Chapter 37. Unicode and Byte Strings

In Chapter 7, the Python string story was watered down on purpose to help you get started with the fundamentals. Now that you've learned the basics, this chapter moves on to extend them to include the full Unicode-text and binary-data string tales in Python.

This extension was more optional in earlier editions of this book because Unicode was an afterthought in Python 2.X. Python 3.X elevates it to required reading because its normal strings simply *are* Unicode. Still, how much you need to care about this topic depends in large part upon which of the following categories you fall into:

- If you deal with non-ASCII *Unicode text*—for instance, in the context of internet content, internationalized applications, XML parsers, and some GUIs—you will find direct and seamless support for text encodings in both Python's all-Unicode `str` object, as well as its Unicode-aware text files.
- If you deal with *binary data*—for example, in the form of image or audio files, network transfers, or packed data shared with lower-level tools—you will need to understand Python's `bytes` object and its sharp distinction between text and binary data and files.
- If you fall into *neither* of the prior two categories, you may be able to defer this topic and use strings as you did in Chapter 7: with the general `str` object, text files, and all the familiar string operations. Your strings will be encoded and decoded using your platform's default Unicode encoding, but you won't notice—until, as you'll see, you encounter content or platforms that use a different default!

To be sure, if text is always ASCII in your corner of the software world, you might be able to get by with simple string objects and text files and can avoid much of the story that follows. As you'll learn in a moment, ASCII is a simple kind of Unicode and a subset of other common encodings, so string operations and files "just work" if your programs process ASCII text only and will never deviate from this limitation.

Even if this chapter's topics seems remote to you today, though, a basic understanding of Python's string model can both demystify some of the underlying details now and prepare you for Unicode or binary-data issues that may impact you in the future. Given the prominence of the web in most software careers today, that impact may be more a matter of *when* than *if*.

Unicode Foundations

Before jumping into code, let's begin with a general overview of the Unicode model and Python's support for it. To fully understand both, we have to start with a brief look at how characters are actually represented in computers.

Character Representations

Most programmers think of strings as a series of characters (really, their integer codes) used to represent textual data. That's still true in the brave new world of Unicode, but the way characters are stored in a computer's memory and files can vary, depending on both what sort of characters are recorded and how programmers choose to record them.

For many programmers in the US, *ASCII* formed their original notion of text strings. ASCII is a standard that defines character codes 0…127 (which always means an inclusive range in this chapter) and thus allows each character to be stored in one 8-bit byte (using 7 bits). For example, the ASCII standard maps character `a` to the integer value 97 (`0x61` in hex), which can be stored in a single byte both in computer memory and on files.

To witness this for yourself, Python's built-in `ord` function shows the integer code of a given character; `chr` reveals the character of a given integer code; and `hex` gives the code's byte value as two hex digits, each of which fits a 4-bit "nibble." The first of these, `ord` , is the value of a character's representation code—and byte—in ASCII:

```
$ python3           # Or py -3 on Windows
>>> ord('a')        # Character => code
97
>>> chr(97)         # Code => character
'a'
>>> hex(97)         # Byte value: fits 8 bits
'0x61'
```

```
>>> 0b0111_1111    # Limit of ASCII's 7-bit range
127
```

ASCII makes text processing simple, because characters directly correlate to bytes. Sometimes, though, this isn't enough. Accented characters and special symbols, for example, do not fit into the range of character codes defined by ASCII. To allow for some such extra characters, other standards allow all possible values in an 8-bit byte, 0…255, to be used as codes, and assign values 128…255 to additional characters.

One such standard is known as *Latin-1* and is widely used in Western Europe. In Latin-1, character codes above 127 are assigned to accented and otherwise special characters. For instance, the character that Latin-1 assigns to code 196 (a.k.a. byte value `0xc4`) is a specially marked and non-ASCII character, `Ä`. In Python:

```
>>> chr(196)         # Too big for ASCII
'Ä'
>>> ord('Ä')         # Okay for Latin-1
196
>>> hex(ord('Ä'))    # Byte value in Latin-1
'0xc4'
>>> bin(ord('Ä'))    # Latin-1 uses all 8 bits
'0b11000100'
```

Still, some alphabets define so many characters that it is impossible to represent them as one byte-sized code per character. The integer codes of the symbols and characters in the following, for example, require more space than a byte—as do those of all the emojis that may not work in some tools but manage to crop up in your emails and texts anyhow:

```
>>> ord('
☞
')
9758
>>> hex(ord('
☞
'))                    # Too big for one byte
'0x261e'

>>> [hex(ord(c)) for c in '
真
Л
➡
```

```
']     # Ditto: Unicode required
['0x771f', '0x41b', '0x21e8']

>>> [hex(ord(c)) for c in '
🙂
🙊
👍
']    # Emojis: > two bytes (16 bits)
['0x1f642', '0x1f64a', '0x1f44d']
```

Unicode provides the generality we need to deal with text containing non-ASCII characters and symbols like these. In fact, it defines and assigns enough *character codes* to represent almost every natural language in use, plus a large set of symbols and emojis. In Unicode speak, these codes that stand for characters take the form of numbers (integers), and are usually called *code points*. The code points that Unicode assigns to characters `a`, `Ä`, and 🙂, for instance, are 97, 196, and 128578 (`0x61`, `0xc4`, and `0x1f642` in hex), respectively:

```
>>> [f'{c} is {ord(c)} and {hex(ord(c))}' for c in 'aÄ
🙂
']
['a is 97 and 0x61', 'Ä is 196 and 0xc4', '
🙂
 is 128578 and 0x1f642']
```

Unicode is sometimes referred to as “wide-character” strings because its range of characters is so broad that multiple bytes may be needed to represent individual character codes. Such text is readily stored in computer memory because each character code can simply span as many bytes as its code-point number requires (the exact way this is done can vary by programming language and isn’t consequential to your Python code).

Once text leaves your computer, though, its storage is more constrained: bytes are a bad thing to waste on your drives and networks, and text used across platforms must follow the same formatting rules. To allow for this, Unicode also defines standard ways to map character codes to and from bytes for storage and transmission that are both platform- and language-neutral—the *encodings* we’ll explore in the next section.

The takeaway here is that Unicode’s combination of all-encompassing character codes and their predefined encodings make it a portable and flexible model, and the standard way that programs deal with non-English and other

text that may have more characters than 8-bit bytes can handle. As an added bonus, earlier schemes like ASCII also fall under the Unicode umbrella unchanged, but we have to move on to the next section to see how.

Character Encodings

One of the keys to understanding how Unicode works lies in the way its integer character codes (a.k.a. code points) that represent characters are mapped to their encoded forms for efficient storage or transfer. Code points in memory are just integers of arbitrary size, but storage and transfer, by nature, impose constraints on time, space, and interoperability that warrant extra formatting steps.

In the Unicode world, we say that characters are translated to and from raw bytes using an *encoding*—the rules for translating a Unicode-text string into a sequence of bytes and extracting the same string from its sequence of bytes. More procedurally, this translation back and forth between bytes and strings is defined by two terms (the first of which doubles as a noun and verb, confusingly!):

- *Encoding* is the process of translating a string of characters into its raw-bytes form, per any desired encoding that's broad enough to store the string's characters.
- *Decoding* is the process of translating a string of raw bytes into its character-string form, per the encoding originally used to create the bytes string.

As we've seen, Unicode defines both character codes and a set of standard encodings. For some of the encodings it defines, the translation process is trivial—ASCII and Latin-1, for instance, map each character to a single byte, so little or no work is required to encode and decode if characters are the same bytes in memory too.

You can view this for yourself with the `encode` method available on all Python text strings, which simply returns the bytes used to encode the string. The following means that the ASCII character `a` occupies just one byte when encoded per the ASCII encoding:

```
>>> len('a'.encode('ASCII'))      # ASCII 'a' encodes i
```

```
1
```

For other encodings, the mapping can be more complex and yield multiple bytes per character. The widely used *UTF-8* encoding, for example, allows more characters to be represented by employing a variable-number-of-bytes scheme that's both general and economical. In fact, because UTF-8 can handle any Unicode code point, it's become a de facto standard of sorts for text.

In UTF-8, character codes less than 128 are represented as a single byte; codes between 128 and `0x7ff` (2047) are turned into two bytes, where each byte has a value between 128 and 255; and codes above `0x7ff` are turned into three- or four-byte sequences having values between 128 and 255. This keeps simple ASCII strings compact, sidesteps byte ordering issues, and avoids null (zero) bytes that can cause problems for C libraries and networking. In Python:

```
>>> len('a'.encode('UTF-8'))      # ASCII: encodes in 1
1
>>> len('Ä'.encode('UTF-8'))      # Non-ASCII: encodes
2
>>> len('
🙂
'.encode('UTF-8'))     # Emoji: encodes in 4 bytes
4
```

Despite such details, it's important to note that ASCII is a *subset* of both Latin-1 and UTF-8. This is true because these encodings encode ASCII characters to bytes the same way as ASCII. That, in turn, makes these encodings backward compatible with existing ASCII data: every character string encoded per ASCII is also valid according to the Latin-1 and UTF-8 encodings, and every ASCII file is a valid Latin-1 and UTF-8 file.

Technically, the ASCII encoding is a 7-bit subset of the other two: it's binary compatible with all character codes less than 128. Latin-1 and UTF-8 simply allow for additional characters: Latin-1 for characters mapped to values 128…255 within a byte, and UTF-8 for characters that may be represented with multiple bytes. The converse is not true, however: UTF-8 and Latin-1 text is not compatible with the ASCII encoding unless its text's code-point values are all less than 128; otherwise, encoding or decoding per ASCII fails.

In Python again, the mapping is easy to observe. Per the following, an ASCII character encodes to the same single byte in ASCII, UTF-8, and Latin-1 encodings, but non-ASCII characters do not, and require more general encodings than ASCII to encode at all (the `b'...'` here is the Python `bytes` object, which will soon play a leading role in this chapter):

```
>>> 'a'.encode('ASCII')          # ASCII encodes to the
b'a'
>>> 'a'.encode('UTF-8')
b'a'
>>> 'a'.encode('Latin-1')
b'a'

>>> 'Ä'.encode('UTF-8')          # But non-ASCII requir
b'\xc3\x84'
>>> '
🙂
'.encode('UTF-8')          # And more inclusive encoding
b'\xf0\x9f\x99\x82'

>>> '
🙂
'.encode('ASCII')
UnicodeEncodeError: 'ascii' codec can't encode characte
>>> '
🙂
'.encode('Latin-1')
UnicodeEncodeError: 'latin-1' codec can't encode charac
```

Other encodings support richer character sets in other ways. For instance, *UTF-16* and *UTF-32* use a fixed and larger 2 and 4 bytes per character, respectively, the former with a special *surrogate-pair* protocol for codes too large for 2 bytes. Both of these, along with UTF-8, may also allow or require a *BOM* (Byte Order Marker) preamble at the start of encoded text, which can designate byte order and encoding type, may be present in encoded text stored in files or memory, and is automatically handled for text-mode files.

We'll skip further details here for space (watch for the BOM to drop at the end of this chapter, along with the thorny topic of Unicode normalization), but keep in mind that all of these—ASCII, Latin-1, UTF-8, and others—are simply alternative Unicode encodings that yield the same Unicode code-point

text when decoded. The net effect ensures that text is *portable* across all the tools that use it in exchange for minor translation costs:

- When *decoded*, character code points may or may not occupy multiple bytes in memory, depending on programming-language implementation. Some recent Pythons, for example, use a variable-length scheme to store decoded text with 1, 2, or 4 bytes per character, depending on string content. Earlier Pythons instead store each character in a fixed 2 or 4 bytes, depending on compilation settings.
- When *encoded*, the format of character code points is wholly determined by the standard Unicode encoding applied. This format is the same regardless of which programming language creates or processes the text, making it ideal for storage and transfer—especially in the diverse realm of the internet. This format is often less ideal for programs to use, though, which is why it's normally decoded when loaded.

To Python programmers, an encoding is specified as a string containing the encoding's name. Python comes with roughly 100 different encodings out of the box; see the `codecs` module in the Python Library Reference for a list. Importing module `encodings` and asking for `help(encodings)` shows you many as well. Some encodings are implemented in Python, and some in C, and many have multiple names; for example, `latin-1`, `iso_8859_1`, and `8859` are all synonyms for the same encoding, `Latin-1`. We'll revisit encodings later in this chapter when we study Unicode coding techniques.

For another take on the Unicode backstory, see the Python standard manual at *python.org*. It includes a Unicode HOWTO section, which provides additional minutiae that we will skip here to focus on the fundamentals.

