Functional Programming in Python

David Mertz

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Functional Programming in Python
by David Mertz

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Meghan Blanchette
Production Editor: Shiny Kalapurakkel
Proofreader: Charles Roumeliotis
Interior Designer: David Futato
Cover Designer: Karen Montgomery

May 2015: First Edition

Revision History for the First Edition
2015-05-27: First Release

978-1-491-92856-1

[LSI]

Table of Contents

Preface v

(Avoiding) Flow Control 1
Encapsulation 1
Comprehensions 2
Recursion 5
Eliminating Loops 7

Callables 11
Named Functions and Lambdas 12
Closures and Callable Instances 13
Methods of Classes 15
Multiple Dispatch 19

Lazy Evaluation 25
The Iterator Protocol 27
Module: itertools 29

Higher-Order Functions 33
Utility Higher-Order Functions 35
The operator Module 36
The functools Module 36
Decorators 37

Preface

What Is Functional Programming?

We'd better start with the hardest question: "What is functional programming (FP), anyway?"

One answer would be to say that functional programming is what you do when you program in languages like Lisp, Scheme, Clojure, Scala, Haskell, ML, OCAML, 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 functional programming is, even from functional programmers themselves. A story about elephants and blind men seems apropos here. It is also safe to contrast functional programming with "imperative programming" (what you do in languages like C, Pascal, C++, Java, Perl, Awk, TCL, and most others, at least for the most part). Functional programming is also not object-oriented programming (OOP), although some languages are both. And it is not Logic Programming (e.g., Prolog), but again some languages are multiparadigm.

Personally, I would roughly characterize functional programming as having at least several of the following characteristics. Languages that get called functional make these things easy, and make other things either hard or impossible:

- 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).
- Recursion is used as a primary control structure. In some languages, no other "loop" construct exists.

- There is a focus on list processing (for example, it is the source of the name Lisp). Lists are often used with recursion on sublists 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 the program state.
- Functional programming either discourages or outright disallows statements, and instead works with the evaluation of expressions (in other words, functions plus arguments). In the pure case, one program is one expression (plus supporting definitions).
- Functional programming worries about *what* is to be computed rather than *how* it is to be computed.
- Much functional programming utilizes "higher order" functions (in other words, functions that operate on functions that operate on functions).

Advocates of functional programming argue that all these characteristics 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. One crucial concept in functional programming is that of a "pure function"—one that always returns the same result given the same arguments—which is more closely akin to the meaning of "function" in mathematics than that in imperative programming.

Python is most definitely not a "*pure* functional programming language"; side effects are widespread in most Python programs. That is, variables are frequently rebound, mutable data collections often change contents, and I/O is freely interleaved with computation. It is also not even a "functional programming language" more generally. However, Python *is* a multiparadigm language that makes functional programming easy to do when desired, and easy to mix with other programming styles.

Beyond the Standard Library

While they will not be discussed withing the limited space of this report, a large number of useful third-party Python libraries for

functional programming are available. The one exception here is that I will discuss Matthew Rocklin's `multipledispatch` as the best current implementation of the concept it implements.

Most third-party libraries around functional programming are collections of higher-order functions, and sometimes enhancements to the tools for working lazily with iterators contained in `itertools`. Some notable examples include the following, but this list should not be taken as exhaustive:

- `pyrsistent` contains a number of immutable collections. All methods on a data structure that would normally mutate it instead return a new copy of the structure containing the requested updates. The original structure is left untouched.
- `toolz` provides a set of utility functions for iterators, functions, and dictionaries. These functions interoperate well and form the building blocks of common data analytic operations. They extend the standard libraries `itertools` and `functools` and borrow heavily from the standard libraries of contemporary functional languages.
- `hypothesis` is a library for creating unit tests for finding edge cases in your code you wouldn't have thought to look for. It works by generating random data matching your specification and checking that your guarantee still holds in that case. This is often called property-based testing, and was popularized by the Haskell library QuickCheck.
- `more_itertools` tries to collect useful compositions of iterators that neither `itertools` nor the recipes included in its docs address. These compositions are deceptively tricky to get right and this well-crafted library helps users avoid pitfalls of rolling them themselves.

Resources

There are a large number of other papers, articles, and books written about functional programming, in Python and otherwise. The Python standard documentation itself contains an excellent introduction called "Functional Programming HOWTO," by Andrew Kuchling, that discusses some of the motivation for functional programming styles, as well as particular capabilities in Python.

Mentioned in Kuchling's introduction are several very old public domain articles this author wrote in the 2000s, on which portions of this report are based. These include:

- The first chapter of my book *Text Processing in Python*, which discusses functional programming for text processing, in the section titled "Utilizing Higher-Order Functions in Text Processing."

I also wrote several articles, mentioned by Kuchling, for IBM's developerWorks site that discussed using functional programming in an early version of Python 2.x:

- Charming Python: Functional programming in Python, Part 1: Making more out of your favorite scripting language
- Charming Python: Functional programming in Python, Part 2: Wading into functional programming?
- Charming Python: Functional programming in Python, Part 3: Currying and other higher-order functions

Not mentioned by Kuchling, and also for an older version of Python, I discussed multiple dispatch in another article for the same column. The implementation I created there has no advantages over the more recent `multipledispatch` library, but it provides a longer conceptual explanation than this report can:

- Charming Python: Multiple dispatch: Generalizing polymorphism with multimethods

A Stylistic Note

As in most programming texts, a fixed font will be used both for inline and block samples of code, including simple command or function names. Within code blocks, a notional segment of pseudocode is indicated with a word surrounded by angle brackets (i.e., not valid Python), such as `<code-block>`. In other cases, syntactically valid but undefined functions are used with descriptive names, such as `get_the_data()`.

(Avoiding) Flow Control

In typical imperative Python programs—including those that make use of classes and methods to hold their imperative code—a block of code generally consists of some outside loops (`for` or `while`), assignment of state variables within those loops, modification of data structures like dicts, lists, and sets (or various other structures, either from the standard library or from third-party packages), and some branch statements (`if`/`elif`/`else` or `try`/`except`/`finally`). All of this is both natural and seems at first easy to reason about. The problems often arise, however, precisely with those side effects that come with state variables and mutable data structures; they often model our concepts from the physical world of containers fairly well, but it is also difficult to reason accurately about what state data is in at a given point in a program.

One solution is to focus not on constructing a data collection but rather on describing "what" that data collection consists of. When one simply thinks, "Here's some data, what do I need to do with it?" rather than the mechanism of constructing the data, more direct reasoning is often possible. The imperative flow control described in the last paragraph is much more about the "how" than the "what" and we can often shift the question.

