Functional Python Programming

Create succinct and expressive implementations with functional programming in Python

Steven Lott
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 “Introducing Functional Programming”
- A synopsis of the book’s content
- More information on Functional Python Programming

About the Author

Steven Lott has been programming since the 1970s, when computers were large, expensive, and rare. As a contract software developer and architect, he has worked on hundreds of projects, from very small to very large. He's been using Python to solve business problems for over 10 years. He's particularly adept struggling with gnarly data representation problems. His other titles include Mastering Object-Oriented Python and Python for Secret Agents, both by Packt Publishing. After spending years as a technomad—living in various places on the east coast of the US—he has dropped the hook in the Chesapeake Bay. He blogs at http://slott-softwarearchitect.blogspot.com.
Functional Python Programming

Programming languages sometimes fit neatly into tidy categories like imperative and functional. Imperative languages might further subdivide into those that are procedural and those that include features for object-oriented programming. The Python language, however, contains aspects of all of these three language categories. Though Python is not a purely functional programming language, we can do a great deal of functional programming in Python.

Most importantly, we can leverage many design patterns and techniques from other functional languages and apply them to Python programming. These borrowed concepts can lead us to create succinct and elegant programs. Python's generator expressions, in particular, avoid the need to create large in-memory data structures, leading to programs which may execute more quickly because they use fewer resources.

We can't easily create purely functional programs in Python. Python lacks a number of features that would be required for this. For example, we don't have unlimited recursion, lazy evaluation of all expressions, and an optimizing compiler.

Generally, Python emphasizes strict evaluation rules. This means that statements are executed in order and expressions are evaluated from left to right. While this deviates from functional purity, it allows us to perform manual optimizations when writing in Python. We'll take a hybrid approach to functional programming using Python's functional features when they can add clarity or simplify the code and use ordinary imperative features for optimization.

There are several key features of functional programming languages that are available in Python. One of the most important is the idea that functions are first-class objects. In some languages, functions exist only as a source code construct: they don't exist as proper data structures at runtime. In Python, functions can use functions as arguments and return functions as results.

Python offers a number of higher-order functions. Functions like map(), filter(), and functools.reduce() are widely used in this role. Less obvious functions like sorted(), min(), and max() are also higher-order functions; they have a default function and, consequently, different syntax from the more common examples.
Functional programs often exploit immutable data structures. The emphasis on stateless objects permits flexible optimization. Python offers tuples and namedtuples as complex but immutable objects. We can leverage these structures to adapt some design practices from other functional programming languages.

Many functional languages emphasize recursion but exploit Tail-Call Optimization (TCO). Python tends to limit recursion to a relatively small number of stack frames. In many cases, we can think of a recursion as a generator function. We can then simply rewrite it to use a `yield from` statement, doing the tail-call optimization ourselves.

We'll look at the core features of functional programming from a Python point of view. Our objective is to borrow good ideas from functional programming languages, and use these ideas to create expressive and succinct applications in Python.

**What This Book Covers**

*Chapter 1, Introducing Functional Programming*, introduces some of the techniques that characterize functional programming. We'll identify some of the ways to map these features to Python, and finally, we'll also address some ways that the benefits of functional programming accrue when we use these design patterns to build Python applications.

*Chapter 2, Introducing Some Functional Features*, will delve into six central features of the functional programming paradigm. We'll look at each in some detail to see how they're implemented in Python. We'll also point out some features of functional languages that don't apply well to Python. In particular, many functional languages have complex type-matching rules required to support compilation and optimization.

*Chapter 3, Functions, Iterators, and Generators*, will show how to leverage immutable Python objects and generator expressions, and adapt functional programming concepts to the Python language. We'll look at some of the built-in Python collection and how we can leverage them without departing too far from functional programming concepts.

*Chapter 4, Working with Collections*, shows how we can use a number of built-in Python functions to operate on collections of data. This section will focus on a number of relatively simple functions such as `any()` and `all()`, which will reduce a collection of values to a single result.