Introducing Python String Tools

At a more concrete level, the Python language provides multiple string data types to represent content in your script: both *textual data*—integer code-point values of decoded Unicode characters in memory—as well as *binary data*—raw byte values, including text that is in encoded form. All told, Python comes with three string object types:

- `str` —for representing Unicode text (decoded code points)
- `bytes` —for representing binary data (including encoded text)

- `bytearray` —a mutable flavor of the `bytes` type

All three types support similar operation sets but have very different roles and cannot generally be mixed because of this. Moreover, files and other content tools reflect the text/binary dichotomy, too, and use specific string types in different nodes. The next sections introduce the salient points of this model.

The str Object

First up, the basic `str` type (e.g., `'text'`) is for decoded Unicode text. It's formally defined as an *immutable sequence of characters*—which means code points that are not necessarily bytes. Its content may contain both simple text, such as ASCII, whose encoded and decoded forms might yield one byte per character, as well as richer Unicode text, whose encoded and decoded forms may both require multiple bytes per character.

In memory, a `str` is just an ordered collection of Unicode code-point integers, which print as *glyph*s—visual representations that may vary from host to host—of the characters that the code points represent. When transferred to and from files, a `str` is automatically encoded to and decoded from a sequence of bytes using either the host platform's default or a provided encoding name to translate with an explicit scheme. `str` objects themselves, however, have *no notion of an encoding*; they are just character code points.

The bytes Object

While `str` is great for Unicode text, many programs need to process raw binary content that is not encoded per any Unicode format—as well as the bytes used to store text when it is encoded. Image files and packed data you might process with Python's `struct` module fall into this category. To accommodate this, the `bytes` type supports processing of truly binary data. `bytes` is just raw bytes, not Unicode-text characters, though its content may include the bytes of still-encoded text, which always has an implied encoding.

The `bytes` type is formally defined as an *immutable sequence of 8-bit integers*. Its content represents byte values, and it supports almost all the same operations that the `str` type does; this includes string methods, sequence operations, and even `re` module pattern matching (formatting works on `bytes` today, too, but was added later in 3.X's evolution).

A `bytes` object really is a sequence of small integers, each of which is in the range 0…255: indexing a `bytes` returns an `int`, slicing one returns another `bytes`, and running `list` on one returns a list of integers, not characters. However, when processed with operations that assume characters (e.g., the `isalpha` method), the contents of `bytes` objects are assumed to be ASCII-encoded bytes. Further, `bytes` items whose values fall in the range of ASCII character codes are printed as ASCII-character glyphs instead of integers or their hex escapes; this is done for convenience, though it may also confuse the distinction between text and binary data.

The bytearray Object

Though less commonly used, Python also comes with `bytearray`, a variant of `bytes` that is *mutable* and so supports in-place changes. The `bytearray` type provides the usual string operations that `str` and `bytes` do but also has many of the same in-place change operations as lists (e.g., `append` and `extend` methods and assignment to indexes). Assuming your strings can be treated as raw bytes, `bytearray` adds direct in-place mutability for string data—something long prohibited by `str` and `bytes`.

Text and Binary Files

Because file I/O is one of the main benefactors of encodings, it's also a core Unicode tool. As we've seen, text is really just decoded integer character codes when it is in memory; it's when text is transferred to and from *external interfaces* like files that Unicode encodings come into play. By contrast, truly binary data may have nothing at all to do with encodings—or text at all. Because of this, Python makes a sharp platform-independent distinction between text and binary files accessed with the built-in `open` function:

- When a file is opened in *text mode*, reading its data automatically decodes its content and returns it as a `st`r, and writing takes a `str` and automatically encodes it before transferring it to the file. In both cases, the encoding to use is either a platform default or a provided `encoding` argument to `open`. Text mode files also support universal newline (a.k.a. end-of-line) translation, BOMs, and other encoding arguments.
- When a file is opened in *binary mode* by adding a `b` to the mode string argument in the `open` call, reading its data does not decode it in any way and simply returns its content raw and unchanged as a `bytes` object.

Writing similarly takes a `bytes` object and transfers it to the file unchanged, and binary-mode files also accept a `bytearray` object for the content to be written to the file.

Because `str` and `bytes` are sharply differentiated by the language this way, you must decide whether your data is text or binary in nature and use `str` or `bytes` objects to represent its content in your script, respectively. Ultimately, the mode in which you open a file will dictate which type of *object* your script will use to represent its content:

- If you are processing image or audio files, packed data created by other programs whose content you must extract, and some device data streams, chances are good that you will want to deal with it using `bytes` and binary-mode files. You might also opt for `bytearray` to update the data without making copies of it in memory.
- If instead you are processing something that is textual in nature, such as program output, HTML or JSON content, and CVS or XML files, you probably want to use `str` and text-mode files.

Subtly, the mode-string argument to `open` (its `mode` keyword and second positional argument) becomes fairly crucial—its content not only specifies a file processing *mode* but also implies a Python object *type*. By adding a `b` (lowercase only) to the mode string, you specify a binary mode file and will receive, or usually provide, a `bytes` object to represent the file's content when reading or writing. Without the `b`, your file is processed in text mode, and you'll use `str` objects to represent its content. For example, modes `rb`, `wb`, and `rb+` imply `bytes`, but `r`, `w+`, and `rt` (the default: read text) imply `str`.

If you're anxious to see files in action, watch for the examples ahead, especially those of Unicode text files. To understand file usage in full, though, we first need to explore string operations as they extend to Unicode and bytes.

Using Text Strings

Let's step through a few examples that demo the prior section's string types live. Here, our primary focus is on using these types for text (we'll explore binary roles later). Along the way, you'll see how literals, conversions, and non-ASCII text are coded in Python.

Literals and Basic Properties

Most Python string objects are born when you call a built-in function such as `str` or `bytes`, process a file created by calling `open`, or code literal syntax in your script. For the latter, the usual `'...'` makes a `str`; a unique literal form `b'...'` is used to create a `bytes`; and `bytearray` objects may be made by calling the same-named function with a variety of possible arguments.

More formally, all the usual string literal forms we met in Chapter 7— `'...'`, `"..."`, and triple-quoted blocks—generate a `str`; adding a `b` or `B` just before them creates a `bytes` instead. This `b'...'` (and equivalently, `B'...'`) bytes literal is similar in spirit to the `r'...'` raw string we've also met, which suppresses backslash escapes; in fact, the two prefixes can be combined to use backlashes verbatim in a `bytes`. Consider the following:

```
>>> B = b'code'                  # Make a bytes object: &
>>> S = 'hack'                   # Make a str object: Uni

>>> type(B), type(S)
(<class 'bytes'>, <class 'str'>)

>>> B                            # Sequence of ints, prir
b'code'
>>> S                            # Sequence of code point
'hack'

>>> B2 = B"""                    # bytes prefix works on
xxxx
yyyy
"""
>>> B2
b'\nxxxx\nyyyy\n'

>>> b'A\nB\rC', br'A\nB\rC', rb'A\nB\rC'     # Raw-strir
(b'A\nB\rC', b'A\\nB\\rC', b'A\\nB\\rC')
```

Once you have a string, all the usual operations we met earlier work, but their results are type specific (`bytes` is integers and `str` is characters), and immutability still applies:

```
>>> B, S
(b'code', 'hack')

>>> B[0], S[0]                  # Indexing returns an ir
(99, 'h')

>>> B[1:], S[1:]                # Slicing makes another
(b'ode', 'ack')

>>> list(B), list(S)
([99, 111, 100, 101], ['h', 'a', 'c', 'k'])     # bytes

>>> B[0] = 'x'                                  # Both
TypeError: 'bytes' object does not support item assignm
>>> S[0] = 'x'
TypeError: 'str' object does not support item assignmer
```

Because they apply to more than text and have some unique behaviors, we'll defer `bytearray` and more about `bytes` to a dedicated section later in this chapter.

NOTE

Blast from the past: Python 3.X also recognizes Python 2.X's Unicode string literals to ease migration of 2.X code: a 2.X `u'…'` literal in Python 3.X is just a synonym for a 3.X `'…'` `str` literal. This makes sense, given that 3.X's `str` is all Unicode, and allows some 3.X code to run on 2.X and vice versa. In today's 3.X world, though, there's no compelling reason to use the `u'…'` literal anymore (unless you're a fan of superfluous prefixes), but it may crop up in Python code you'll encounter in the wild. 2.X had a very long shelf life, after all.

String Type Conversions

Syntax aside, the first thing you might notice about Python strings is what they *cannot* do— `str` and `bytes` never mix automatically in expressions and generally are not converted to one another automatically when passed to functions. A function that expects an argument to be a `str` may not accept a `bytes` (and vice versa), and operators are fully rigid:

```
>>> 'hack' + b'code'
TypeError: can only concatenate str (not "bytes") to st
```

This is easier to understand if you remember that a text string may be radically different in its encoded and decoded forms, and Python has no idea what the content of a `bytes` is: if the `bytes` is encoded text, its encoding is unknown, but it may also be binary data (e.g., a loaded audio file) that has nothing to do with text at all.

Because of this ambiguity, Python basically requires that you either commit to one type or the other or perform manual, explicit conversions with the following tools (where `?` means optional):

`S.encode(encoding?)` and `bytes(S, encoding)`

Encode a `str` object *`S`* to a new `bytes` per *`encoding`*

`B.decode(encoding?)` and `str(B, encoding)`

Decode a `bytes` object *`B`* to a new `str` per *`encoding`*

Both the preceding *`S`*`.encode()` and *`B`*`.decode()` methods and the file `open` call we'll explore ahead use either an explicitly passed-in encoding name or a default. The methods' default is always UTF-8 (by contrast, `open` uses a value in the `locale` module you'll meet shortly that may vary per platform, settings, and run and should usually be avoided):

```
>>> S = 'hack'
>>> S.encode()                      # str to bytes: encc
b'hack'

>>> bytes(S, encoding='ascii')      # str to bytes, alte
b'hack'

>>> B = b'code'
>>> B.decode()                      # bytes to str: decc
'code'

>>> str(B, encoding='ascii')        # bytes to str, alte
'code'
```

Putting this together solves our original type error and allows us to mix strings and bytes as either encoded or decoded text:

```
>>> S, B
('hack', b'code')

>>> S.encode('ascii') + B          # bytes + bytes (enc
b'hackcode'

>>> S + B.decode('ascii')          # str + str (code pc
'hackcode'
```

A few cautions on defaults here. First of all, the encoding argument to `bytes` is *not optional*, even though it is in `S.encode()` (and `B.decode()`). More subtly, although `str` does not require the encoding argument like `bytes` does, leaving it off in `str` calls does not mean it defaults—instead, due to Python history, a `str` without an encoding returns the `bytes` object's *print string*, not its decoded and converted `str` form (this is usually not what you'll want!).

Assuming again that `B` and `S` are still as in the prior listing:

```
>>> bytes(S)
TypeError: string argument without an encoding

>>> str(B)                                  # str() works
"b'code'"                                   # But print st
>>> len(str(B))
7

>>> len(str(B, encoding='ascii'))           # Pass encodir
4
```

Also in the defaults department, your platform's various default encodings are available in the `sys` and `locale` modules but aren't as trustworthy as you might think:

```
$ py -3
>>> import sys, locale
>>> sys.platform                            # Underlying p
```

```
'win32'
>>> sys.getdefaultencoding()              # Methods defa
'utf-8'
>>> locale.getpreferredencoding(False)    # open() defau
'cp1252'
```

As shown, the `open` function's default file encoding lives in the `locale` module. On the PC used for Windows examples in this chapter, it's *cp1252*—a superset of Latin-1 that adds characters like slanted quotes. Defaults may differ, however, on other *platforms* and technically can even depend on environment-variable settings, command-line arguments, and settings on individual host machines.

For example, here's the differing case on this chapter's macOS host—like most Unix platforms, its `open` defaults to *UTF-8*:

```
$ python3
>>> import sys, locale
>>> sys.platform                          # Underlying p
'darwin'
>>> sys.getdefaultencoding()              # The default
'utf-8'
>>> locale.getpreferredencoding(False)    # But this dif
'utf-8'
```

Besides such program-host differences, keep in mind that text *content* you receive from disparate sources might also use any Unicode encoding at all, making your host's default a moot point. Hence, your programs shouldn't generally rely on `open` defaults if they may need to care about portability now or in the future—always pass an explicit encoding to `open` when interoperability counts. We'll revisit encoding defaults and learn how to provide an explicit encoding to `open` when we explore Unicode files later in this chapter.

Having said all that, it's important to also note that encoding and decoding are substantially more than simple programming-language type conversions; really, they produce very different kinds of data. Encoding returns the bytes that result from transforming a text string per a Unicode scheme, and decoding returns the text string that is produced by undoing that transformation. While this is a conversion of sorts, and the mapping may seem

trivial for simple text like ASCII, Unicode tends to make much more sense if you avoid blurring the distinction—especially for richer types of text like that in the next section.

