Charming Python #13 (20000155)

Functional Programming in Python

David Mertz, Ph.D.
Applied Metaphysician, Gnosis Software, Inc.
January, 2001

Although users usually think of Python has a procedural and object oriented language, it actually contains everything one needs for a completely functional approach to programming. This article discusses general concepts of functional programming, and illustrates ways of implementing functional techniques in Python.

What Is Python?

Python is a freely available, very-high-level, interpreted language developed by Guido van Rossum. It combines a clear syntax with powerful (but optional) object-oriented semantics. Python is available for almost every computer platform you might find yourself working on, and has strong portability between platforms.

What Is Functional Programming?

We better start with the hardest question: "What is functional programming (FP), anyway?" One approach would be to say that FP is what you do when you program in languages like Lisp, Scheme, Haskell, ML, OCAML, Clean, Mercury or Erlang (or a few others). That is a safe answer, but not one that clarifies very much. Unfortunately, it is hard to get a consistent opinion on just what FP is, even from functional programmers themselves. A story about elephants and blind men seems apropos here. It is also safe to contrast FP with "imperative programming" (what one does in languages like C, Pascal, C++, Java, Perl, Awk, TCL, and most others, at least for the most part).

While the author by all means welcomes the advice of those who know better, he would roughly characterize functional programming as having at least several of the following characteristics. Languages that get called functional make these things easy, and other things either hard or disallowed:

* Functions are first class (objects). That is, everything you can do with "data" can be done with functions themselves (such as passing a function to another function).

* Use of recursion as a primary control structure. In some languages, no other "loop" construct exists other than recursion.

* Focus on LISt Processing (e.g. the name `Lisp`). Lists are often used with recursion on sub-lists as a substitute for loops.

* "Pure" functional languages eschew side-effects. This excludes the almost ubiquitous pattern in imperative languages of assigning first one, then another value, to the same variable to track

program state.

* FP either discourages or outright disallows *statements*, and instead works with the evaluation of expressions (i.e. functions plus arguments). In the pure case, one program is one expression (plus supporting definitions).

* FP worries about *what* is to be computed rather than *how* it is to be computed.

* Much FP utilizes "higher order" functions (i.e. functions that operate on functions that operate on functions).

Advocates of functional programming argue that all these characteristic make for more rapidly developed, shorter, and less bug-prone code. Moreover, high theorists of computer science, logic, and math find it a lot easier to prove formal properties of functional languages and programs than of imperative languages and programs.

Inherent Python Functional Capabilities

Python has had most of the characteristics of FP listed above since Python 1.0. But as with most Python features, they have been present in a very mixed language. Much as with Python's OOP features, you can use what you want and ignore the rest (until you need it later). With Python 2.0, a *very* nice bit of "syntactic sugar" was added with *list comprehensions*. While list comprehensions add no entirely new capability, they make a lot of the old capabilities look *a lot* nicer.

The basic elements of FP in Python are the functions `map()`, `reduce()` and `filter()`, and the operator `lambda`. In Python 1.x, the `apply()` function also comes in handy for direct application of one function's list return value to another function. Python 2.0 provides an improved syntax for this purpose. Perhaps surprisingly, these very few functions (and the basic operators), are almost sufficient to write any Python program; specifically, the flow control statements (`if`, `elif`, `else`, `assert`, `try`, `except`, `finally`, `for`, `break`, `continue`, `while`, `def`) can all be handled in a functional style using exclusively the FP functions and operators. While actually eliminating all flow control commands in a program is probably only useful for entering an "obfuscated Python" contest (with code that will look a lot like Lisp), it is worth understanding how FP expresses flow control with functions and recursion.

Eliminating Flow Control Statements

The first thing to think about in our elimination exercise is the fact that Python "short circuits" evaluation of boolean expressions. This turns out to provide an expression version of `if`/ `elif`/ `else` blocks (assuming each block calls one function, which is always possible to arrange). Here is how:

"Short-circuit" conditional calls in Python

```
# Normal statement-based flow control
if <cond1>:   func1()
elif <cond2>: func2()
else:         func3()

# Equivalent "short circuit" expression
(<cond1> and func1()) or (<cond2> and func2()) or (func3())

# Example "short circuit" expression
>>> x = 3
>>> def pr(s): return s
>>> (x==1 and pr('one')) or (x==2 and pr('two')) or (pr('other'))
'other'
>>> x = 2
>>> (x==1 and pr('one')) or (x==2 and pr('two')) or (pr('other'))
'two'
```

Our expression version of conditional calls might seem to be nothing but a parlor trick; however, it is more interesting when we notice that the `lambda` operator must return an expression. Since--as we have shown--expressions can contain conditional blocks via short circuiting, a `lambda` expression is fully general in expressing conditional return values. Building on our example:

