

## CHAPTER 4. STRINGS

---

“ I’m telling you this ’cause you’re one of my friends.

My alphabet starts where your alphabet ends! ”

— Dr. Seuss, *On Beyond Zebra!*

### SOME BORING STUFF YOU NEED TO UNDERSTAND BEFORE YOU CAN DIVE IN #

Few people think about it, but text is incredibly complicated. Start with the alphabet. The people of Bougainville have the smallest alphabet in the world; their Rotokas alphabet is composed of only 12 letters: A, E, G, I, K, O, P, R, S, T, U, and V. On the other end of the spectrum, languages like Chinese, Japanese, and Korean have thousands of characters. English, of course, has 26 letters — 52 if you count uppercase and lowercase separately — plus a handful of `!@#$%&` punctuation marks.

When you talk about “text,” you’re probably thinking of “characters and symbols on my computer screen.” But computers don’t deal in characters and symbols; they deal in bits and bytes. Every piece of text you’ve ever seen on a computer screen is actually stored in a particular *character encoding*. Very roughly speaking, the character encoding provides a mapping between the stuff you see on your screen and the stuff your computer actually stores in memory and on disk. There are many different character encodings, some optimized for particular languages like Russian or Chinese or English, and others that can be used for multiple languages.

In reality, it’s more complicated than that. Many characters are common to multiple encodings, but each encoding may use a different sequence of bytes to actually store those characters in memory or on disk. So you can think of the character encoding as a kind of decryption key. Whenever someone gives you a sequence of bytes — a file, a web page, whatever — and claims it’s “text,” you need to know what character encoding they used so you can decode the bytes into characters. If they give you the wrong key or no key at all, you’re left with the unenviable task of cracking the code yourself. Chances are you’ll get it wrong, and the result will be gibberish.

Surely you’ve seen web pages like this, with strange question-mark-like characters where apostrophes should be. That usually means the page author didn’t declare their character encoding correctly, your browser was

left guessing, and the result was a mix of expected and unexpected characters. In English it's merely annoying; in other languages, the result can be completely unreadable.

There are character encodings for each major language in the world. Since each language is different, and memory and disk space have historically been expensive, each character encoding is optimized for a particular language. By that, I mean each encoding using the same numbers (0–255) to represent that language's characters. For instance, you're probably familiar with the `ASCII` encoding, which stores English characters as numbers ranging from 0 to 127. (65 is capital "A", 97 is lowercase "a", &c.) English has a very simple alphabet, so it can be completely expressed in less than 128 numbers. For those of you who can count in base 2, that's 7 out of the 8 bits in a byte.

*Everything  
you thought  
you knew  
about strings  
is wrong.*

Western European languages like French, Spanish, and German have more letters than English. Or, more precisely, they have letters combined with various diacritical marks, like the ñ character in Spanish. The most common encoding for these languages is `CP-1252`, also called "windows-1252" because it is widely used on Microsoft Windows. The `CP-1252` encoding shares characters with `ASCII` in the 0–127 range, but then extends into the 128–255 range for characters like n-with-a-tilde-over-it (241), u-with-two-dots-over-it (252), &c. It's still a single-byte encoding, though; the highest possible number, 255, still fits in one byte.

Then there are languages like Chinese, Japanese, and Korean, which have so many characters that they require multiple-byte character sets. That is, each "character" is represented by a two-byte number from 0–65535. But different multi-byte encodings still share the same problem as different single-byte encodings, namely that they each use the same numbers to mean different things. It's just that the range of numbers is broader, because there are many more characters to represent.

That was mostly OK in a non-networked world, where "text" was something you typed yourself and occasionally printed. There wasn't much "plain text". Source code was `ASCII`, and everyone else used word processors, which defined their own (non-text) formats that tracked character encoding information along

with rich styling, &c. People read these documents with the same word processing program as the original author, so everything worked, more or less.

Now think about the rise of global networks like email and the web. Lots of “plain text” flying around the globe, being authored on one computer, transmitted through a second computer, and received and displayed by a third computer. Computers can only see numbers, but the numbers could mean different things. Oh no! What to do? Well, systems had to be designed to carry encoding information along with every piece of “plain text.” Remember, it’s the decryption key that maps computer-readable numbers to human-readable characters. A missing decryption key means garbled text, gibberish, or worse.

Now think about trying to store multiple pieces of text in the same place, like in the same database table that holds all the email you’ve ever received. You still need to store the character encoding alongside each piece of text so you can display it properly. Think that’s hard? Try searching your email database, which means converting between multiple encodings on the fly. Doesn’t that sound fun?

Now think about the possibility of multilingual documents, where characters from several languages are next to each other in the same document. (Hint: programs that tried to do this typically used escape codes to switch “modes.” Poof, you’re in Russian koi8-r mode, so 241 means Я; poof, now you’re in Mac Greek mode, so 241 means ώ.) And of course you’ll want to search *those* documents, too.

