Chapter 4. Introducing Python Objects

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, such as strings and lists, or add-on objects we create with Python classes or external-language tools. As you'll find, these objects are essentially just pieces of memory, with values and associated operations. Moreover, *everything* is an object in a Python script. Even simple numbers qualify, with values (e.g., `99`) and supported operations (`+`, `-`, and so on).

Because objects are also the most fundamental notion in Python programming, this chapter gets us started with a survey that previews Python's built-in object types. Later chapters in this part provide a second pass that fills in details we'll gloss over in this survey. Here, our goal is a brief tour to introduce the basics.

The Python Conceptual Hierarchy

Before we get to the code, 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.

We'll move on to statements in the next part of the book, though you will find that they largely exist to manage the objects you'll meet here. Furthermore, by the time we reach classes in the OOP part of this book, you'll discover that they allow you to define new object types of your own, by both using and emulating the object types you will explore here. Because of all this, built-in objects are a mandatory point of embarkation for all Python journeys.

NOTE

Terminology moment: Traditional introductions to programming often stress its three pillars of *sequence* ("Do this, then that"), *selection* ("Do this if that is true"), and *repetition* ("Do this many times"). Python has tools in all three categories, and these terms might help you organize your thinking early on. But they are also artificial and simplistic, and prone to confuse. For example, tools such as comprehensions are both repetition and selection; these terms have other, more specific meanings in Python; and many later concepts won't seem to fit this mold at all. In Python, the more strongly unifying principle is *objects* and what we can do with them. To see why, read on.

Why Use Built-in Objects?

If you've used lower-level programming languages, 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 may 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 objects 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 simpler tasks, built-in objects 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 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 objects 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 objects employ algorithms that have already been optimized and are often implemented in a lower-level language like C for speed. Although you can write similar object types on your own, you'll usually be hard-pressed to match the level of performance that 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 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 toolkits, on the other hand, tend to differ from site to site.

In other words, not only do built-in object types make programming easier, they're also more powerful and accessible 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 Object Types

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

To some readers, though, the object types in Table 4-1 may be more general and flexible than what you are 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, and both come with automatic memory management, support arbitrarily nesting, can grow and shrink on demand, and may contain objects of any kind.

Table 4-1. Python built-in (core) objects

Object type	**Example literals/creation**
Numbers	`1234`, `3.1415`, `0b111`, `1_234`, `3+4j`, `Decimal`, `Fraction`
Strings	`'code'`, `"app's"`, `b'a\x01c'`, `'h\u00c4ck'`, `'hÄck'`, `'`
Lists	`[1, [2, 'three'], 4.5]`, `list(range(10))`
Dictionaries	`{'job': 'dev', 'years': 40}`, `dict(hours=10)`
Tuples	`(1, 'app', 4, 'U')`, `tuple('hack')`, `namedtuple`
Files	`open('docs.txt')`, `open(r'C:\data.bin', 'wb')`
Sets	`set('abc')`, `{'a', 'b', 'c'}`
Other core objects	Booleans, types, `None`
Program-unit objects	Functions, modules, classes (Parts IV, V, and VI)
Implementation objects	Compiled code, stack tracebacks (Parts IV and VII)

Also shown in Table 4-1, *program units* such as functions, modules, and classes—which you'll meet in later parts of this book—are objects in Python too; they are created with statements and expressions such as `def`, `class`, `import`, and `lambda` and may be passed around scripts freely, stored within other objects, and so on. Python also provides a set of *implementation-*

related objects such as compiled-code objects, which are generally of interest to tool builders more than application developers; we'll explore these later, though in less depth due to their specialized roles.

Despite its title, 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 do network scripting, we use socket objects. These other kinds of objects are generally created by importing and using functions in standard or add-on library modules, and have behavior all their own. Patterns and sockets, for example, are made by calling tools in the standard library's `re` and `socket` modules, respectively.

We usually call the objects in Table 4-1 *core object types*, though, because they are effectively built into the Python language itself—that is, there is specific expression syntax for generating most of them. For instance, when you run the following code with characters surrounded by quotes in a REPL or program file:

```
>>> 'Python'
```

you are, technically speaking, running a *literal expression* that generates and returns a new *string* object. There is Python language syntax to make this object. Similarly, an expression wrapped in square brackets makes a *list*, one in curly braces makes a *dictionary* or *set*, and so on. Even though, as you'll see, Python does not require or use type declarations, 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 list. In formal terms, this means that Python is *dynamically typed*, a model that keeps track of object types for you automatically instead of requiring declaration code, but it is also *strongly typed*, a constraint that means you can perform on an object only operations that are valid for its type.

We'll study each of the object types in Table 4-1 completely in upcoming chapters. Before digging into the full details, though, let's begin by taking a

quicker 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 object-wrangling 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 objects set includes the usual suspects: *integers* that have no fractional part, *floating-point* numbers that do, and more exotic types—*complex* numbers with imaginary parts, *decimals* with flexible precision, *rationals* with numerator and denominator, and full-featured *sets*. Built-in numbers are enough to represent most numeric quantities—from your age to your bank balance—but specialized numeric objects like vectors and matrixes are also available as third-party add-ons.

Although they offer fancier options, Python's basic number objects 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. Here is a demo in a Python REPL of the sort we learned about in Chapter 3:

```
$ python3                          # Start up a REPL
>>> 123 + 222                      # Integer addition
345
>>> 1.5 * 4                        # Floating-point multi
6.0
>>> 1_234_567, 0x15, bin(21)       # Separators, hex, bir
(1234567, 21, '0b10101')
>>> 2 ** 100                       # 2 to the power 100,
1267650600228229401496703205376
```

Notice the last result here: Python's integer object automatically provides extra precision for large numbers like this when needed. You can, for instance, compute 2 to the power 12,345 as an integer in Python, but you probably shouldn't try to grok the result—with nearly 4K digits, it's a lot to take in:

```
>>> len(str(2 ** 12345))          # How many digits in c
3717
```

This nested-call form works *from inside out*—first computing the `**` result, then converting it to a string of digits with the built-in `str` function, and finally getting the length of the resulting string with `len`. The end result is the number of digits in the number. `str` and `len` both work on many object types, and we'll use them again as we move along.

Besides expressions, there are a handful of useful numeric modules that ship with Python in *modules*—which are just packages of additional tools that we import to use, using statements like `import` introduced in Chapter 3:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

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* coded in square brackets—an ordered collection of other objects to be introduced later in this chapter):

```
>>> import random
>>> random.random()
0.7082048489415967
>>> random.choice([1, 2, 3, 4])
2
```

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

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

NOTE

Big numbers versus DOS attacks: Python integers can be arbitrarily large and may even be used to represent floating-point values with extended precision. While integer size is limited only by your computer's memory, though, Python 3.11 and later require extra steps to convert pathologically large numbers to decimal strings (e.g., for `str` or display). This avoids rare denial-of-service attacks with a 4,300-digit limit that can be lifted with a `sys` module tool, `set_int_max_str_digits` ; call this to count digits in larger numbers, and see Python's docs for all the sordid details.