Lambda with short-circuiting in Python

```
>>> pr = lambda s:s
>>> namenum = lambda x: (x==1 and pr("one")) \
...                  or (x==2 and pr("two")) \
...                  or (pr("other"))
>>> namenum(1)
'one'
>>> namenum(2)
'two'
>>> namenum(3)
'other'
```

Functions As First Class Objects

The above examples have already witnessed the first class status of functions in Python, but in a subtle way. When we create a *function object* with the `lambda` operation we have something entirely general. As such, we were able to bind our objects to the names "pr" and "namenum", in exactly the same way we might have bound the number 23 or the string "spam" to those names. But just like we can use the number 23 without binding it to any name (i.e. as a function argument), we can use the function object we created with `lambda` without binding it to any name. A function is simply another value we might do something with in Python.

The main thing we do with our first class objects, is pass them to our FP builtin functions `map()`, `reduce()` and `filter()`. Each of these functions accepts a function object as its first argument. `map()` performs the passed function on each corresponding item in the specified list(s), and returns a list of results. `reduce()` performs the passed function on each subsequent item and an internal accumulator of a final result; for example, `reduce(lambda n,m:n*m, range(1,10))` means "factorial of 10" (i.e. multiply each item by the product of previous multiplications). `filter()` uses the passed function to "evaluate" each item in a list, and return a winnowed list of the items that pass the function test. We also often pass function objects to our own custom functions, but usually those amount to combinations of the mentioned builtins.

By combining these three FP builtin functions, a surprising range of "flow" operations can be performed (all without statements, only expressions).

Functional Looping In Python

Replacing loops is as simple as was replacing conditional blocks. `for` can be directly translated to `map()`. As with our conditional execution, we will need to simplify statement blocks to single function calls (we are getting close to being able to generally):

Functional 'for' looping in Python

```
for e in lst:  func(e)      # statement-based loop
map(func,lst)               # map()-based loop
```

By the way, a similar technique is available for a functional approach to sequential program flow. That is, imperative programming mostly consists of statements that amount to "do this, then do that, then do the other thing." `map()` lets us do just this:

Functional sequential actions in Python

```
# let's create an execution utility function
do_it = lambda f: f()

# let f1, f2, f3 (etc) be functions that perform actions
map(do_it, [f1,f2,f3])    # map()-based action sequence
```

In general, the whole of our main program can be a `map()` expression with a list of functions to execute to complete the program. Another handy feature of first class functions is that you can put them in a list.

Translating `while` is slightly more complicated, but is still possible directly:

Functional 'while' looping in Python

```
# statement-based while loop
while <cond>:
    <pre-suite>
    if <break_condition>:
        break
    else:
        <suite>

# FP-style recursive while loop
def while_block():
    <pre-suite>
    if <break_condition>:
        return 1
    else:
        <suite>
    return 0

while_FP = lambda: (<cond> and while_block()) or while_FP()
while_FP()
```

Our translation of `while` still requires a `while_block()` function that may itself contain statements rather than just expressions. But we might be able to apply further eliminations to that function (such as short circuiting the `if/else` in the template. Also, it is hard for <cond> to be useful with the usual tests, such as `while myvar==7`, since the loop body (by design) cannot change any variable values (well, globals could be modified in `while_block()`). One way to add a more useful condition is to let `while_block()` return a more interesting value, and compare that return for a termination condition. It is worth looking at a concrete example of eliminating statements:

Functional 'echo' loop in Python

```
# imperative version of "echo()"
def echo_IMP():
    while 1:
        x = raw_input("IMP -- ")
        if x == 'quit':
            break
        else:
            print x
echo_IMP()

# utility function for "identity with side-effect"
def monadic_print(x):
    print x
    return x

# FP version of "echo()"
echo_FP = lambda: monadic_print(raw_input("FP -- "))=='quit' or echo_FP()
echo_FP()
```

What we have accomplished is that we have managed to express a little program that involves I/O, looping, and conditional statments as a pure expression with recursion (in fact, as a function object that can be passed elsewhere if desired). We *do* still utilize the utility function `monadic_print()`, but this function is completely general, and can be reused in every functional program expression we might create later (it's a one-time cost). Notice that any expression containing `monadic_print(x)` *evaluates* to the same thing as if it had simply contained `x`. FP (particularly Haskell) has the notion of a "monad" for a function that "does nothing, and has a side effect in the process."

Eliminating Side-effects

After all this work in getting rid of perfectly sensible statements and substituting obscure nested expressions for them, a naturaly question is "Why?!" Reading over my descriptions of FP, we can see that all of them are achieved in Python. But the most important characteristic--and the one likely to be concretely useful--is the elimination of side-effects (or at least their containment to special areas like monads). A very large percentage of program errors--and the problem that drives programmers to debuggers--occur because variables obtain unexpected values during the course of program execution. Functional programs bypasses this particular issue by simply not assigning values to variables at all.

