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

01 June 2001

Author David Mertz touched on many basic concepts of functional programming in earlier Charming Python articles: "Functional programming in Python", Part 1 and Part 2. Here he continues the discussion by illustrating additional capabilities, like currying and other higher-order functions contained in the Xoltar Toolkit.

View more content in this series

Expression bindings

Never content with partial solutions, one reader -- Richard Davies -- raised the issue of whether we might move bindings all the way into individual expressions. Let's take a quick look at why we might want to do that, and also show a remarkably elegant means of expression provided by a comp.lang.python contributor.

Let's first recall the `Bindings` class of the `functional` module. Using the attributes of that class, we were able to assure that a particular name means only one thing within a given block scope:

Listing 1: 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.
>>> let.car(range(10))
0
```

The `Bindings` class does what we want within a module or function `def` scope, but there is no way to make it work within a single expression. In ML-family languages, however, it is natural to create bindings within a single expression:

Listing 2: Haskell expression-level name bindings

```
-- car (x:xs) = x  -- *could* create module-level binding
list_of_list = [[1,2,3],[4,5,6],[7,8,9]]

-- 'where' clause for expression-level binding
firsts1 = [car x | x <- list_of_list] where car (x:xs) = x

-- 'let' clause for expression-level binding
firsts2 = let car (x:xs) = x in [car x | x <- list_of_list]

-- more idiomatic higher-order 'map' technique
firsts3 = map car list_of_list where car (x:xs) = x

-- Result: firsts1 == firsts2 == firsts3 == [1,4,7]
```

Greg Ewing observed that it is possible to accomplish the same effect using Python's list comprehensions; we can even do it in a way that is nearly as clean as Haskell's syntax:

Listing 3: Python 2.0+ expression-level name bindings

```
>>> list_of_list = [[1,2,3],[4,5,6],[7,8,9]]
>>> [car_x for x in list_of_list for car_x in
 (x[0],)]
[1, 4, 7]
```

This trick of putting an expression inside a single-item tuple in a list comprehension does not provide any way of using expression-level bindings with higher-order functions. To use the higher-order functions, we still need to use block-level bindings, as with:

Listing 4: Python block-level bindings with 'map()'

```
>>> list_of_list = [[1,2,3],[4,5,6],[7,8,9]]
>>> let = Bindings()
>>> let.car = lambda l: l[0]
>>> map(let.car,list_of_list)
[1, 4, 7]
```

Not bad, but if we want to use `map()`, the scope of the binding remains a little broader than we might want. Nonetheless, it is possible to coax list comprehensions into doing our name bindings for us, even in cases where a list is not what we finally want:

Listing 5: "Stepping down" from Python list comprehension

```
# Compare Haskell expression:
# result = func car_car
#          where
#              car (x:xs) = x
#              car_car = car (car list_of_list)
#              func x = x + x^2
>>> [func for x in list_of_list
...
for car in (x[0],)
...
for func in (car+car**2,)][0]
2
```

We have performed an arithmetic calculation on the first element of the first element of `list_of_list` while also naming the arithmetic calculation (but only in expression scope). As an "optimization" we might not bother to create a list longer than one element to start with, since we choose only the first element with the ending index `0`:

Listing 6: Efficient stepping down from list comprehension

```
>>> [func for x in list_of_list[:1]
...       for car in (x[0],)
...       for func in (car+car**2,)][0]
2
```

Higher-order functions: currying

Three of the most general higher-order functions are built into Python: `map()`, `reduce()`, and `filter()`. What these functions do -- and the reason we call them "higher-order" -- is take other functions as (some of) their arguments. Other higher-order functions, but not these built-ins, return function objects.

Python has always given users the ability to construct their own higher-order functions by virtue of the first-class status of function objects. A trivial case might look like this:

Listing 7: Trivial Python function factory

```
>>> def foo_factory():
...
def foo():
...
print
"Foo function from factory"
...
return foo
...
>>> f = foo_factory()
>>> f()
Foo function from factory
```

The Xoltar Toolkit, which I discussed in Part 2 of this series, comes with a nice collection of higher-order functions. Most of the functions that Xoltar's `functional` module provides are ones developed in various traditionally functional languages, and whose usefulness have been proven over many years.

Possibly the most famous and most important higher-order function is `curry()`. `curry()` is named after the logician Haskell Curry, whose first name is also used to name the above-mentioned programming language. The underlying insight of "currying" is that it is possible to treat (almost) every function as a partial function of just one argument. All that is necessary for currying to work is to allow the return value of functions to themselves be functions, but with the returned functions "narrowed" or "closer to completion." This works quite similarly to the closures I wrote about in Part 2 -- each successive call to a curried return function "fills in" more of the data involved in a final computation (data attached to a procedure).

Let's illustrate currying first with a very simple example in Haskell, then with the same example repeated in Python using the `functional` module:

Listing 8: Currying a Haskell computation

```
computation a b c d = (a + b^2+ c^3 + d^4)
check = 1 + 2^2 + 3^3 + 5^4

fillOne   = computation 1
-- specify "a"
fillTwo   = fillOne 2
-- specify "b"
fillThree = fillTwo 3
-- specify "c"
answer    = fillThree 5
-- specify "d"
-- Result: check == answer == 657
```

Now in Python:

Listing 9: Currying a Python computation