Coding Unicode Strings in Python

Encoding and decoding grow more meaningful when you start dealing with non-ASCII Unicode text. To code Unicode characters that may be difficult to type on your keyboard, Python string literals support both:

- `\x`*`NN`* hex escapes, where two hex digits (*`NN`*) specify a character code as a 1-byte (8-bit) numeric value
- `\u`*`NNNN`* and `\U`*`NNNNNNNN`* Unicode escapes, where the first *lowercase* form gives 4 hex digits to denote a 2-byte (16-bit) character code, and the second *uppercase* form gives 8 hex digits for a 4-byte (32-bit) code

Importantly, in `str` objects, all three of these escapes are used to give a Unicode character's *code-point* value—not its encoded bytes. By contrast, `bytes` objects allow only hex escapes for byte values; for text, this gives its *encoded form*—not its decoded code points.

Let's see how this all translates to code. Simple 7-bit *ASCII* text is formatted with one character per byte under most of the encoding schemes described near the start of this chapter (again, this is why ASCII passes as a binary-compatible subset of many other schemes):

```
>>> ord('X')                # Character 'X' has code-pc
88
>>> chr(88)                 # Code-point 88 stands for
'X'

>>> S = 'XYZ'               # str: code points display
>>> S
'XYZ'
>>> len(S)                  # 3 characters (not necessc
3

>>> S.encode('ascii')       # Values 0…127 in 1 byte ec
b'XYZ'
>>> S.encode('latin-1')     # Values 0…255 in 1 byte ec
b'XYZ'
```

```
>>> S.encode('utf-8')       # Values 0…127 in 1 byte, 1
b'XYZ'
```

By contrast, the less common *UTF-16* and *UTF-32* use 2 and 4 bytes for every character, respectively, even for simple text like ASCII. This makes these encodings' data fast to process but may consume extra space and bandwidth, which renders them subpar in some applications. In the following, ASCII bytes print as characters, non-ASCIIs print as `\xNN` escapes, padding bytes follow text, and each result has a 2- or 4-byte BOM header at the front whose details we're largely ignoring here (again, stay tuned for more on BOMs near the end of this chapter):

```
>>> S
'XYZ'

>>> S.encode('utf-16')      # Always 2 or 4 bytes per c
b'\xff\xfeX\x00Y\x00Z\x00'

>>> S.encode('utf-32')
b'\xff\xfe\x00\x00X\x00\x00\x00Y\x00\x00\x00Z\x00\x00\x
```

To code *non-ASCII* characters, you can use hex and Unicode escapes in your strings. The numeric values coded as hexadecimal literals `0xC4` and `0xE8`, for instance, are the Unicode code points used to represent two special characters outside the 7-bit range of ASCII; we can embed them in `str` objects anyhow because `str` supports Unicode in full:

```
>>> chr(0xc4)               # 0xC4 and 0xE8 are accente
'Ä'
>>> chr(0xe8)
'è'

>>> S = '\xc4\xe8'          # Hex escapes: code-point v
>>> S
'Äè'

>>> S = '\u00c4\u00e8'      # Unicode escapes: 16-bits
>>> S
'Äè'
```

```
>>> len(S)                  # 2 characters long (not nu
2
```

Now, if we try to encode a non-ASCII string like this to raw bytes as ASCII, we'll get an error. Encoding as Latin-1 works, though, and allocates 1 byte per character; encoding as UTF-8 allocates 2 bytes per character instead. If you write this string to a text-mode file, the raw bytes shown are what is actually stored on the file for the encoding types given:

```
>>> S = '\u00c4\u00e8'
>>> S.encode('ascii')
UnicodeEncodeError: 'ascii' codec can't encode characte
ordinal not in range(128)

>>> S.encode('latin-1')             # 1 byte per charc
b'\xc4\xe8'

>>> S.encode('utf-8')               # 2 bytes per char
b'\xc3\x84\xc3\xa8'

>>> len(S.encode('latin-1'))        # 2 bytes in latir
2
>>> len(S.encode('utf-8'))
4
```

You can also go the other way—from raw bytes back to a Unicode string. You could read raw bytes from a file and decode manually this way, but the encoding mode you give to the `open` call causes this decoding to be done for you automatically (and avoids issues that may arise from reading partial character sequences when reading by blocks of bytes):

```
>>> B = b'\xc4\xe8'
>>> B
b'\xc4\xe8'
>>> len(B)                          # 2 raw bytes, 2
2
>>> B.decode('latin-1')             # Decode to lati
'Äè'

>>> B = b'\xc3\x84\xc3\xa8'
>>> len(B)                          # 4 raw bytes
4
```

```
>>> B.decode('utf-8')
'Äè'
>>> len(B.decode('utf-8'))              # 2 Unicode char
2
```

When needed, you can also specify both 16- and 32-bit Unicode code-point values for characters in your `str` strings: use `\u…` with 4 hex digits for the former and `\U…` with 8 hex digits for the latter. As the last example in the following shows, you can also build such strings up piecemeal using `chr`, but it might become tedious for large strings:

```
>>> S = 'A\u00c4B\U000000e8C'
>>> S                                   # A, B, C, and 2
'AÄBèC'
>>> len(S)                              # 5 characters l
5

>>> S.encode('latin-1')
b'A\xc4B\xe8C'
>>> len(S.encode('latin-1'))            # 5 bytes in lat
5

>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'
>>> len(S.encode('utf-8'))              # 7 bytes in utf
7

>>> S.encode('cp500')                   # Two other West
b'\xc1c\xc2T\xc3'
>>> S.encode('cp850')                   # 5 bytes each
b'A\x8eB\x8aC'

>>> S = 'code'                          # ASCII text is
>>> S.encode('latin-1')
b'code'
>>> S.encode('utf-8')
b'code'
>>> S.encode('cp500')                   # But not in cp5
b'\x83\x96\x84\x85'
>>> S.encode('cp850')
b'code'

>>> S = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'    # st
```

```
>>> S
'AÄBèC'
```

Another distinction to keep in mind: Python allows special characters' code points to be coded with both hex and Unicode escapes in `str` string literals but allows only *hex* escapes in `bytes` literals—and prints a warning in its recent versions if you violate this rule. In fact, Unicode escape sequences are taken *verbatim* in `bytes` and not as escapes.

This makes sense if you remember that `bytes` objects hold binary data, both textual and not. When they contain text, they hold characters' *encoded* bytes—not their decoded code points. Thus, Unicode code-point escapes simply don't apply to `bytes` , and hex escapes in their literals yield raw byte values, not characters.

This is true even though code-point and encoded-byte values happen to be the same for some characters in some encodings (confusingly!). Because `bytes` are not code points, they also must be decoded to `str` to print their non-ASCII characters properly:

```
>>> S = 'A\xC4B\xE8C'            # str recognizes hex c
>>> S
'AÄBèC'

>>> S = 'A\u00C4B\U000000E8C'    # 4- and 8-digit Unicc
>>> S
'AÄBèC'

>>> B = b'A\xC4B\xE8C'           # bytes recognizes hex
>>> B
b'A\xc4B\xe8C'

>>> B = b'A\u00C4B\U000000E8C'   # Unicode escape seque
<stdin>:1: SyntaxWarning: invalid escape sequence '\u'
>>> B                            # bytes is encoded byt
b'A\\u00C4B\\U000000E8C'

>>> B = b'A\xC4B\xE8C'           # Use hex escapes for
>>> B                            # Prints non-ASCII byt
b'A\xc4B\xe8C'
>>> print(B)                     # For both interactive
b'A\xc4B\xe8C'
```

```
>>> B.decode('latin-1')          # Decode to str to int
'AÄBèC'
```

Finally, notice that `bytes` literals assume that textual characters embedded within them are ASCII and require escapes for byte values > 127. By contrast, `str` literals in code like that in the following allow embedding any character supported by the source code encoding of the hosting file or GUI (as you'll learn in a moment, the encoding used for code defaults to UTF-8, sans declarations in the code's file):

```
>>> S = 'AÄBèC'                  # Chars from UTF-8 if
>>> S                            # Decoded to str when
'AÄBèC'

>>> B = b'AÄBèC'
SyntaxError: bytes can only contain ASCII literal chara

>>> B = b'A\xC4B\xE8C'           # Chars must be ASCII,
>>> B                            # Non-ASCIIs are latir
b'A\xc4B\xe8C'
>>> B.decode('latin-1')
'AÄBèC'

>>> S.encode()                   # Source code encoded
b'A\xc3\x84B\xc3\xa8C'           # Methods use UTF-8 tc
>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'
>>> B.decode()                   # Raw bytes do not cor
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc

>>> S = 'AÄBèC'
>>> S.encode()                   # Method's default utf
b'A\xc3\x84B\xc3\xa8C'
>>>
>>> T = S.encode('cp500')        # "Convert" to EBCDIC
>>> T
b'\xc1c\xc2T\xc3'
>>>
>>> U = T.decode('cp500')        # Back to Unicode code
>>> U
'AÄBèC'
>>>
```

```
>>> U.encode()                    # Back to UTF-8 bytes,
b'A\xc3\x84B\xc3\xa8C'
```

Notice how the last part of the preceding code seems to "convert" encodings from UTF-8 to cp500 and back again. Really, this just creates different encoded representations of the same Unicode code points, but this pattern can be used to translate encoded text when needed. Text in a file, for instance, can be re-encoded with a decode (to `str`) plus an encode (to `bytes`) combination that changes its stored encoding; as you'll see ahead, the `open` function does most of this work for you.

Also, note how the preceding code is able to use a `str` literal `'AÄBèC'` with raw Unicode characters for its non-ASCII characters. This is noticeably simpler than coding escapes and works as long as your code file (and GUI) support it, as the next section will explain.

Source-File Encoding Declarations

Unicode escapes suffice for the occasional Unicode character in string literals, but they can become tedious if you need to code non-ASCII text in your strings frequently. For string literals and other text that you embed in your script files (or paste into your coding GUI), Python uses the *UTF-8* encoding by default to read your code's text but allows you to change this per file to use an arbitrary encoding. With this support, your code can directly embed any unescaped characters that the chosen encoding supports.

To make this work, simply use Python's default UTF-8 encoding to save your source code file in your text editor, or include a comment that names the Unicode encoding that you used for the save if it differs. This special encoding-declaration comment must appear as either the first or second line in your script (e.g., a `#!` line works before it: see Appendix A) and is usually of the following form (see Python's manuals for other forms it accepts):

```
# -*- coding: latin-1 -*-
```

When present, Python will recognize text in your code represented natively (unescaped) in the given encoding. That way, you can edit your script file in a text editor that accepts, displays, and saves accented and other non-ASCII

characters, and Python will correctly decode them when reading your string literals and other program-file text.

For example, notice the `coding` comment at the top of Example 37-1: when this file is saved in the nondefault Latin-1 encoding, it allows Python to recognize Latin-1 characters embedded in the string literal to be assigned to `myStr1` in the text of the source file. This file also neatly summarizes the various ways to code non-ASCII text in Python.

Example 37-1. source-encoding-latin1.py

```
# -*- coding: Latin-1 -*-

#-------------------------------------------------------
# Demo all the ways to code non-ASCII text in Python, p
#
# If this file is saved as Latin-1 text, it works as is
# coding line above to either ASCII or UTF-8 will then
# Latin-1 0xc4 and 0xe8 saved in myStr1's value are not
#
# A UTF-8 line works if this file is also saved as UTF-
# text match.  Because UTF-8 is the default for source,
# optional if the file is saved as UTF-8 or its text is
# (e.g., ASCII, which is a subset of both the Latin-1 a
#-------------------------------------------------------

myStr1 = 'AÄBèC'                                        #

myStr2 = 'A\xc4B\xe8C'                                  #

myStr3 = 'A\u00c4B\U000000e8C'                          #

myStr4 = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'        #

import sys, locale
print('Sys hosting platform: ', sys.platform)
print('Sys default encoding: ', sys.getdefaultencoding(
print('Open default encoding:', locale.getpreferredencc

for aStr in (myStr1, myStr2, myStr3, myStr4):
    print(f'{aStr}, strlen={len(aStr)}', end=', ')    #

    bytes1 = aStr.encode()                  # Default UTF-
    bytes2 = aStr.encode('latin-1')         # Explicit Lat
```

```
#bytes3 = aStr.encode('ascii')        # ASCII fails:

print(f'byteslen1={len(bytes1)}, byteslen2={len(byt
```

After saving this file in a text editor with encoding Latin-1 (or its default cp1252 superset on some Windows), running it as a script prints its four strings, their character code-point lengths, and their byte lengths in two encodings that work—the `encode` method's default UTF-8, and an explicit Latin-1 (ASCII is too narrow to use). The Python default encodings it also prints may vary across host platforms, but the rest of the output will not:

```
$ python3 source-encoding-latin1.py
Sys hosting platform:  darwin
Sys default encoding:  utf-8
Open default encoding: UTF-8
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
```

To change this file to use UTF-8 instead, first save it with the UTF-8 encoding in your text editor or by running Python code like the following to make its embedded literal match (you'll learn how and why this code works when we explore Unicode text files ahead):

```
>>> text = open('source-encoding-latin1.py', encoding='
>>> open('source-encoding-utf8.py', 'w', encoding='utf8
```

Then, either mod its first line to name UTF-8, or delete the first line altogether (UTF-8 is Python's default for source code). After you're done, the only difference in the two versions of the source file will be the way the embedded Unicode literal is rendered on your platform; here's the verdict on the UTF-8-centric macOS using its `diff` to compare (use `fc` instead on Windows):

```
$ diff source-encoding-latin1.py source-encoding-utf8.p
1c1
< # -*- coding: Latin-1 -*-
---
> # -*- coding: UTF-8 -*-
13c13
```

```
< myStr1 = 'A?B?C'
---
> myStr1 = 'AÄBèC'

$ python3 source-encoding-utf8.py     # Same output as l
```

Since most programmers are likely to fall back on the default and general (really, *universal*) UTF-8 encoding in Python, we'll defer to Python's standard manual set for more details on this option, as well as its more advanced and obscure Unicode support such as properties and character-name escapes in strings that we'll skip here.

