CHAPTER 4

Introducing Python Object Types

This chapter begins our tour of the Python language. In an informal sense, in Python, we do things with stuff. "Things" take the form of operations like addition and concatenation, and "stuff" refers to the objects on which we perform those operations. In this part of the book, our focus is on that stuff, and the things our programs can do with it.

Somewhat more formally, in Python, data takes the form of *objects*—either built-in objects that Python provides, or objects we create using Python or external language tools such as C extension libraries. Although we'll firm up this definition later, objects are essentially just pieces of memory, with values and sets of associated operations.

Because objects are the most fundamental notion in Python programming, we'll start this chapter with a survey of Python's built-in object types.

By way of introduction, however, let's first establish a clear picture of how this chapter fits into the overall Python picture. From a more concrete perspective, Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

The discussion of modules in Chapter 3 introduced the highest level of this hierarchy. This part's chapters begin at the bottom, exploring both built-in objects and the expressions you can code to use them.

Why Use Built-in Types?

If you've used lower-level languages such as C or C++, you know that much of your work centers on implementing *objects*—also known as *data structures*—to represent the components in your application's domain. You need to lay out memory structures, manage memory allocation, implement search and access routines, and so on. These chores are about as tedious (and error prone) as they sound, and they usually distract from your program's real goals.

In typical Python programs, most of this grunt work goes away. Because Python provides powerful object types as an intrinsic part of the language, there's usually no need to code object implementations before you start solving problems. In fact, unless you have a need for special processing that built-in types don't provide, you're almost always better off using a built-in object instead of implementing your own. Here are some reasons why:

- **Built-in objects make programs easy to write.** For simple tasks, built-in types are often all you need to represent the structure of problem domains. Because you get powerful tools such as collections (lists) and search tables (dictionaries) for free, you can use them immediately. You can get a lot of work done with Python's built-in object types alone.
- **Built-in objects are components of extensions.** For more complex tasks, you still may need to provide your own objects, using Python classes or C language interfaces. But as you'll see in later parts of this book, objects implemented manually are often built on top of built-in types such as lists and dictionaries. For instance, a stack data structure may be implemented as a class that manages or customizes a built-in list.
- **Built-in objects are often more efficient than custom data structures.** Python's built-in types employ already optimized data structure algorithms that are implemented in C for speed. Although you can write similar object types on your own, you'll usually be hard-pressed to get the level of performance built-in object types provide.
- **Built-in objects are a standard part of the language.** In some ways, Python borrows both from languages that rely on built-in tools (e.g., LISP) and languages that rely on the programmer to provide tool implementations or frameworks of their own (e.g., C++). Although you can implement unique object types in Python, you don't need to do so just to get started. Moreover, because Python's built-ins are standard, they're always the same; proprietary frameworks, on the other hand, tend to differ from site to site.

In other words, not only do built-in object types make programming easier, but they're also more powerful and efficient than most of what can be created from scratch. Regardless of whether you implement new object types, built-in objects form the core of every Python program.

Python's Core Data Types

Table 4-1 previews Python's built-in object types and some of the syntax used to code their *literals*—that is, the expressions that generate these objects.[*] Some of these types will probably seem familiar if you've used other languages; for instance, numbers and strings represent numeric and textual values, respectively, and files provide an interface for processing files stored on your computer.

Table 4-1. Built-in objects preview

Object type	Example literals/creation
Numbers	`1234, 3.1415, 999L, 3+4j, Decimal`
Strings	`'spam', "guido's"`
Lists	`[1, [2, 'three'], 4]`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}`
Tuples	`(1,'spam', 4, 'U')`
Files	`myfile = open('eggs', 'r')`
Other types	Sets, types, `None`, Booleans

Table 4-1 isn't really complete, because everything we process in Python programs is a kind of object. For instance, when we perform text pattern matching in Python, we create pattern objects, and when we perform network scripting, we use socket objects. These other kinds of objects are generally created by importing and using modules, and they have behavior all their own.

We call the object types in Table 4-1 *core* data types because they are effectively built into the Python language—that is, there is specific syntax for generating most of them. For instance, when you run the following code:

```
>>> 'spam'
```

you are, technically speaking, running a literal expression, which generates and returns a new string object. There is specific Python language syntax to make this object. Similarly, an expression wrapped in square brackets makes a list, one in curly braces makes a dictionary, and so on. Even though, as we'll see, there are no type declarations in Python, the syntax of the expressions you run determines the types of objects you create and use. In fact, object-generation expressions like those in Table 4-1 are generally where types originate in the Python language.

Just as importantly, once you create an object, you bind its operation set for all time—you can perform only string operations on a string and list operations on a

[*] In this book, the term *literal* simply means an expression whose syntax generates an object—sometimes also called a *constant*. Note that the term "constant" does not imply objects or variables that can never be changed (i.e., this term is unrelated to C++'s `const` or Python's "immutable"—a topic explored later in this chapter).

list. As you'll learn, Python is *dynamically typed* (it keeps track of types for you automatically instead of requiring declaration code), but it is also *strongly typed* (you can only perform on an object operations that are valid for its type).

Functionally, the object types in Table 4-1 are more general and powerful than what you may be accustomed to. For instance, you'll find that lists and dictionaries alone are powerful data representation tools that obviate most of the work you do to support collections and searching in lower-level languages. In short, lists provide ordered collections of other objects, while dictionaries store objects by key; both lists and dictionaries may be nested, can grow and shrink on demand, and may contain objects of any type.

We'll study each of the object types in Table 4-1 in detail in upcoming chapters. Before digging into the details, though, let's begin by taking a quick look at Python's core objects in action. The rest of this chapter provides a preview of the operations we'll explore in more depth in the chapters that follow. Don't expect to find the full story here—the goal of this chapter is just to whet your appetite and introduce some key ideas. Still, the best way to get started is to get started, so let's jump right into some real code.

