write and debug. Errors, when they do occur, tend to be larger and thus easier to find and eliminate. The difference between a functional and an imperative style is best illustrated by examples, as we will present shortly.

Mapping, Filtering, and Reduction

The process of transformation can be subdivided into several common forms. The three most common varieties of transformation are mapping, filtering, and reduction.

A *mapping* is a one-to-one transformation. Each element in the source is converted into a new value. The new values are gathered into a collection, leaving the original collection unchanged. For example, suppose that you begin with the list [1, 2, 3, 4, 5] and map using the transformation x*2+1. The result would be the list [3, 5, 7, 9, 11].

A *filtering* is the process of testing each value in a list with a function and retaining only those for which the function is true. If you begin with the list [1, 2, 3, 4, 5] and filter with a function that returns true on the odd values, the result would be the list [1, 3, 5].

A *reduction* is the process of applying a binary function to each member of a list in a cumulative fashion. If you begin with the list [1, 2, 3, 4, 5] and reduce using the addition operation, the result would be ((((1 + 2) + 3) + 4) + 5), or 15.

Each of these three basic tasks is provided by a function in the Python library. Notice that the definition of each of these functions refers to invoking another function as part of the process. The function used in this case is passed as an argument. A function that uses another function that is passed as an argument is sometimes referred to as a *higher order* function.

Lambda Functions

When a function is required as an argument, one possibility is to simply pass the name of a previously defined function:

```
def even(x):
  return x % 2 == 0

>>> a = [1, 2, 3, 4, 5]
>>> print filter(even, a)
[2, 4]
```

However, because the functions that are passed as argument to maps, reductions, and filters are often simple and are usually used nowhere else, it is inconvenient to require the programmer to define them using the standard `def` keyword. An alternative is a mechanism to define a nameless function as an expression. This type of expression is termed a *lambda*. The following example illustrates the syntax:

```
lambda x, y : x + y
```

The body of the lambda function must be a simple expression. Because it must be written on one line, it cannot contain any complex logic, such as conditional

statements or loops. Generally a `lambda` is passed as argument to `map`, `filter`, or `reduce`. The following illustrates the application of each of these functions:

```
>>> a = [1, 2, 3, 4, 5]
>>> print map(lambda x : x * 2 + 1, a)
[3, 5, 7, 9, 11]
>>> print filter(lambda x: x % 2 == 0, a)
[2, 4]
>>> print reduce(lambda x, y: x + y, a)
15
```

Notice that the original list, held in the variable named `a`, remains unchanged. The functions `map`, `filter`, and `reduce` produce new lists that are transformations of the argument.

The function `filter` requires an argument that is itself a function that takes only one argument and returns a Boolean value. A one-argument function that returns a Boolean result is termed a *predicate*.

List Comprehensions

An even simpler form of functional programming is provided by a *list comprehension*. Instead of defining a list by a sequence of elements, lists can be characterized by a process. This process is described by a series of keywords:

```
[ expr for var in list if expr ]
```

Here `var` is a variable name, and `list` is an existing sequence. The optional `if` part requires an expression that evaluates to `true` or `false`. Only those elements that evaluate to `true` are examined. To construct the new sequence, each element in the original list is examined. If it passes the `if` expression test, the initial expression is evaluated and the resulting value added to the new list. In this fashion, the list completion combines aspects of both a filter and a map. The following example illustrates the use of a list:

```
>>> a = [1, 2, 3, 4, 5]
>>> print [x*2 for x in a if x < 4]
[2, 4, 6]
```

List comprehensions are often simpler to read than the equivalent expression formed using filter and map, in part because lists do not require an explicit lambda function. However, both forms are useful, and a Python programmer should be familiar with both.

List comprehensions are often used as the body of a function. The function definition provides a convenient syntax and a way to provide names to arguments. The list comprehension is an easy-to-understand way to write the body of the function:

```
>>> def listOfSquares(a):
  return [x*x for x in a]
>>> listOfSquares([1, 2, 3])
[1, 4, 9]
```