Encapsulation

One obvious way of focusing more on "what" than "how" is simply to refactor code, and to put the data construction in a more isolated place—i.e., in a function or method. For example, consider an existing snippet of imperative code that looks like this:

```
# configure the data to start with
collection = get_initial_state()
state_var = None
for datum in data_set:
    if condition(state_var):
        state_var = calculate_from(datum)
        new = modify(datum, state_var)
        collection.add_to(new)
    else:
        new = modify_differently(datum)
        collection.add_to(new)

# Now actually work with the data
for thing in collection:
    process(thing)
```

We might simply remove the "how" of the data construction from the current scope, and tuck it away in a function that we can think about in isolation (or not think about at all once it is sufficiently abstracted). For example:

```
# tuck away construction of data
def make_collection(data_set):
    collection = get_initial_state()
    state_var = None
    for datum in data_set:
        if condition(state_var):
            state_var = calculate_from(datum, state_var)
            new = modify(datum, state_var)
            collection.add_to(new)
        else:
            new = modify_differently(datum)
            collection.add_to(new)
    return collection

# Now actually work with the data
for thing in make_collection(data_set):
    process(thing)
```

We haven't changed the programming logic, nor even the lines of code, at all, but we have still shifted the focus from "How do we construct `collection`?" to "What does `make_collection()` create?"

Comprehensions

Using comprehensions is often a way both to make code more compact and to shift our focus from the "how" to the "what." A comprehension is an expression that uses the same keywords as loop and conditional blocks, but inverts their order to focus on the data

rather than on the procedure. Simply changing the form of expression can often make a surprisingly large difference in how we reason about code and how easy it is to understand. The ternary operator also performs a similar restructuring of our focus, using the same keywords in a different order. For example, if our original code was:

```
collection = list()
for datum in data_set:
    if condition(datum):
        collection.append(datum)
    else:
        new = modify(datum)
        collection.append(new)
```

Somewhat more compactly we could write this as:

```
collection = [d if condition(d) else modify(d)
              for d in data_set]
```

Far more important than simply saving a few characters and lines is the mental shift enacted by thinking of what `collection` is, and by avoiding needing to think about or debug "What is the state of `collection` at this point in the loop?"

List comprehensions have been in Python the longest, and are in some ways the simplest. We now also have generator comprehensions, set comprehensions, and dict comprehensions available in Python syntax. As a caveat though, while you can nest comprehensions to arbitrary depth, past a fairly simple level they tend to stop clarifying and start obscuring. For genuinely complex construction of a data collection, refactoring into functions remains more readable.

Generators

Generator comprehensions have the same syntax as list comprehensions—other than that there are no square brackets around them (but parentheses are needed syntactically in some contexts, in place of brackets)—but they are also *lazy*. That is to say that they are merely a description of "how to get the data" that is not realized until one explicitly asks for it, either by calling `.next()` on the object, or by looping over it. This often saves memory for large sequences and defers computation until it is actually needed. For example:

```
log_lines = (line for line in read_line(huge_log_file)
                  if complex_condition(line))
```

For typical uses, the behavior is the same as if you had constructed a list, but runtime behavior is nicer. Obviously, this generator comprehension also has imperative versions, for example:

```
def get_log_lines(log_file):
    line = read_line(log_file)
    while True:
        try:
            if complex_condition(line):
                yield line
            line = read_line(log_file)
        except StopIteration:
            raise

log_lines = get_log_lines(huge_log_file)
```

Yes, the imperative version could be simplified too, but the version shown is meant to illustrate the behind-the-scenes "how" of a `for` loop over an iteratable—more details we also want to abstract from in our thinking. In fact, even using `yield` is somewhat of an abstraction from the underlying "iterator protocol." We could do this with a class that had `.__next__()` and `.__iter__()` methods. For example:

```
class GetLogLines(object):
    def __init__(self, log_file):
        self.log_file = log_file
        self.line = None
    def __iter__(self):
        return self
    def __next__(self):
        if self.line is None:
            self.line = read_line(log_file)
        while not complex_condition(self.line):
            self.line = read_line(self.log_file)
        return self.line

log_lines = GetLogLines(huge_log_file)
```

Aside from the digression into the iterator protocol and laziness more generally, the reader should see that the comprehension focuses attention much better on the "what," whereas the imperative version—although successful as refactorings perhaps—retains the focus on the "how."

Dicts and Sets

In the same fashion that lists can be created in comprehensions rather than by creating an empty list, looping, and repeatedly call-

ing `.append()`, dictionaries and sets can be created "all at once" rather than by repeatedly calling `.update()` or `.add()` in a loop. For example:

```
>>> {i:chr(65+i) for i in range(6)}
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}
>>> {chr(65+i) for i in range(6)}
{'A', 'B', 'C', 'D', 'E', 'F'}
```

The imperative versions of these comprehensions would look very similar to the examples shown earlier for other built-in datatypes.

Recursion

Functional programmers often put weight in expressing flow control through recursion rather than through loops. Done this way, we can avoid altering the state of any variables or data structures within an algorithm, and more importantly get more at the "what" than the "how" of a computation. However, in considering using recursive styles we should distinguish between the cases where recursion is just "iteration by another name" and those where a problem can readily be partitioned into smaller problems, each approached in a similar way.

There are two reasons why we should make the distinction mentioned. On the one hand, using recursion effectively as a way of marching through a sequence of elements is, while possible, really not "Pythonic." It matches the style of other languages like Lisp, definitely, but it often feels contrived in Python. On the other hand, Python is simply comparatively slow at recursion, and has a limited stack depth limit. Yes, you can change this with `sys.setrecursionlimit()` to more than the default 1000; but if you find yourself doing so it is probably a mistake. Python lacks an internal feature called *tail call elimination* that makes deep recursion computationally efficient in some languages. Let us find a trivial example where recursion is really just a kind of iteration:

```
def running_sum(numbers, start=0):
    if len(numbers) == 0:
        print()
        return
    total = numbers[0] + start
    print(total, end=" ")
    running_sum(numbers[1:], total)
```

There is little to recommend this approach, however; an iteration that simply repeatedly modified the `total` state variable would be more readable, and moreover this function is perfectly reasonable to want to call against sequences of much larger length than 1000. However, in other cases, recursive style, even over sequential operations, still expresses algorithms more intuitively and in a way that is easier to reason about. A slightly less trivial example, factorial in recursive and iterative style:

```
def factorialR(N):
    "Recursive factorial function"
    assert isinstance(N, int) and N >= 1
    return 1 if N <= 1 else N * factorialR(N-1)

def factorialI(N):
    "Iterative factorial function"
    assert isinstance(N, int) and N >= 1
    product = 1
    while N >= 1:
        product *= N
        N -= 1
    return product
```

Although this algorithm can also be expressed easily enough with a running product variable, the recursive expression still comes closer to the "what" than the "how" of the algorithm. The details of repeatedly changing the values of `product` and `N` in the iterative version feels like it's just bookkeeping, not the nature of the computation itself (but the iterative version is probably faster, and it is easy to reach the recursion limit if it is not adjusted).