Numbers

If you've done any programming or scripting in the past, some of the object types in Table 4-1 will probably seem familiar. Even if you haven't, numbers are fairly straightforward. Python's core object set includes the usual suspects: integers (numbers without a fractional part), floating-point numbers (roughly, numbers with a decimal point in them), and more exotic types (unlimited-precision "long" integers, complex numbers with imaginary parts, fixed-precision decimals, and sets).

Although it offers some fancier options, Python's basic number types are, well, basic. Numbers in Python support the normal mathematical operations. For instance, the plus sign (+) performs addition, a star (*) is used for multiplication, and two stars (**) are used for exponentiation:

```
>>> 123 + 222                       # Integer addition
345
>>> 1.5 * 4                         # Floating-point multiplication
6.0
>>> 2 ** 100                        # 2 to the power 100
1267650600228229401496703205376L
```

Notice the *L* at the end of the last operation's result here: Python automatically converts up to a long integer type when extra precision is needed. You can, for instance, compute 2 to the 1,000,000 power in Python (but you probably shouldn't try to print the result—with more than 300,000 digits, you may be waiting awhile!). Watch what happens when some floating-point numbers are printed:

```
>>> 3.1415 * 2                               # repr: as code
6.2830000000000004
>>> print 3.1415 * 2                         # str: user-friendly
6.283
```

The first result isn't a bug; it's a display issue. It turns out that there are two ways to print every object: with full precision (as in the first result shown here), and in a user-friendly form (as in the second). Formally, the first form is known as an object's as-code `repr`, and the second is its user-friendly `str`. The difference can matter when we step up to using classes; for now, if something looks odd, try showing it with a `print` statement.

Besides expressions, there are a handful of useful numeric modules that ship with Python:

```
>>> import math
>>> math.pi
3.1415926535897931
>>> math.sqrt(85)
9.2195444572928871
```

The `math` module contains more advanced numeric tools as functions, while the `random` module performs random number generation and random selections (here, from a Python list, introduced later in this chapter):

```
>>> import random
>>> random.random( )
0.59268735266273953
>>> random.choice([1, 2, 3, 4])
1
```

Python also includes more exotic number objects—such as complex numbers, fixed-precision decimal numbers, and sets—and the third-party open source extension domain has even more (e.g., matrixes and vectors). We'll defer discussion of the details of these types until later in the book.

So far, we've been using Python much like a simple calculator; to do better justice to its built-in types, let's move on to explore strings.

Strings

Strings are used to record textual information as well as arbitrary collections of bytes. They are our first example of what we call a *sequence* in Python—that is, a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative position. Strictly speaking, strings are sequences of one-character strings; other types of sequences include lists and tuples (covered later).

Sequence Operations

As sequences, strings support operations that assume a positional ordering among items. For example, if we have a four-character string, we can verify its length with the built-in `len` function and fetch its components with *indexing* expressions:

```
>>> S = 'Spam'
>>> len(S)                # Length
4
>>> S[0]                  # The first item in S, indexing by zero-based position
'S'
>>> S[1]                  # The second item from the left
'p'
```

In Python, indexes are coded as offsets from the front, and so start from 0: the first item is at index 0, the second is at index 1, and so on. In Python, we can also index backward, from the end:

```
>>> S[-1]                 # The last item from the end in S
'm'
>>> S[-2]                 # The second to last item from the end
'a'
```

Formally, a negative index is simply added to the string's size, so the following two operations are equivalent (though the first is easier to code and less easy to get wrong):

```
>>> S[-1]                 # The last item in S
'm'
>>> S[len(S)-1]           # Negative indexing, the hard way
'm'
```

Notice that we can use an arbitrary expression in the square brackets, not just a hardcoded number literal—anywhere that Python expects a value, we can use a literal, a variable, or any expression. Python's syntax is completely general this way.

In addition to simple positional indexing, sequences also support a more general form of indexing known as *slicing*, which is a way to extract an entire section (slice) in a single step. For example:

```
>>> S                     # A 4-character string
'Spam'
>>> S[1:3]                # Slice of S from offsets 1 through 2 (not 3)
'pa'
```

Probably the easiest way to think of slices is that they are a way to extract an entire *column* from a string in a single step. Their general form, `X[I:J]`, means "give me everything in `X` from offset `I` up to but not including offset `J`." The result is returned in a new object. The last operation above, for instance, gives us all the characters in string `S` from offsets 1 through 2 (that is, 3–1) as a new string. The effect is to slice or "parse out" the two characters in the middle.

In a slice, the left bound defaults to zero, and the right bound defaults to the length of the sequence being sliced. This leads to some common usage variations:

```
>>> S[1:]                    # Everything past the first (1:len(S))
'pam'
>>> S                        # S itself hasn't changed
'Spam'
>>> S[0:3]                   # Everything but the last
'Spa'
>>> S[:3]                    # Same as S[0:3]
'Spa'
>>> S[:-1]                   # Everything but the last again, but simpler (0:-1)
'Spa'
>>> S[:]                     # All of S as a top-level copy (0:len(S))
'Spam'
```

Note how negative offsets can be used to give bounds for slices, too, and how the last operation effectively copies the entire string. As you'll learn later, there is no reason to copy a string, but this form can be useful for sequences like lists.

Finally, as sequences, strings also support *concatenation* with the plus sign (joining two strings into a new string), and *repetition* (making a new string by repeating another):

```
>>> S
'Spam'
>>> S + 'xyz'                          # Concatenation
'Spamxyz'
>>> S                                  # S is unchanged
'Spam'
>>> S * 8                              # Repetition
'SpamSpamSpamSpamSpamSpamSpamSpam'
```

Notice that the plus sign (+) means different things for different objects: addition for numbers, and concatenation for strings. This is a general property of Python that we'll call *polymorphism* later in the book—in sum, the meaning of an operation depends on the objects being operated on. As you'll see when we study dynamic typing, this polymorphism property accounts for much of the conciseness and flexibility of Python code. Because types aren't constrained, a Python-coded operation can normally work on many different types of objects automatically, as long as they support a compatible interface (like the + operation here). This turns out to be a huge idea in Python; you'll learn more about it later on our tour.