```
>>> from functional import curry
>>> computation = lambda a,b,c,d: (a + b**2 + c**3 + d**4)
>>> computation(1,2,3,5)
657
>>> fillZero  = curry(computation)
>>> fillOne   = fillZero(1)
# specify "a"
>>> fillTwo   = fillOne(2)
# specify "b"
>>> fillThree = fillTwo(3)
# specify "c"
>>> answer    = fillThree(5)
# specify "d"
>>> answer
657
```

It is possible to further illustrate the parallel with closures by presenting the same simple tax-calculation program used in Part 2 (this time using `curry()`):

Listing 10: Python curried tax calculations

```
from functional import *

taxcalc = lambda income,rate,deduct: (income-(deduct))*rate

taxCurry = curry(taxcalc)
taxCurry = taxCurry(50000)
taxCurry = taxCurry(0.30)
taxCurry = taxCurry(10000)
print "Curried taxes due =",taxCurry

print "Curried expression taxes due =", \
      curry(taxcalc)(50000)(0.30)(10000)
```

Unlike with closures, we need to curry the arguments in a specific order (left to right). But note that `functional` also contains an `rcurry()` class that will start at the other end (right to left).

The second `print` statement in the example at one level is a trivial spelling change from simply calling the normal `taxcalc(50000,0.30,10000)`. In a different level, however, it makes rather clear the concept that every function can be a function of just one argument -- a rather surprising idea to those new to it.

Miscellaneous higher-order functions

Beyond the "fundamental" operation of currying, `functional` provides a grab-bag of interesting higher-order functions. Moreover, it is really not hard to write your own higher-order functions -- either with or without `functional`. The ones in `functional` provide some interesting ideas, at the least.

For the most part, higher-order functions feel like "enhanced" versions of the standard `map()`, `filter()`, and `reduce()`. Often, the pattern in these functions is roughly "take a function or functions and some lists as arguments, then apply the function(s) to list arguments." There are a surprising number of interesting and useful ways to play on this theme. Another pattern is "take a collection of functions and create a function that combines their functionality." Again, numerous variations are possible. Let's look at some of what `functional` provides.

The functions `sequential()` and `also()` both create a function based on a sequence of component functions. The component functions can then be called with the same argument(s). The main difference between the two is simply that `sequential()` expects a single list as an argument, while `also()` takes a list of arguments. In most cases, these are useful for function side effects, but `sequential()` optionally lets you choose which function provides the combined return value:

Listing 11: Sequential calls to functions (with same args)

```
>>> def a(x):
...     print x,
...     return "a"
...
>>> def b(x):
...     print x*2,
...     return "b"
...
>>> def c(x):
...     print x*3,
...     return "c"
...
>>> r = also(a,b,c)
>>> r
<functional.sequential instance at 0xb86ac>
>>> r(5)
5 10 15
'a'
>>> sequential([a,b,c],main=c)('x')
x xx xxx
'c'
```

The functions `disjoin()` and `conjoin()` are similar to `sequential()` and `also()` in terms of creating new functions that apply argument(s) to several component functions. But `disjoin()` asks whether *any* component functions return true (given the argument(s)), and `conjoin()` asks whether *all* components return true. Logical shortcutting is applied, where possible, so some side effects might not occur with `disjoin()`. `joinfuncs()` is similar to `also()`, but returns a tuple of the components' return values rather than selecting a main one.

Where the previous functions let you call multiple functions with the same argument(s), `any()`, `all()`, and `none_of()` let you call the same function against a list of arguments. In general structure, these are a bit like the built-in `map()`, `reduce()`, `filter()` functions. But these particular higher-order functions from `functional` ask Boolean questions about collections of return values. For example:

Listing 12: Ask about collections of return values

```
>>> from functional import *
>>> isEven = lambda n: (n%2 == 0)
>>> any([1,3,5,8], isEven)
1
>>> any([1,3,5,7], isEven)
0
>>> none_of([1,3,5,7], isEven)
1
>>> all([2,4,6,8], isEven)
1
>>> all([2,4,6,7], isEven)
0
```

A particularly interesting higher-order function for those with a little bit of mathematics background is `compose()`. The composition of several functions is a "chaining together" of the return value of one function to the input of the next function. The programmer who composes several functions is responsible for making sure the outputs and inputs match up -- but then, that is true any time a programmer uses a return value. A simple example makes it clear:

Listing 13: Creating compositional functions

```
>>> def minus7(n): return n-7
...
>>> def times3(n): return n*3
...
>>> minus7(10)
3
>>> minustimes = compose(times3,minus7)
>>> minustimes(10)
9
>>> times3(minus7(10))
9
>>> timesminus = compose(minus7,times3)
>>> timesminus(10)
23
>>> minus7(times3(10))
23
```

Until next time

I hope this latest look at higher-order functions will arouse readers' interest in a certain style of thinking. By all means, play with it. Try to create some of your own higher-order functions; some might well prove useful and powerful. Let me know how it goes; perhaps a later installment of this ad hoc series will discuss the novel and fascinating ideas that readers continue to provide.

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.
- 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 welcome.

© Copyright IBM Corporation 2001
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)