As a footnote, the fastest version I know of for `factorial()` in Python is in a functional programming style, and also expresses the "what" of the algorithm well once some higher-order functions are familiar:

```
from functools import reduce
from operator import mul
def factorialHOF(n):
    return reduce(mul, range(1, n+1), 1)
```

Where recursion is compelling, and sometimes even the only really obvious way to express a solution, is when a problem offers itself to a "divide and conquer" approach. That is, if we can do a similar computation on two halves (or anyway, several similarly sized chunks) of a larger collection. In that case, the recursion depth is only O(log N) of the size of the collection, which is unlikely to be

overly deep. For example, the quicksort algorithm is very elegantly expressed without any state variables or loops, but wholly through recursion:

```
def quicksort(lst):
    "Quicksort over a list-like sequence"
    if len(lst) == 0:
        return lst
    pivot = lst[0]
    pivots = [x for x in lst if x == pivot]
    small = quicksort([x for x in lst if x < pivot])
    large = quicksort([x for x in lst if x > pivot])
    return small + pivots + large
```

Some names are used in the function body to hold convenient values, but they are never mutated. It would not be as readable, but the definition could be written as a single expression if we wanted to do so. In fact, it is somewhat difficult, and certainly less intuitive, to transform this into a stateful iterative version.

As general advice, it is good practice to look for possibilities of recursive expression—and especially for versions that avoid the need for state variables or mutable data collections—whenever a problem looks partitionable into smaller problems. It is not a good idea in Python—most of the time—to use recursion merely for "iteration by other means."

Eliminating Loops

Just for fun, let us take a quick look at how we could take out all loops from any Python program. Most of the time this is a bad idea, both for readability and performance, but it is worth looking at how simple it is to do in a systematic fashion as background to contemplate those cases where it *is* actually a good idea.

If we simply call a function inside a `for` loop, the built-in higher-order function `map()` comes to our aid:

```
for e in it:    # statement-based loop
    func(e)
```

The following code is entirely equivalent to the functional version, except there is no repeated rebinding of the variable `e` involved, and hence no state:

```
map(func, it)   # map()-based "loop"
```

A similar technique is available for a functional approach to sequential program flow. Most imperative programming consists of statements that amount to "do this, then do that, then do the other thing." If those individual actions are wrapped in functions, `map()` lets us do just this:

```
# let f1, f2, f3 (etc) be functions that perform actions
# an execution utility function
do_it = lambda f, *args: f(*args)
# map()-based action sequence
map(do_it, [f1, f2, f3])
```

We can combine the sequencing of function calls with passing arguments from iterables:

```
>>> hello = lambda first, last: print("Hello", first, last)
>>> bye = lambda first, last: print("Bye", first, last)
>>> _ = list(map(do_it, [hello, bye],
>>>                     ['David','Jane'], ['Mertz','Doe']))
Hello David Mertz
Bye Jane Doe
```

Of course, looking at the example, one suspects the result one really wants is actually to pass all the arguments to each of the functions rather than one argument from each list to each function. Expressing that is difficult without using a list comprehension, but easy enough using one:

```
>>> do_all_funcs = lambda fns, *args: [
                           list(map(fn, *args)) for fn in fns]
>>> _ = do_all_funcs([hello, bye],
                     ['David','Jane'], ['Mertz','Doe'])
Hello David Mertz
Hello Jane Doe
Bye David Mertz
Bye Jane Doe
```

In general, the whole of our main program could, in principle, be a `map()` expression with an iterable of functions to execute to complete the program.

Translating `while` is slightly more complicated, but is possible to do directly using recursion:

```
# 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. We could go further in turning suites into function sequences, using `map()` as above. If we did this, we could, moreover, also return a single ternary expression. The details of this further purely functional refactoring is left to readers as an exercise (hint: it will be ugly; fun to play with, but not good production code).

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. One way to add a more useful condition is to let `while_block()` return a more interesting value, and compare that return value for a termination condition. Here is a concrete example of eliminating statements:

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

Now let's remove the `while` loop for the functional version:

```
# FP version of "echo()"
def identity_print(x):      # "identity with side-effect"
    print(x)
    return x
echo_FP = lambda: identity_print(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 statements 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 `identity_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 `identity_print(x)` evaluates to the same thing as if it had simply contained `x`; it is only called for its I/O side effect.

Eliminating Recursion

As with the simple factorial example given above, sometimes we can perform "recursion without recursion" by using `functools.reduce()` or other *folding* operations (other "folds" are not in the Python standard library, but can easily be constructed and/or occur in third-party libraries). A recursion is often simply a way of combining something simpler with an accumulated intermediate result, and that is exactly what `reduce()` does at heart. A slightly longer discussion of `functools.reduce()` occurs in the chapter on higher-order functions.

Callables

The emphasis in functional programming is, somewhat tautologously, on calling functions. Python actually gives us several different ways to create functions, or at least something very function-like (i.e., that can be called). They are:

- Regular functions created with `def` and given a name at definition time
- Anonymous functions created with `lambda`
- Instances of classes that define a `__call()__` method
- Closures returned by function factories
- Static methods of instances, either via the `@staticmethod` decorator or via the class `__dict__`
- Generator functions

This list is probably not exhaustive, but it gives a sense of the numerous slightly different ways one can create something callable. Of course, a plain method of a class instance is also a callable, but one generally uses those where the emphasis is on accessing and modifying mutable state.

Python is a multiple paradigm language, but it has an emphasis on object-oriented styles. When one defines a class, it is generally to generate instances meant as containers for data that change as one calls methods of the class. This style is in some ways opposite to a functional programming approach, which emphasizes immutability and pure functions.

Any method that accesses the state of an instance (in any degree) to determine what result to return is not a pure function. Of course, all the other types of callables we discuss also allow reliance on state in various ways. The author of this report has long pondered whether he could use some dark magic within Python explicitly to declare a function as pure—say by decorating it with a hypothetical `@purefunction` decorator that would raise an exception if the function can have side effects—but consensus seems to be that it would be impossible to guard against every edge case in Python's internal machinery.

The advantage of a *pure function* and side-effect-free code is that it is generally easier to debug and test. Callables that freely intersperse statefulness with their returned results cannot be examined independently of their running context to see how they behave, at least not entirely so. For example, a unit test (using `doctest` or `unittest`, or some third-party testing framework such as `py.test` or `nose`) might succeed in one context but fail when identical calls are made within a running, stateful program. Of course, at the very least, any program that *does* anything must have some kind of output (whether to console, a file, a database, over the network, or whatever) in it to do anything useful, so side effects cannot be entirely eliminated, only isolated to a degree when thinking in functional programming terms.