Immutability

Notice that in the prior examples, we were not changing the original string with any of the operations we ran on it. Every string operation is defined to produce a new string as its result, because strings are *immutable* in Python—they cannot be changed in-place after they are created. For example, you can't change a string by assigning to one of its positions, but you can always build a new one and assign it to the same name. Because Python cleans up old objects as you go (as you'll see later), this isn't as inefficient as it may sound:

```
>>> S
'Spam'
>>> S[0] = 'z'              # Immutable objects cannot be changed
...error text omittted...
TypeError: 'str' object does not support item assignment

>>> S = 'z' + S[1:]         # But we can run expressions to make new objects
>>> S
'zpam'
```

Every object in Python is classified as immutable (unchangeable) or not. In terms of the core types, numbers, strings, and tuples are immutable; lists and dictionaries are not (they can be changed in-place freely). Among other things, immutability can be used to guarantee that an object remains constant throughout your program.

Type-Specific Methods

Every string operation we've studied so far is really a sequence operation—that is, these operations will work on other sequences in Python as well, including lists and tuples. In addition to generic sequence operations, though, strings also have operations all their own, available as *methods* (functions attached to the object, which are triggered with a call expression).

For example, the string `find` method is the basic substring search operation (it returns the offset of the passed-in substring, or `-1` if it is not present), and the string `replace` method performs global searches and replacements:

```
>>> S.find('pa')             # Find the offset of a substring
1
>>> S
'Spam'
>>> S.replace('pa', 'XYZ')   # Replace occurrences of a substring with another
'SXYZm'
>>> S
'Spam'
```

Again, despite the names of these string methods, we are not changing the original strings here, but creating new strings as the results—because strings are immutable, we have to do it this way. String methods are the first line of text-processing tools in Python; other methods split a string into substrings on a delimiter (handy as a simple form of parsing), perform case conversions, test the content of the string (digits, letters, and so on), and strip whitespace characters off the ends of the string:

```
>>> line = 'aaa,bbb,ccccc,dd'
>>> line.split(',')          # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'ccccc', 'dd']
```

```
>>> S = 'spam'
>>> S.upper( )                # Upper- and lowercase conversions
'SPAM'

>>> S.isalpha( )              # Content tests: isalpha, isdigit, etc.
True

>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line = line.rstrip( )     # Remove whitespace characters on the right side
>>> line
'aaa,bbb,ccccc,dd'
```

One note here: although sequence operations are generic, methods are not—string method operations work only on strings, and nothing else. As a rule of thumb, Python's toolset is layered: generic operations that span multiple types show up as built-in functions or expressions (e.g., `len(X)`, `X[0]`), but type-specific operations are method calls (e.g., `aString.upper( )`). Finding the tools you need among all these categories will become more natural as you use Python more, but the next section gives a few tips you can use right now.

Getting Help

The methods introduced in the prior section are a representative, but small, sample of what is available for string objects. In general, this book is not exhaustive in its look at object methods. For more details, you can always call the built-in `dir` function, which returns a list of all the attributes available in a given object. Because methods are function attributes, they will show up in this list:

```
>>> dir(S)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__',
'__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__str__', 'capitalize', 'center',
'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

You probably won't care about the names with underscores in this list until later in the book, when we study operator overloading in classes—they represent the implementation of the string object, and are available to support customization. In general, leading and trailing double underscores is the naming pattern Python uses for implementation details. The names without the underscores in this list are the callable methods on string objects.

The `dir` function simply gives the methods' names. To ask what they do, you can pass them to the `help` function:

```
>>> help(S.index)
Help on built-in function index:

index(...)
    S.index(sub [,start [,end]]) -> int

    Like S.find( ) but raise ValueError when the substring is not found.
```

`help` is one of a handful of interfaces to a system of code that ships with Python known as *PyDoc*—a tool for extracting documentation from objects. Later in the book, you'll see that PyDoc can also render its reports in HTML format.

You can also ask for help on an entire string (e.g., `help(S)`), but you may get more help than you want to see—i.e., information about every string method. It's generally better to ask about a specific method, as we did above.

For more details, you can also consult Python's standard library reference manual, or commercially published reference books, but `dir` and `help` are the first line of documentation in Python.

Other Ways to Code Strings

So far, we've looked at the string object's sequence operations and type-specific methods. Python also provides a variety of ways for us to code strings, which we'll explore further later (with special characters represented as backslash escape sequences, for instance):

```
>>> S = 'A\nB\tC'        # \n is end-of-line, \t is tab
>>> len(S)               # Each stands for just one character
5

>>> ord('\n')            # \n is a byte with the binary value 10 in ASCII
10

>>> S = 'A\0B\0C'        # \0, the binary zero byte, does not terminate the string
>>> len(S)
5
```

Python allows strings to be enclosed in single or double quote characters (they mean the same thing). It also has a multiline string literal form enclosed in triple quotes (single or double)—when this form is used, all the lines are concatenated together, and end-of-line characters are added where line breaks appear. This is a minor syntactic convenience, but it's useful for embedding things like HTML and XML code in a Python script:

```
>>> msg = """
aaaaaaaaaaaaa
bbb'''bbbbbbbbbb""bbbbbbb'bbbb
cccccccccccccc"""
```