Now cry a lot, because everything you thought you knew about strings is wrong, and there ain’t no such thing as “plain text.”



## 4.2. UNICODE #

*Enter Unicode.*

Unicode is a system designed to represent every character from every language. Unicode represents each letter, character, or ideograph as a 4-byte number. Each number represents a unique character used in at least one of the world’s languages. (Not all the numbers are used, but more than 65535 of them are, so 2 bytes wouldn’t be sufficient.) Characters that are used in multiple languages generally have the same number, unless

there is a good etymological reason not to. Regardless, there is exactly 1 number per character, and exactly 1 character per number. Every number always means just one thing; there are no “modes” to keep track of. `U+0041` is always 'A', even if your language doesn't have an 'A' in it.

On the face of it, this seems like a great idea. One encoding to rule them all. Multiple languages per document. No more “mode switching” to switch between encodings mid-stream. But right away, the obvious question should leap out at you. Four bytes? For every single character? That seems awfully wasteful, especially for languages like English and Spanish, which need less than one byte (256 numbers) to express every possible character. In fact, it's wasteful even for ideograph-based languages (like Chinese), which never need more than two bytes per character.

There is a Unicode encoding that uses four bytes per character. It's called UTF-32, because 32 bits = 4 bytes. UTF-32 is a straightforward encoding; it takes each Unicode character (a 4-byte number) and represents the character with that same number. This has some advantages, the most important being that you can find the  $N$ th character of a string in constant time, because the  $N$ th character starts at the  $4 \times N$ th byte. It also has several disadvantages, the most obvious being that it takes four freaking bytes to store every freaking character.

Even though there are a lot of Unicode characters, it turns out that most people will never use anything beyond the first 65535. Thus, there is another Unicode encoding, called UTF-16 (because 16 bits = 2 bytes). UTF-16 encodes every character from 0–65535 as two bytes, then uses some dirty hacks if you actually need to represent the rarely-used “astral plane” Unicode characters beyond 65535. Most obvious advantage: UTF-16 is twice as space-efficient as UTF-32, because every character requires only two bytes to store instead of four bytes (except for the ones that don't). And you can still easily find the  $N$ th character of a string in constant time, if you assume that the string doesn't include any astral plane characters, which is a good assumption right up until the moment that it's not.

But there are also non-obvious disadvantages to both UTF-32 and UTF-16. Different computer systems store individual bytes in different ways. That means that the character `U+4E2D` could be stored in UTF-16 as either `4E 2D` or `2D 4E`, depending on whether the system is big-endian or little-endian. (For UTF-32, there are even more possible byte orderings.) As long as your documents never leave your computer, you're safe — different applications on the same computer will all use the same byte order. But the minute you want to transfer documents between systems, perhaps on a world wide web of some sort, you're going to need a way to indicate which order your bytes are stored. Otherwise, the receiving system has no way of knowing whether the two-byte sequence `4E 2D` means `U+4E2D` or `U+2D4E`.

To solve *this* problem, the multi-byte Unicode encodings define a “Byte Order Mark,” which is a special non-printable character that you can include at the beginning of your document to indicate what order your bytes are in. For UTF-16, the Byte Order Mark is U+FEFF. If you receive a UTF-16 document that starts with the bytes FF FE, you know the byte ordering is one way; if it starts with FE FF, you know the byte ordering is reversed.

Still, UTF-16 isn’t exactly ideal, especially if you’re dealing with a lot of ASCII characters. If you think about it, even a Chinese web page is going to contain a lot of ASCII characters — all the elements and attributes surrounding the printable Chinese characters. Being able to find the Nth character in constant time is nice, but there’s still the nagging problem of those astral plane characters, which mean that you can’t *guarantee* that every character is exactly two bytes, so you can’t *really* find the Nth character in constant time unless you maintain a separate index. And boy, there sure is a lot of ASCII text in the world...

Other people pondered these questions, and they came up with a solution:

# UTF-8

UTF-8 is a *variable-length* encoding system for Unicode. That is, different characters take up a different number of bytes. For ASCII characters (A-Z, &c.) UTF-8 uses just one byte per character. In fact, it uses the exact same bytes; the first 128 characters (0–127) in UTF-8 are indistinguishable from ASCII. “Extended Latin” characters like ñ and ö end up taking two bytes. (The bytes are not simply the Unicode code point like they would be in UTF-16; there is some serious bit-twiddling involved.) Chinese characters like 中 end up taking three bytes. The rarely-used “astral plane” characters take four bytes.

**Disadvantages:** because each character can take a different number of bytes, finding the  $N$ th character is an  $O(N)$  operation — that is, the longer the string, the longer it takes to find a specific character. Also, there is bit-twiddling involved to encode characters into bytes and decode bytes into characters.

**Advantages:** super-efficient encoding of common `ASCII` characters. No worse than `UTF-16` for extended Latin characters. Better than `UTF-32` for Chinese characters. Also (and you'll have to trust me on this, because I'm not going to show you the math), due to the exact nature of the bit twiddling, there are no byte-ordering issues. A document encoded in `UTF-8` uses the exact same stream of bytes on any computer.