Named Functions and Lambdas

The most obvious ways to create callables in Python are, in definite order of obviousness, named functions and lambdas. The only in-principle difference between them is simply whether they have a `.__qualname__` attribute, since both can very well be bound to one or more names. In most cases, `lambda` expressions are used within Python only for callbacks and other uses where a simple action is *inlined* into a function call. But as we have shown in this report, flow control in general can be incorporated into single-expression lambdas if we really want. Let's define a simple example to illustrate:

```
>>> def hello1(name):
.....     print("Hello", name)
.....
>>> hello2 = lambda name: print("Hello", name)
>>> hello1('David')
Hello David
```

```
>>> hello2('David')
Hello David
>>> hello1.__qualname__
'hello1'
>>> hello2.__qualname__
'<lambda>'
>>> hello3 = hello2         # can bind func to other names
>>> hello3.__qualname__
'<lambda>'
>>> hello3.__qualname__ = 'hello3'
>>> hello3.__qualname__
'hello3'
```

One of the reasons that functions are useful is that they isolate state lexically, and avoid contamination of enclosing namespaces. This is a limited form of nonmutability in that (by default) nothing you do within a function will bind state variables outside the function. Of course, this guarantee is very limited in that both the `global` and `nonlocal` statements explicitly allow state to "leak out" of a function. Moreover, many data types are themselves mutable, so if they are passed into a function that function might change their contents. Furthermore, doing I/O can also change the "state of the world" and hence alter results of functions (e.g., by changing the contents of a file or a database that is itself read elsewhere).

Notwithstanding all the caveats and limits mentioned above, a programmer who wants to focus on a functional programming style can intentionally decide to write many functions as pure functions to allow mathematical and formal reasoning about them. In most cases, one only leaks state intentionally, and creating a certain subset of all your functionality as pure functions allows for cleaner code. They might perhaps be broken up by "pure" modules, or annotated in the function names or docstrings.

Closures and Callable Instances

There is a saying in computer science that a class is "data with operations attached" while a closure is "operations with data attached." In some sense they accomplish much the same thing of putting logic and data in the same object. But there is definitely a philosophical difference in the approaches, with classes emphasizing mutable or rebindable state, and closures emphasizing immutability and pure functions. Neither side of this divide is absolute—at least in Python—but different attitudes motivate the use of each.

Let us construct a toy example that shows this, something just past a "hello world" of the different styles:

```
# A class that creates callable adder instances
class Adder(object):
    def __init__(self, n):
        self.n = n
    def __call__(self, m):
        return self.n + m
add5_i = Adder(5)       # "instance" or "imperative"
```

We have constructed something callable that adds five to an argument passed in. Seems simple and mathematical enough. Let us also try it as a closure:

```
def make_adder(n):
    def adder(m):
        return m + n
    return adder
add5_f = make_adder(5)  # "functional"
```

So far these seem to amount to pretty much the same thing, but the mutable state in the instance provides a attractive nuisance:

```
>>> add5_i(10)
15
>>> add5_f(10)          # only argument affects result
15
>>> add5_i.n = 10       # state is readily changeable
>>> add5_i(10)          # result is dependent on prior flow
20
```

The behavior of an "adder" created by either `Adder()` or `make_adder()` is, of course, not determined until runtime in general. But once the object exists, the closure behaves in a pure functional way, while the class instance remains state dependent. One might simply settle for "don't change that state"—and indeed that is possible (if no one else with poorer understanding imports and uses your code)—but one is accustomed to changing the state of instances, and a style that prevents abuse programmatically encourages better habits.

There *is* a little "gotcha" about how Python binds variables in closures. It does so by name rather than value, and that can cause confusion, but also has an easy solution. For example, what if we want to manufacture several related closures encapsulating different data:

```
# almost surely not the behavior we intended!
>>> adders = []
>>> for n in range(5):
        adders.append(lambda m: m+n)
>>> [adder(10) for adder in adders]
[14, 14, 14, 14, 14]
>>> n = 10
>>> [adder(10) for adder in adders]
[20, 20, 20, 20, 20]
```

Fortunately, a small change brings behavior that probably better meets our goal:

```
>>> adders = []
>>> for n in range(5):
....     adders.append(lambda m, n=n: m+n)
....
>>> [adder(10) for adder in adders]
[10, 11, 12, 13, 14]
>>> n = 10
>>> [adder(10) for adder in adders]
[10, 11, 12, 13, 14]
>>> add4 = adders[4]
>>> add4(10, 100)       # Can override the bound value
110
```

Notice that using the keyword argument scope-binding trick allows you to change the closed-over value; but this poses much less of a danger for confusion than in the class instance. The overriding value for the named variable must be passed explictly in the call itself, not rebound somewhere remote in the program flow. Yes, the name `add4` is no longer accurately descriptive for "add any two numbers," but at least the change in result is syntactically local.

Methods of Classes

All methods of classes are callables. For the most part, however, calling a method of an instance goes against the grain of functional programming styles. Usually we use methods because we want to reference mutable data that is bundled in the attributes of the instance, and hence each call to a method may produce a different result that varies independently of the arguments passed to it.

Accessors and Operators

Even accessors, whether created with the `@property` decorator or otherwise, are technically callables, albeit accessors are callables with

a limited use (from a functional programming perspective) in that they take no arguments as getters, and return no value as setters:

```
class Car(object):
    def __init__(self):
        self._speed = 100

    @property
    def speed(self):
        print("Speed is", self._speed)
        return self._speed

    @speed.setter
    def speed(self, value):
        print("Setting to", value)
        self._speed = value

# >> car = Car()
# >>> car.speed = 80    # Odd syntax to pass one argument
# Setting to 80
# >>> x = car.speed
# Speed is 80
```

In an accessor, we co-opt the Python syntax of assignment to pass an argument instead. That in itself is fairly easy for much Python syntax though, for example:

```
>>> class TalkativeInt(int):
        def __lshift__(self, other):
            print("Shift", self, "by", other)
            return int.__lshift__(self, other)
....
>>> t = TalkativeInt(8)
>>> t << 3
Shift 8 by 3
64
```

Every operator in Python is basically a method call "under the hood." But while occasionally producing a more readable "domain specific language" (DSL), defining special callable meanings for operators adds no improvement to the underlying capabilities of function calls.