```
>>> msg
'\naaaaaaaaaaaaa\nbbb\'\'\'bbbbbbbbbb""bbbbbbb\'bbbb\ncccccccccccccc'
```

Python also supports a "raw" string literal that turns off the backslash escape mechanism (they start with the letter *r*), as well as a Unicode string form that supports internationalization (they begin with the letter *u* and contain multibyte characters). Technically, Unicode string is a different data type than normal string, but it supports all the same string operations. We'll meet all these special string forms in later chapters.

Pattern Matching

One point worth noting before we move on is that none of the string object's methods support pattern-based text processing. Text pattern matching is an advanced tool outside this book's scope, but readers with backgrounds in other scripting languages may be interested to know that to do pattern matching in Python, we import a module called re. This module has analogous calls for searching, splitting, and replacement, but because we can use patterns to specify substrings, we can be much more general:

```
>>> import re
>>> match = re.match('Hello[ \t]*(.*)world', 'Hello    Python world')
>>> match.group(1)
'Python '
```

This example searches for a substring that begins with the word "Hello," followed by zero or more tabs or spaces, followed by arbitrary characters to be saved as a matched group, terminated by the word "world." If such as substring is found, portions of the substring matched by parts of the pattern enclosed in parentheses are available as groups. The following pattern, for example, picks out three groups separated by slashes:

```
>>> match = re.match('/(.*)/(.*)/(.*)', '/usr/home/lumberjack')
>>> match.groups()
('usr', 'home', 'lumberjack')
```

Pattern matching is a fairly advanced text-processing tool by itself, but there is also support in Python for even more advanced language processing, including natural language processing. I've already said enough about strings for this tutorial, though, so let's move on to the next type.

Lists

The Python list object is the most general sequence provided by the language. Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size. They are also mutable—unlike strings, lists can be modified in-place by assignment to offsets as well as a variety of list method calls.

Sequence Operations

Because they are sequences, lists support all the sequence operations we discussed for strings; the only difference is that results are usually lists instead of strings. For instance, given a three-item list:

```
>>> L = [123, 'spam', 1.23]          # A list of three different-type objects
>>> len(L)                           # Number of items in the list
3
```

we can index, slice, and so on, just as for strings:

```
>>> L[0]                             # Indexing by position
123

>>> L[:-1]                           # Slicing a list returns a new list
[123, 'spam']

>>> L + [4, 5, 6]                    # Concatenation makes a new list too
[123, 'spam', 1.23, 4, 5, 6]

>>> L                                # We're not changing the original list
[123, 'spam', 1.23]
```

Type-Specific Operations

Python's lists are related to arrays in other languages, but they tend to be more powerful. For one thing, they have no fixed type constraint—the list we just looked at, for example, contains three objects of completely different types (an integer, a string, and a floating-point number). Further, lists have no fixed size. That is, they can grow and shrink on demand, in response to list-specific operations:

```
>>> L.append('NI')                   # Growing: add object at end of list
>>> L
[123, 'spam', 1.23, 'NI']

>>> L.pop(2)                         # Shrinking: delete an item in the middle
1.23

>>> L                               # "del L[2]" deletes from a list too
[123, 'spam', 'NI']
```

Here, the list `append` method expands the list's size and inserts an item at the end; the `pop` method (or an equivalent `del` statement) then removes an item at a given offset, causing the list to shrink. Other list methods insert items at an arbitrary position (`insert`), remove a given item by value (`remove`), and so on. Because lists are mutable, most list methods also change the list object in-place, instead of creating a new one:

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']
```

```
>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

The list sort method here, for example, orders the list in ascending fashion by default, and reverse reverses it—in both cases, the methods modify the list directly.

Bounds Checking

Although lists have no fixed size, Python still doesn't allow us to reference items that are not present. Indexing off the end of a list is always a mistake, but so is assigning off the end:

```
>>> L
[123, 'spam', 'NI']

>>> L[99]
...error text omitted...
IndexError: list index out of range

>>> L[99] = 1
...error text omitted...
IndexError: list assignment index out of range
```

This is on purpose, as it's usually an error to try to assign off the end of a list (and a particularly nasty one in the C language, which doesn't do as much error checking as Python). Rather than silently growing the list in response, Python reports an error. To grow a list, we call list methods such as append instead.

Nesting

One nice feature of Python's core data types is that they support arbitrary nesting—we can nest them in any combination, and as deeply as we like (for example, we can have a list that contains a dictionary, which contains another list, and so on). One immediate application of this feature is to represent matrixes, or "multidimensional arrays" in Python. A list with nested lists will do the job for basic applications:

```
>>> M = [[1, 2, 3],               # A 3 x 3 matrix, as nested lists
         [4, 5, 6],
         [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Here, we've coded a list that contains three other lists. The effect is to represent a 3× 3 matrix of numbers. Such a structure can be accessed in a variety of ways:

```
>>> M[1]                          # Get row 2
[4, 5, 6]

>>> M[1][2]                       # Get row 2, then get item 3 within the row
6
```

The first operation here fetches the entire second row, and the second grabs the third item within that row—stringing together index operations takes us deeper and deeper into our nested-object structure.[*]

List Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation known as a *list comprehension expression*, which turns out to be a powerful way to process structures like our matrix. Suppose, for instance, that we need to extract the second column of our sample matrix. It's easy to grab rows by simple indexing because the matrix is stored by rows, but it's almost as easy to get a column with a list comprehension:

```
>>> col2 = [row[1] for row in M]              # Collect the items in column 2
>>> col2
[2, 5, 8]

>>> M                                         # The matrix is unchanged
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

List comprehensions derive from set notation; they are a way to build a new list by running an expression on each item in a sequence, one at a time, from left to right. List comprehensions are coded in square brackets (to tip you off to the fact that they make a list), and are composed of an expression and a looping construct that share a variable name (`row`, here). The preceding list comprehension means basically what it says: "Give me `row[1]` for each row in matrix `M`, in a new list." The result is a new list containing column 2 of the matrix.

List comprehensions can be more complex in practice:

```
>>> [row[1] + 1 for row in M]                 # Add 1 to each item in column 2
[3, 6, 9]

>>> [row[1] for row in M if row[1] % 2 == 0]  # Filter out odd items
[2, 8]
```

The first operation here, for instance, adds 1 to each item as it is collected, and the second uses an `if` clause to filter odd numbers out of the result using the `%` modulus expression (remainder of division). List comprehensions make new lists of results, but can be used to iterate over any iterable object—here, for instance, we'll use list comprehensions to step over a hardcoded list of coordinates, and a string:

[*] This matrix structure works for small-scale tasks, but for more serious number crunching, you will probably want to use one of the numeric extensions to Python, such as the open source NumPy system. Such tools can store and process large matrixes much more efficiently than our nested list structure. NumPy has been said to turn Python into the equivalent of a free and more powerful version of the MatLab system, and organizations such as NASA, Los Alamos, and JPMorgan Chase use this tool for scientific and financial tasks. Search the Web for more details.

```
>>> diag = [M[i][i] for i in [0, 1, 2]]          # Collect a diagonal from matrix
>>> diag
[1, 5, 9]

>>> doubles = [c * 2 for c in 'spam']            # Repeat characters in a string
>>> doubles
['ss', 'pp', 'aa', 'mm']
```

List comprehensions are a bit too involved for me to say more about them here. The main point of this brief introduction is to illustrate that Python includes both simple and advanced tools in its arsenal. List comprehensions are an optional feature, but they tend to be handy in practice, and often provide a substantial processing speed advantage. They also work on any type that is a sequence in Python, as well as some types that are not. You'll hear more about them later in this book.

Dictionaries

Python dictionaries are something completely different (Monty Python reference intended)—they are not sequences at all, but are instead known as *mappings*. Mappings are also collections of other objects, but they store objects by key instead of by relative position. In fact, mappings don't maintain any reliable left-to-right order; they simply map keys to associated values. Dictionaries, the only mapping type in Python's core objects set, are also mutable: they may be changed in-place, and can grow and shrink on demand, like lists.

Mapping Operations

When written as literals, dictionaries are coded in curly braces, and consist of a series of "key: value" pairs. Dictionaries are useful anytime we need to associate a set of values with keys—to describe the properties of something, for instance. As an example, consider the following three-item dictionary (with keys "food," "quantity," and "color"):

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```

We can index this dictionary by key to fetch and change the keys' associated values. The dictionary index operation uses the same syntax as that used for sequences, but the item in the square brackets is a key, not a relative position:

```
>>> D['food']                          # Fetch value of key 'food'
'Spam'

>>> D['quantity'] += 1                 # Add 1 to 'quantity' value
>>> D
{'food': 'Spam', 'color': 'pink', 'quantity': 5}
```

Although the curly-braces literal form does see use, it is perhaps more common to see dictionaries built up in different ways. The following, for example, starts with an empty dictionary, and fills it out one key at a time. Unlike out-of-bounds assignments in lists, which are forbidden, an assignment to new dictionary key creates that key:

```
>>> D = {}
>>> D['name'] = 'Bob'                # Create keys by assignment
>>> D['job']  = 'dev'
>>> D['age']  = 40

>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}

>>> print D['name']
Bob
```

Here, we're effectively using dictionary keys as field names in a record that describes someone. In other applications, dictionaries can also be used to replace searching operations—indexing a dictionary by key is often the fastest way to code a search in Python.

Nesting Revisited

In the prior example, we used a dictionary to describe a hypothetical person, with three keys. Suppose, though, that the information is more complex. Perhaps we need to record a first name and a last name, along with multiple job titles. This leads to another application of Python's object nesting in action. The following dictionary, coded all at once as a literal, captures more structured information:

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
           'job':  ['dev', 'mgr'],
           'age':  40.5}
```

Here, we again have a three-key dictionary at the top (keys "name," "job," and "age"), but the values have become more complex: a nested dictionary for the name to support multiple parts, and a nested list for the job to support multiple roles and future expansion. We can access the components of this structure much as we did for our matrix earlier, but this time some of our indexes are dictionary keys, not list offsets:

```
>>> rec['name']                       # 'Name' is a nested dictionary
{'last': 'Smith', 'first': 'Bob'}

>>> rec['name']['last']               # Index the nested dictionary
'Smith'

>>> rec['job']                        # 'Job' is a nested list
['dev', 'mgr']
```

```
>>> rec['job'][-1]                       # Index the nested list
'mgr'

>>> rec['job'].append('janitor')         # Expand Bob's job description in-place
>>> rec
{'age': 40.5, 'job': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith', 'first':
'Bob'}}
```

Notice how the last operation here expands the nested job list—because the job list is a separate piece of memory from the dictionary that contains it, it can grow and shrink freely (object memory layout will be discussed further later in this book).

The real reason for showing you this example is to demonstrate the flexibility of Python's core data types. As you can see, nesting allows us to build up complex information structures directly and easily. Building a similar structure in a low-level language like C would be tedious and require much more code: we would have to lay out and declare structures and arrays, fill out values, link everything together, and so on. In Python, this is all automatic—running the expression creates the entire nested object structure for us. In fact, this is one of the main benefits of scripting languages like Python.

Just as importantly, in a lower-level language, we would have to be careful to clean up all of the object's space when we no longer need it. In Python, when we lose the last reference to object—by assigning its variable to something else, for example—all of the memory space occupied by that object's structure is automatically cleaned up for us:

```
>>> rec = 0                              # Now the object's space is reclaimed
```

Technically speaking, Python has a feature known as *garbage collection* that cleans up unused memory as your program runs and frees you from having to manage such details in your code. In Python, the space is reclaimed immediately, as soon as the last reference to an object is removed. We'll study how this works later in this book; for now, it's enough to know that you can use objects freely, without worrying about creating their space or cleaning up as you go.*

Sorting Keys: for Loops

As mappings, as we've already seen, dictionaries only support accessing items by key. However, they also support type-specific operations with method calls that are useful in a variety of common use cases.

