1.2.3 Python's Built-In Classes

Table 1.2 provides a summary of commonly used, built-in classes in Python; we take particular note of which classes are mutable and which are immutable. A class is ***immutable*** if each object of that class has a fixed value upon instantiation that cannot subsequently be changed. For example, the float class is immutable. Once an instance has been created, its value cannot be changed (although an identifier referencing that object can be reassigned to a different value).

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Table 1.2: Commonly used built-in classes for Python

In this section, we provide an introduction to these classes, discussing their purpose and presenting several means for creating instances of the classes. Literal forms (such as 98.6) exist for most of the built-in classes, and all of the classes support a traditional constructor form that creates instances that are based upon one or more existing values. Operators supported by these classes are described in Section 1.3. More detailed information about these classes can be found in later chapters as follows: lists and tuples (Chapter 5); strings (Chapters 5 and 13, and Appendix A); sets and dictionaries (Chapter 10).

The bool Class

The **bool** class is used to manipulate logical (Boolean) values, and the only two instances of that class are expressed as the literals True and False. The default constructor, bool( ), returns False, but there is no reason to use that syntax rather than the more direct literal form. Python allows the creation of a Boolean value from a nonboolean type using the syntax bool(foo) for value foo. The interpretation depends upon the type of the parameter. Numbers evaluate to False if zero, and True if nonzero. Sequences and other container types, such as strings and lists, evaluate to False if empty and True if nonempty. An important application of this interpretation is the use of a nonboolean value as a condition in a control structure.

The int Class

The **int** and **float** classes are the primary numeric types in Python. The **int** class is designed to represent integer values with arbitrary magnitude. Unlike Java and C++, which support different integral types with different precisions (e.g., int, short, long), Python automatically chooses the internal representation for an integer based upon the magnitude of its value. Typical literals for integers include 0, 137, and −23. In some contexts, it is convenient to express an integral value using binary, octal, or hexadecimal. That can be done by using a prefix of the number 0 and then a character to describe the base. Example of such literals are respectively 0b1011, 0o52, and 0x7f.

The integer constructor, int( ), returns value 0 by default. But this constructor can be used to construct an integer value based upon an existing value of another type. For example, if f represents a floating-point value, the syntax int(f) produces the *truncated* value of f. For example, both int(3.14) and int(3.99) produce the value 3, while int(−3.9) produces the value −3. The constructor can also be used to parse a string that is presumed to represent an integral value (such as one entered by a user). If s represents a string, then int(s) produces the integral value that string represents. For example, the expression int('137') produces the integer value 137. If an invalid string is given as a parameter, as in int('hello'), a ValueError is raised (see Section 1.7 for discussion of Python's exceptions). By default, the string must use base 10. If conversion from a different base is desired, that base can be indicated as a second, optional, parameter. For example, the expression int('7f', 16) evaluates to the integer 127.

The float Class

The **float** class is the sole floating-point type in Python, using a fixed-precision representation. Its precision is more akin to a double in Java or C++, rather than those languages' float type. We have already discussed a typical literal form, 98.6. We note that the floating-point equivalent of an integral number can be expressed directly as 2.0. Technically, the trailing zero is optional, so some programmers might use the expression 2. to designate this floating-point literal. One other form of literal for floating-point values uses scientific notation. For example, the literal 6.022e23 represents the mathematical value 6.022×10^{23}.

The constructor form of float( ) returns 0.0. When given a parameter, the constructor attempts to return the equivalent floating-point value. For example, the call float(2) returns the floating-point value 2.0. If the parameter to the constructor is a string, as with float('3.14'), it attempts to parse that string as a floating-point value, raising a ValueError as an exception.

Sequence Types: The list, tuple, and str Classes

The **list**, **tuple**, and **str** classes are ***sequence*** types in Python, representing a collection of values in which the order is significant. The list class is the most general, representing a sequence of arbitrary objects (akin to an "array" in other languages). The tuple class is an ***immutable*** version of the list class, benefiting from a streamlined internal representation. The str class is specially designed for representing an immutable sequence of text characters. We note that Python does not have a separate class for characters; they are just strings with length one.

The list Class

A **list** instance stores a sequence of objects. A list is a ***referential*** structure, as it technically stores a sequence of ***references*** to its elements (see Figure 1.4). Elements of a list may be arbitrary objects (including the None object). Lists are ***array-based*** sequences and are ***zero-indexed***, thus a list of length n has elements indexed from 0 to $n-1$ inclusive. Lists are perhaps the most used container type in Python and they will be extremely central to our study of data structures and algorithms. They have many valuable behaviors, including the ability to dynamically expand and contract their capacities as needed. In this chapter, we will discuss only the most basic properties of lists. We revisit the inner working of all of Python's sequence types as the focus of Chapter 5.

Python uses the characters [ ] as delimiters for a list literal, with [ ] itself being an empty list. As another example, `['red', 'green', 'blue']` is a list containing three string instances. The contents of a list literal need not be expressed as literals; if identifiers a and b have been established, then syntax [a, b] is legitimate.