Static Methods of Instances

One use of classes and their methods that is more closely aligned with a functional style of programming is to use them simply as namespaces to hold a variety of related functions:

```
import math
class RightTriangle(object):
    "Class used solely as namespace for related functions"
    @staticmethod
    def hypotenuse(a, b):
        return math.sqrt(a**2 + b**2)

    @staticmethod
    def sin(a, b):
        return a / RightTriangle.hypotenuse(a, b)

    @staticmethod
    def cos(a, b):
        return b / RightTriangle.hypotenuse(a, b)
```

Keeping this functionality in a class avoids polluting the global (or module, etc.) namespace, and lets us name either the class or an instance of it when we make calls to pure functions. For example:

```
>>> RightTriangle.hypotenuse(3,4)
5.0
>>> rt = RightTriangle()
>>> rt.sin(3,4)
0.6
>>> rt.cos(3,4)
0.8
```

By far the most straightforward way to define static methods is with the decorator named in the obvious way. However, in Python 3.x, you can pull out functions that have not been so decorated too—i.e., the concept of an "unbound method" is no longer needed in modern Python versions:

```
>>> import functools, operator
>>> class Math(object):
...     def product(*nums):
...         return functools.reduce(operator.mul, nums)
...     def power_chain(*nums):
...         return functools.reduce(operator.pow, nums)
...
>>> Math.product(3,4,5)
60
>>> Math.power_chain(3,4,5)
3486784401
```

Without `@staticmethod`, however, this will not work on the instances since they still expect to be passed `self`:

```
>>> m = Math()
>>> m.product(3,4,5)
----------------------------------------------------------------
```

```
TypeError
Traceback (most recent call last)
<ipython-input-5-e1de62cf88af> in <module>()
----> 1 m.product(3,4,5)

<ipython-input-2-535194f57a64> in product(*nums)
      2 class Math(object):
      3     def product(*nums):
----> 4         return functools.reduce(operator.mul, nums)
      5     def power_chain(*nums):
      6         return functools.reduce(operator.pow, nums)

TypeError: unsupported operand type(s) for *: 'Math' and 'int'
```

If your namespace is entirely a bag for pure functions, there is no reason not to call via the class rather than the instance. But if you wish to mix some pure functions with some other stateful methods that rely on instance mutable state, you should use the `@staticmethod` decorator.

Generator Functions

A special sort of function in Python is one that contains a `yield` statement, which turns it into a generator. What is returned from calling such a function is not a regular value, but rather an *iterator* that produces a sequence of values as you call the `next()` function on it or loop over it. This is discussed in more detail in the chapter entitled "Lazy Evaluation."

While like any Python object, there are many ways to introduce statefulness into a generator, in principle a generator can be "pure" in the sense of a pure function. It is merely a pure function that produces a (potentially infinite) sequence of values rather than a single value, but still based only on the arguments passed into it. Notice, however, that generator functions typically have a great deal of *internal* state; it is at the boundaries of call signature and return value that they act like a side-effect-free "black box." A simple example:

```
>>> def get_primes():
...     "Simple lazy Sieve of Eratosthenes"
...     candidate = 2
...     found = []
...     while True:
...         if all(candidate % prime != 0 for prime in found):
...             yield candidate
...             found.append(candidate)
...         candidate += 1
```

```
...
>>> primes = get_primes()
>>> next(primes), next(primes), next(primes)
(2, 3, 5)
>>> for _, prime in zip(range(10), primes):
...     print(prime, end=" ")
....
7 11 13 17 19 23 29 31 37 41
```

Every time you create a new object with `get_primes()` the iterator is the same infinite lazy sequence—another example might pass in some initializing values that affected the result—but the object itself is stateful as it is consumed incrementally.

Multiple Dispatch

A very interesting approach to programming multiple paths of execution is a technique called "multiple dispatch" or sometimes "multimethods." The idea here is to declare multiple signatures for a single function and call the actual computation that matches the types or properties of the calling arguments. This technique often allows one to avoid or reduce the use of explicitly conditional branching, and instead substitute the use of more intuitive pattern descriptions of arguments.

A long time ago, this author wrote a module called `multimethods` that was quite flexible in its options for resolving "dispatch linearization" but is also so old as only to work with Python 2.x, and was even written before Python had decorators for more elegant expression of the concept. Matthew Rocklin's more recent `multipledispatch` is a modern approach for recent Python versions, albeit it lacks some of the theoretical arcana I explored in my ancient module. Ideally, in this author's opinion, a future Python version would include a standardized syntax or API for multiple dispatch (but more likely the task will always be the domain of third-party libraries).

To explain how multiple dispatch can make more readable and less bug-prone code, let us implement the game of rock/paper/scissors in three styles. Let us create the classes to play the game for all the versions:

```
class Thing(object): pass
class Rock(Thing): pass
```

```
class Paper(Thing): pass
class Scissors(Thing): pass
```

Many Branches

First a purely imperative version. This is going to have a lot of repetitive, nested, conditional blocks that are easy to get wrong:

```
def beats(x, y):
    if isinstance(x, Rock):
        if isinstance(y, Rock):
            return None         # No winner
        elif isinstance(y, Paper):
            return y
        elif isinstance(y, Scissors):
            return x
        else:
            raise TypeError("Unknown second thing")
    elif isinstance(x, Paper):
        if isinstance(y, Rock):
            return x
        elif isinstance(y, Paper):
            return None         # No winner
        elif isinstance(y, Scissors):
            return y
        else:
            raise TypeError("Unknown second thing")
    elif isinstance(x, Scissors):
        if isinstance(y, Rock):
            return y
        elif isinstance(y, Paper):
            return x
        elif isinstance(y, Scissors):
            return None         # No winner
        else:
            raise TypeError("Unknown second thing")
    else:
        raise TypeError("Unknown first thing")

rock, paper, scissors = Rock(), Paper(), Scissors()
# >>> beats(paper, rock)
# <__main__.Paper at 0x103b96b00>
# >>> beats(paper, 3)
# TypeError: Unknown second thing
```

Delegating to the Object

As a second try we might try to eliminate some of the fragile repitition with Python's "duck typing"—that is, maybe we can have different things share a common method that is called as needed:

```
class DuckRock(Rock):
    def beats(self, other):
        if isinstance(other, Rock):
            return None         # No winner
        elif isinstance(other, Paper):
            return other
        elif isinstance(other, Scissors):
            return self
        else:
            raise TypeError("Unknown second thing")

class DuckPaper(Paper):
    def beats(self, other):
        if isinstance(other, Rock):
            return self
        elif isinstance(other, Paper):
            return None         # No winner
        elif isinstance(other, Scissors):
            return other
        else:
            raise TypeError("Unknown second thing")

class DuckScissors(Scissors):
    def beats(self, other):
        if isinstance(other, Rock):
            return other
        elif isinstance(other, Paper):
            return self
        elif isinstance(other, Scissors):
            return None         # No winner
        else:
            raise TypeError("Unknown second thing")

def beats2(x, y):
    if hasattr(x, 'beats'):
        return x.beats(y)
    else:
        raise TypeError("Unknown first thing")

rock, paper, scissors = DuckRock(), DuckPaper(), DuckScissors()
# >>> beats2(rock, paper)
# <__main__.DuckPaper at 0x103b894a8>
# >>> beats2(3, rock)
# TypeError: Unknown first thing
```

We haven't actually reduced the amount of code, but this version somewhat reduces the complexity within each individual callable, and reduces the level of nested conditionals by one. Most of the logic is pushed into separate classes rather than deep branching. In

object-oriented programming we can "delgate dispatch to the object" (but only to the one controlling object).

