Chapter 2: Building Abstractions with Data

2.1 Introduction

We concentrated in Chapter 1 on computational processes and on the role of functions in program design. We saw how to use primitive data (numbers) and primitive operations (arithmetic), how to form compound functions through composition and control, and how to create functional abstractions by giving names to processes. We also saw that higher-order functions enhance the power of our language by enabling us to manipulate, and thereby to reason, in terms of general methods of computation. This is much of the essence of programming.

This chapter focuses on data. The techniques we investigate here will allow us to represent and manipulate information about many different domains. Due to the explosive growth of the Internet, a vast amount of structured information is freely available to all of us online, and computation can be applied to a vast range of different problems. Effective use of built-in and user-defined data types are fundamental to data processing applications.

2.1.1 Native Data Types

Every value in Python has a *class* that determines what type of value it is. Values that share a class also share behavior. For example, the integers `1` and `2` are both instances of the `int` class. These two values can be treated similarly. For example, they can both be negated or added to another integer. The built-in `type` function allows us to inspect the class of any value.

```
>>> type(2)
<class 'int'>
```

The values we have used so far are instances of a small number of *native* data types that are built into the Python language. Native data types have the following properties:

1. There are expressions that evaluate to values of native types, called *literals*.
2. There are built-in functions and operators to manipulate values of native types.

The `int` class is the native data type used to represent integers. Integer literals (sequences of adjacent numerals) evaluate to `int` values, and mathematical operators manipulate these values.

```
>>> 12 + 3000000000000000000000000
3000000000000000000000012
```

Python includes three native numeric types: integers (`int`), real numbers (`float`), and complex numbers (`complex`).

```
>>> type(1.5)
<class 'float'>
>>> type(1+1j)
<class 'complex'>
```

Floats. The name `float` comes from the way in which real numbers are represented in Python and many other programming languages: a "floating point" representation. While the details of how numbers are represented is not a topic for this text, some high-level differences between `int` and `float` objects are important to know. In particular, `int` objects represent integers exactly, without any approximation or limits on their size. On the other hand, `float` objects can represent a wide range of fractional numbers, but not all numbers can be represented exactly, and there are minimum and maximum values. Therefore, `float` values should be treated as approximations to real values. These approximations have only a finite amount of precision. Combining `float` values can lead to approximation errors; both of the following expressions would evaluate to `7` if not for approximation.

```
>>> 7 / 3 * 3
7.0
```

```
>>> 1 / 3 * 7 * 3
6.999999999999999
```

Although `int` values are combined above, dividing one `int` by another yields a `float` value: a truncated finite approximation to the actual ratio of the two integers divided.

```
>>> type(1/3)
<class 'float'>
>>> 1/3
0.3333333333333333
```

Problems with this approximation appear when we conduct equality tests.

```
>>> 1/3 == 0.333333333333333312345  # Beware of float approximation
True
```

These subtle differences between the `int` and `float` class have wide-ranging consequences for writing programs, and so they are details that must be memorized by programmers. Fortunately, there are only a handful of native data types, limiting the amount of memorization required to become proficient in a programming language. Moreover, these same details are consistent across many programming languages, enforced by community guidelines such as the IEEE 754 floating point standard.

Non-numeric types. Values can represent many other types of data, such as sounds, images, locations, web addresses, network connections, and more. A few are represented by native data types, such as the `bool` class for values `True` and `False`. The type for most values must be defined by programmers using the means of combination and abstraction that we will develop in this chapter.

The following sections introduce more of Python's native data types, focusing on the role they play in creating useful data abstractions. For those interested in further details, a chapter on native data types in the online book Dive Into Python 3 gives a pragmatic overview of all Python's native data types and how to manipulate them, including numerous usage examples and practical tips.

Continue: 2.2 Data Abstraction