NOTE

Unicode in variable names: Source-file encoding declarations apply to a file's content in general and support arbitrary kinds of text. The rules for variable names within a file's code, however, are more stringent.

As noted briefly in Chapter 11, Python allows some, but not all, non-ASCII Unicode characters to be used for variables in your code. Roughly, number- and letter-like characters work, but symbols and emojis are not allowed. For instance, `hÄck` is a valid variable, but `hÄck`
🐍
is not. You can check whether a specific string passes as a variable with the `isidentifier` method of `str`, but the rules behind this are complex and best had in Python's language manual.

Also, keep in mind that, even when valid, non-ASCIIs in variables may make your code difficult to use on some keyboards and devices outside a given language's locale. In fact, all code in the Python standard library must use ASCII-only identifiers for this reason. As usual, use with care.

Using Byte Strings

We'll be able to see strings in action again when we study files ahead. First, though, let's take a brief side trip to dig a bit deeper into the operation sets provided by objects geared for binary data—the `bytes` type and its `bytearray` mutable kin. While these types can be used to hold encoded text too (as in prior sections), their scope is much broader: anything that can be stored as bytes works, and that's everything digital.

As mentioned earlier, Python's `bytes` type supports sequence operations and most of the same methods available on `str`. Even so, because you cannot mix and match `bytes` and `str` without explicit conversions, you'll generally use `str` objects and text files for text data and `bytes` objects and binary files for binary data. This makes `bytes` a crucial tool in many roles and worthy of a quick demo here.

Methods

If you really want to see what attributes `str` has that `bytes` doesn't, you can always check their `dir` results (review: `set(X)-set(Y)` is items in *X* but not in *Y*). This can also tell you something about the expression operators they support (e.g., `__mod__` and `__rmod__` implement the `%` operator, and they're present in both today):

```
$ python3
Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024,…

# Attributes unique to str

>>> sorted(set(dir('abc')) - set(dir(b'abc')))
['casefold', 'encode', 'format', 'format_map', 'isdecim
'isnumeric', 'isprintable']

# Attributes unique to bytes

>>> sorted(set(dir(b'abc')) - set(dir('abc')))
['__buffer__', '__bytes__', 'decode', 'fromhex', 'hex']
```

As you can see, `str` and `bytes` have almost identical functionality; their unique attributes are generally methods that don't apply to the other (`format` is an outlier you'll meet shortly). For instance, `decode` translates a raw `bytes` into its `str` representation, and `encode` translates a `str` into its raw `bytes` representation. Most other methods are shared between the two types. Moreover, `bytes` are immutable just like `str`:

```
>>> B = b'code'                      # b'…' bytes literal
>>> B.find(b'od')                    # Search for substr
1

>>> B.replace(b'od', b'XY')          # New bytes with rep
```

```
b'cXYe'
>>> B
b'code'

>>> B[0] = 'x'
TypeError: 'bytes' object does not support item assignm
```

For more `bytes` methods, see the earlier coverage of string fundamentals in Chapter 7; `bytes` do most of the same work, though their methods generally return a new `bytes` instead of a `str`.

Sequence Operations

Besides method calls, all the usual generic sequence operations you know from other sequences, like lists, work as expected on both `str` and `bytes`. This includes indexing, slicing, concatenation, and so on. As we've learned, `str` is a sequence of character code points, and `bytes` is a sequence of byte-size integers, but their sequence operations' semantics are the same.

Notice in the following, though, that indexing `bytes` returns an integer giving the byte's binary value; `bytes` really is a sequence of 8-bit integers in the 0…255 range, but when displayed, its components print as either ASCII characters if their values fall into ASCII's 0…127 code-point range, or as hex escapes otherwise:

```
>>> B = b'code'           # bytes and str are both sequ
>>> B                     # Bytes in the 0…127 range di
b'code'

>>> B[0], B[-1]           # Indexing: returns a byte's
(99, 101)
>>> 'code'[0]             # But indexing a str returns
'c'

>>> chr(B[0])             # Unicode character (code poi
'c'
>>> list(B)               # But it's really integer byt
[99, 111, 100, 101]

>>> b'A\x42C\xFF\x63'     # That happen to display as A
b'ABC\xffc'
>>> chr(0x63), hex(B[0])  # And accept hex escapes for
```

```
('c', '0x63')

>>> B[1:], B[:-1]          # Slicing: bytes (that displ
(b'ode', b'cod')
>>> len(B)                 # Length: number bytes (not r
4

>>> B + b'lmn'             # Concatenation: bytes
b'codelmn'
>>> B * 4                  # Repetition: bytes
b'codecodecodecode'
```

Formatting

One notable exception to the same-operations rule for strings: string formatting `%` expressions work on `bytes` too (as of Python 3.5), but neither the `format` method nor `f'…'` f-string formatting is available for `bytes` — an odd bifurcation that seems to forget that formatting comes in multiple flavors today:

```
>>> b'a %s string' % b'fine'                  # Py 3.5+
b'a fine string'
>>> b'a %s string' % bytes([0xFF, 0xFE])      # Non-ASC
b'a \xff\xfe string'

>>> 'a {} string'.format('fine')              # But for
'a fine string'
>>> b'a {} string'.format(b'fine')
AttributeError: 'bytes' object has no attribute 'format

>>> kind = 'fine'
>>> f'a {kind} string'                        # But f-s
'a fine string'
>>> bf'a {kind} string'
SyntaxError: invalid syntax
```

There are arguably sound reasons that formatting shouldn't work on `bytes` —it's just raw bytes, after all, which happens to accept and print ASCII characters as their ASCII code-point values, and may use any encoding if it's text, or none at all if it's not. Plugging ASCII text into an encoded UTF-16

string or loaded image, for instance, makes no sense. But these sound reasons are inconsistently violated for the sake of the `%` expression alone.

Moreover, post-operation conversion by encoding isn't the same as `bytes` operations, and will fail if the default or explicit encoding isn't inclusive enough to handle the text:

```
>>> kind = 'fine'
>>> f'a {kind} string'.encode()                # Encodir
b'a fine string'

>>> kind = 'AÄBèC'                             # And nar
>>> f'a {kind} string'.encode('ascii')
UnicodeEncodeError: 'ascii' codec can't encode characte
```

This inconsistency is prone to change over time, but today, the embarrassment of riches in the formatting department has also given birth to an embarrassment of special cases.

Other Ways to Make Bytes

So far in this section, we've been making `bytes` objects with the `b'…'` literal syntax, but they can also be created by calling the `bytes` constructor with a `str` and an encoding name, by calling `bytes` with an iterable of integers representing byte values, or by encoding a `str` object per the default (or passed-in) encoding. We met some of these earlier in the guise of conversions, but they're more general than previously told.

For example, encoding takes a `str` and returns the raw binary `bytes` value of the string according to its encoding specification; decoding takes a raw `bytes` sequence and encodes it to its string representation—a series of Unicode characters:

```
>>> B = b'abc'
>>> B
b'abc'

>>> B = bytes('abc', 'ascii')
>>> B
b'abc'
```

```
>>> ord('a')
97
>>> B = bytes([97, 98, 99])
>>> B
b'abc'

>>> B = 'code'.encode()       # Or bytes()
>>> B
b'code'

>>> S = B.decode()            # Or str()
>>> S
'code'
```

As we saw earlier, the last two of these operations can also be thought of as tools for converting between `str` and `bytes`, as expanded upon in the next section.

Mixing String Types

When we used the `replace` earlier when sampling `bytes` methods, you may have noticed that we had to pass in two `bytes` objects for from and to — `str` types won't work there. More generally, Python requires specific string types in some contexts and expects manual conversions if needed. Here's the story for function and method calls:

```
>>> B = b'code'

>>> B.replace('od', 'XY')
TypeError: a bytes-like object is required, not 'str'

>>> B.replace(b'od', b'XY')
b'cXYe'

>>> B.replace(bytes('od'), bytes('XY'))
TypeError: string argument without an encoding

>>> B.replace(bytes('od', 'ascii'), bytes('XY', 'utf-8'
b'cXYe'
```

The same holds for mixed-type expressions: you should try to keep your text and binary data separate, but if you must mix, you generally also must convert:

```
>>> b'ab' + 'cd'
TypeError: can't concat str to bytes

>>> b'ab'.decode() + 'cd'                    # bytes to
'abcd'

>>> b'ab' + 'cd'.encode()                    # str to by
b'abcd'

>>> b'ab' + bytes('cd', 'ascii')             # str to by
b'abcd'
```

Two notes here. First, remember that encoding and decoding are more than a simple type conversion; as we learned in the coverage earlier, they create different types of data altogether. Second, although you can create `bytes` objects yourself to represent packed binary data, they can also be made automatically by reading files opened in binary mode, as we will later in this chapter. First, though, let's briefly explore `bytes` ' elusive and changeable colleague.

The bytearray Object

So far in this chapter, we've focused on `str` and `bytes` because they will be your go-to string tools. Python, however, has a third string type—`bytearray` , which is essentially a mutable variant of `bytes` , and thus a mutable sequence of integers in the range 0…255. As such, it supports the same string methods and sequence operations as `bytes` , as well as the mutable in-place-change operations found on lists.

We've already seen most operations that apply to `bytearray` , so we'll just take a quick tour here to sample their flavor. First off, you can call `bytearray` as a function passing a `bytes` (not a `str`) to make a new mutable sequence of small (0…255) integers:

```
>>> B = b'code'                 # A str 'code' does not w
>>> C = bytearray(B)
```

```
>>> C
bytearray(b'code')
>>> C[0], chr(C[0])           # ASCII code-point intege
(99, 'c')
```

Once you've got a `bytearray`, you can change it in place using the same sorts of operations available to modify a list, but keep in mind that you must assign just integers to its cells, not `str` text strings, and not arbitrary objects:

```
>>> C[0] = 'x'
TypeError: 'str' object cannot be interpreted as an int

>>> C[0] = b'x'
TypeError: 'bytes' object cannot be interpreted as an i

>>> C[0] = ord('x')
>>> C
bytearray(b'xode')

>>> C[1] = b'Y'[0]
>>> C
bytearray(b'xYde')
```

The `bytearray`'s methods set overlaps broadly with both `str` and `bytes` because it's a kind of string sequence, but it also has the list object's in-place change methods because it's mutable too (the second of the following means attributes unique to `bytearray`):

```
>>> sorted(set(dir(b'abc')) - set(dir(bytearray(b'abc')
{'__bytes__', '__getnewargs__'}

>>> sorted(set(dir(bytearray(b'abc'))) - set(dir(b'abc'
['__alloc__', '__delitem__', '__iadd__', '__imul__', '_
'__setitem__', 'append', 'clear', 'copy', 'extend', 'ir
```

Hence, it's something of a combo platter—you get methods for in-place changes:

```
>>> C
bytearray(b'xYde')
```

```
>>> C.append(b'LMN')
TypeError: 'bytes' object cannot be interpreted as an i

>>> C.append(b'LMN'[0])
>>> C
bytearray(b'xYdeL')

>>> C.append(ord('M'))
>>> C
bytearray(b'xYdeLM')

>>> C.extend(b'NO')
>>> C
bytearray(b'xYdeLMNO')
```