Pattern Matching

As a final try, we can express all the logic more directly using multiple dispatch. This should be more readable, albeit there are still a number of cases to define:

```
from multipledispatch import dispatch

@dispatch(Rock, Rock)
def beats3(x, y): return None

@dispatch(Rock, Paper)
def beats3(x, y): return y

@dispatch(Rock, Scissors)
def beats3(x, y): return x

@dispatch(Paper, Rock)
def beats3(x, y): return x

@dispatch(Paper, Paper)
def beats3(x, y): return None

@dispatch(Paper, Scissors)
def beats3(x, y): return x

@dispatch(Scissors, Rock)
def beats3(x, y): return y

@dispatch(Scissors, Paper)
def beats3(x, y): return x

@dispatch(Scissors, Scissors)
def beats3(x, y): return None

@dispatch(object, object)
def beats3(x, y):
    if not isinstance(x, (Rock, Paper, Scissors)):
        raise TypeError("Unknown first thing")
    else:
        raise TypeError("Unknown second thing")

# >>> beats3(rock, paper)
# <__main__.DuckPaper at 0x103b894a8>
# >>> beats3(rock, 3)
# TypeError: Unknown second thing
```

Predicate-Based Dispatch

A really exotic approach to expressing conditionals as dispatch decisions is to include predicates directly within the function signatures (or perhaps within decorators on them, as with `multipledispatch`). I do not know of any well-maintained Python library that does this, but let us simply stipulate a hypothetical library briefly to illustrate the concept. This imaginary library might be aptly named `predicative_dispatch`:

```
from predicative_dispatch import predicate

@predicate(lambda x: x < 0, lambda y: True)
def sign(x, y):
    print("x is negative; y is", y)

@predicate(lambda x: x == 0, lambda y: True)
def sign(x, y):
    print("x is zero; y is", y)

@predicate(lambda x: x > 0, lambda y: True)
def sign(x, y):
    print("x is positive; y is", y)
```

While this small example is obviously not a full specification, the reader can see how we might move much or all of the conditional branching into the function call signatures themselves, and this might result in smaller, more easily understood and debugged functions.

Lazy Evaluation

A powerful feature of Python is its *iterator protocol* (which we will get to shortly). This capability is only loosely connected to functional programming per se, since Python does not quite offer *lazy data structures* in the sense of a language like Haskell. However, use of the iterator protocol—and Python's many built-in or standard library iteratables—accomplish much the same effect as an actual lazy data structure.

Let us explain the contrast here in slightly more detail. In a language like Haskell, which is inherently lazily evaluated, we might define a list of all the prime numbers in a manner like the following:

```
-- Define a list of ALL the prime numbers
primes = sieve [2 ..]
  where sieve (p:xs) = p : sieve [x | x <- xs, (x `rem` p)/=0]
```

This report is not the place to try to teach Haskell, but you can see a comprehension in there, which is in fact the model that Python used in introducing its own comprehensions. There is also deep recursion involved, which is not going to work in Python.

Apart from syntactic differences, or even the ability to recurse to indefinite depth, the significant difference here is that the Haskell version of `primes` is an actual (infinite) sequence, not just an object capable of sequentially producing elements (as was the `primes` object we demonstrated in the chapter entitled "Callables"). In particular, you can index into an arbitrary element of the infinite list of primes in Haskell, and the intermediate values will be produced internally as needed based on the syntactic construction of the list itself.

Mind you, one can replicate this in Python too, it just isn't in the inherent syntax of the language and takes more manual construction. Given the `get_primes()` generator function discussed earlier, we might write our own container to simulate the same thing, for example:

```
from collections.abc import Sequence
class ExpandingSequence(Sequence):
    def __init__(self, it):
        self.it = it
        self._cache = []
    def __getitem__(self, index):
        while len(self._cache) <= index:
            self._cache.append(next(self.it))
        return self._cache[index]
    def __len__(self):
        return len(self._cache)
```

This new container can be both lazy and also indexible:

```
>>> primes = ExpandingSequence(get_primes())
>>> for _, p in zip(range(10), primes):
....     print(p, end=" ")
....
2 3 5 7 11 13 17 19 23 29
>>> primes[10]
31
>>> primes[5]
13
>>> len(primes)
11
>>> primes[100]
547
>>> len(primes)
101
```

Of course, there are other custom capabilities we might want to engineer in, since lazy data structures are not inherently intertwined into Python. Maybe we'd like to be able to slice this special sequence. Maybe we'd like a prettier representation of the object when printed. Maybe we should report the length as `inf` if we somehow signaled it was meant to be infinite. All of this is possible, but it takes a little bit of code to add each behavior rather than simply being the default assumption of Python data structures.

The Iterator Protocol

The easiest way to create an iterator—that is to say, a lazy sequence—in Python is to define a generator function, as was discussed in the chapter entitled "Callables." Simply use the `yield` statement within the body of a function to define the places (usually in a loop) where values are produced.

Or, technically, the *easiest* way is to use one of the many iterable objects already produced by built-ins or the standard library rather than programming a custom one at all. Generator functions are syntax sugar for defining a function that returns an iterator.

Many objects have a method named `.__iter__()`, which will return an iterator when it is called, generally via the `iter()` built-in function, or even more often simply by looping over the object (e.g., `for item in collection: ...`).

What an iterator *is* is the object returned by a call to `iter(something)`, which itself has a method named `.__iter__()` that simply returns the object itself, and another method named `.__next__()`. The reason the iterable itself still has an `.__iter__()` method is to make `iter()` *idempotent*. That is, this identity should always hold (or raise `TypeError("object is not iterable")`):

```
iter_seq = iter(sequence)
iter(iter_seq) == iter_seq
```

The above remarks are a bit abstract, so let us look at a few concrete examples:

```
>>> lazy = open('06-laziness.md')  # iterate over lines of file
>>> '__iter__' in dir(lazy) and '__next__' in dir(lazy)
True
>>> plus1 = map(lambda x: x+1, range(10))
>>> plus1                  # iterate over deferred computations
<map at 0x103b002b0>
>>> '__iter__' in dir(plus1) and '__next__' in dir(plus1)
True
>>> def to10():
...     for i in range(10):
...         yield i
...
>>> '__iter__' in dir(to10)
False
>>> '__iter__' in dir(to10()) and '__next__' in dir(to10())
True
```

```
>>> l = [1,2,3]
>>> '__iter__' in dir(l)
True
>>> '__next__' in dir(l)
False
>>> li = iter(l)          # iterate over concrete collection
>>> li
<list_iterator at 0x103b11278>
>>> li == iter(li)
True
```

In a functional programming style—or even just generally for readability—writing custom iterators as generator functions is most natural. However, we can also create custom classes that obey the protocol; often these will have other behaviors (i.e., methods) as well, but most such behaviors necessarily rely on statefulness and side effects to be meaningful. For example:

```
from collections.abc import Iterable
class Fibonacci(Iterable):
    def __init__(self):
        self.a, self.b = 0, 1
        self.total = 0
    def __iter__(self):
        return self
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        self.total += self.a
        return self.a
    def running_sum(self):
        return self.total

# >>> fib = Fibonacci()
# >>> fib.running_sum()
# 0
# >>> for _, i in zip(range(10), fib):
# ...     print(i, end=" ")
# ...
# 1 1 2 3 5 8 13 21 34 55
# >>> fib.running_sum()
# 143
# >>> next(fib)
# 89
```

This example is trivial, of course, but it shows a class that both implements the iterator protocol and also provides an additional method to return something stateful about its instance. Since statefulness is for object-oriented programmers, in a functional programming style we will generally avoid classes like this.