Let's look at a fairly ordinary bit of imperative code. The goal here is to print out a list of pairs of numbers whose product is more than 25. The numbers that make up the pairs are themselves taken from two other lists. This sort of thing is moderately similar to things that programmers actually do in segments of their programs. An imperative approach to the goal might look like:

Imperative Python code for "print big products"

```
# Nested loop procedural style for finding big products
xs = (1,2,3,4)
ys = (10,15,3,22)
bigmuls = []
# ...more stuff...
for x in xs:
    for y in ys:
        # ...more stuff...
        if x*y > 25:
            bigmuls.append((x,y))
            # ...more stuff...
# ...more stuff...
print bigmuls
```

This project is small enough that nothing is likely to go wrong. But perhaps our goal is embedded in code that accomplishes a number of other goals at the same time. The sections commented with "more stuff" are the places where side-effects are likely to lead to bugs. At any of these points, the variables `xs`, `ys`, `bigmuls`, `x`, `y` might acquire unexpected values in the hypothetical abbreviated code. Futhermore, after this bit of code is done, all the variables have values that may or may not be expected and wanted by later code. Obviously, encapsulation in functions/instances and care as to scoping can be used to guard against this type of error. And you can always `del` your variables when you are done with them. But in practice, the types of errors pointed to are common.

A functional approach to our goal eliminates these side-effect errors altogether. A possible bit of code is:

Functional Python code for "print big products"

```
bigmuls = lambda xs,ys: filter(lambda (x,y):x*y > 25, combine(xs,ys))
combine = lambda xs,ys: map(None, xs*len(ys), dupelms(ys,len(xs)))
dupelms = lambda lst,n: reduce(lambda s,t:s+t, map(lambda l,n=n: [l]*n, lst))
print bigmuls((1,2,3,4),(10,15,3,22))
```

We bind our anonymous (`lambda`) function objects to names in the example, but that is not strictly necessary. We could instead simply nest the definitions. For readability we do it this way; but also because `combine()` is a nice utility function to have anyway (produce a list of all pairs of elements from two input lists). `dupelms()` in turn is mostly just a way of helping out `combine()`. Even though this functional example is more verbose than the imperative example, once you consider the utility functions for reuse, the new code in `bigmuls()` itself is probably slightly less than in the imperative version.

The real advantage of this functional example is that absolutely no variables change any values within it. There are no **possible** unanticipated side-effects on later code (or from earlier code). Obviously, the lack of side-effects, in itself, does not guarantee that the code is *correct*, but it is nonetheless an advantage. Notice, however, that Python (unlike many functional languages) does *not* prevent rebinding of the names `bigmuls`, `combine` and `dupelms`. If `combine()` starts meaning something different later in the program, all bets are off. One could work up a Singleton class to contain this type of immutable bindings (as, say, `s.bigmuls` and so on); but this column does not have room for that.

One thing distinctly worth noticing is that our particular goal is one tailor-made for a new feature of Python 2. Rather than either the imperative or functional examples given, the best (and functional) technique is:

List-comprehension Python code for "bigmuls"

```
print [(x,y) for x in (1,2,3,4) for y in (10,15,3,22) if x*y > 25]
```

Closure

This column has demonstrated ways to replace just about every Python flow-control construct with a functional equivalent (sparing side effects in the process). Translating a particular program efficiently takes some additional thinking, but we have seen that the functional built-ins are general and complete. In subsequent columns, we will look at more advanced techniques for functional programming; and hopefully we will be able to explore some more of the pros and cons of functional styles.

Resources

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. The toolkit can be found at:

> http://sourceforge.net/projects/xoltar-toolkit

Peter Norvig has written an interesting article, *Python for Lisp Programmers*. While the focus there is somewhat the reverse of my column, it provides very good general comparisons between Python and Lisp:

> http://www.norvig.com/python-lisp.html

A good starting point for functional programming is the *Frequently Asked Questions for comp.lang.functional* :

> http://www.cs.nott.ac.uk/~gmh//faq.html#functional-languages

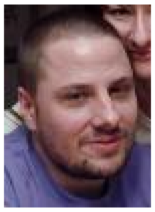
The author has found it much easier to get a grasp of functional programming via the language Haskell than in Lisp/Scheme (even though the latter is probably more widely used, if only in Emacs). Other Python programmers might similarly have an easier time without quite so many parentheses and prefix (Polish) operators.

> http://www.haskell.org/

An excellent introductory book is:

> Haskell: The Craft of Functional Programming (2nd Edition), Simon Thompson, Addison-Wesley (1999).

About The Author



Since conceptions without intuitions are empty, and intutions 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/publish/. Suggestions and recommendations on this, past, or future, columns are welcomed.