Plus all the usual sequence operations and string methods:

```
>>> C + b'!#'
bytearray(b'xYdeLMNO!#')

>>> C[0], chr(C[0])
(120, 'x')

>>> C[1:]
bytearray(b'YdeLMNO')

>>> len(C)
8
>>> C
bytearray(b'xYdeLMNO')

>>> C.replace('xY', 'co')
TypeError: a bytes-like object is required, not 'str'

>>> C.replace(b'xY', b'co')
bytearray(b'codeLMNO')

>>> C
bytearray(b'xYamLMNO')

>>> C * 4
bytearray(b'xYdeLMNOxYdeLMNOxYdeLMNOxYdeLMNO')
```

For completeness, this section would be remiss if it didn't call out that you can also make a `bytes` or `bytearray` by passing in any sort of integer sequence or generator, not just text strings. This also provides a sort of conversion from an integer character code to a `bytes`, but only if the integer is in the range 0…255, and only if it's embedded in a sequence (else you get a `bytes` with that many zeroes—surprisingly!). See Python's manuals for more esoteric bits that we don't have space to cover here:

```
>>> bytes([1, 2, 3, 4]), bytearray([1, 2, 3, 4])
(b'\x01\x02\x03\x04', bytearray(b'\x01\x02\x03\x04'))

>>> bytes([115])        # int => bytes
b's'

>>> bytes([115])[0]     # bytes => int
115

>>> bytes([999])        # Too big for a byte
ValueError: bytes must be in range(0, 256)

>>> bytes(115)          # Lots of zeroes!
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\

>>> bytes(range(97, 104)), bytes(c + 97 for c in range(
(b'abcdefg', b'abcdefg')
```

Finally, by way of summary, the following examples demonstrate how `bytes` and `bytearray` are sequences of integers, and `str` is a sequence of characters (i.e., decoded Unicode code points); although all three can contain textual content and support many of the same operations, you should use `str` for decoded text, `bytes` for binary data including encoded text, and `bytearray` for binary data you wish to change in place to avoid the time and space overheads of generating copies for each change you make:

```
>>> B = b'code'                    # Bytes
>>> list(B)
[99, 111, 100, 101]

>>> C = bytearray(b'code')         # Changeable bytes
>>> list(C)
[99, 111, 100, 101]
```

```
>>> S = 'code'                          # Unicode text
>>> list(S)
['c', 'o', 'd', 'e']
```

Using Text and Binary Files

Now that we've learned all about Python's string types, let's return to their roles in files—the main context in which most programmers will likely encounter Unicode and bytes.

As mentioned earlier, the *mode* in which you open a file is crucial in Python: it determines both how the file's content is interpreted as well as the object type you will use to process that content in your script. By way of review, text mode implies `str` objects and binary mode implies `bytes`, as follows:

Text-mode files

Interpret file contents according to an encoding—either the default for your platform or one whose name you pass in to `open`. By passing in an encoding name, you can force conversions for various types of Unicode files. Text-mode files may also handle BOM headers for some encodings (deferred till the end of this chapter) and may perform universal newline translations for you or not; by default, all newline forms map to the `\n` character in your script, regardless of which platform you are on.

Binary-mode files

Instead return file content to you raw as a sequence of integers representing byte values, with no encoding or decoding, no BOM handling, and no newline translations.

In terms of code, the second positional argument to `open` (a.k.a. `mode` when passed by keyword) determines whether you want text or binary processing and types—adding a `b` to the mode string implies binary mode. The default mode is `rt`, which is the same as `r`, and means text input. In addition, the mode argument to `open` also implies an object type for file content representation regardless of the underlying platform—text files return a `str` for reads and expect one for writes, but binary files return a `bytes` for reads and expect `bytes` (or `bytearray`) for writes.

Text-File Basics

To demonstrate, let's review basic file I/O. As long as you're processing simple text files that adhere to your platform's default encoding, files look and feel much as they do in this book's earlier coverage (for that matter, so do strings in general). The next example, for instance, writes one line of text to a file and reads it back:

```
>>> file = open('temp.txt', 'w')      # Use default encc
>>> size = file.write('abc\n')        # Returns number c
>>> file.close()                      # Manual close to

>>> file = open('temp.txt')           # Default mode is
>>> text = file.read()
>>> text
'abc\n'
```

As a refresher, the first argument to `open` is the file's *pathname*—the address of a file in the host's folder hierarchy that's either absolute or relative to the current directory. The second argument to open is *mode*—where `w` means write text, and the default means read it, and `write` methods return the number of written *items*—either characters for text mode or bytes for binary mode. Also, the `close` call here is optional in some contexts (e.g., the widely used *CPython* auto-closes files when their objects are garbage collected) but is generally advised to flush changes and avoid memory growth.[1]

Technically, the preceding example writes and reads Unicode text, but it's hardly noticeable: the ASCII text string is encoded and decoded per the hosting platform's encoding default. We're also relying on platform-agnostic newline handling for `\n`, but we must move ahead for more on encodings and newlines.

Text and Binary Modes

Next, let's write a *text file* and read it back in both text and binary modes. Notice in the following how text mode requires us to provide a `str` for writing, `rb` distinguishes binary-mode input, and reading gives us a `str` or `bytes` depending on the mode (opens and transfer operations are strung

together here into one-liners just for brevity; again, remember to `close` explicitly in production code, and possibly in some IDEs and outside CPython):

```
$ py -3                                    # Run on W
>>> open('temp.txt', 'w').write('abc\n')   # Text-mod
4

>>> open('temp.txt', 'r').read()           # Text-mod
'abc\n'

>>> open('temp.txt', 'rb').read()          # Binary-m
b'abc\r\n'
```

Observe how the *newline character* is always `\n` when writing and reading in text mode with `str` but `\r\n` when reading in binary mode with `bytes`. This reflects the fact that this was run on Windows. Though it has nothing to do with Unicode, text-mode files automatically map all `\n` in `str` to and from the host platform's newline separator: `\r\n` on Windows and just `\n` on Unix. When reading in binary mode, though, we get what's actually in the file—with neither newline mapping nor Unicode decoding.

Now, let's do the same, but with a *binary file*. We provide a `bytes` to write and still get back a `str` or `bytes` depending on the input mode, though the `\n` isn't expanded to `\r\n` on Windows this time:

```
>>> open('temp.bin', 'wb').write(b'abc\n')  # Binary-m
4

>>> open('temp.bin', 'r').read()           # Text-mod
'abc\n'

>>> open('temp.bin', 'rb').read()          # Binary-m
b'abc\n'
```

This holds true even if the data we're writing to the binary file is truly binary in nature. In the following, the `\x00` is a binary zero byte and not a printable character, though it works in the middle of a `bytes` and qualifies as a text code point in the default encoding (strictly speaking, zero is a character called null or NUL in ASCII and its supersets like UTF-8):

```
>>> open('temp.bin', 'wb').write(b'a\x00c')     # Binary
3

>>> open('temp.bin', 'r').read()               # Text-m
'a\x00c'

>>> open('temp.bin', 'rb').read()              # Binary
b'a\x00c'
```

Binary mode files always return contents as a `bytes` object but accept either a `bytes` or `bytearray` object for writing. This naturally follows, given that `bytearray` is mostly just a mutable variant of `bytes`. In fact, most APIs in Python that accept a `bytes` also allow a `bytearray` (the bytes here are also ASCII characters too obscure for this note):

```
>>> BA = bytearray(b'\x01\x02\x03')
>>> open('temp.bin', 'wb').write(BA)
3

>>> open('temp.bin', 'r').read()
'\x01\x02\x03'

>>> open('temp.bin', 'rb').read()
b'\x01\x02\x03'
```

Finally, notice that you can't get away with violating Python's `str` / `bytes` (i.e., text/binary) distinction when it comes to files; in the following, we get errors if we try to write a `bytes` to a text file or a `str` to a binary file. Remember, although it is often possible to convert between these two types (as described earlier in this chapter), you will usually want to stick to `str` for text data and `bytes` for binary data:

```
>>> open('temp.txt', 'w').write('abc\n')           # A
4
>>> open('temp.txt', 'w').write(b'abc\n')          # B
TypeError: write() argument must be str, not bytes

>>> open('temp.bin', 'wb').write(b'abc\n')         # W
4
```

```
>>> open('temp.bin', 'wb').write('abc\n')           # E
TypeError: a bytes-like object is required, not 'str'
```

This may seem strict, but Python cannot guess how you wish to interpret the contents of a `bytes` or `str` when used in the opposite context and wisely refuses to convert implicitly (a `bytes` might be an image, after all). Moreover, because `str` and `bytes` operation sets largely intersect, the choice of types won't be much of a dilemma for most programs. Watch for the `struct` module coverage ahead for another binary-file example.

Unicode-Text Files

And now for the featured attraction of our files tour: text files with non-ASCII text. Beyond their text/binary distinction, Python files come with additional requirements and tools for dealing with Unicode text. In short, text files allow a specific Unicode encoding-scheme name to be passed in with an `encoding` argument to `open` and use it to automatically *decode* and *encode* text on input and output, respectively. As abstract examples, for a file identified by a `pathname` string:

```
open(pathname, 'r', encoding='utf8')
```

Returns a file object that decodes text from UTF-8 on reads

```
open(pathname, 'w', encoding='latin1')
```

Returns a file object that encodes text to Latin-1 on writes

The file object returned by the first of the preceding assumes the file's content is encoded per UTF-8 and automatically decodes it to `str` Unicode code points when read by the program. Similarly, the result of the second line of code encodes `str` code points to their Latin-1 format as they are output to the file. Here's the text-file story with the universal *UTF-8* encoding and three non-ASCII characters in the content:

```
>>> file = open('uni.txt', 'w', encoding='utf8')
>>> file.write('
💛
 2 hÄck
🐍
')
10
>>> file.close()
```

```
>>> text = open('uni.txt', 'r', encoding='utf8').read()
>>> text
'
💛
 2 hÄck
🐍
'

>>> [ord(c) for c in text]
[128155, 32, 50, 32, 104, 196, 99, 107, 32, 128013]

>>> raw = open('uni.txt', 'rb').read()
>>> raw
b'\xf0\x9f\x92\x9b 2 h\xc3\x84ck \xf0\x9f\x90\x8d'
```

File transfers raise exceptions whenever a requested encoding doesn't work, so the encoding you pass must match the data. For example, *ASCII* is not inclusive enough to handle the augmented and emoji characters in the text we're writing here and fails on both reads and writes:

```
>>> open('uni.txt', 'r', encoding='ascii').read()
UnicodeDecodeError: 'ascii' codec can't decode byte 0xf

>>> open('ascii.txt', 'w', encoding='ascii').write('
💛
 2 hÄck
🐍
')
UnicodeEncodeError: 'ascii' codec can't encode characte

>>> hex(ord('
💛
'))    # ASCII will always break your heart?
'0x1f49b'
```

By contrast, using the broader and more general *UTF-16* on both ends handles this text in full, though its encoded bytes stored in the file naturally differ from those of UTF-8:

```
>>> file = open('uni2.txt', 'w', encoding='utf16')
>>> file.write('
💛
 2 hÄck
🐍
')
```

```
10
>>> file.close()

>>> text = open('uni2.txt', 'r', encoding='utf16').reac
>>> text
'
💛
 2 hÄck
🐍
'


>>> [ord(c) for c in text]
[128155, 32, 50, 32, 104, 196, 99, 107, 32, 128013]

>>> open('uni2.txt', 'rb').read()
b'\xff\xfe=\xd8\x9b\xdc \x002\x00 \x00h\x00\xc4\x00c\x0
```

Even for broader encodings like UTF-8 and UTF-16, though, you cannot *mix and match*: using an incompatible encoding still won't work because encoded bytes in the file differ. Although you can sometimes handle unknown encodings with binary-mode files, `open` error handlers, and other techniques, your encoding must generally match your file's data:

```
>>> open('uni.txt', 'r', encoding='utf16').read()
UnicodeDecodeError: 'utf-16-le' codec can't decode byte

>>> open('uni2.txt', 'r', encoding='utf8').read()
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf
```

In the *absence* of an `encoding` argument, text files still encode and decode per a host- and platform-specific default in `locale` discussed earlier. But as a reminder: although you may not notice these translations if your default and files agree, you generally should not rely on the default; it makes your programs dependent on the context in which their files were created and can lead to portability issues (and nightmares; see the upcoming sidebar "Unicode Defaults and UTF-8 Mode").

For instance, a program run on a UTF-8 default platform (the macOS and Android norm) may have trouble using a file made under a cp1252 default (the ASCII-superset default on some Windows hosts) and vice versa, and content fetched from other devices may be encoded arbitrarily. Use explicit `open` encodings as a rule. This may also help if environment settings are not

reliable (e.g., when running as a generic user for security in server-side web scripts).