*Chapter 5, Higher-order Functions*, examines the commonly used higher order functions such as `map()` and `filter()`. The chapter also includes a number of other functions that are also higher-order functions, as well as how we can create our own higher-order functions.
Chapter 6, *Recursions and Reductions*, shows how we can design an algorithm using recursion and then optimize it into a high performance for loop. We'll also look at some other reductions that are widely used, including the `collections.Counter()` function.

Chapter 7, *Additional Tuple Techniques*, shows a number of ways in which we can use immutable tuples and namedtuples instead of stateful objects. Immutable objects have a much simpler interface: we never have to worry about abusing an attribute and setting an object into some inconsistent or invalid state.

Chapter 8, *The Itertools Module*, examines a number of functions in the standard library module. This collection of functions simplifies writing programs that deal with collections or generator functions.

Chapter 9, *More Itertools Techniques*, covers the combinatoric functions in the `itertools` module. These functions are somewhat less useful. This chapter includes some examples that illustrate ill-considered uses of these functions and the consequences of combinatoric explosion.

Chapter 10, *The Functools Module*, will show how to use some of the functions in this module for functional programming. A few of the functions in this module are more appropriate for building decorators, and are left for the next chapter. The other functions, however, provide several more ways to design and implement function programs.

Chapter 11, *Decorator Design Techniques*, shows how we can look at a decorator as a way to build a composite function. While there is considerable flexibility here, there are also some conceptual limitations: we'll look at ways in which overly complex decorators can become confusing rather than helpful.

Chapter 12, *The Multiprocessing and Threading Modules*, points out an important consequence of good functional design: we can distribute the processing workload. Using immutable objects means that we can't corrupt an object because of poorly synchronized write operations.

Chapter 13, *Conditional Expressions and the Operator Module*, will show some ways in which we can break out of Python's strict order of evaluation. There are limitations to what we can achieve here. We'll also look at the operator module and how the operator module can lead to some slight clarification of some simple kinds of processing.
Chapter 14, *The PyMonad Library*, examines some of the features of the PyMonad library. This provides some additional functional programming features. This also provides a way to learn more about monads. In some functional languages, monads are an important way to force a particular order for operations that might get optimized into an undesirable order. Since Python already has strict ordering of expressions and statements, the monad feature is more instructive than practical.

Chapter 15, *A Functional Approach to Web Services*, shows how we can think of web services as a nested collection of functions that transform a request into a reply. We'll see ways in which we can leverage functional programming concepts for building responsive, dynamic web content.

Chapter 16, *Optimizations and Improvements*, includes some additional tips on performance and optimization. We'll emphasize techniques like memorization because they're easy to implement and can—in the right context—yield dramatic performance improvements.
Introducing Functional Programming

Functional programming defines a computation using expressions and evaluation—often encapsulated in function definitions. It de-emphasizes or avoids the complexity of state change and mutable objects. This tends to create programs that are more succinct and expressive. In this chapter, we’ll introduce some of the techniques that characterize functional programming. We’ll identify some of the ways to map these features to Python. Finally, we’ll also address some ways in which the benefits of functional programming accrue when we use these design patterns to build Python applications.

Python has numerous functional programming features. It is not a purely functional programming language. It offers enough of the right kinds of features that it confers to the benefits of functional programming. It also retains all optimization power available from an imperative programming language.

We’ll also look at a problem domain that we’ll use for many of the examples in this book. We’ll try to stick closely to Exploratory Data Analysis (EDA) because its algorithms are often good examples of functional programming. Furthermore, the benefits of functional programming accrue rapidly in this problem domain.

Our goal is to establish some essential principles of functional programming. The more serious Python code will begin in Chapter 2, Introducing Some Functional Features.

We’ll focus on Python 3 features in this book. However, some of the examples might also work in Python 2.
Identifying a paradigm

It's difficult to be definitive on what fills the universe of programming paradigms. For our purposes, we will distinguish between just two of the many programming paradigms: Functional programming and Imperative programming. One important distinguishing feature between these two is the concept of state.