Strings

Strings are used to record both textual information (your name, for instance) as well as arbitrary collections of bytes (such as an image file's contents). They are our first example of what in Python we call a *sequence*—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 positions. Strictly speaking, strings are sequences of one-character strings; other, more general sequence objects 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 coded inside *quotes* (of the single or double straight kind—they mean the same thing, but single is more common and less busy), we can verify its length with the built-in `len` function and fetch its components with *indexing* expressions:

```
>>> S = 'Code'            # Make a 4-character string, c
>>> len(S)                # Length: number characters
4
>>> S[0]                  # The first item in S, indexir
'C'
>>> S[1]                  # The second item from the lef
'o'
```

In Python, indexes are coded in square brackets as offsets from the front, so start from 0: the first item is at index 0, the second is at index 1, and so on.

Notice how we assign the string to a *variable* named `S` here. We'll go into detail on how this works later (especially in Chapter 6), but Python variables never need to be declared ahead of time. A variable is created when you assign it a value, may be assigned any type of object, and is replaced with its value when it shows up in an expression. It must also have been previously assigned by the time you use its value. For the purposes of this chapter, it's enough to know that we need to assign an object to a variable in order to save it for later use.

In Python, we can also index backward, from the *end*—positive indexes count forward from the left, and negative indexes count backward from the right. Continuing our REPL session:

```
>>> S[-1]               # The last item from the end i
'e'
>>> S[-2]               # The second-to-last item from
'd'
```

Formally, a negative index is simply added to the string's length to yield a positive offset, 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
'e'
>>> S[len(S)-1]         # Negative indexing, the hard
'e'
```

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 we wish. 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 (a.k.a. slice) in a single step. For example:

```
>>> S                        # A 4-character string
'Code'
```

```
>>> S[1:3]                # Slice of S from offsets 1 t
'od'
```

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 second of the preceding operations, for instance, gives us all the characters in string `S` from offsets 1 through 2 (that is, 1 through 3 – 1) as a new string. The effect is to slice or "parse out" the two characters in the middle at offsets 1 and 2.

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 (
'ode'
>>> S                     # S itself hasn't changed
'Code'
>>> S[0:3]                # Everything but the last
'Cod'
>>> S[:3]                 # Same as S[0:3]
'Cod'
>>> S[:-1]                # Everything but the last agc
'Cod'
>>> S[:]                  # All of S as a top-level cop
'Code'
```

Note in the second-to-last command 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 other 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
'Code'
>>> S + 'xyz'             # Concatenation
'Codexyz'
```

```
>>> S                       # S is unchanged
'Code'
>>> S * 8                   # Repetition
'CodeCodeCodeCodeCodeCodeCodeCode'
```

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 regularly call *polymorphism* in this book—in short, this means that 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 object 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 this tour.

Immutability

Also notice in the prior examples that we were not changing the original string with any of the operations we ran on it. As it turns out, 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.

More generally, you can never overwrite the values of immutable objects. 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 automatically cleans up old objects as you go (as you'll learn later), this isn't as inefficient as it may sound:

```
>>> S
'Code'

>>> S[0] = 'z'              # Immutable objects cannot b
…error text omitted…
TypeError: 'str' object does not support item assignmer

>>> S = 'Z' + S[1:]         # But we can run expressions
>>> S
'Zode'
```

Every object in Python is classified as either immutable (unchangeable) or not. In terms of the core objects, *numbers*, *strings*, and *tuples* are immutable; but *lists*, *dictionaries*, and *sets* are not—they can be changed in place freely, as can most new objects you'll code with classes. This distinction turns out to be crucial in Python work, in ways that we can't yet fully explore. Among other things, immutability can be used to guarantee that an object remains constant throughout your program; mutable objects' values, by contrast, can be changed at any time and place (and whether your code expects it or not!).

Strictly speaking, you can change text-based data *in place* if you either expand it into a *list* of individual characters and join it back together with nothing between, or use the special-purpose `bytearray` object:

```
>>> S = 'Python'
>>> L = list(S)                              # Expand to
>>> L
['P', 'y', 't', 'h', 'o', 'n']
>>> L[0] = 'C'                               # Change it
>>> ''.join(L)                               # Join with
'Cython'

>>> B = bytearray(b'app')                    # A bytes/l
>>> B.extend(b'lication')                    # 'b' means
>>> B                                        # B[i] = or
bytearray(b'application')
>>> B.decode()                               # Translate
'application'
```

The `bytearray` supports in-place changes for text, but only for text whose characters are all at most 8-bits wide (e.g., ASCII). All other strings are still immutable— `bytearray` is a distinct hybrid of immutable *bytes* strings (whose `b'…'` syntax distinguishes them from normal text strings) and mutable *lists* (coded and displayed in `[]`), but we have to wait until we learn more about both these and Unicode text ahead to fully grasp this code.

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 that are attached to and act upon a specific object, which are triggered with a call expression using parentheses.

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; both act on the subject that they are attached to and called from:

```
>>> S = 'Code'
>>> S.find('od')                 # Find the offset of c
1
>>> S
'Code'
>>> S.replace('od', 'abl')       # Replace all substrir
'Cable'
>>> S
'Code'
```

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, this is the only way this can work. 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
['aaa', 'bbb', 'ccccc', 'dd']

>>> S = 'code'
>>> S.upper()                    # Upper- and lowercase
'CODE'
>>> S.isalpha()                  # Content tests: isalp
True

>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line.rstrip()                # Remove whitespace ch
'aaa,bbb,ccccc,dd'
```

```
>>> line.rstrip().split(',')     # Combine two operatic
['aaa', 'bbb', 'ccccc', 'dd']
```

Notice the last command here—it strips before it splits because Python runs it *from left to right*, making a temporary result along the way. You can chain method calls this way, as long as the prior call returns an object with methods.