* One footnote here: keep in mind that the rec record we just created really could be a database record, when we employ Python's *object persistence* system—an easy way to store native Python objects in files or access-by-key databases. We won't go into more details here; see Python's pickle and shelve modules for more details.

As mentioned earlier, because dictionaries are not sequences, they don't maintain any dependable left-to-right order. This means that if we make a dictionary, and print it back, its keys may come back in a different order than how we typed them:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

What do we do, though, if we do need to impose an ordering on a dictionary's items? One common solution is to grab a list of keys with the dictionary `keys` method, sort that with the list `sort` method, and then step through the result with a Python `for` loop:

```
>>> Ks = D.keys( )                       # Unordered keys list
>>> Ks
['a', 'c', 'b']

>>> Ks.sort( )                           # Sorted keys list
>>> Ks
['a', 'b', 'c']

>>> for key in Ks:                       # Iterate though sorted keys
        print key, '=>', D[key]

a => 1
b => 2
c => 3
```

This is a three-step process, though, as we'll see in later chapters, in recent versions of Python it can be done in one step with the newer `sorted` built-in function (`sorted` returns the result and sorts a variety of object types):

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> for key in sorted(D):
        print key, '=>', D[key]

a => 1
b => 2
c => 3
```

This case serves as an excuse to introduce the Python `for` loop. The `for` loop is a simple and efficient way to step through all the items in a sequence and run a block of code for each item in turn. A user-defined loop variable (`key`, here) is used to reference the current item each time through. The net effect in our example is to print the unordered dictionary's keys and values, in sorted-key order.

The `for` loop, and its more general cousin the `while` loop, are the main ways we code repetitive tasks as statements in our scripts. Really, though, the `for` loop, like its relative the list comprehension (which we met earlier) is a sequence operation. It works on any object that is a sequence and, also like the list comprehension, even on some things that are not. Here, for example, it is stepping across the characters in a string, printing the uppercase version of each as it goes:

```
>>> for c in 'spam':
        print c.upper( )

S
P
A
M
```

We'll discuss looping statements further later in the book.

Iteration and Optimization

If the `for` loop looks like the list comprehension expression introduced earlier, it should: both are really general iteration tools. In fact, both will work on any object that follows the *iteration protocol*—an idea introduced recently in Python that essentially means a physically stored sequence in memory, or an object that generates one item at a time in the context of an iteration operation. This is why the `sorted` call used in the prior section works on the dictionary directly—we don't have to call the `keys` method to get a sequence because dictionaries are iterable objects.

I'll have more to say about the iteration protocol later in this book. For now, keep in mind that any list comprehension expression, such as this one, which computes the squares of a list of numbers:

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]
>>> squares
[1, 4, 9, 16, 25]
```

can always be coded as an equivalent `for` loop that builds the result list manually by appending as it goes:

```
>>> squares = []
>>> for x in [1, 2, 3, 4, 5]:          # This is what a list comp does
        squares.append(x ** 2)

>>> squares
[1, 4, 9, 16, 25]
```

The list comprehension, though, will generally run faster (perhaps even twice as fast)—a property that could matter in your programs for large data sets. Having said that, though, I should point out that performance measures are tricky business in Python because it optimizes so much, and can vary from release to release.

A major rule of thumb in Python is to code for simplicity and readability first, and worry about performance later, after your program is working, and after you've proved that there is a genuine performance concern. More often than not, your code will be quick enough as it is. If you do need to tweak code for performance, though, Python includes tools to help you out, including the `time` and `timeit` modules and the `profile` module. You'll find more on these later in this book, and in the Python manuals.

Missing Keys: if Tests

One other note about dictionaries before we move on. Although we can assign to a new key to expand a dictionary, fetching a nonexistent key is still a mistake:

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> D['e'] = 99                          # Assigning new keys grows dictionaries
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}

>>> D['f']                               # Referencing one is an error
...error text omitted...
KeyError: 'f'
```

This is what we want—it's usually a programming error to fetch something that isn't really there. But, in some generic programs, we can't always know what keys will be present when we write our code. How do we handle such cases and avoid the errors? One trick here is to test ahead of time. The dictionary `has_key` method allows us to query the existence of a key and branch on the result with a Python `if` statement:

```
>>> D.has_key('f')
False

>>> if not D.has_key('f'):
        print 'missing'

missing
```

I'll have much more to say about the `if` statement and statement syntax in general later in this book, but the form we're using here is straightforward: it consists of the word `if`, followed by an expression that is interpreted as a true or false result, followed by a block of code to run if the test is true. In its full form, the `if` statement can also have an `else` clause for a default case, and one or more `elif` (else if) clauses for other tests. It's the main selection tool in Python, and it's the way we code logic in our scripts.

There are other ways to create dictionaries and avoid accessing a nonexistent dictionary key (including the `get` method; the `in` membership expression; and the `try` statement, a tool we'll first meet in Chapter 10 that catches and recovers from exceptions altogether), but we'll save the details on those until a later chapter. Now, let's move on to tuples.

Tuples

The tuple object (pronounced "toople" or "tuhple," depending on who you ask) is roughly like a list that cannot be changed—tuples are sequences, like lists, but they are immutable, like strings. Syntactically, they are coded in parentheses instead of square brackets, and they support arbitrary types, nesting, and the usual sequence operations:

```
>>> T = (1, 2, 3, 4)                # A 4-item tuple
>>> len(T)                          # Length
4