The list( ) constructor produces an empty list by default. However, the constructor will accept any parameter that is of an ***iterable*** type. We will discuss iteration further in Section 1.8, but examples of iterable types include all of the standard container types (e.g., strings, list, tuples, sets, dictionaries). For example, the syntax list(`'hello'`) produces a list of individual characters, `['h', 'e', 'l', 'l', 'o']`. Because an existing list is itself iterable, the syntax backup = list(data) can be used to construct a new list instance referencing the same contents as the original.

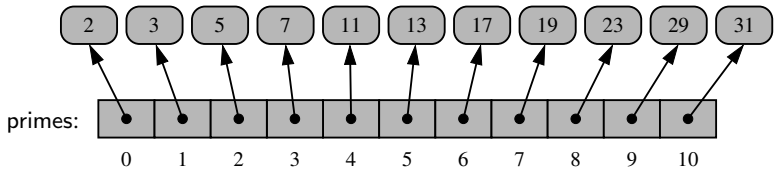

Figure 1.4: Python's internal representation of a list of integers, instantiated as prime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]. The implicit indices of the elements are shown below each entry.

The tuple Class

The **tuple** class provides an immutable version of a sequence, and therefore its instances have an internal representation that may be more streamlined than that of a list. While Python uses the [ ] characters to delimit a list, parentheses delimit a tuple, with ( ) being an empty tuple. There is one important subtlety. To express a tuple of length one as a literal, a comma must be placed after the element, but within the parentheses. For example, (17,) is a one-element tuple. The reason for this requirement is that, without the trailing comma, the expression (17) is viewed as a simple parenthesized numeric expression.

The str Class

Python's **str** class is specifically designed to efficiently represent an immutable sequence of characters, based upon the Unicode international character set. Strings have a more compact internal representation than the referential lists and tuples, as portrayed in Figure 1.5.

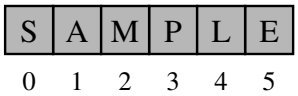

Figure 1.5: A Python string, which is an indexed sequence of characters.

String literals can be enclosed in single quotes, as in `'hello'`, or double quotes, as in `"hello"`. This choice is convenient, especially when using another of the quotation characters as an actual character in the sequence, as in `"Don't worry"`. Alternatively, the quote delimiter can be designated using a backslash as a so-called ***escape character***, as in `'Don\'t worry'`. Because the backslash has this purpose, the backslash must itself be escaped to occur as a natural character of the string literal, as in `'C:\\Python\\'`, for a string that would be displayed as `C:\Python\`. Other commonly escaped characters are `\n` for newline and `\t` for tab. Unicode characters can be included, such as `'20\u20AC'` for the string 20€.

Python also supports using the delimiter `'''` or `"""` to begin and end a string literal. The advantage of such triple-quoted strings is that newline characters can be embedded naturally (rather than escaped as `\n`). This can greatly improve the readability of long, multiline strings in source code. For example, at the beginning of Code Fragment 1.1, rather than use separate print statements for each line of introductory output, we can use a single print statement, as follows:

```
print("""Welcome to the GPA calculator.
Please enter all your letter grades, one per line.
Enter a blank line to designate the end.""")
```

The set and frozenset Classes

Python's **set** class represents the mathematical notion of a set, namely a collection of elements, without duplicates, and without an inherent order to those elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a ***hash table*** (which will be the primary topic of Chapter 10). However, there are two important restrictions due to the algorithmic underpinnings. The first is that the set does not maintain the elements in any particular order. The second is that only instances of ***immutable*** types can be added to a Python set. Therefore, objects such as integers, floating-point numbers, and character strings are eligible to be elements of a set. It is possible to maintain a set of tuples, but not a set of lists or a set of sets, as lists and sets are mutable. The **frozenset** class is an immutable form of the set type, so it is legal to have a set of frozensets.

Python uses curly braces { and } as delimiters for a set, for example, as {17} or {'red', 'green', 'blue'}. The exception to this rule is that { } does not represent an empty set; for historical reasons, it represents an empty dictionary (see next paragraph). Instead, the constructor syntax set( ) produces an empty set. If an iterable parameter is sent to the constructor, then the set of distinct elements is produced. For example, set('hello') produces {'h', 'e', 'l', 'o'}.

The dict Class

Python's **dict** class represents a ***dictionary***, or ***mapping***, from a set of distinct ***keys*** to associated ***values***. For example, a dictionary might map from unique student ID numbers, to larger student records (such as the student's name, address, and course grades). Python implements a dict using an almost identical approach to that of a set, but with storage of the associated values.

A dictionary literal also uses curly braces, and because dictionaries were introduced in Python prior to sets, the literal form { } produces an empty dictionary. A nonempty dictionary is expressed using a comma-separated series of key:value pairs. For example, the dictionary {'ga' : 'Irish', 'de' : 'German'} maps 'ga' to 'Irish' and 'de' to 'German'.

The constructor for the dict class accepts an existing mapping as a parameter, in which case it creates a new dictionary with identical associations as the existing one. Alternatively, the constructor accepts a sequence of key-value pairs as a parameter, as in dict(pairs) with pairs = [('ga', 'Irish'), ('de', 'German')].