All that being said, you'll probably find files to be easier in practice than the full details may suggest. Accessing web pages and images, for example, soon becomes second nature and simple. For variety, the following first omits mode (again, its default is the same as `r`) and then passes mode by explicit keyword (instead of position), and this example is abstract (substitute pathnames of real and accessible files on your device to run live):

```
>>> text = open('Websites/about-lp5e.html', encoding='u
>>> text[:79]
'<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitic

>>> image = open('Websites/lp5e-large.jpg', mode='rb').
>>> image[:20]
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x00\x00\x01\
```

Once you've loaded content this way, all the `str` and `bytes` operations we've seen in this chapter are at your disposal for wrangling text and binary data, and saving the result follows the same pattern—simply use an encoding for text and binary mode for bytes:

```
>>> text = text.replace('2013', '2024')
>>> open('Websites/about-lp6e.html', mode='w', encoding

>>> image = some_sort_of_modding(image)
>>> open('Websites/lp6e-large.jpg', mode='wb').write(im
```

You might mod images, for example, with the many tools provided by the third-party *Pillow* (f.k.a. PIL) imaging library for Python or similar. With Python's files and strings, content possibilities are largely endless.

In the interest of full disclosure, Python's `open` accepts additional arguments that modify its behavior. Among them, `newline` changes newline mapping; `errors` specifies handling of encoding and decoding errors (e.g., `'surrogateescape'` replaces failing bytes with sequences that can be used to restore them on output); and `buffering` alters, well, buffering. Because their defaults are generally what you'll use, we'll defer to the Python

standard-library manual for the fine print on these and other advanced file options.

NOTE

Blast from the past: Python's `codecs` module also has an `open` that returns an auto-decode/encode file object just like the built-in `open` function. This was used for Python 2.X backward compatibility in times gone by, but there's no obvious reason to rely on it in new code. Still, you might see it in legacy code (and code that relies on any of its unique behaviors) anyhow:

```
>>> import codecs
>>> f = codecs.open('uni.txt', 'r', encoding='utf8')
>>> f.read()
'
💛
 2 hÄck
🐍
'
```

Unicode, Bytes, and Other String Tools

Besides built-in strings and files, many of the popular string-processing tools in Python's standard library installed with Python itself also adhere to the `str` / `bytes` dichotomy. We won't cover any of these application-focused topics in detail in this core-language book, but as a sample, here's a *very* brief look at how some of these tools handle the split. See Python's Library Reference for more on the tools used here if any pique your interest.

The re Pattern-Matching Module

Python's `re` pattern-matching module has been generalized to work on objects of any string type— `str`, `bytes`, and `bytearray` (a `(.*)` means any run, saved as a group):

```
>>> import re
>>> S = '
🐍
 is the fastest way to

!'
>>> B = b'Python is the fastest way to pizza!'
```

```
>>> re.match('(.*) the (.*) way (.*)', S).groups()
('
 is', 'fastest', 'to

!')

>>> re.match(b'(.*) the (.*) way (.*)', B).groups()
(b'Python is', b'fastest', b'to pizza!')

>>> re.match(b'(.*) the (.*) way (.*)', bytearray(B)).g
(b'Python is', b'fastest', b'to pizza!')
```

Like many tools, though, its result types depend on the type of strings you pass in, and you can't mix `str` and `bytes` types in its calls' arguments (sans explicit conversions, of course); `bytearray` is also unusable here as a pattern because it's mutable (and changeable means "unhashable"):

```
>>> re.match('(.*) the (.*) way (.*)', B).groups()
TypeError: cannot use a string pattern on a bytes-like
>>> re.match(b'(.*) the (.*) way (.*)', S).groups()
TypeError: cannot use a bytes pattern on a string-like

>>> re.match('(.*) the (.*) way (.*)', bytearray(B)).gr
TypeError: cannot use a string pattern on a bytes-like
>>> re.match(bytearray(b'(.*) the (.*) way (.*)'), byte
TypeError: unhashable type: 'bytearray'
```

The struct Binary-Data Module

The Python `struct` module, used to create and extract packed binary data from strings, operates on `bytes` and `bytearray` only, not `str` —which makes sense, given that it's intended for processing binary data, not decoded text. This module uses a format string to specify the types and sizes of objects to pack to and from a bytes string, along with an endianness (i.e., significant side of bitstrings) specifier like `>` for big-endian:

```
>>> import struct
>>> B = struct.pack('>i4sh', 7, b'code', 8)    # int ,
>>> B
b'\x00\x00\x00\x07code\x00\x08'
```

```
>>> vals = struct.unpack('>i4sh', B)           # Packed
>>> vals
(7, b'code', 8)

>>> vals = struct.unpack('>i4sh', B.decode())
TypeError: a bytes-like object is required, not 'str'
```

You'll often use this in conjunction with binary-mode files to make the packed bytes persistent across program runs (e.g., for saving and restoring app user settings):

```
>>> F = open('data.bin', 'wb')                 # Ope
>>> data = struct.pack('>i4sh', 7, b'code', 8)  # Cre
>>> data                                       # byt
b'\x00\x00\x00\x07code\x00\x08'
>>> F.write(data)                              # Wri
10
>>> F.close()                                  # Flu

>>> F = open('data.bin', 'rb')                 # Ope
>>> data = F.read()                            # Rea
>>> data
b'\x00\x00\x00\x07code\x00\x08'
>>> values = struct.unpack('>i4sh', data)      # Ext
>>> values                                     # Bac
(7, b'code', 8)
```

The pickle and json Serialization Modules

Python's `pickle` module provides another way to save data in files but allows saved data to be nearly arbitrary Python objects instead of values coerced to bytes as in `struct`. We met this module briefly in Chapters 9, 28, and 31. For completeness here, keep in mind that the `pickle` module always creates a `bytes` object, regardless of the default or passed-in `protocol` (data format level). You can see this by using the module's `dumps` call to return an object's pickle string:

```
>>> import pickle                              # dump
>>> pickle.dumps(['code', 4, '
🐍
'])              # Default protocol=binary
```

```
b'\x80\x04\x95\x15\x00\x00\x00\x00 …etc… \x04\x8c\x04\x
```

```
>>> pickle.dumps(['code', 4, '
🐍
'], protocol=0)   # ASCII protocol 0, still bytes!
b'(lp0\nVcode\np1\naI4\naV\\U0001f40d\np2\na.'
```

This implies that files used to store pickled objects must always be opened in *binary mode* because text files use `str` strings to represent data, not `bytes`, and the `dump` call simply attempts to write the pickled byte string to an open output file:

```
>>> pickle.dump(['code', 4, '
🐍
'], open('temp.pkl', 'w'))   # bytes+text mode fail
TypeError: write() argument must be str, not bytes

>>> pickle.dump(['code', 4, '
🐍
'], open('temp.pkl', 'w'), protocol=0)
TypeError: write() argument must be str, not bytes

>>> pickle.dump(['code', 4, '
🐍
'], open('temp.pkl', 'wb'))  # Always binary mode

>>> open('temp.pkl', 'r').read()
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x8
```

Notice the last result fails here in text mode on macOS; if it doesn't fail for you, it's only because the stored binary data is compatible with your platform's default decoding (i.e., just by luck!). A Windows host, for example, prints gibberish for the last result instead of an error message. Because pickle data is not generally decodable Unicode text, the same rule holds on input as output—correct usage always requires both writing and reading pickle data with binary-mode files, whether unpickling or not:

```
>>> pickle.dump(['code', 4, '
🐍
'], open('temp.pkl', 'wb'))    # Save to file

>>> pickle.load(open('temp.pkl', 'rb'))
['code', 4, '
🐍
```

```
']

>>> open('temp.pkl', 'rb').read()
b'\x80\x04\x95\x15\x00\x00\x00\x00 …etc… \x04\x8c\x04\x
```

The Python `json` module, introduced in Chapter 9, is a related tool: it converts a nested-object tree to a text string that can be saved on a file to make it persistent. Unlike `pickle`, `json` doesn't support arbitrary objects (just basic types and built-in containers) but uses a language-neutral scheme that can make it more interoperable with other programs.

Also unlike `pickle`, the `json` module always produces a `str`, so files for saves and loads should generally use *text mode* (JSON files are commonly encoded per UTF-8 for portability, but any encoding works as long as it's consistent and expected):

```
>>> import json
>>> vals = ['code', {'app': ('
🙂
', None, 1.23, 99)}]
>>> json.dumps(vals)
'["code", {"app": ["\\ud83d\\ude42", null, 1.23, 99]}]'

>>> text = json.dumps(vals)                          #
>>> anew = json.loads(text)
>>> anew
['code', {'app': ['
🙂
', None, 1.23, 99]}]

>>> file = open('data.txt', 'w', encoding='utf8')    #
>>> json.dump(vals, file)
>>> file.close()
>>> file = open('data.txt', 'r', encoding='utf8')
>>> json.load(file)
['code', {'app': ['
🙂
', None, 1.23, 99]}]
```

Filenames in open and Other Filename Tools

So far in this chapter, we've focused on the *content* of files, but their *names* have a Unicode story too. Its short version is that `str` is the norm for file

pathnames in Python and is recommended for portability. If your code, like all the examples so far, uses `str` for filenames and pathnames, they simply work: Python's file tools automatically translate them to and from the encoding used by the host platform and device.

It turns out, though, that Python's file tools also allow you to specify file pathnames as `bytes` to skip automatic filesystem encoding in some contexts. This `bytes` mode may come in handy if you need more control over filename encodings in cross-platform code, but it is rarely needed in typical programs. Per Python's manuals, in fact, this mode need be used only on some Unix systems where undecodable filenames may be present.

Nevertheless, `bytes` filenames do work in these tools, as shown in the following demo, which runs the same on macOS, Windows, Linux, and Android devices tested. As shown, the `open` function happily accepts text or bytes for a file's non-ASCII pathname (this section's examples were run in subfolder *_filenames* in the book's examples package):

```
>>> import sys, os, glob
>>> sys.getfilesystemencoding()          # Filesystem c
'utf-8'

>>> name1 = 'hÄck
🙂
1'                      # str filenames
>>> name2 = 'hÄck
🙂
2'
>>> name2.encode('utf8')                 # bytes equivc
b'h\xc3\x84ck\xf0\x9f\x99\x822'

>>> os.listdir()                         # Make files v
[]
>>> open(name1, mode='w', encoding='utf8').write('text1
5
>>> open(b'h\xc3\x84ck\xf0\x9f\x99\x822', 'w', encoding
5
>>> os.listdir()                         # Both show up
['hÄck
🙂
1', 'hÄck
🙂
2']
```

```
>>> open('hÄck
🙂
2').read()                # And can be accessed either w
'text2'
>>> open('hÄck
🙂
2'.encode('utf8')).read()
'text2'
```

For `bytes` filenames, Python uses UTF-8 encoding on macOS and on Windows since 3.6 (with extra translation for the Windows API's UTF-16). Linux, and by extension Android, accepts any sort of bytes for filenames, but Python tries to use UTF-8 when converting to and from `str` (with surrogate escapes for encoding errors: see Python's manuals).

Whether `bytes` filenames are a useful tool or neat parlor trick depends on your use cases, but keep in mind that `bytes` filenames will only work in `open` if their encoding matches the expectations of Python or the underlying filesystem. To demo, the following passes Latin-1 bytes `b'h\xc4ck'` to `open` : this *fails* on both macOS and Windows as shown (though with a `UnicodeDecodeError` on the latter because it's trapped by Python):

```
>>> 'hÄck'.encode('latin-1'), 'hÄck'.encode('utf-8')
(b'h\xc4ck', b'h\xc3\x84ck')

>>> b'h\xc4ck'.decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc

>>> open(b'h\xc4ck', 'w')
OSError: [Errno 92] Illegal byte sequence: b'h\xc4ck'

# open(b'h\xc4ck'.decode('latin1'), 'w')    # Works: cc
# open(b'h\xc3\x84ck', 'w')                 # Works: us
```

By contrast, the preceding code's `open` *works* on Linux and Android—which later decode the Latin-1 name `b'h\xc4ck'` to `str` by UTF-8, as surrogate-escaped `'h\udcc4ck'` . Still, this works only on drives using a Linux-native *filesystem*. For both its `bytes` and escaped `str` forms, the Latin-1 name fails in `open` on Linux and Android when the target is a removable drive formatted as exFAT. Hence, the effect of using arbitrary `bytes` in `open` varies by both platform and filesystem (that is, don't do that!).

