**Understanding Functional Programming**

Functional programming defines a computation using expressions and evaluation; often these are 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 a functional programming language. It offers enough of the right kinds of features that it confers the benefits of functional programming. It also retains all the optimization power of an imperative programming language.

# Identifying a paradigm

It's difficult to be definitive on the universe of programming paradigms. For our purposes, we will distinguish between only two of the many paradigms: **functional programming** and **imperative programming**. One important distinguishing feature between these two is the concept of state.

In an imperative language, such as Python, the state of the computation is reflected by the values of the variables in the various namespaces; some kinds of statements make 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 the state. Python has other statements, such as **global** or **nonlocal**, which modify the rules for variables in a particular namespace. Statements such as **def**, **class**, and **import** change the processing context. Other statements such as **try**, **except**, **if**, **elif**, and **else** act as guards to modify how a collection of statements will change the computation's state. Statements such as **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 assignment statement advances the state of the computation from an initial condition toward the desired final outcome. This advancing 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 the 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 functions, we can design lower-level functions that are easy to understand, and then 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 compared to imperative (object-oriented or procedural) programs. The benefit isn't automatic; it requires a careful design. This design effort for functional programming is often easier than for procedural programming.

# Using the functional paradigm

In a functional sense, the sum of the multiples of three and five 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:

**def sumr(seq):**

**if len(seq) == 0: return 0**

**return seq[0] + sumr(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.

# Higher-order functions

We can achieve expressive, succinct programs using higher-order functions. These are functions that accept a function as an argument or return a function as a value. We can use higher-order functions as a way to create composite functions from simpler functions.

Consider the Python **max()** function. We can provide a function as an argument and modify how the **max()** function behaves.

Here's some data we might want to process:

**>>> year\_cheese = [(2000, 29.87), (2001, 30.12), (2002, 30.6), (2003,**   
**30.66),(2004, 31.33), (2005, 32.62), (2006, 32.73), (2007, 33.5),**   
**(2008, 32.84), (2009, 33.02), (2010, 32.92)]**

We can apply the **max()** function, as follows:

**>>> max(year\_cheese)**

**(2010, 32.92)**

# Strict and non-strict evaluation

Functional programming's efficiency stems, in part, from being able to defer a computation until it's required. The idea of lazy or non-strict evaluation is very helpful. To an extent, Python offers this feature.

In Python, the logical expression operators **and**, **or**, and **if-then-else** are all non-strict. We sometimes call them short-circuit operators because they don't need to evaluate all arguments to determine the resulting value.

The following command snippet shows the **and** operator's non-strict feature:

**>>> 0 and print("right")**

**0**

**>>> True and print("right")**

**right**

When we execute the first of the preceding command snippet, the left-hand side of the **and** operator is equivalent to **False**; the right-hand side is not evaluated. In the second example, when the left-hand side is equivalent to **True**, the right-hand side is evaluated.

Other parts of Python are strict. Outside the logical operators, an expression is evaluated eagerly from left to right. A sequence of statement lines is also evaluated strictly in order. Literal lists and **tuples** require eager evaluation.

When a class is created, the method functions are defined in a strict order. In the case of a class definition, the method functions are collected into a dictionary (by default) and order is not maintained after they're created. If we provide two methods with the same name, the second one is retained because of the strict evaluation order.

Python's generator expressions and generator functions, however, are lazy. These expressions don't create all possible results immediately. It's difficult to see this without explicitly logging the details of a calculation. Here is an example of the version of the **range()** function that has the side effect of showing the numbers it creates:

**def numbers():  
 for i in range(1024):**  **print(f"= {i}")  
 yield i**

To provide some debugging hints, this function prints each value as the value is yielded. If this function were eager, it would create all 1,024 numbers. Since it's lazy, it only creates numbers as requested.

*The older Python 2***range()***function was eager and created an actual list object with all of the requested numbers. The Python 3***range()***object is lazy, and will not create a large data structure.*

We can use this noisy **numbers()** function in a way that will show lazy evaluation. We'll write a function that evaluates some, but not all, of the values from this iterator:

**def sum\_to(n: int) -> int:  
 sum: int = 0  
 for i in numbers():  
 if i == n: break  
 sum += i  
 return sum**

The **sum\_to()** function has type hints to show that it should accept an integer value for the **n** parameter and return an integer result. The **sum** variable also includes Python 3 syntax  **:int**, which is a hint that it should be considered to be an integer. This function will not evaluate the entire result of the **numbers()** function. It will break after only consuming a few values from the **numbers()** function. We can see this consumption of values in the following log:

**>>> sum\_to(5)**

**= 0**

**= 1**

**= 2**

**= 3**

**= 4**

**= 5**

**10**

As we'll see later, Python generator functions have some properties that make them a little awkward for simple functional programming. Specifically, a generator can only be used once in Python. We have to be cautious with how we use the lazy Python generator expressions.