Strings methods are also one way to run an advanced substitution operation known as *formatting*, available as an expression (the original), a string method call (newer), and a literal form called f-strings (newest). The first two replace keys with separate values, the third replaces embedded Python expressions with their results, and all three resolve substitution values and build new strings when they are run:

```
>>> tool  = 'Python'
>>> major = 3
>>> minor = 3

>>> 'Using %s version %s.%s' % (tool, major, minor + 9)
'Using Python version 3.12'

>>> 'Using {} version {}.{}'.format(tool, major, minor
'Using Python version 3.12'

>>> f'Using {tool} version {major}.{minor + 9}'
'Using Python version 3.12'
```

And yes, this comes in *three* flavors today. While you may want to pick one for your own code, all three are fair game in code you may read (and inclusive books). Each form is rich with features, which we'll postpone discussing until later in this book, and which tend to matter most when you must generate readable output and numeric reports:

```
>>> '%.2f | %+05d' % (3.14159, -62)                 #
'3.14 | -0062'

>>> '{1:,.2f} | {0}'.format('sapp'[1:], 296999.256)   #
'296,999.26 | app'
```

```
>>> f'{296999.256:,.2f} | {'sapp'[1:]}'                #
'296,999.26 | app'
```

Although sequence operations are generic, methods are not—while some objects share some method names, string method operations generally work only on strings, and nothing else. As a rule of thumb, Python's *toolset* is layered: generic operations that span multiple object 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, 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 coverage of object methods, but for more details, you can always call the built-in `dir` function. This function lists variables assigned in the caller's scope when called with no argument; more usefully, it returns a list of all the attributes available for any object passed to it. Because methods are callable attributes, they will show up in this list. Assuming `S` is still the string `'Code'`, here are its attributes:

```
>>> dir(S)
['__add__', '__class__', '__contains__', '__delattr__',
'__eq__', '__format__', '__ge__', '__getattribute__', '
…etc…
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 's
'startswith', 'strip', 'swapcase', 'title', 'translate'
```

Run this live for the full list; its middle was cut at …*etc*… for space here (strings have 81 attributes today!). The names without underscores in the second half of this list are all the callable methods on string objects. The names with *double underscores* won't be important until later in the book, when we study operator overloading in classes. In short, they represent the implementation of the string object and support customization. The `__add__` method of strings, for example, is how concatenation ultimately works; Python maps the first of the following to the second internally, though you

shouldn't usually use the second form yourself (it's less intuitive, and might even run slower):

```
>>> S + 'head!'
'Codehead!'
>>> S .__add__('head!')
'Codehead!'
```

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

```
>>> help(S.replace)                              # Or help
Help on built-in function replace:
replace(old, new, count=-1, /) method of builtins.str i
    Return a copy with all occurrences of substring olc
…etc…
```

Press the `Q` key to exit a `help` display in most console windows. `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 for display in a web browser.

You can also ask for help on an *entire string* with `help(S)`, but it may not help: the string's value is used instead of the string itself, unless it's empty. To do better, you need to know either the name of the string type, or that the `type` built-in returns any object's type as another object: `help(str)` and `help(type(S))` both give help for strings but may be more than you want—because they describe every method, `help` may be best used on specific methods.

For more info, you can also consult Python's standard-library reference manual, but `dir` and `help` are the first level of documentation in Python, especially when working in an interactive REPL. Try them soon on an object near you.

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 in greater depth later. For instance, special characters can be represented as *backslash escape* sequences, which Python displays in `\xNN` hexadecimal escape notation, unless they stand for printable characters:

```
>>> S = 'A\nB\tC'            # Escapes: \n is newline,
>>> len(S)                   # Each stands for just one
5

>>> S = 'A\0B\0C'            # \0, a binary zero byte,
>>> len(S)
5
>>> S                        # Nonprintables are displa
'A\x00B\x00C'
```

As hinted earlier, Python allows strings to be enclosed in *single* or *double* quote characters—they mean the same thing but allow the other type of quote to be embedded without an escape (most programmers prefer single quotes for less clutter, unless they're pining for other-language pasts). You can also code multiline string literals enclosed in triple quotes (single or double)—when used, all the lines are concatenated together, and newline characters (`\n`) are added where line breaks appear. This is useful for embedding things like multiline HTML, YAML, or JSON code in a Python script, as well as stubbing out lines of code temporarily—just add three quotes above and below:

```
>>> msg = """
... aaaaaaaaaaa
... bbb'''bbb""bbb
... cccccccc
... """
>>> msg
'\naaaaaaaaaaa\nbbb\'\'\'bbb""bbb\ncccccccc\n'
```

Python also supports a *raw* string literal that turns off the backslash escape mechanism. Such literals start with the letter `r` and are useful for strings like

regular-expression patterns, and directory paths on Windows sans doubled-up backslashes (e.g., `r'C:\Users\you\code'`).

Unicode Strings

Python's strings also come with full Unicode support required for processing *non-ASCII text*. Such text includes characters in non-English languages, as well as symbols and emojis, and is common today in web pages, emails, GUIs, documents, and data. Python's string objects let you process such text seamlessly: its normal `str` string handles Unicode text (including ASCII, which is just a simple kind of Unicode); and its `bytes` string, together with its `bytearray` mutable cousin used earlier, handles raw byte values (including media and encoded text):

```
>>> 'hÄck'                    # Normal str strings are l
'hÄck'
>>> b'a\x01c'                 # bytes strings are byte-l
b'a\x01c'
```

Formally, Python's byte strings are sequences of *8-bit bytes* that print with ASCII characters when possible, and its text strings are sequences of *Unicode code points*—identifying numbers for characters, which print as the usual glyphs we've come to know and do not necessarily map to single bytes when stored in memory or encoded in files. In fact, the notion of bytes doesn't quite apply to Unicode: it includes a host of character code points too large to fit in a byte, and defines encodings for storage and transmission in which characters may be any size at all:

```
>>> 'Code'                              # Characters mc
'Code'
>>> 'Code'.encode('utf-8')              # Encoded to 4
b'Code'
>>> 'Code'.encode('utf-16')             # But encoded t
b'\xff\xfeC\x00o\x00d\x00e\x00'
```

This is especially true for richer text (`ord` gives a character's code point, and `hex` gives its hexadecimal string):

```
>>> hex(ord('
🐍
'))                      # Code points too big for a by
'0x1f40d'
>>> len('
🐍
hÄck
👏
')                     # One character (code point) eac
6
>>> len('
🐍
hÄck
👏
'.encode('utf-8'))     # But encoded bytes sizes vary
13
>>> len('
🐍
hÄck
👏
'.encode('utf-16'))
18
```

To code non-ASCII characters in text strings, use `\x` hexadecimal escapes; short `\u` or long `\U` Unicode escapes; or raw text interpreted per source encodings optionally declared in program files when code is read (the default for code is the universal UTF-8). To illustrate, here's our non-ASCII A-with-diaeresis character `Ä` coded four ways in Python:

```
>>> 'h\xc4\u00c4\U000000c4Äck'          # Coding non-ASC
'hÄÄÄÄck'
```

In text strings, all these forms specify Unicode *code points* that stand for characters. By contrast, byte strings use only `\x` hexadecimal escapes to embed the values of *raw bytes*; this isn't always text, but when it is, it's the encoded form of text, not its decoded code points, and encoded bytes are the same as code points only for simple text and encodings:

```
>>> '\u00A3', '\u00A3'.encode('latin1'), b'\xA3'.decode
('£', b'\xa3', '£')
```

Apart from these string types, Unicode processing often reduces to transferring text data to and from *files*—which automatically *encode* text to bytes when stored in a file and *decode* it to characters (a.k.a. code points) when read back into memory. Once loaded, we usually process text as strings in decoded form only. To make this work, *text files* implement encodings and accept and return text strings, but *binary files* instead deal in `bytes` strings for raw data.