In an imperative language, like Python, the state of the computation is reflected by the values of the variables in the various namespaces. The values of the variables establish the state of a computation; each kind of statement makes a well-defined change to the state by adding or changing (or even removing) a variable. A language is imperative because each statement is a command, which changes the state in some way.

Our general focus is on the assignment statement and how it changes state. Python has other statements, such as `global` or `nonlocal`, which modify the rules for variables in a particular namespace. Statements like `def`, `class`, and `import` change the processing context. Other statements like `try`, `except`, `if`, `elif`, and `else` act as guards to modify how a collection of statements will change the computation's state. Statements like `for` and `while`, similarly, wrap a block of statements so that the statements can make repeated changes to the state of the computation. The focus of all these various statement types, however, is on changing the state of the variables.

Ideally, each statement advances the state of the computation from an initial condition toward the desired final outcome. This "advances the computation" assertion can be challenging to prove. One approach is to define the final state, identify a statement that will establish this final state, and then deduce the precondition required for this final statement to work. This design process can be iterated until an acceptable initial state is derived.

In a functional language, we replace state—the changing values of variables—with a simpler notion of evaluating functions. Each function evaluation creates a new object or objects from existing objects. Since a functional program is a composition of a function, we can design lower-level functions that are easy to understand, and we will design higher-level compositions that can also be easier to visualize than a complex sequence of statements.

Function evaluation more closely parallels mathematical formalisms. Because of this, we can often use simple algebra to design an algorithm, which clearly handles the edge cases and boundary conditions. This makes us more confident that the functions work. It also makes it easy to locate test cases for formal unit testing.
It's important to note that functional programs tend to be relatively succinct, expressive, and efficient when compared to imperative (object-oriented or procedural) programs. The benefit isn't automatic; it requires a careful design. This design effort is often easier than functionally similar procedural programming.

Subdividing the procedural paradigm

We can subdivide imperative languages into a number of discrete categories. In this section, we'll glance quickly at the procedural versus object-oriented distinction. What's important here is to see how object-oriented programming is a subset of imperative programming. The distinction between procedural and object-orientation doesn't reflect the kind of fundamental difference that functional programming represents.

We'll use code examples to illustrate the concepts. For some, this will feel like reinventing a wheel. For others, it provides a concrete expression of abstract concepts.

For some kinds of computations, we can ignore Python's object-oriented features and write simple numeric algorithms. For example, we might write something like the following to get the range of numbers:

```python
s = 0
for n in range(1, 10):
    if n % 3 == 0 or n % 5 == 0:
        s += n
print(s)
```

We've made this program strictly procedural, avoiding any explicit use of Python's object features. The program's state is defined by the values of the variables `s` and `n`. The variable `n`, takes on values such that `1 ≤ n < 10`. As the `for` loop involves an ordered exploration of values of `n`, we can prove that it will terminate when `n == 10`. Similar code would work in C or Java using their primitive (non-object) data types.

We can exploit Python's Object-Oriented Programming (OOP) features and create a similar program:

```python
m = list()
for n in range(1, 10):
    if n % 3 == 0 or n % 5 == 0:
        m.append(n)
print(sum(m))
```

This program produces the same result but it accumulates a stateful collection object, `m`, as it proceeds. The state of the computation is defined by the values of the variables `m` and `n`. 
Introducing Functional Programming

The syntax of `m.append(n)` and `sum(m)` can be confusing. It causes some programmers to insist (wrongly) that Python is somehow not purely Object-oriented because it has a mixture of the `function()` and `object.method()` syntax. Rest assured, Python is purely Object-oriented. Some languages, like C++, allow the use of primitive data type such as `int`, `float`, and `long`, which are not objects. Python doesn't have these primitive types. The presence of prefix syntax doesn't change the nature of the language.

To be pedantic, we could fully embrace the object model, the subclass, the `list` class, and add a `sum` method:

```python
class SummableList(list):
    def sum(self):
        s = 0
        for v in self.__iter__():
            s += v
        return s
```

If we initialize the variable, `m`, with the `SummableList()` class instead of the `list()` method, we can use the `m.sum()` method instead of the `sum(m)` method. This kind of change can help to clarify the idea that Python is truly and completely object-oriented. The use of prefix function notation is purely syntactic sugar.