Module: itertools

The module `itertools` is a collection of very powerful—and carefully designed—functions for performing *iterator algebra*. That is, these allow you to combine iterators in sophisticated ways without having to concretely instantiate anything more than is currently required. As well as the basic functions in the module itself, the module documentation provides a number of short, but easy to get subtly wrong, recipes for additional functions that each utilize two or three of the basic functions in combination. The third-party module `more_itertools` mentioned in the Preface provides additional functions that are likewise designed to avoid common pitfalls and edge cases.

The basic goal of using the building blocks inside `itertools` is to avoid performing computations before they are required, to avoid the memory requirements of a large instantiated collection, to avoid potentially slow I/O until it is stricly required, and so on. Iterators are lazy sequences rather than realized collections, and when combined with functions or recipes in `itertools` they retain this property.

Here is a quick example of combining a few things. Rather than the stateful `Fibonacci` class to let us keep a running sum, we might simply create a single lazy iterator to generate both the current number and this sum:

```
>>> def fibonacci():
...     a, b = 1, 1
...     while True:
...         yield a
...         a, b = b, a+b
...
>>> from itertools import tee, accumulate
>>> s, t = tee(fibonacci())
>>> pairs = zip(t, accumulate(s))
>>> for _, (fib, total) in zip(range(7), pairs):
...     print(fib, total)
...
1 1
1 2
2 4
3 7
5 12
8 20
13 33
```

Figuring out exactly *how* to use functions in `itertools` correctly and optimally often requires careful thought, but once combined, remarkable power is obtained for dealing with large, or even infinite, iterators that could not be done with concrete collections.

The documentation for the `itertools` module contain details on its combinatorial functions as well as a number of short recipes for combining them. This paper does not have space to repeat those descriptions, so just exhibiting a few of them above will suffice. Note that for practical purposes, `zip()`, `map()`, `filter()`, and `range()` (which is, in a sense, just a terminating `itertools.count()`) could well live in `itertools` if they were not built-ins. That is, all of those functions lazily generate sequential items (mostly based on existing iterables) without creating a concrete sequence. Built-ins like `all()`, `any()`, `sum()`, `min()`, `max()`, and `functools.reduce()` also act on iterables, but all of them, in the general case, need to exhaust the iterator rather than remain lazy. The function `itertools.product()` might be out of place in its module since it also creates concrete cached sequences, and cannot operate on infinite iterators.

Chaining Iterables

The `itertools.chain()` and `itertools.chain.from_iterable()` functions combine multiple iterables. Built-in `zip()` and `itertools.zip_longest()` also do this, of course, but in manners that allow incremental advancement through the iterables. A consequence of this is that while chaining infinite iterables is valid syntactically and semantically, no actual program will exhaust the earlier iterable. For example:

```
from itertools import chain, count
thrice_to_inf = chain(count(), count(), count())
```

Conceptually, `thrice_to_inf` will count to infinity three times, but in practice once would always be enough. However, for merely *large* iterables—not for infinite ones—chaining can be very useful and parsimonious:

```
def from_logs(fnames):
    yield from (open(file) for file in fnames)
lines = chain.from_iterable(from_logs(
              ['huge.log', 'gigantic.log']))
```

Notice that in the example given, we didn't even need to pass in a concrete list of files—that sequence of filenames itself could be a lazy iterable per the API given.

Besides the chaining with `itertools`, we should mention `collections.ChainMap()` in the same breath. Dictionaries (or generally any `collections.abc.Mapping`) are iterable (over their keys). Just as we might want to chain multiple sequence-like iterables, we sometimes want to chain together multiple mappings without needing to create a single larger concrete one. `ChainMap()` is handy, and does not alter the underlying mappings used to construct it.

Higher-Order Functions

In the last chapter we saw an iterator algebra that builds on the `itertools` module. In some ways, higher-order functions (often abbreviated as "HOFs") provide similar building blocks to express complex concepts by combining simpler functions into new functions. In general, a *higher-order function* is simply a function that takes one or more functions as arguments and/or produces a function as a result. Many interesting abstractions are available here. They allow chaining and combining higher-order functions in a manner analogous to how we can combine functions in `itertools` to produce new iterables.

A few useful higher-order functions are contained in the `functools` module, and a few others are built-ins. It is common the think of `map()`, `filter()`, and `functools.reduce()` as the most basic building blocks of higher-order functions, and most functional programming languages use these functions as their primitives (occasionally under other names). Almost as basic as map/filter/reduce as a building block is currying. In Python, currying is spelled as `partial()`, and is contained in the `functools` module—this is a function that will take another function, along with zero or more arguments to pre-fill, and return a function of fewer arguments that operates as the input function would when those arguments are passed to it.

The built-in functions `map()` and `filter()` are equivalent to comprehensions—especially now that generator comprehensions are available—and most Python programmers find the comprehension versions more readable. For example, here are some (almost) equivalent pairs:

```
# Classic "FP-style"
transformed = map(tranformation, iterator)
# Comprehension
transformed = (transformation(x) for x in iterator)

# Classic "FP-style"
filtered = filter(predicate, iterator)
# Comprehension
filtered = (x for x in iterator if predicate(x))
```

The function `functools.reduce()` is very general, very powerful, and very subtle to use to its full power. It takes successive items of an iterable, and combines them in some way. The most common use case for `reduce()` is probably covered by the built-in `sum()`, which is a more compact spelling of:

```
from functools import reduce
total = reduce(operator.add, it, 0)
# total = sum(it)
```

It may or may not be obvious that `map()` and `filter()` are also a special cases of `reduce()`. That is:

```
>>> add5 = lambda n: n+5
>>> reduce(lambda l, x: l+[add5(x)], range(10), [])
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> # simpler: map(add5, range(10))
>>> isOdd = lambda n: n%2
>>> reduce(lambda l, x: l+[x] if isOdd(x) else l, range(10),
[])
[1, 3, 5, 7, 9]
>>> # simpler: filter(isOdd, range(10))
```

These `reduce()` expressions are awkward, but they also illustrate how powerful the function is in its generality: *anything* that can be computed from a sequence of successive elements can (if awkwardly) be expressed as a reduction.