The `os` standard-library module's directory-listing `listdir` similarly accepts `str` or `bytes` and returns a folder's names in the same form, ready to be used in other file tools (per the earlier listing, it also defaults to the current directory if no folder is passed):

```
>>> os.listdir('.')                         # str gives st
['hÄck
🙂
1', 'hÄck
🙂
2']

>>> os.listdir(b'.')
[b'h\xc3\x84ck\xf0\x9f\x99\x821', b'h\xc3\x84ck\xf0\x9f

>>> for name in os.listdir('.'):
        print(name, '=>', open(name).read())

hÄck
🙂
1 => text1
hÄck
🙂
2 => text2

>>> for name in os.listdir(b'.'):
        print(name, '=>', open(name).read())

b'h\xc3\x84ck\xf0\x9f\x99\x821' => text1
b'h\xc3\x84ck\xf0\x9f\x99\x822' => text2
```

The `glob` module does filename expansion both ways, too (as `str` or `bytes`), and `os.fsdecode` decodes filenames per the host's default automatically:

```
>>> glob.glob('h*ck*')                      # str gives st
['hÄck
🙂
1', 'hÄck
🙂
2']

>>> glob.glob(b'h*ck*')
[b'h\xc3\x84ck\xf0\x9f\x99\x821', b'h\xc3\x84ck\xf0\x9f
```

```
>>> for name in glob.glob(b'h*ck*'):
        print(name.decode(sys.getfilesystemencoding()),

hÄck
🙂
1 hÄck
🙂
1
hÄck
🙂
2 hÄck
🙂
2
```

In addition, tools that create and walk folders are similarly flexible for names and paths (demoed on Windows—your path separators and order may vary):

```
>>> os.mkdir('sub
🐵
')                          # Make dirs with str or byte
>>> os.mkdir('sub
🐵
bytes'.encode('utf8'))      # and walk them with both
>>> os.mkdir('sub
🐵
bytes/subsub
👍
')

>>> os.listdir()
['hÄck
🙂
1', 'hÄck
🙂
2', 'sub
🐵
', 'sub
🐵
bytes']
>>> os.listdir('sub
🐵
bytes')
['subsub
👍
']

>>> for (dirhere, subshere, fileshere) in os.walk('.'):
        print(dirhere, '=>', subshere, fileshere)

. => ['sub
🐵
```

```
', 'sub
🐵
bytes'] ['hÄck
🙂
1', 'hÄck
🙂
2']
.\sub
🐵
 => [] []
.\sub
🐵
bytes => ['subsub
👍
'] []
.\sub
🐵
bytes\subsub
👍
 => [] []
>>>
>>> for (dirhere, subshere, fileshere) in os.walk(b'.')
        print(dirhere, '=>', subshere, fileshere)

b'.' => [b'sub\xf0\x9f\x99\x8a', b'sub\xf0\x9f\x99\x8ab
b'.\\sub\xf0\x9f\x99\x8a' => [] []
b'.\\sub\xf0\x9f\x99\x8abytes' => [b'subsub\xf0\x9f\x91
b'.\\sub\xf0\x9f\x99\x8abytes\\subsub\xf0\x9f\x91\x8d'
```

While `bytes` pathnames work as shown and may have some valid roles, it's important to stress that you're almost always better off using `str` strings to name files instead. Doing so leverages tools that already go to great lengths to make pathnames do the right thing and might just avoid at least some portability issues that can arise in apps whose scope must span platforms and devices.

And that's all the time and space we have for this tools survey. Really, Python's standard library and third-party domain are large and evolving toolsets that will likely occupy much of your attention after you've finished this book and move on to real programming tasks. Again, be sure to consult the Python Library Reference soon and often for more info.

UNICODE DEFAULTS AND UTF-8 MODE

So what's the default encoding, then? This turns out to be a weirdly convoluted story, which we've touched on lightly a few times. Now that we've seen all the players in action, we can finally summarize and finalize this thread.

In short, Python's default Unicode encoding for both *source code* and string encoding/decoding *methods* is always UTF-8 everywhere; for *filenames* can be had with `sys.getfilesystemencoding()`; and for *file content* accessed via `open` is fetched with `locale.getpreferred encoding(False)`.

The *methods* default is `sys.getdefaultencoding()`, but it can no longer be changed as of Python 3.2 (and probably shouldn't have been changed earlier). The *filenames* default is usually UTF-8, including on Windows as of Python 3.6, but is moot if your names are always `str` (as we learned in the preceding section).

The `open` default for *file content* is bifurcated most and badly. Contrary to Python's current docs, it's not simply the result of `locale.getencoding()` on all platforms. Rather, it's either UTF-8 if the *UTF-8 mode* introduced in Python 3.7 is enabled or else `getencoding()`. This matters on Windows today, where `getencoding()` may be a code-page encoding like cp1252; because Unix is usually UTF-8, an `open` sans `encoding` isn't portable for non-ASCII content.

UTF-8 mode addresses this, but it must be enabled today by setting the environment variable `PYTHONUTF8` to `1` or using command-line switch `-X utf8`. The result of `getencoding()`, added in Python 3.11, is not influenced by UTF-8 mode the way that `open` and the older `getpreferredencoding(False)` in `locale` are. Hence, `open`'s conditional default.

Both of the `locale` encoding results may also be influenced by platform, environment variables, and command-line switches…*except* on Android, which is just plain UTF-8, and on Windows in the future, when *Python 3.15* will turn UTF-8 mode on by default to match other platforms for code that lacks explicit encodings—a good idea, though too late to the party to avoid making a scene.

Separately, environment variable `PYTHONIOENCODING` can be used to give the encoding of *stdio streams* (e.g., `sys.stdout`) when they are redirected to files on Windows and others (e.g., `> output.txt`)…*except* when you or a Python of the future enable 3.7's UTF-8 mode, which applies to redirected streams too…*unless* variable `PYTHONIOENCODING` is also set.

Beyond all this, the encoding fate of unredirected console streams on Windows can be sealed with `PYTHONLEGACYWINDOWSSTDIO`; filename encodings on Windows may also backslide to their code-page roots for the brazenly grandiose `PYTHONLEGACYWINDOWSFSENCODING`; Unix encodings can be forced to skip UTF-8 similarly with `PYTHONCOERCECLOCALE`; and UTF-8 mode modulates additional textual tools omitted here for space (and humanity).

Right. If you don't care to remember all that—and prefer to sleep well at night—enable UTF-8 mode on Windows today before Python 3.15 does; use `str` for filenames and pass explicit encodings to `open` instead of relying on tangled and morphing defaults; and see Python's manuals for the full, if frightening, story.

The Unicode Twilight Zone

To wrap up, this chapter is going to briefly present two curated topics from the Unicode realm—BOMs and normalization—that are too convoluted for full coverage in this book and probably too arcane to matter to most Python newcomers. If and when these become important to your projects, you'll find ample resources in both Python's docs and the web at large to fill in the bits glossed over here for space. For now, let's jump right into the first of these arguably explosive topics.

Dropping the BOM in Python

As noted briefly earlier in this chapter, some encoding schemes store a special *byte order marker* (BOM) sequence at the start of files to specify data endianness (which end of a string of bits is most significant to its value) or declare the encoding type in general. Python's text-mode files both skip this marker on input and write it on output if the encoding implies presence, but we sometimes must use a specific encoding name to force BOM processing explicitly and may need to accommodate it when encoding manually.

For example, in the UTF-16 and UTF-32 encodings, the BOM both identifies the encoding and specifies big- or little-endian format. A UTF-8 text file may also include a BOM, but this isn't guaranteed and serves only to declare that it is UTF-8 in general.

When reading and writing data using these encoding schemes, Python automatically skips or writes the BOM if it is either implied by a general encoding name or if you provide a more specific encoding name to force the issue. More concretely:

- In *UTF-16*, the BOM is always processed for encoding name `utf-16`, and the more specific encoding name `utf-16-le` denotes little-endian format.
- In *UTF-8*, the more specific encoding name `utf-8-sig` forces Python to both skip and write a BOM on input and output, respectively, but the general `utf-8` does not.

Making BOMs in Text Editors

Let's make some files with BOMs to see how this works in practice. We're going to do this in Python in a moment, too, but let's start by creating a file in a text editor to underscore that your programs will also have to process content from other sources. If you wish to work along, you can use any text editor that's Unicode aware for saves, and most editors on PCs, tablets, and phones are today. This demo uses Windows Notepad just because it's widespread (and documenting multiple editors' usage here is right out).

When you save a text file in Windows Notepad, you can specify its encoding type in a drop-down list—simple text, little- or big-endian UTF-16, or UTF-8 with or without a BOM. For instance, if you use Notepad to save the two lines "code" and "CODE" in a text file named *code.txt* with encoding type *ANSI*, the file is written as simple ASCII text without a BOM. Technically, the save uses the cp1252 default encoding on the Windows host used, but cp1252 is an ASCII superset, and the content is all ASCII.

In Python after the save, when this file is read in binary mode, we can see the actual bytes stored in the file, including `\r` in newlines. When it's read as text, Python performs newline translation by default, and we can also decode it explicitly as UTF-8 text since ASCII is a subset of both this encoding and Windows' default cp1252:

```
$ py -3                                  # On Windows,
>>> import locale
>>> locale.getpreferredencoding(False)   # open defaul
'cp1252'
>>> open('code.txt', 'rb').read()        # ASCII (and
b'code\r\nCODE\r\n'
>>> open('code.txt', 'r').read()         # Text mode d
'code\nCODE\n'
>>> open('code.txt', 'r', encoding='utf-8').read()
'code\nCODE\n'
```

Now, if this file is instead saved as *UTF-8 with BOM* in Notepad (*UTF* in its prior versions), its text is prepended with a three-byte UTF-8 BOM sequence (which prints as non-ASCII `\x`*`NN`* hex escapes in `bytes` in Python), and we need to give the more specific encoding name `utf-8-sig` to force Python to skip the marker automatically on input (else it prints as a `\u`*`NNNN`* Unicode code-point escapes in `str`):

```
>>> open('code.txt', 'rb').read()       # After resaved
b'\xef\xbb\xbfcode\r\nCODE\r\n'
>>> open('code.txt', 'r').read()        # Default+utf8
'ï»¿code\nCODE\n'
>>> open('code.txt', 'r', encoding='utf-8').read()
'\ufeffcode\nCODE\n'
>>> open('code.txt', 'r', encoding='utf-8-sig').read()
'code\nCODE\n'
```

And if the file is stored as *UTF-16 BE* in Notepad (*Unicode big endian* formerly), we get UTF-16 big-endian format data in the file, with two bytes (16 bits) prepended to record a two-byte BOM sequence. The encoding name `utf-16` in Python skips the BOM because it is implied (since all UTF-16 files have a BOM), and `utf-16-be` handles the big-endian format but does not skip the BOM in the input result:

```
>>> open('code.txt', 'rb').read()
b'\xfe\xff\x00c\x00o\x00d\x00e\x00\r\x00\n\x00C\x00O\x0

>>> open('code.txt', 'r', encoding='utf-16').read()
'code\nCODE\n'
```

```
>>> open('code.txt', 'r', encoding='utf-16-be').read()
'\ufeffcode\nCODE\n'
```

Experiment with other save/open combinations for more insights. The default encoding in the last example, for example, would print garbage characters because it's not valid for the text, and UTF-16 little-endian swaps byte order for the BOM and each 2-byte character.

NOTE

Notepad flux: Notepad recently changed its encoding options for saves. Today, it offers ANSI (the host's default), UTF-16 in little- and big-endian flavors, and UTF-8 with and without a BOM, and defaults to UTF-8. When this book's prior edition was penned, Notepad had UTF-8 with an implied BOM, and its "Unicode" meant UTF-16—which, of course, is just one of the very many kinds of Unicode encoding. The narrative here has been updated to reflect the new choices, but this naturally is just the story today. Because this book staunchly refuses to become a Notepad doc, translate save options as needed to the Notepad on a PC near you.