All three of these examples rely on variables to explicitly show the state of the program. They rely on the assignment statements to change the values of the variables and advance the computation toward completion. We can insert the `assert` statements throughout these examples to demonstrate that the expected state changes are implemented properly.

The point is not that imperative programming is broken in some way. The point is that functional programming leads to a change in viewpoint, which can, in many cases, be very helpful. We'll show a function view of the same algorithm. Functional programming doesn't make this example dramatically shorter or faster.

### Using the functional paradigm

In a functional sense, the sum of the multiples of 3 and 5 can be defined in two parts:

- The sum of a sequence of numbers
- A sequence of values that pass a simple test condition, for example, being multiples of three and five
The sum of a sequence has a simple, recursive definition:

```python
def sum(seq):
    if len(seq) == 0: return 0
    return seq[0] + sum(seq[1:])
```

We've defined the sum of a sequence in two cases: the base case states that the sum of a zero length sequence is 0, while the recursive case states that the sum of a sequence is the first value plus the sum of the rest of the sequence. Since the recursive definition depends on a shorter sequence, we can be sure that it will (eventually) devolve to the base case.

The + operator on the last line of the preceding example and the initial value of 0 in the base case characterize the equation as a sum. If we change the operator to * and the initial value to 1, it would just as easily compute a product. We'll return to this simple idea of generalization in the following chapters.

Similarly, a sequence of values can have a simple, recursive definition, as follows:

```python
def until(n, filter_func, v):
    if v == n: return []
    if filter_func(v): return [v] + until(n, filter_func, v+1)
    else: return until(n, filter_func, v+1)
```

In this function, we've compared a given value, v, against the upper bound, n. If v reaches the upper bound, the resulting list must be empty. This is the base case for the given recursion.

There are two more cases defined by the given filter_func() function. If the value of v is passed by the filter_func() function, we'll create a very small list, containing one element, and append the remaining values of the until() function to this list. If the value of v is rejected by the filter_func() function, this value is ignored and the result is simply defined by the remaining values of the until() function.

We can see that the value of v will increase from an initial value until it reaches n, assuring us that we'll reach the base case soon.

Here's how we can use the until() function to generate the multiples of 3 or 5. First, we'll define a handy lambda object to filter values:

```python
mult_3_5= lambda x: x%3==0 or x%5==0
```

(We will use lambdas to emphasize succinct definitions of simple functions. Anything more complex than a one-line expression requires the def statement.)
Introducing Functional Programming

We can see how this lambda works from the command prompt in the following example:

```python
>>> mult_3_5(3)
True
>>> mult_3_5(4)
False
>>> mult_3_5(5)
True
```

This function can be used with the `until()` function to generate a sequence of values, which are multiples of 3 or 5.

```python
>>> until(10, lambda x: x%3==0 or x%5==0, 0)
[0, 3, 5, 6, 9]
```

We can use our recursive `sum()` function to compute the sum of this sequence of values. The various functions, such as `sum()`, `until()`, and `mult_3_5()` are defined as simple recursive functions. The values are computed without restoring to use intermediate variables to store state.

We'll return to the ideas behind this purely functional recursive function definition in several places. It's important to note here that many functional programming language compilers can optimize these kinds of simple recursive functions. Python can't do the same optimizations.

Using a functional hybrid

We'll continue this example with a mostly functional version of the previous example to compute the sum of the multiples of 3 and 5. Our hybrid functional version might look like the following:

```python
print( sum(n for n in range(1, 10) if n%3==0 or n%5==0) )
```

We've used nested generator expressions to iterate through a collection of values and compute the sum of these values. The `range(1, 10)` method is an iterable and, consequently, a kind of generator expression; it generates a sequence of values `{n|1≤n<10}`. The more complex expression, `n for n in range(1, 10) if n%3==0 or n%5==0`, is also an iterable expression. It produces a set of values `{n|1≤n<10∧(nmod3=0∨nmod5=0)}`. A variable, n, is bound to each value, more as a way of expressing the contents of the set than as an indicator of the state of the computation. The `sum()` function consumes the iterable expression, creating a final object, 23.
The bound variable doesn't change once a value is bound to it. The variable, \( n \), in the loop is essentially a shorthand for the values available from the `range()` function.