You'll meet Unicode again in the files coverage later in this chapter, but we will save the rest of the Unicode story for later in this book. It crops up briefly in Chapters 7, 9, and 15, but for the most part is postponed until this book's advanced topics part, in Chapter 37. Unicode is crucial in many (or most) domains today, but many Python newcomers can get by with just a passing acquaintance until they've mastered string basics.

In addition to its built-in string objects, Python's standard toolset includes support for text pattern matching with its `re` module, as well as parsing textual data like JSON, CSV, XML, and HTML. You'll meet additional examples of some of these tools later in this book, but this tutorial intro has already said enough about strings and must move on.

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. Accordingly, they provide a very flexible tool for representing arbitrary collections—files in a folder, employees in a company, emails in your inbox, and so on.

Sequence Operations

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

```
>>> L = [123, 'text', 1.23]            # A list of thre
>>> len(L)                             # Number of iten
```

```
3
```

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

```
>>> L[0]                                # Indexing by pc
123
>>> L[:-1]                              # Slicing a list
[123, 'text']

>>> L + [4, 5, 6]                       # Concat/repeat
[123, 'text', 1.23, 4, 5, 6]
>>> L * 2
[123, 'text', 1.23, 123, 'text', 1.23]

>>> L                                   # We're not char
[123, 'text', 1.23]
```

Type-Specific Operations

Python's lists may be reminiscent of *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('Py')                      # Growing: add c
>>> L
[123, 'text', 1.23, 'Py']

>>> L.pop(2)                            # Shrinking: del
1.23
>>> L                                   # "del L[2]" del
[123, 'text', 'Py']
```

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 an item at an arbitrary position (`insert`), remove a given item by value (`remove`), add multiple items at the end (`extend`), and more. Because lists

are mutable, most list methods also change the list object *in place*, instead of making 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 (though similarly named functions we'll explore later do not).

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 (error messages are condensed here and elsewhere):

```
>>> L
[123, 'text', 'Py']

>>> L[99]
IndexError: list index out of range

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

This is intentional, 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 (or make a new list). Unlike indexing, slicing *scales* offsets to be in bounds, but this will have to await coverage in the in-depth chapters ahead.

Nesting

One nice feature of Python's core object 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—as deeply and mixed as needed to describe things in our real world.

An immediate application of this feature is to represent matrixes, or "multidimensional arrays" in Python. A list with nested lists like the following will do the job for basic applications (coding fine print: indentation doesn't matter in this, Python expressions with unclosed brackets can span multiple lines this way, and some REPLs show "..." continuation-line prompts, which are often omitted in this book for easier copy and paste from emedia):

```
>>> M = [[1, 2, 3],                # A 3 × 3 matrix, as
         [4, 5, 6],                # Code can span lines
         [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
6
```

The first operation here fetches the entire second row, and the second grabs the third item within that row (it runs left to right, like the earlier string strip and split combo). Stringing together index operations takes us deeper and deeper into our nested-object structure. Tip: this matrix structure works for small-scale tasks, but for more serious number crunching you will probably want to use the open source *NumPy* extension for Python, which can store and process large matrixes much more efficiently than our nested list structure (and is well out of scope here; see Chapter 1).

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 the prior section's 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
>>> col2
[2, 5, 8]

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

List comprehensions 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. They 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 t
[3, 6, 9]

>>> [row[1] for row in M if row[1] % 2 == 0]  # Filter
[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 they can be used to iterate over any *iterable* object—a term we'll flesh out later in this book, but which simply means either a physical sequence or a virtual one that produces its items on request. Here, for

instance, we use list comprehensions to step over a hardcoded list of coordinates and a string:

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

>>> doubles = [c * 2 for c in 'hack']        # Repeat c
>>> doubles
['hh', 'aa', 'cc', 'kk']
```

These expressions can also be used to collect multiple values, as long as we wrap those values in a nested collection. The following illustrates using `range` —a built-in that generates successive integers and requires a surrounding `list` to force it to yield all its values for display in the REPL (there's more on this mystery ahead):

```
>>> list(range(4))                           # Integers
[0, 1, 2, 3]
>>> list(range(-6, 7, 2))                    # -6 to +6
[-6, -4, -2, 0, 2, 4, 6]

>>> [[x ** 2, x ** 3] for x in range(4)]     # Multiple
[[0, 0], [1, 1], [4, 8], [9, 27]]

>>> [[x, x // 2, x * 2] for x in range(-6, 7, 2) if x >
[[2, 1, 4], [4, 2, 8], [6, 3, 12]]
```

As you can probably tell, list comprehensions are too involved to cover more formally in this preview. The main point of this brief introduction is to illustrate that Python includes both simple and advanced tools in its arsenal. List comprehensions are optional, but they can be very useful in practice and often provide a processing speed advantage.

In fact, comprehension syntax is not just for making lists: enclosing it in *parentheses* can also be used to create an iterable object known as a *generator*, which produces results on demand per Python's *iteration protocol* —in the following, summing items in a matrix row with the built-in `sum`, on each call to `next`:

```
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> G = (sum(row) for row in M)             # Make a g
>>> next(G)                                 # Run the
6
>>> next(G)                                 # A new ro
15
>>> next(G)                                 # Row 3: 7
24
```

And comprehension syntax can also be used to create *sets* and *dictionaries* when enclosed in curly braces:

```
>>> {sum(row) for row in M}                 # Makes ar
{24, 6, 15}

>>> {i: sum(M[i]) for i in range(3)}        # Makes ke
{0: 6, 1: 15, 2: 24}
```

But to grasp concepts like the iteration protocol and objects like sets and dictionaries, we must move ahead.

Dictionaries

Unlike strings and lists, Python dictionaries are not sequences at all but are instead the only core member of a category known as *mappings*. Mappings are also collections of other objects, but they store objects by *key* instead of by relative position. While dictionaries retain the insertion order of their keys today, this may not apply to your goals, and key-to-value mapping remains their main role. Dictionaries are also *mutable*: like lists, they may be changed in place and can grow and shrink on demand. Also like lists, they are a flexible tool for coding collections, but their more *mnemonic* keys are better suited when a collection's items are named or labeled—as in fields of a database record.

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. As an example, consider the following three-item dictionary with keys "name," "job," and "age," that record the attributes of a fictitious (and wholly generic and neutral) worker:

```
>>> D = {'name': 'Pat', 'job': 'dev', 'age': 40}
```

We can index this dictionary by key to fetch and change its 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['name']                       # Fetch value of ke
'Pat'

>>> D['job'] = 'mgr'                # Change Pat's job
>>> D
{'name': 'Pat', 'job': 'mgr', 'age': 40}
```

Although the curly-braces literal form does see use, it is perhaps more common to see dictionaries built up in different ways (after all, it's rare to know all your program's data before your program runs). The following code, 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, assignments to new dictionary keys create those keys:

```
>>> D = {}
>>> D['name'] = 'Pat'               # Create keys by as
>>> D['job']  = 'dev'
>>> D['age']  = 40
>>> D
{'name': 'Pat', 'job': 'dev', 'age': 40}
```

Here, we're effectively using dictionary keys as field names in a record that describes an imaginary person. In other roles, 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.

As you'll learn later, we can also make dictionaries by passing to the `dict` type name either *keyword arguments* (a special `name=value` syntax in function calls), or the result of *zipping* together sequences of keys and values obtained at runtime (e.g., from files). Both of the following make the same dictionary as the prior example and its equivalent `{}` literal form; the first requires string keys but tends to make for less typing (and subjective noise), and the second uses the `zip` built-in we'll study later in this part of the book:

```
>>> pat1 = dict(name='Pat', job='dev', age=40)
>>> pat1
{'name': 'Pat', 'job': 'dev', 'age': 40}

>>> pat2 = dict(zip(['name', 'job', 'age'], ['Pat', 'de
>>> pat2
{'name': 'Pat', 'job': 'dev', 'age': 40}
```

Notice how the left-to-right order of dictionary keys is always the same. Though not sequences, dictionary keys retain their *insertion order*, even in the presence of changes: the order of keys is the order in which keys were added. This wasn't the norm until Python 3.7, and prior to this, key order was scrambled and sometimes required separate ordering. The new ordering may be more intuitive and will be assumed in this book but adds a sequence flavor to dictionaries not shared by other nonsequences. Like most mods, it also rewrites history and invalidates Python learning resources that cannot be updated as frequently as Python. Change is almost always a double-edged sword.

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 (again, indentation here and "…" in some REPLs are moot):

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

Here, we again have a three-key dictionary at the top (keys “name,” “jobs,” 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 jobs to support multiple roles and future expansion. We can access the components of this structure much as we did for our list-based matrix earlier, but this time most indexes are dictionary keys, not list offsets:

```
>>> rec['name']                          # 'name' is a r
{'first': 'Pat', 'last': 'Smith'}

>>> rec['name']['last']                  # Index the nes
'Smith'

>>> rec['jobs']                          # 'jobs' is a r
['dev', 'mgr']
>>> rec['jobs'][-1]                      # Index the nes
'mgr'

>>> rec['jobs'].append('janitor')        # Expand Pat's
>>> rec
{'name': {'first': 'Pat', 'last': 'Smith'}, 'jobs': ['d
```

Notice how the last operation here *expands* the nested jobs list—because the jobs list is a separate piece of memory from the dictionary that contains it, it can grow and shrink freely (the `pop` method we met earlier is one way to shrink objects). This may seem quite a trick to readers with backgrounds in more rigid languages, but is natural in Python.

More fundamentally, this example demos the *flexibility* of Python’s core object 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 lower-level languages we may have to allocate *memory* space for objects ahead of time and be careful to release it when we no longer need it. In Python, this is all automatic: object memory is allotted as needed and freed when we lose the last reference to the object—by assigning its variable to something else, for example:

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

Technically speaking, Python uses a scheme called *garbage collection* that reclaims unused memory as your program runs and frees you from having to manage such details in your code. In standard Python (a.k.a. CPython) this uses object *reference counts* primarily, along with a supplemental garbage collector for cycles. We'll study how this works later in Chapter 6; for now, it's enough to know that you can use objects freely, while Python handles their memory.

Watch for a record structure similar to the one we just coded in Chapters 8, 9, and 27, where we'll use it to compare and contrast lists, dictionaries, tuples, named tuples, and classes—an array of data structure options with trade-offs we'll cover in full later. It's also worth noting that the record structure we used here can be saved in a file with a variety of techniques in Python, including its `pickle` module and support for language-neutral *JSON* (a data format that's strikingly similar to Python dictionary objects); more on such tools later in this book.

Missing Keys: if Tests

As mappings, dictionaries support accessing items by key only, with the sorts of operations we've just seen. In addition, though, they also support type-specific operations with *methods* that are useful in a variety of common roles. For example, the dictionary `copy` and `update` methods copy a dictionary and merge one into another in place, respectively (the `|` operator does the same, but makes a new dictionary for its result: it's just a `copy` plus an `update`).

Dictionary methods also play parts in common key use cases. For instance, while we can assign to a new key to expand a dictionary, fetching a nonexistent key is still a mistake:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D['d'] = 4                          # Assigning new k
>>> D
{'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> D['e']                              # Referencing a r
…error text omitted…
KeyError: 'e'
```

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

Which brings us to our first Python *compound statement*—one with nested parts. If you're working along, here are a few practical bits: in the following, press the Enter key twice to run the `if` interactively after typing its code (an empty line means "go" in most REPLs, as explained in Chapter 3); the prompt changes to a "..." after the first line in some interfaces (as for the earlier multiline dictionaries and lists); and indentation matters this time (for reasons up next):

```
>>> 'e' in D                            # Boolean result:
False

>>> if not 'e' in D:                    # Python's main s
        print('missing key!')

missing key!
```

This book has more to say about the `if` statement in later chapters, but its syntax is straightforward. It consists of the word `if`, followed by an expression whose result is interpreted as true or false, followed by a block of

code to run if the expression result is true. In its full regalia, the `if` statement can also have an `else` clause for the false case, and one or more `elif` ("else if") clauses for other tests.

Functionally, the `if` is the main *selection* tool in Python, and how we code most of the logic of choices and decisions in our scripts. It's joined by its ternary `if`/`else` expression cousin you'll meet in a moment, the `if` comprehension filter lookalike we used earlier, and the newer `match` multiple-selection statement you'll meet later in this book.

If you've used some other programming languages in the past, you might now be wondering how Python knows when the `if` statement ends. We'll explore Python's syntax rules in depth in later chapters, but in short, if you have more than one action to run in a statement block, you simply indent all their statements the same way—which both promotes readable code and reduces the number of characters you have to type. All multiline statements follow this pattern: a header line ending in ":" and a block of (usually) indented code, with no "{}" around blocks, and no ";" required after statements (though fair warning: forgetting the ":" is the most common beginner's mistake in Python!):