There are a few common higher-order functions that are not among the "batteries included" with Python, but that are very easy to create as utilities (and are included with many third-party collections of functional programming tools). Different libraries—and other programming languages—may use different names for the utility functions I describe, but analogous capabilities are widespread (as are the names I choose).

Utility Higher-Order Functions

A handy utility is `compose()`. This is a function that takes a sequence of functions and returns a function that represents the application of each of these argument functions to a data argument:

```
def compose(*funcs):
    """Return a new function s.t.
       compose(f,g,...)(x) == f(g(...(x)))"""
    def inner(data, funcs=funcs):
        result = data
        for f in reversed(funcs):
            result = f(result)
        return result
    return inner

# >>> times2 = lambda x: x*2
# >>> minus3 = lambda x: x-3
# >>> mod6 = lambda x: x%6
# >>> f = compose(mod6, times2, minus3)
# >>> all(f(i)==((i-3)*2)%6 for i in range(1000000))
# True
```

For these one-line math operations (`times2`, `minus3`, etc.), we could have simply written the underlying math expression at least as easily; but if the composite calculations each involved branching, flow control, complex logic, etc., this would not be true.

The built-in functions `all()` and `any()` are useful for asking whether a predicate holds of elements of an iterable. But it is also nice to be able to ask whether any/all of a collection of predicates hold for a particular data item in a composable way. We might implement these as:

```
all_pred = lambda item, *tests: all(p(item) for p in tests)
any_pred = lambda item, *tests: any(p(item) for p in tests)
```

To show the use, let us make a few predicates:

```
>>> is_lt100 = partial(operator.ge, 100)    # less than 100?
>>> is_gt10 = partial(operator.le, 10)      # greater than 10?
>>> from nums import is_prime          # implemented elsewhere
>>> all_pred(71, is_lt100, is_gt10, is_prime)
True
>>> predicates = (is_lt100, is_gt10, is_prime)
>>> all_pred(107, *predicates)
False
```

The library `toolz` has what might be a more general version of this called `juxt()` that creates a function that calls several functions with

the same arguments and returns a tuple of results. We could use that, for example, to do:

```
>>> from toolz.functoolz import juxt
>>> juxt([is_lt100, is_gt10, is_prime])(71)
(True, True, True)
>>> all(juxt([is_lt100, is_gt10, is_prime])(71))
True
>>> juxt([is_lt100, is_gt10, is_prime])(107)
(False, True, True)
```

The utility higher-order functions shown here are just a small selection to illustrate composability. Look at a longer text on functional programming—or, for example, read the Haskell prelude—for many other ideas on useful utility higher-order-functions.

The operator Module

As has been shown in a few of the examples, every operation that can be done with Python's infix and prefix operators corresponds to a named function in the `operator` module. For places where you want to be able to pass a function performing the equivalent of some syntactic operation to some higher-order function, using the name from `operator` is faster and looks nicer than a corresponding lambda. For example:

```
# Compare ad hoc lambda with `operator` function
sum1 = reduce(lambda a, b: a+b, iterable, 0)
sum2 = reduce(operator.add, iterable, 0)
sum3 = sum(iterable)    # The actual Pythonic way
```

The functools Module

The obvious place for Python to include higher-order functions is in the `functools` module, and indeed a few are in there. However, there are surprisingly few utility higher-order functions in that module. It has gained a few interesting ones over Python versions, but core developers have a resistence to going in the direction of a full functional programming language. On the other hand, as we have seen in a few example above, many of the most useful higher-order functions only take a few lines (sometimes a single line) to write yourself.

Apart from `reduce()`, which is discussed at the start of this chapter, the main facility in the module is `partial()`, which has also been

mentioned. This operation is called "currying" (after Haskell Curry) in many languages. There are also some examples of using `partial()` discussed above.

The remainder of the `functools` module is generally devoted to useful *decorators*, which is the topic of the next section.

Decorators

Although it is—by design—easy to forget it, probably the most common use of higher-order functions in Python is as decorators. A decorator is just syntax sugar that takes a function as an argument, and if it is programmed correctly, returns a new function that is in some way an *enhancement* of the original function (or method, or class). Just to remind readers, these two snippets of code defining `some_func` and `other_func` are equivalent:

```
@enhanced
def some_func(*args):
    pass

def other_func(*args):
    pass
other_func = enhanced(other_func)
```

Used with the decorator syntax, of course, the higher-order function is necessarily used at definition time for a function. For their intended purpose, this is usually when they are best applied. But the same decorator function can always, in principle, be used elsewhere in a program, for example in a more dynamic way (e.g., mapping a decorator function across a runtime-generated collection of other functions). That would be an unusual use case, however.

Decorators are used in many places in the standard library and in common third-party libraries. In some ways they tie in with an idea that used to be called "aspect-oriented programming." For example, the decorator function `asyncio.coroutine` is used to mark a function as a coroutine. Within `functools` the three important decorator functions are `functools.lru_cache`, `functools.total_ordering`, and `functools.wraps`. The first "memoizes" a function (i.e., it caches the arguments passed and returns stored values rather than performing new computation or I/O). The second makes it easier to write custom classes that want to use inequality operators. The last makes it easier to write new decorators. All of these are important

and worthwhile purposes, but they are also more in the spirit of making the plumbing of Python programming easier in a general—almost syntactic—way rather than the composable higher-order functions this chapter focuses on.

Decorators in general are more useful when you want to poke into the guts of a function than when you want to treat it as a pluggable component in a flow or composition of functions, often done to mark the purpose or capabilities of a particular function.

This report has given only a glimpse into some techniques for programming Python in a more functional style, and only some suggestions as to the advantages one often finds in aspiring in that direction. Programs that use functional programming are usually shorter than more traditional imperative ones, but much more importantly, they are also usually both more composable and more provably correct. A large class of difficult to debug errors in program logic are avoided by writing functions without side effects, and even more errors are avoided by writing small units of functionality whose operation can be understood and tested more reliably.

A rich literature on functional programming as a general technique —often in particular languages which are not Python—is available and well respected. Studying one of many such classic books, some published by O'Reilly (including very nice video training on functional programming in Python), can give readers further insight into the nitty-gritty of functional programming techniques. Almost everything one might do in a more purely functional language can be done with very little adjustment in Python as well.

About the Author

David Mertz is a director of the PSF, and chair of its Trademarks Committee and Outreach & Education Committee. He wrote the columns *Charming Python* and *XML Matters* for IBM developerWorks and the Addison-Wesley book *Text Processing in Python*, has spoken at multiple OSCONs and PyCons, and was invited to be a keynote speaker at PyCon India, PyCon UK, PyCon ZA, and PyCon Belarus.

In the distant past, David spent some time as a university professor, teaching in areas far removed from computer programming, but gained some familiarity with the vicissitudes of pedagogy.

Since 2008, David has worked with folks who have built the world's fastest supercomputer for performing molecular dynamics. He is pleased to find Python becoming the default high-level language for most scientific computing projects.