The `if` clause of the expression can be extracted into a separate function, allowing us to easily repurpose this with other rules. We could also use a higher-order function named `filter()` instead of the `if` clause of the generator expression. We'll save this for Chapter 5, *Higher-order Functions*.

As we work with generator expressions, we'll see that the bound variable is at the blurry edge of defining the state of the computation. The variable, \( n \), in this example isn't directly comparable to the variable, \( n \), in the first two imperative examples. The `for` statement creates a proper variable in the local namespace. The generator expression does not create a variable in the same way as a `for` statement does:

```python
>>> sum( n for n in range(1, 10) if n%3==0 or n%5==0 )
23
```

```python
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Because of the way Python uses namespaces, it might be possible to write a function that can observe the \( n \) variable in a generator expression. However, we won't. Our objective is to exploit the functional features of Python, not to detect how those features have an object-oriented implementation under the hood.

### Looking at object creation

In some cases, it might help to look at intermediate objects as a history of the computation. What's important is that the history of a computation is not fixed. When functions are commutative or associative, then changes to the order of evaluation might lead to different objects being created. This might have performance improvements with no changes to the correctness of the results.

Consider this expression:

```python
>>> 1+2+3+4
10
```

We are looking at a variety of potential computation histories with the same result. Because the `+` operator is commutative and associative, there are a large number of candidate histories that lead to the same result.
Introducing Functional Programming

Of the candidate sequences, there are two important alternatives, which are as follows:

```python
>>> ((1+2)+3)+4
10
>>> 1+(2+(3+4))
10
```

In the first case, we fold in values working from left to right. This is the way Python works implicitly. Intermediate objects 3 and 6 are created as part of this evaluation.

In the second case, we fold from right-to-left. In this case, intermediate objects 7 and 9 are created. In the case of simple integer arithmetic, the two results have identical performance; there's no optimization benefit.

When we work with something like the `list` append, we might see some optimization improvements when we change the association rules.

Here's a simple example:

```python
>>> import timeit
>>> timeit.timeit("(((\[\]+[1])+[2])+[3])+[4]")
0.8846941249794327
>>> timeit.timeit("[[1]+([2]+([3]+[4]))]")
1.0207440659869462
```

In this case, there's some benefit in working from left to right.

What's important for functional design is the idea that the `+` operator (or `add()` function) can be used in any order to produce the same results. The `+` operator has no hidden side effects that restrict the way this operator can be used.

**The stack of turtles**

When we use Python for functional programming, we embark down a path that will involve a hybrid that's not strictly functional. Python is not Haskell, OCaml, or Erlang. For that matter, our underlying processor hardware is not functional; it's not even strictly object-oriented—CPUs are generally procedural.
All programming languages rest on abstractions, libraries, frameworks and virtual machines. These abstractions, in turn, may rely on other abstractions, libraries, frameworks and virtual machines. The most apt metaphor is this: the world is carried on the back of a giant turtle. The turtle stands on the back of another giant turtle. And that turtle, in turn, is standing on the back of yet another turtle.

It's turtles all the way down.

– Anonymous

There's no practical end to the layers of abstractions.

More importantly, the presence of abstractions and virtual machines doesn't materially change our approach to designing software to exploit the functional programming features of Python.

Even within the functional programming community, there are more pure and less pure functional programming languages. Some languages make extensive use of monads to handle stateful things like filesystem input and output. Other languages rely on a hybridized environment that's similar to the way we use Python. We write software that's generally functional with carefully chosen procedural exceptions.

Our functional Python programs will rely on the following three stacks of abstractions:

- Our applications will be functions—all the way down—until we hit the objects
- The underlying Python runtime environment that supports our functional programming is objects—all the way down—until we hit the turtles
- The libraries that support Python are a turtle on which Python stands

The operating system and hardware form their own stack of turtles. These details aren't relevant to the problems we're going to solve.

A classic example of functional programming

As part of our introduction, we'll look at a classic example of functional programming. This is based on the classic paper Why Functional Programming Matters by John Hughes. The article appeared in a paper called Research Topics in Functional Programming, edited by D. Turner, published by Addison-Wesley in 1990.
Introducing Functional Programming

Here's a link to the paper Research Topics in Functional Programming:

http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf

This discussion of functional programming in general is profound. There are several examples given in the paper. We'll look at just one: the Newton-Raphson algorithm for locating the roots of a function. In this case, the function is the square root.

It's important because many versions of this algorithm rely on the explicit state managed via loops. Indeed, the Hughes paper provides a snippet of the Fortran code that emphasizes stateful, imperative processing.

The backbone of this approximation is the calculation of the next approximation from the current approximation. The next_() function takes x, an approximation to the sqrt(n) method and calculates a next value that brackets the proper root. Take a look at the following example:

```python
def next_(n, x):
    return (x+n/x)/2
```

This function computes a series of values \( a_{i+1} = (a_i + n/a_i) / 2 \). The distance between the values is halved each time, so they'll quickly get to converge on the value such that \( a = n/a \), which means \( a = \sqrt{n} \). We don't want to call the method next() because this name would collide with a built-in function. We call it the next_() method so that we can follow the original presentation as closely as possible.

Here's how the function looks when used in the command prompt:

```python
>>> n= 2
>>> f= lambda x: next_(n, x)
>>> a0= 1.0
>>> [ round(x,4) for x in (a0, f(a0), f(f(a0)), f(f(f(a0))))) ]
[1.0, 1.5, 1.4167, 1.4142]
```

We've defined the f() method as a lambda that will converge on \( \sqrt{2} \). We started with 1.0 as the initial value for \( a_0 \). Then we evaluated a sequence of recursive evaluations: \( a_1 = f(a_0), a_2 = f(f(a_0)) \) and so on. We evaluated these functions using a generator expression so that we could round off each value. This makes the output easier to read and easier to use with doctest. The sequence appears to converge rapidly on \( \sqrt{2} \).

We can write a function, which will (in principle) generate an infinite sequence of \( a_i \) values converging on the proper square root:

```python
def repeat(f, a):
    yield a
    for v in repeat(f, f(a)):
        yield v
```
This function will generate approximations using a function, \( f() \), and an initial value, \( a \). If we provide the \( \text{next}() \) function defined earlier, we'll get a sequence of approximations to the square root of the \( n \) argument.

We have two ways to return all the values instead of returning a generator expression, which are as follows:

- We can write an explicit \texttt{for} loop as follows:
  
  ```python
  for x in some_iter: yield x.
  ```

- We can use the \texttt{yield from} statement as follows:
  
  ```python
  yield from some_iter.
  ```

Both techniques of yielding the values of a recursive generator function are equivalent. We'll try to emphasize \texttt{yield from}. In some cases, however, the \texttt{yield} with a complex expression will be more clear than the equivalent mapping or generator expression.

Of course, we don't want the entire infinite sequence. We will stop generating values when two values are so close to each other that we can call either one the square root we're looking for. The common symbol for the value, which is close enough, is the Greek letter \textit{Epsilon}, \( \varepsilon \), which can be thought of as the largest error we will tolerate.

In Python, we'll have to be a little clever about taking items from an infinite sequence one at a time. It works out well to use a simple interface function that wraps a slightly more complex recursion. Take a look at the following code snippet:

```python
def within(\varepsilon, iterable):
    def head_tail(\varepsilon, a, iterable):
        b= next(iterable)
        if abs(a-b) <= \varepsilon: return b
        return head_tail(\varepsilon, b, iterable)
    return head_tail(\varepsilon, next(iterable), iterable)
```
We've defined an internal function, `head_tail()`, which accepts the tolerance, $\varepsilon$, an item from the iterable sequence, $a$, and the rest of the iterable sequence, $\text{iterable}$. The next item from the iterable bound to a name $b$. If $|a-b| \leq \varepsilon$, then the two values that are close enough together that we've found the square root. Otherwise, we use the $b$ value in a recursive invocation of the `head_tail()` function to examine the next pair of values.

Our `within()` function merely seeks to properly initialize the internal `head_tail()` function with the first value from the `iterable` parameter.

Some functional programming languages offer a technique that will put a value back into an iterable sequence. In Python, this might be a kind of `unget()` or `previous()` method that pushes a value back into the iterator. Python iterables don't offer this kind of rich functionality.

We can use the three functions `next_()`, `repeat()`, and `within()` to create a square root function, as follows:

```python
def sqrt(a0, $\varepsilon$, n):
    return within($\varepsilon$, repeat(lambda x: next_(n,x), a0))
```

We've used the `repeat()` function to generate a (potentially) infinite sequence of values based on the `next_(n,x)` function. Our `within()` function will stop generating values in the sequence when it locates two values with a difference less than $\varepsilon$.

When we use this version of the `sqrt()` method, we need to provide an initial seed value, $a_0$, and an $\varepsilon$ value. An expression like $\text{sqrt}(1.0, .0001, 3)$ will start with an approximation of 1.0 and compute the value of $\sqrt{3}$ to within 0.0001. For most applications, the initial $a_0$ value can be 1.0. However, the closer it is to the actual square root, the more rapidly this method converges.

The original example of this approximation algorithm was shown in the Miranda language. It's easy to see that there are few profound differences between Miranda and Python. The biggest difference is Miranda's ability to construct `cons`, a value back into an `iterable`, doing a kind of `unget`. This parallelism between Miranda and Python gives us confidence that many kinds of functional programming can be easily done in Python.

**Exploratory Data Analysis**

Later in this book, we'll use the field of EDA as a source for concrete examples of functional programming. This field is rich with algorithms and approaches to working with complex datasets; functional programming is often a very good fit between the problem domain and automated solutions.
While details vary from author to author, there are several widely accepted stages of EDA. These include the following:

- **Data preparation**: This might involve extraction and transformation for source applications. It might involve parsing a source data format and doing some kinds of data scrubbing to remove unusable or invalid data. This is an excellent application of functional design techniques.

- **Data exploration**: This is a description of the available data. This usually involves the essential statistical functions. This is another excellent place to explore functional programming. We can describe our focus as univariate and bivariate statistics but that sounds too daunting and complex. What this really means is that we'll focus on mean, median, mode, and other related descriptive statistics. Data exploration may also involve data visualization. We'll skirt this issue because it doesn't involve very much functional programming. I'll suggest that you use a toolkit like SciPy.

  Visit the following link to get more information how SciPy works and its usage:


- **Data modeling and machine learning**: This tends to be prescriptive as it involves extending a model to new data. We're going to skirt this because some of the models can become mathematically complex. If we spend too much time on these topics, we won't be able to focus on functional programming.

- **Evaluation and comparison**: When there are alternative models, each must be evaluated to determine which is a better fit for the available data. This can involve ordinary descriptive statistics of model outputs. This can benefit from functional design techniques.

The goal of EDA is often to create a model that can be deployed as a decision support application. In many cases, a model might be a simple function. A simple functional programming approach can apply the model to new data and display results for human consumption.
Summary
We've looked at programming paradigms with an eye toward distinguishing the functional paradigm from two common imperative paradigms. Our objective in this book is to explore the functional programming features of Python. We've noted that some parts of Python don't allow purely functional programming; we'll be using some hybrid techniques that meld the good features of succinct, expressive functional programming with some high-performance optimizations in Python.

In the next chapter, we'll look at five specific functional programming techniques in detail. These techniques will form the essential foundation for our hybridized functional programming in Python.
Where to buy this book

You can buy Functional Python Programming from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.