```
>>> if not 'e' in D:
        print('missing')                # Statement block
        print('no, really...')          # (Unless they're

missing
no, really...
```

Besides the `in` test, there are a variety of other ways to avoid accessing nonexistent keys in the dictionaries we create: the dictionary `get` method, which is a conditional index with a default; the `if`/`else` ternary (three-part) expression, which is essentially a limited `if` statement squeezed onto a single line; and the `try` statement, which is a tool we'll first use in Chapter 10 that catches and recovers from errors altogether. Here are the first two in action:

```
>>> D.get('a', 'missing')               # Like D['a'] but
1
>>> D.get('e', 'missing')               # Default returne
'missing'
```

```
>>> D['e'] if 'e' in D else 0          # if/else ternary
0
```

We'll save the details on such alternatives until a later chapter. For now, let's turn to other dictionary methods' roles in another common use case.

Item Iteration: for Loops

Dictionaries collect a lot of useful info, but what do we do if we need to process their items one at a time? As it turns out, dictionaries come with methods designed for the job:

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

>>> list(D.keys())                     # Keys, values,
['a', 'b', 'c']
>>> list(D.values())                   # list forces re
[1, 2, 3]
>>> list(D.items())
[('a', 1), ('b', 2), ('c', 3)]
```

As shown, a dictionary's `keys`, `values`, and `items` methods return its keys, values, and key/value pairs (the latter is tuples, up next on this tour). Really, though, these methods all return an object that produces results one at a time, which is why they've been wrapped in `list` calls, as we did for `range` earlier. This reflects the *iteration protocol* in Python—a concept we'll explore in full later, but which boils down to an iterable object, which has an iterator object, which responds to `next` calls to produce one result at a time:

```
>>> D.keys()                           # Get an iterabl
dict_keys(['a', 'b', 'c'])

>>> I = iter(D.keys())                 # Get an iterato
>>> next(I)                            # Get one result
'a'
>>> next(I)                            # This is most c
'b'
```

We used the same `next` built-in to force results from a generator comprehension earlier, but without the `iter` step: because generators don't support multiple scans, they are their own iterators, and `iter` is a no-op.

Tools that support this protocol can both save memory and minimize delays, because they don't produce all their results at once. The iteration protocol works on all sorts of objects in Python, but you can usually forget its details if you use the Python `for` loop, which runs the iteration protocol automatically to step through items one at a time—both for physical collections like strings and lists, and virtual sequences like generators, `range`, and `keys`:

```
>>> for key in D.keys():                # Auto run the i
        print(key, '=>', D[key])        # Display multip

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

To code a `for`, provide a variable (e.g., `key`) and an iterable object (e.g., `D.keys()`); for each item in the object, the `for` assigns the item to the variable and runs the nested (and usually indented) block of code—which uses the variable to refer to the current item each time through. The `for` is one of the main *repetition* tools in Python, together with the comprehensions you met earlier, and the more general-purpose `while` loop you'll meet later in this book.

Because dictionary iteration is so common, the `for`, and similar iteration tools, can also step through keys implicitly, as well as key/value pairs. The following loops, for example, produce the same output as the preceding example; the choice between all these forms is partly a matter of personal preference, though explicit is generally better:

```
>>> for key in D:                       # Implicit keys(
        print(key, '=>', D[key])

>>> for (key, value) in D.items():      # Key/value-pair
        print(key, '=>', value)
```

The last of these uses something known as *tuple assignment*, which automatically unpacks items into variables. But to fully understand the sorts of stuff we get back from the dictionary `items` method, we have to move ahead.

Tuples

The tuple object (pronounced “toople” or “tuhple,” depending on whom you ask) is roughly like a list that cannot be changed—tuples are *sequences*, like lists, but they are *immutable*, like strings. Functionally, they’re used to represent fixed collections of items: the components of a specific calendar date, for instance. Syntactically, they are normally coded in parentheses instead of square brackets and support arbitrary object types, arbitrary nesting, and the usual sequence operations that we used on strings and lists earlier:

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

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

>>> T[0], T[1:]                 # Indexing, slicing, ar
(1, (2, 3, 4))
```

As usual, tuples also have type-specific callable methods, but not nearly as many as lists:

```
>>> T.index(4)                  # Tuple methods: 4 appe
3
>>> T.count(4)                  # 4 appears once
1
```

The primary distinction for tuples is that they cannot be changed once created. That is, they are immutable sequences (quirk: one-item tuples like the one here require a trailing comma to distinguish them from simple expressions):

```
>>> T[0] = 2                    # Tuples are immutable
TypeError: 'tuple' object does not support item assignm

>>> T = (2,) + T[1:]            # Make a new tuple for
>>> T
(2, 2, 3, 4)
```

Like lists and dictionaries, tuples support mixed types and nesting, but they don’t grow and shrink like lists and dictionaries because they are immutable:

```
>>> T = 'hack', 3.0, [11, 22, 33]
>>> T
('hack', 3.0, [11, 22, 33])

>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'appenc
```

Notice the first line in this: the *parentheses* enclosing a tuple’s items can often be omitted, as done here. In contexts where commas don’t otherwise matter, the *commas* are what actually builds a tuple. This also explains why REPLs show results in parentheses when you enter multiple items separated by commas: the input is really a tuple.

Why Tuples?

So, why have a kind of object that is like a list, but supports fewer operations? Frankly, tuples are not 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 here. We’ll talk more about tuples later in the book, including an extension that builds upon them called *named tuples*. For now, though, let’s move on to this tour’s last major object.

Files

File objects are the main way your Python code will access the content of files on your computer. They can be used to read and write text memos, audio clips, Excel documents, saved emails, and whatever else you have stored on your device. Files are a core object type, but they're something of an oddball—there is no specific literal syntax for creating them. Rather, you create a file object by calling the built-in `open` function with an external filename, and perhaps more depending on your goals.