Making BOMs in Python

The preceding section used Python to read files made in an editor, but the same patterns apply when Python also makes the files: Python will automatically add a BOM when we write a Unicode file in code, but we need to use a more explicit encoding name to force the BOM in UTF-8 mode—`utf-8` does not write the BOM on output but `utf-8-sig` does, and the same goes for skipping UTF-8 BOMs on input. When run on Windows (Unix is the same, but doesn't add `\r` to newlines, and usually defaults to UTF-8):

```
>>> open('temp.txt', 'w', encoding='utf-8').write('code
10
>>> open('temp.txt', 'rb').read()
b'code\r\nCODE\r\n'

>>> open('temp.txt', 'w', encoding='utf-8-sig').write('
10
>>> open('temp.txt', 'rb').read()
b'\xef\xbb\xbfcode\r\nCODE\r\n'

>>> open('temp.txt', 'r').read()
'ï»¿code\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()
```

```
'\ufeffcode\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()
'code\nCODE\n'
```

Per this code, although `utf-8` does not drop the BOM when one is present, data *without* a BOM can be read with both `utf-8` and `utf-8-sig` —which means you can use the latter for input if you're not sure whether a BOM is present in a file or not (and don't read this paragraph out loud in an airport security line!):

```
>>> open('temp.txt', 'w').write('code\nCODE\n')
10
>>> open('temp.txt', 'rb').read()
b'code\r\nCODE\r\n'

>>> open('temp.txt', 'r').read()
'code\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()
'code\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()
'code\nCODE\n'
```

For the encoding name `utf-16` , the BOM is handled automatically: on *output*, data is written in the platform's native endianness, and the BOM is always written; on *input*, data is decoded per the BOM, and the BOM is always stripped. This reflects the fact that BOMs are standard and required in the UTF-16 encoding scheme:

```
>>> import sys
>>> sys.byteorder          # Windows host's default endi
'little'
>>> open('temp.txt', 'w', encoding='utf-16').write('coc
10

>>> open('temp.txt', 'rb').read()
b'\xff\xfec\x00o\x00d\x00e\x00\r\x00\n\x00C\x00O\x00D\x

>>> open('temp.txt', 'r', encoding='utf-16').read()
'code\nCODE\n'
```

More specific UTF-16 encoding names can specify different endianness, though you may have to manually write and skip the BOM yourself in some scenarios if it is required or present—study the following examples for more BOM-making instructions (sorry):

```
>>> open('temp.txt', 'r', encoding='utf-16-le').read()
'\ufeffcode\nCODE\n'

>>> open('temp.txt', 'w', encoding='utf-16-be').write('
11
>>> open('temp.txt', 'rb').read()
b'\xfe\xff\x00c\x00o\x00d\x00e\x00\r\x00\n\x00C\x00O\x0

>>> open('temp.txt', 'r', encoding='utf-16').read()
'code\nCODE\n'
>>> open('temp.txt', 'r', encoding='utf-16-be').read()
'\ufeffcode\nCODE\n'
```

The more specific UTF-16 encoding names by themselves create and work fine with BOM-less files, though `utf-16` requires one on input in order to determine byte order:

```
>>> open('temp.txt', 'w', encoding='utf-16-le').write('
4
>>> open('temp.txt', 'rb').read()       # Okay if BOM r
b'C\x00O\x00D\x00E\x00'

>>> open('temp.txt', 'r', encoding='utf-16-le').read()
'CODE'
>>> open('temp.txt', 'r', encoding='utf-16').read()
UnicodeError: UTF-16 stream does not start with BOM
```

Experiment with these encodings yourself, or see Python's library manuals for more details on the BOM. And if you really want to drop the bomb in Python (and stretch this section's silly bit to its breaking point), Unicode emoji characters do the job:

```
>>> open('boms.txt', 'w', encoding='utf-8-sig').write('

' * 10)
10
```

```
>>> open('boms.txt', encoding='utf-8-sig').read()
'


'
```

Unicode Normalization: Whither Standard?

Last but not least, after devoting dozens of pages to Unicode, we'll close by explaining one way in which it, at least arguably, falls short. In brief, this standard *failed to standardize* code points for a handful of characters: it allows the same character to be represented in more than one way, which breaks equality testing and can wreak interoperability havoc with text-processing programs. While there may have been valid rationales for this policy, it adds a special case for programmers and undoubtedly breaks many a tool and app.

For example, the character `ñ` (an `n` augmented with a tilde, commonly used in Spanish) can be represented with two different code-point sequences in the Unicode standard:

- As a single character with code point `\u00F1` —an augmented `n`
- As a two-codepoint sequence `\u006E` and `\u0303` —a naked `n` followed by a combining tilde

The first of these is called the *composed* form, known as *NFC*, because it combines letter and accent. The second is called the *decomposed* form, known as *NFD*, because its parts are split. Despite their glaring difference, the Unicode standard mandates that these two forms represent the same character `ñ` and must be treated as such by programs.

It's easy to observe these alter egos in Python: the following uses `\u…` Unicode code-point escapes to specify the code points for `ñ` in both forms (Windows users: don't be alarmed if some characters, especially NFDs, render differently in command-line interfaces due to settings that are well beyond this book's scope):

```
>>> L = '\u00F1'             # NFC form ('\xF1' works tc
>>> M = '\u006E\u0303'       # NFD form
>>> L, M                     # The same character
('ñ', 'ñ')
```

Nor is this character alone in its split personality. Other characters such as `Å`, `è`, `é`, and

have both NFC and NFD representations as well:

```
>>> '\u00C5', '\u0041\u030A'      # NFD (1 code point),
('Å', 'Å')
>>> '\u00E8', '\u0065\u0300'      # But it's the same c
('è', 'è')
>>> '\u00E9', '\u0065\u0301'
('é', 'é')
>>> '\u03D4', '\u03D2\u0308'
('

', '

')
```

Importantly, this is not about the *encoded* representation of this character in bytes, which naturally varies across different encodings. The two alternative representations for each preceding character differ for their decoded, in-memory representation as integer *code points*. Encoded forms (e.g., stored in files) differ, too, but decode to the same differing code points:

```
>>> '\u00F1'.encode('utf8'), '\u006E\u0303'.encode('utf
(b'\xc3\xb1', b'n\xcc\x83')

>>> b'\xc3\xb1'.decode('utf8'), b'n\xcc\x83'.decode('ut
('ñ', 'ñ')
```

The unfortunate consequence of this plurality is that it *breaks equality testing*: because the same character can be represented in multiple ways, it's impossible to compare with the usual tools. In Python specifically, the `==` equal-by-value operator, which compares characters (really, code points), won't suffice in the presence of Unicode doppelgängers:

```
>>> L = '\u00F1'
>>> M = '\u006E\u0303'

>>> L, M                        # The same character
('ñ', 'ñ')

>>> L == M                      # But equality says no!
False

>>> len(L), len(M)              # Because their code poi
(1, 2)
```

All of which might not be a problem in an ideal computing world that settled on one form as a de facto standard for portability's sake. Unfortunately, that's not the world we occupy. Computer vendors, being computer vendors, have opted to favor different forms: broadly speaking, macOS prefers NFD, Windows and others prefer NFC, this can also vary by filesystem, and tolerance of nonnative forms is less than complete. The net effect is that the Unicode standard created an interoperability problem while trying to fix another!

So what to do with a world that just won't standardize? The trick is that, for every tool that processes file contents or names across divergent platforms, text must be converted to a common form before comparisons. And luckily, Python makes this remarkably easy:

```
>>> from unicodedata import normalize             # Pyt
>>> L = '\u00F1'
>>> M = '\u006E\u0303'
>>> L, M                                          # San
('ñ', 'ñ')

>>> L == M                                        # Equ
False

>>> normalize('NFC', L) == normalize('NFC', M)    # Com
True
>>> normalize('NFD', L) == normalize('NFD', M)    # Eit
True
```

Programs that compare filenames sent between platforms, for example, should be careful to normalize this way. Otherwise, files whose names look the same to users (and really *are* the same per Unicode) will fail to match by simple equality and derail searches and syncs. The same goes for text files obtained from arbitrary sources—like most internet content.

There's more to this story (e.g., the *canonical* equivalence of NFC and NFD normalized forms is still not the same as *compatibility*, though most programs don't need to care). Because we've run tight on space, though, we'll stop short. If you'd like to dig deeper, you'll find ample follow-up coverage both on the web and in Python's manual set (see the latter's Unicode HOWTO and its coverage of the related string `casefold` method).

At the least, though, you now shouldn't be wholly surprised when your text-processing code is bitten by this curious choice of the Unicode standard.

Chapter Summary

This chapter explored the advanced string support available in Python for processing Unicode text and binary data. While some programmers use ASCII text and may get by with the basic string type and its operations, Python's string model fully supports both richer Unicode text via the normal `str` string type and byte-oriented binary data with `bytes` and `bytearray`.

In addition, we learned how Python's file object automatically encodes and decodes Unicode text, and deals with byte strings for binary-mode files. Finally, we briefly met some text and binary tools in Python's library and sampled their behavior with strings, and took a look at the darker Unicode corners of BOMs and normalization.

In the next chapter, we'll shift our focus to tool-builder topics, with a survey of ways to manage access to object attributes by inserting automatically run code. Before we move on, though, here's a set of questions to review what we've learned. This has been a substantial chapter, so be sure to read the quiz answers eventually for a more in-depth summary.

Test Your Knowledge: Quiz

1. What are the names and roles of string object types in Python?
2. How do Python's string types differ in terms of operations?
3. How can you code non-ASCII Unicode characters in a string in Python?
4. What are the main differences between text- and binary-mode files in Python?
5. How would you read a Unicode text file that contains text in a different encoding than the default for your platform?
6. How can you create a Unicode text file in a specific encoding format?
7. Why is ASCII text considered to be a kind of Unicode text?
8. What do BOM and normalization mean in Unicode?
9. How large an impact does Python's text/binary string dichotomy have on your code?

Test Your Knowledge: Answers

1. Python has three string types: `str` (for Unicode text, including ASCII), `bytes` (for binary data with absolute byte values), and `bytearray` (a mutable flavor of `bytes`). The `str` type usually represents content stored in text files, and the other two types generally represent content stored in binary files (including encoded text).
2. Python's string types share almost all the same operations: method calls, sequence operations, and even larger tools like pattern matching work the same way. On the other hand, only `str` supports string formatting's `format` method and `f'…'` f-strings, and `bytearray` has an additional set of operations that perform in-place changes. The `str` and `bytes` types also have methods for encoding and decoding text, respectively.
3. Non-ASCII Unicode characters can be coded in a `str` string with both hex (`\xNN`) and Unicode (`\uNNNN`, `\UNNNNNNNN`) escapes for code points, both of which denote character code points. They can also be coded in their encoded form as `bytes` using hex escapes and decoded to text. On most devices, non-ASCII characters—accented characters and emojis, for example—can also be typed or pasted directly into code and are interpreted per the UTF-8 default or an encoding-directive comment at the top of a source code file.
4. Text-mode files assume their content is Unicode text (even if it's all ASCII) and automatically decode when reading and encode when writing.

With binary-mode files, bytes are transferred to and from the file unchanged. The contents of text-mode files are usually represented as `str` objects in your script, and the contents of binary files are represented as `bytes` (or `bytearray`) objects. Text-mode files also handle BOMs for certain encoding types and automatically translate newline sequences to and from the single `\n` character on input and output unless this is explicitly disabled; binary-mode files do not perform either of these steps.

5. To read files encoded in a different encoding than the default for your platform, simply pass the name of the file's encoding to the `open` built-in function; data will be decoded per the specified encoding when it is read from the file, and you'll get back a decoded Unicode-text `str` string that has no encoding. You can also read in binary mode and manually decode the bytes to a string by giving an encoding name, but this involves extra work and is somewhat error-prone for multibyte characters (you may accidentally read a partial character sequence when loading content by bytes or chunks).
6. To create a Unicode text file in a specific encoding format, pass the desired encoding name to `open`; strings will be encoded per the desired encoding when they are written to the file. You can also manually encode a string to bytes and write it in binary mode, but this is usually extra work.
7. ASCII text is considered to be a kind of Unicode text because its 7-bit (0..127) range of values is a subset of many Unicode encodings. For example, valid ASCII text is also valid Latin-1 text (Latin-1 simply assigns the remaining possible values in an 8-bit byte to additional characters) and valid UTF-8 text (UTF-8 uses a variable-byte scheme for representing more characters, but ASCII characters are still represented with the same values in a single byte). This makes Unicode backward compatible with ASCII, as long as the encodings used represent ASCII the same way.
8. The Unicode BOM is a sequence of bytes added to the front of a text string or file in some encodings to both identify the encoding and give the string's endianness. Unicode normalization converts characters to a common format to neutralize differences that arise for characters that have multiple code-point values in the Unicode standard and would fail to match by simple code-point (character) equality.
9. The impact of Python's strings model depends upon the types of strings you use. For scripts that use simple ASCII text on platforms with ASCII-compatible default encodings, the impact is probably minor: the `str` string sans encodings suffices in this case. Moreover, although string-

related tools in the standard library, such as `re`, `struct`, `pickle`, and `os`, may technically use different types in different contexts, the effect is largely irrelevant to most programs because Python's `str` and `bytes` support almost identical interfaces. On the other hand, if you process non-ASCII Unicode text, you'll need to use `str` and pass encodings to `open`; if you deal with binary data files, you'll need to deal with content as `bytes` objects; and if your code must work across many platforms or process content from arbitrary sources, you'll want to use explicit encodings for text files. In general, Unicode is the way the text world works today and will be a must-know tool for most Python users.

[1] File fine points: most of this chapter's file examples use filenames relative to, and hence stored in, the current directory, so be sure to run them in a directory (a.k.a. folder) where you have *permission* to create files; `cd` to one in your shell or IDE if needed. This may matter on platforms with major storage constraints like Android, but you'll probably already be in a safe folder in most apps. Also, file *extensions* (e.g., `.txt`) don't mean anything to Python and are technically optional; some IDEs that hang on to objects for debugging may require manual `close` calls to flush changes too; and purists take note that `file` is no longer a built-in name in Python and OK to use as a variable here!