>> T + (5, 6)                       # Concatenation
(1, 2, 3, 4, 5, 6)

>>> T[0]                            # Indexing, slicing, and more
1
```

The only real distinction for tuples is that they cannot be changed once created. That is, they are immutable sequences:

```
>>> T[0] = 2                        # Tuples are immutable
...error text omitted...
TypeError: 'tuple' object does not support item assignment
```

Why Tuples?

So, why have a type that is like a list, but supports fewer operations? Frankly, tuples are not generally used as often as lists in practice, but their immutability is the whole point. If you pass a collection of objects around your program as a list, it can be changed anywhere; if you use a tuple, it cannot. That is, tuples provide a sort of integrity constraint that is convenient in programs larger than those we can write here. We'll talk more about tuples later in the book. For now, though, let's jump ahead to our last major core type, the file.

Files

File objects are Python code's main interface to external files on your computer. They are a core type, but they're something of an oddball—there is no specific literal syntax for creating them. Rather, to create a file object, you call the built-in `open` function, passing in an external filename as a string, and a processing mode string. For example, to create an output file, you would pass in its name and the `'w'` processing mode string to write data:

```
>>> f = open('data.txt', 'w')       # Make a new file in output mode
>>> f.write('Hello\n')              # Write strings of bytes to it
>>> f.write('world\n')
>>> f.close( )                      # Close to flush output buffers to disk
```

This creates a file in the current directory, and writes text to it (the filename can be a full directory path if you need to access a file elsewhere on your computer). To read back what you just wrote, reopen the file in `'r'` processing mode, for reading input (this is the default if you omit the mode in the call). Then read the file's content into a string of bytes, and display it. A file's contents are always a string of bytes to your script, regardless of the type of data the file contains:

```
>>> f = open('data.txt')                # 'r' is the default processing mode
>>> bytes = f.read( )                   # Read entire file into a string
>>> bytes
'Hello\nworld\n'

>>> print bytes                         # Print interprets control characters
Hello
world

>>> bytes.split( )                      # File content is always a string
['Hello', 'world']
```

Other file object methods support additional features we don't have time to cover here. For instance, file objects provide more ways of reading and writing (`read` accepts an optional byte size, `readline` reads one line at a time, and so on), as well as other tools (`seek` moves to a new file position). We'll meet the full set of file methods later in this book, but if you want a quick preview now, run a `dir` call on the word `file` (the name of the file data type), and a `help` on any of the names that come back:

```
>>> dir(file)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__',
'__getattribute__', '__hash__', '__init__', '__iter__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__',
'close', 'closed', 'encoding', 'fileno', 'flush', 'isatty', 'mode',
'name', 'newlines', 'next', 'read', 'readinto', 'readline', 'readlines',
'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']

>>> help(file.seek)
...try it and see...
```

Other File-Like Tools

The `open` function is the workhorse for most file processing you will do in Python. For more advanced tasks, though, Python comes with additional file-like tools: pipes, fifos, sockets, keyed-access files, object persistence, descriptor-based files, relational and object-oriented database interfaces, and more. Descriptor files, for instance, support file locking and other low-level tools, and sockets provide an interface for networking and interprocess communication. We won't cover many of these topics in this book, but you'll find them useful once you start programming Python in earnest.

Other Core Types

Beyond the core types we've seen so far, there are others that may or may not qualify for membership, depending on how broad the category is defined to be. Sets, for example, are a recent addition to the language. Sets are containers of other objects created by calling the built-in `set` function, and they support the usual mathematical set operations:

```
>>> X = set('spam')
>>> Y = set(['h', 'a', 'm'])                # Make 2 sets out of sequences
>>> X, Y
(set(['a', 'p', 's', 'm']), set(['a', 'h', 'm']))

>>> X & Y                                   # Intersection
set(['a', 'm'])

>>> X | Y                                   # Union
set(['a', 'p', 's', 'h', 'm'])

>>> X - Y                                   # Difference
set(['p', 's'])
```

In addition, Python recently added decimal numbers (fixed-precision floating-point numbers) and Booleans (with predefined `True` and `False` objects that are essentially just the integers 1 and 0 with custom display logic), and it has long supported a special placeholder object called `None`:

```
>>> import decimal                          # Decimals
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal("4.141")

>>> 1 > 2, 1 < 2                            # Booleans
(False, True)
>>> bool('spam')
True

>>> X = None                                # None placeholder
>>> print X
None
>>> L = [None] * 100                        # Initialize a list of 100 Nones
>>> L
[None, None, None, None, None, None, None, None, None, None, None, None, None,
...a list of 100 Nones...]

>>> type(L)                                 # Types
<type 'list'>
>>> type(type(L))                           # Even types are objects
<type 'type'>
```

How to Break Your Code's Flexibility

I'll have more to say about all these types later in the book, but the last merits a few more words here. The type object allows code to check the types of the objects it uses. In fact, there are at least three ways to do so in a Python script:

```
>>> if type(L) == type([]):                    # Type testing, if you must...
        print 'yes'

yes
>>> if type(L) == list:                        # Using the type name
        print 'yes'

yes
>>> if isinstance(L, list):                    # Object-oriented tests
         print 'yes'

yes
```

Now that I've shown you all these ways to do type testing, however, I must mention that, as you'll see later in the book, doing so is almost always the wrong thing to do in a Python program (and often a sign of an ex-C programmer first starting to use Python). By checking for specific types in your code, you effectively break its flexibility—you limit it to working on just one type. Without such tests, your code may be able to work on a whole range of types.

This is related to the idea of polymorphism mentioned earlier, and it stems from Python's lack of type declarations. As you'll learn, in Python, we code to object *interfaces* (operations supported), not to types. Not caring about specific types means that code is automatically applicable to many of them—any object with a compatible interface will work, regardless of its specific type. Although type checking is supported—and even required, in some rare cases—you'll see that it's not usually the "Pythonic" way of thinking. In fact, you'll find that polymorphism is probably the key idea behind using Python well.

User-Defined Classes

We'll study object-oriented programming in Python—an optional but powerful feature of the language that cuts development time by supporting programming by customization—in depth later in this book. In abstract terms, though, classes define new types of objects that extend the core set, so they merit a passing glance here. Say, for example, that you wish to have a type of object that models employees. Although there is no such specific core type in Python, the following user-defined class might fit the bill:

```
>>> class Worker:
        def __init__(self, name, pay):          # Initialize when created
            self.name = name                    # Self is the new object
            self.pay  = pay
```

```
    def lastName(self):
        return self.name.split( )[-1]          # Split string on blanks
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)            # Update pay in-place
```

This class defines a new kind of object that will have `name` and `pay` attributes (sometimes called *state information*), as well as two bits of behavior coded as functions (normally called *methods*). Calling the class like a function generates instances of our new type, and the class' methods automatically receive the instance being processed by a given method call (in the `self` argument):

```
>>> bob = Worker('Bob Smith', 50000)          # Make two instances
>>> sue = Worker('Sue Jones', 60000)          # Each has name and pay
>>> bob.lastName( )                           # Call method: bob is self
'Smith'
>>> sue.lastName( )                           # Sue is the self subject
'Jones'
>>> sue.giveRaise(.10)                        # Updates sue's pay
>>> sue.pay
66000.0
```

The implied "self" object is why we call this an object-oriented model: there is always an implied subject in functions within a class. In a sense, though, the class-based type simply builds on and uses core types—a user-defined `Worker` object here, for example, is just a collection of a string and number (`name` and `pay`, respectively), plus functions for processing those two built-in objects.

The larger story of classes is that their inheritance mechanism supports software hierarchies that lend themselves to customization by extension. We extend software by writing new classes, not by changing what already works. You should also know that classes are an optional feature of Python, and simpler built-in types such as lists and dictionaries are often better tools than user-coded classes. This is all well beyond the bounds of our introductory object-type tutorial, though, so for that tale, you'll have to read on to a later chapter.

And Everything Else

As mentioned earlier, everything you can process in a Python script is a type of object, so our object type tour is necessarily incomplete. However, even though everything in Python is an "object," only those types of objects we've met so far are considered part of Python's core type set. Other object types in Python are usually implemented by module functions, not language syntax. They also tend to have application-specific roles—text patterns, database interfaces, network connections, and so on.

Moreover, keep in mind that the objects we've met here are objects, but not necessarily *object-oriented*—a concept that usually requires inheritance and the Python `class` statement, which we'll meet again later in this book. Still, Python's core objects are the workhorses of almost every Python script you're likely to meet, and they usually are the basis of larger noncore types.

Chapter Summary

And that's a wrap for our concise data type tour. This chapter has offered a brief introduction to Python's core object types, and the sorts of operations we can apply to them. We've studied generic operations that work on many object types (sequence operations such as indexing and slicing, for example), as well as type-specific operations available as method calls (for instance, string splits and list appends). We've also defined some key terms along the way, such as immutability, sequences, and polymorphism.

Along the way, we've seen that Python's core object types are more flexible and powerful than what is available in lower-level languages such as C. For instance, lists and dictionaries obviate most of the work you do to support collections and searching in lower-level languages. Lists are ordered collections of other objects, and dictionaries are collections of other objects that are indexed by key instead of by position. Both dictionaries and lists may be nested, can grow and shrink on demand, and may contain objects of any type. Moreover, their space is automatically cleaned up as you go.

I've skipped most of the details here in order to provide a quick tour, so you shouldn't expect all of this chapter to have made sense yet. In the next few chapters, we'll start to dig deeper, filling in details of Python's core object types that were omitted here so you can gain a more complete understanding. We'll start off in the next chapter with an in-depth look at Python numbers. First, though, another quiz to review.

Chapter Quiz

We'll explore the concepts introduced in this chapter in more detail in upcoming chapters, so we'll just cover the big ideas here:

1. Name four of Python's core data types.
2. Why are they called "core" data types?
3. What does "immutable" mean, and which three of Python's core types are considered immutable?
4. What does "sequence" mean, and which three types fall into that category?
5. What does "mapping" mean, and which core type is a mapping?
6. What is "polymorphism," and why should you care?

Quiz Answers

1. Numbers, strings, lists, dictionaries, tuples, and files are generally considered to be the core object (data) types. Sets, types, `None`, and Booleans are sometimes classified this way as well. There are multiple number types (integer, long, floating point, and decimal) and two string types (normal and Unicode).
2. They are known as "core" types because they are part of the Python language itself, and are always available; to create other objects, you generally must call functions in imported modules. Most of the core types have specific syntax for generating the objects: `'spam,'` for example, is an expression that makes a string and determines the set of operations that can be applied to it. Because of this, core types are hardwired into Python's syntax. In contrast, you must call the built-in `open` function to create a file object.
3. An "immutable" object is an object that cannot be changed after it is created. Numbers, strings, and tuples in Python fall into this category. While you cannot change an immutable object in-place, you can always make a new one by running an expression.
4. A "sequence" is a positionally ordered collection of objects. Strings, lists, and tuples are all sequences in Python. They share common sequence operations, such as indexing, concatenation, and slicing, but also have type-specific method calls.
5. The term "mapping" denotes an object that maps keys to associated values. Python's dictionary is the only mapping type in the core type set. Mappings do not maintain any left-to-right positional ordering; they support access to data stored by key, plus type-specific method calls.

6. “Polymorphism” means that the meaning of an operation (like a +) depends on the objects being operated on. This turns out to be a key idea (perhaps *the* key idea) behind using Python well—not constraining code to specific types makes that code automatically applicable to many types.