For example, to create a text output file, pass in its name and the `'w'` processing mode string to *write* text data; Python automatically makes the newline character `\n` portable across platforms when it's transferred to and from files:

```
>>> f = open('data.txt', 'w')      # Open a new file ir
>>> f.write('Hello\n')             # Write strings of c
6
>>> f.write('world!\n')            # Return number of i
7
>>> f.close()                      # Close to flush out
```

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 device). To read back what you just wrote, reopen the file in `'r'` processing mode, for *reading* text input—this is also the default if you omit the mode in the call. Then read the file's content into a string and access it. A file's content is always a string in your script, regardless of the type of data the file contains:

```
>>> f = open('data.txt')           # Open an existing f
>>> text = f.read()                # Read entire file i
>>> text
'Hello\nworld!\n'

>>> print(text)                    # print interprets c
Hello
world!
```

```
>>> text.split()                    # File content is al
['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 maximum byte/character size, `readline` reads one line at a time, and so on), as well as other tools (`seek` moves to a new file position). As you'll see later, though, the best way to read a text file is usually to not read it at all—files support the *iteration protocol* with an iterator that automatically reads line by line in `for` loops and other contexts:

```
>>> for line in open('data.txt'):       # Display lines
        print(line.rstrip())            # Single spaced
```

You'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 any open file and a `help` on any of the method names that come back, as we learned earlier.

Unicode and Byte Files

The prior section's examples illustrate file basics that suffice for many roles. Technically, though, they rely on the platform's Unicode *encoding* default for the host platform. Python text files always use a Unicode encoding to encode strings on writes and decode them on reads. This is often irrelevant for simple ASCII text data, which usually maps to and from file bytes unchanged. But for richer kinds of data, file interfaces can vary by content type.

As hinted when we studied Unicode strings earlier, Python draws a sharp distinction between text and binary data in files: *text files* represent content as normal `str` strings and perform Unicode encoding and decoding automatically as noted, while *binary files* represent content as the special `bytes` string and allow you to access file content unaltered.

For example, *binary files* are useful for processing media, accessing data created by C programs, and so on. What you send and receive is the literal content of the file, whether it's encoded text or a JPEG image. Add a `b` to the mode to invoke binary (and expect a `\r\n` instead of `\n` at the end on Windows, because its newlines vary from Unix):

```
>>> bf = open('data.bin', 'wb')
>>> bf.write(b'h\xFFa\xEEc\xDDk\n')     # Write binary
8
>>> bf.close()
>>> open('data.bin', 'rb').read()       # Read binary c
b'h\xffa\xeec\xddk\n'
```

For *text files*, we can't really talk about content without also asking, "What kind?"—files may use any Unicode encoding, especially if they came from another platform, or the internet at large. This applies to portable programs too: if you want your code to work across platforms, you should generally make encodings explicit to avoid unpleasant surprises. Luckily, this is easier than it may sound—simply pass in an `encoding` name to `open` to force an encoding:

```
>>> tf = open('unidata.txt', 'w', encoding='utf-8')
>>> tf.write('
🐍
h\u00c4ck
👋
')                                  # Encodes to UTF-8
6
>>> tf.close()
```

If you read with the same encoding (or one that's compatible), you get back the same text-character code points that you wrote. The encoded bytes on the file are in UTF-8 form, but your code usually doesn't need to care:

```
>>> open('unidata.txt', 'r', encoding='utf-8').read()
'
🐍
hÄck
👋
'

>>> open('unidata.txt', 'rb').read()
b'\xf0\x9f\x90\x8dh\xc3\x84ck\xf0\x9f\x91\x8f'
```

While files automate most encodings, you can also encode and decode manually if your program gets Unicode data from another source—parsed from an email message or fetched over a network connection, for example:

```
>>> 'hÄck'.encode('utf-8')
b'h\xc3\x84ck'
>>> b'h\xc3\x84ck'.decode('utf-8')
'hÄck'
```

Python also supports non-ASCII file *names* (not just content), but it's largely automatic. For the whole story on Unicode in Python, stay tuned for Chapter 37.

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, persistent object shelves, descriptor-based files, relational and object-oriented database interfaces, and more. We won't cover many of these topics in this language book, but you'll find them useful once you start programming Python in earnest.

Other Object Types

Beyond the core object types we've seen so far, there are others that get less publicity than their cohorts but are useful in their intended roles nonetheless. Let's quickly run down some of the stragglers in this category.

Sets

Python sets are neither mappings nor sequences; rather, they are unordered collections of immutable (technically, "hashable") objects, which store each object just once. You create sets by calling the built-in `set` function with a sequence or other iterable, or by using a set literal expression, and sets support the usual mathematical set operations:

```
>>> X = set('hack')                     # Sequence => se
>>> Y = {'a', 'p', 'p'}                 # Set literal
>>> X, Y
({'c', 'k', 'a', 'h'}, {'p', 'a'})

>>> X & Y, X | Y                        # Intersection,
({'a'}, {'p', 'c', 'k', 'h', 'a'})
```

```
>>> X - Y, X > Y                          # Difference, su
({'c', 'k', 'h'}, False)
```

Even less mathematically inclined programmers often find sets useful for common tasks such as filtering out duplicates, isolating differences, and performing order-neutral equality tests without sorting:

```
>>> list(set([3, 1, 2, 1, 3, 1]))        # Duplicates rem
[1, 2, 3]
>>> set('code') - set('hack')            # Collection dif
{'d', 'o', 'e'}
>>> set('code') == set('deoc')           # Order-neutral
True
```

As you'll see later in this part of the book, normal sets themselves are mutable and can be changed (with their `remove` and `add` methods, for example), though the immutable items within them by definition cannot.

Booleans and None

Python also comes with Booleans, with predefined `True` and `False` objects that are essentially just the integers 1 and 0 with custom display logic; as well as a special placeholder object called `None`, commonly used to initialize names and objects and designate an absence of a result in functions:

```
>>> 1 > 2, 1 < 2                         # Booleans
(False, True)
>>> bool('hack')                         # All objects have
True                                     # Nonempty means Tr

>>> X = None                             # None placeholder
>>> print(X)                             # But None is a thi
None
>>> L = [None] * 100                     # Initialize a list
>>> L
[None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, …etc: c
```

Types

One last core object merits a callout here. The *type* object, returned by the `type` built-in function, is an object that gives the type of another object. We used it earlier when exploring the `help` function, but here's its actual result:

```
>>> L = [1, 2, 3]
>>> type(L)                          # The type of a lis
<class 'list'>
>>> type(type(L))                    # Even types are ob
<class 'type'>
```

Besides allowing you to explore your objects interactively, the `type` object in its most practical application allows code to check the types of the objects it processes. In fact, there are at least three ways to do so in a Python script:

```
>>> type(L) == type([])              # Type testing, if
True                                 # Using a real obje

>>> type(L) == list                  # Using a type name
True

>>> isinstance(L, list)              # The object-orient
True
```

But now that this book has shown you all these ways to do type testing, it's required by Python law to tell you that doing so is almost always the wrong thing to do in a Python program (and often a sign of an ex-Java programmer first starting to use Python!). The reason won't become completely clear until later in the book when we start writing larger code units like functions, but it's a—and perhaps *the*—core Python concept. By checking for specific types in your code, you effectively break its flexibility: you limit it to working on just one type. Without such checks, your code may be able to work on a whole range of types automatically.

This is part of the *polymorphism* mentioned earlier, and it stems from Python's lack of type declarations. As you'll learn more when we step up to coding functions and classes, in Python, we code to object *interfaces* (operations supported), not to types. That is, we care what an object *does*, not what it *is*.

Not caring about specific types means that code can be applied 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 probably find that polymorphism is the key to using Python well.

Type Hinting

That being said, Python has slowly accumulated a type-declaration facility known as *type hinting*, based originally on its earlier function annotations and inspired by the *TypeScript* dialect of JavaScript. With these syntax and module extensions, it is possible to name expected object types of function arguments and results, attributes in class-based objects, and even simple variables in Python code, and these hints may be used by external type checkers like *mypy*:

```
>>> x: int = 1           # Optional hint: x might be ar
>>> x = 'anything'       # But it doesn't have to be!
```

Importantly, though, Python type hinting is meant only for documentation and use by third-party tools. The Python language *does not itself mandate or use type declarations* and has no plans to ever do so. Hence, type hinting by most measures is largely academic, no more useful than in-program comments, and oddly contrary to Python's core ideas. The fact that it was nevertheless elevated to language syntax and complex subdomain arguably does a disservice to Python learners and users alike. Especially for beginners, this is an optional and peripheral topic that's best deferred until you master the flexibility of Python's dynamic typing. We'll study it only briefly in this book, in Chapter 6.

To be sure, type hinting does not mean that Python is no longer dynamically typed. Indeed, a statically typed Python that requires type declarations would not be a Python at all! Some programmers accustomed to restrictive languages may regrettably code Python type hints anyhow as a hard-to-break habit (or misguided display of prowess), but good programmers focus instead on Python's polymorphism. As you'll find, it's how to code Python in Python.

User-Defined Objects

We won't study *object-oriented programming* (OOP) in Python and its `class` statement until 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. Suppose, for example, that you wish to have a kind of object that models workers in a company. Although there is no such specific core object type in Python (it's not an HR language, after all), a user-defined class might fit the bill:

```
>>> class Worker:
        …stay tuned for Part VI…
```

Most of this code is omitted because it wouldn't make much sense at this point in the book, but such a class might define *attributes* of workers like `name` and `pay`, as well as *behavior* coded as custom methods. Calling the class would generate objects that are instances of our new type, and the class's methods would process them:

```
>>> sue = Worker('Sue Jones', 60000)          # Make two
>>> bob = Worker('Bob Smith', 50000)          # Each has
>>> sue.lastName()                            # Call a m
'Jones'
>>> bob.lastName()                            # Call a m
'Smith'
>>> sue.giveRaise(.10)                        # Update s
>>> sue.pay                                   # Display
66000.0
```

This is called *object-oriented*, because there is always an implied subject in functions within a class. Class-based objects ultimately use built-in objects internally, and we can always describe things like workers with Python's built-in objects instead, as we did with dictionaries and lists earlier. Classes, though, implement operations with meaningful names, add structure to your code, and come with inheritance mechanisms that lend themselves to customization by *extension*. In OOP, we strive to extend software by writing new classes, not by changing what already works.

All of which is well beyond the bounds of this object preview, though, so we must stop short here. For full disclosure on user-defined object types coded with classes, you'll have to read on. Because classes build upon other tools in Python, they are one of the major destinations of this book's journey.

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," not everything is considered a part of Python's *core* toolset. Other object types in Python either are related to program execution (like functions, modules, classes, and compiled code), or are implemented by imported module functions, not language syntax. The latter of these 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 the Python `class` statement, which you'll meet again later in this book. Still, Python's core objects are the workhorses of all Python scripts you're likely to meet and are often the basis of larger noncore objects.

Chapter Summary

And that's a wrap for our object tour. This chapter has previewed 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 (string splits and list appends, for instance). We've also defined some key terms, such as immutability, sequences, and polymorphism.

Along the way, we've learned that Python's core object types are more flexible and powerful than what is available in lower-level languages. For instance, Python's lists and dictionaries can nest, grow and shrink, and contain objects of any type, and their space is automatically created and cleaned up as you go. We've also glimpsed the ways that strings and files work hand in hand to support binary and text data, peeked at the iteration protocol and OOP,

discussed the perils of type hinting, and introduced the `if` and `for` statements we'll be using ahead.

This chapter skipped many of the details in order to provide a first 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, taking a second pass over Python's object types that will fill in details omitted here and give you a deeper understanding. We'll start off the next chapter with an in-depth look at Python numbers. First, though, here is another quiz to review.

Test Your Knowledge: 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 object types.
2. Why are they called "core" object types?
3. What does "immutable" mean, and which three of Python's object types are considered immutable?
4. What does "sequence" mean, which objects fall into this category, and how is "iterable" related?
5. What does "mapping" mean, and which core object type is a mapping?
6. What is "polymorphism," and why should you care?

Test Your Knowledge: Answers

1. Numbers, strings, lists, dictionaries, tuples, files, and sets are generally considered to be the core object types. Booleans, `None`, and types themselves are classified this way as well. Some of these types are really categories: there are multiple number types (integer, floating point, complex, fraction, and decimal) and multiple string types (text strings, byte strings, and mutable byte strings).
2. They are known as "core" object 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 objects have specific syntax for generating their objects: `'hack'`, 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 objects are

hardwired into Python's syntax. In contrast, you must call the built-in `open` function to create a file object, even though this is usually considered a core object type too.

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. `bytearrays` offer mutability for strings, but they only apply directly to text if it's a simple 8-bit kind (e.g., ASCII).
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. The related term "iterable" means either a physical sequence, or a virtual one that produces its items on request. Sequences are iterable, but so are generators, files, results of functions like `range` , and the dictionary object (which produces its keys when iterated, just like its `keys` method).
5. The term "mapping" denotes an object that maps keys to associated values. Python's dictionary is the only mapping among its core object types. Mappings retain insertion order (as of Python 3.7), and support access to data stored by key, plus type-specific method calls that enable key tests, iteration, and more.
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 (and perhaps the largest) behind using Python well—not constraining code to specific types makes that code automatically applicable to many types. This becomes more obvious when coding functions and classes in Python. While Python today has type hinting, it's not used by Python itself and is meant only for documentation and tools.