## 4.3. DIVING IN #

In Python 3, all strings are sequences of Unicode characters. There is no such thing as a Python string encoded in `UTF-8`, or a Python string encoded as `CP-1252`. “Is this string `UTF-8`?” is an invalid question. `UTF-8` is a way of encoding characters as a sequence of bytes. If you want to take a string and turn it into a sequence of bytes in a particular character encoding, Python 3 can help you with that. If you want to take a sequence of bytes and turn it into a string, Python 3 can help you with that too. Bytes are not characters; bytes are bytes. Characters are an abstraction. A string is a sequence of those abstractions.

```
>>> s = '深入 Python'    ①
>>> len(s)               ②
9
>>> s[0]                 ③
'深'
>>> s + ' 3'             ④
'深入 Python 3'
```

- ① To create a string, enclose it in quotes. Python strings can be defined with either single quotes (`'`) or double quotes (`"`).
- ② The built-in `len()` function returns the length of the string, *i.e.* the number of characters. This is the same function you use to find the length of a list, tuple, set, or dictionary. A string is like a tuple of characters.

- ③ Just like getting individual items out of a list, you can get individual characters out of a string using index notation.
- ④ Just like lists, you can concatenate strings using the + operator.

\*  
\*\*

## 4.4. FORMATTING STRINGS #

Let's take another look at `humansize.py`:

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'], ①
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

```
    '''Convert a file size to human-readable form.
```

```
    Keyword arguments:
```

```
    size -- file size in bytes
```

```
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000
```

```
    Returns: string
```

```
    ...
```

```
    if size < 0:
```

```
        raise ValueError('number must be non-negative') ④
```

```
    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

```
    for suffix in SUFFIXES[multiple]:
```

```
        size /= multiple
```

```
    if size < multiple:
```

```
        return '{0:.1f} {1}'.format(size, suffix) ⑤
```

*Strings can  
be defined<sup>②</sup>  
with either  
single or  
double quotes.*

```
raise ValueError('number too large')
```

- ① 'KB', 'MB', 'GB'... those are each strings.
- ② Function docstrings are strings. This docstring spans multiple lines, so it uses three-in-a-row quotes to start and end the string.
- ③ These three-in-a-row quotes end the docstring.
- ④ There's another string, being passed to the exception as a human-readable error message.
- ⑤ There's a... whoa, what the heck is that?

Python 3 supports formatting values into strings. Although this can include very complicated expressions, the most basic usage is to insert a value into a string with a single placeholder.

```
>>> username = 'mark'
>>> password = 'PapayaWhip' ①
>>> "{0}'s password is {1}".format(username, password) ②
'mark's password is PapayaWhip'
```

- ① No, my password is not really PapayaWhip.
- ② There's a lot going on here. First, that's a method call on a string literal. *Strings are objects*, and objects have methods. Second, the whole expression evaluates to a string. Third, {0} and {1} are *replacement fields*, which are replaced by the arguments passed to the `format()` method.

#### 4.4.1. COMPOUND FIELD NAMES #

The previous example shows the simplest case, where the replacement fields are simply integers. Integer replacement fields are treated as positional indices into the argument list of the `format()` method. That means that {0} is replaced by the first argument (username in this case), {1} is replaced by the second argument (password), &c. You can have as many positional indices as you have arguments, and you can have as many arguments as you want. But replacement fields are much more powerful than that.

```
>>> import humansize
>>> si_suffixes = humansize.SUFFIXES[1000] ①
>>> si_suffixes
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
```



```
>>> '1000{0[0]} = 1{0[1]}'.format(si_suffixes) ②
'1000KB = 1MB'
```

- ① Rather than calling any function in the `humansize` module, you're just grabbing one of the data structures it defines: the list of “SI” (powers-of-1000) suffixes.
- ② This looks complicated, but it's not. `{0}` would refer to the first argument passed to the `format()` method, `si_suffixes`. But `si_suffixes` is a list. So `{0[0]}` refers to the first item of the list which is the first argument passed to the `format()` method: `'KB'`. Meanwhile, `{0[1]}` refers to the second item of the same list: `'MB'`. Everything outside the curly braces — including `1000`, the equals sign, and the spaces — is untouched. The final result is the string `'1000KB = 1MB'`.

What this example shows is that *format specifiers can access items and properties of data structures using (almost) Python syntax*. This is called *compound field names*. The following compound field names “just work”:

- Passing a list, and accessing an item of the list by index (as in the previous example)
- Passing a dictionary, and accessing a value of the dictionary by key
- Passing a module, and accessing its variables and functions by name
- Passing a class instance, and accessing its properties and methods by name
- Any combination of the above

*{0} is replaced by the 1<sup>st</sup> format() argument. {1} is replaced by the 2<sup>nd</sup>.*

Just to blow your mind, here's an example that combines all of the above:

```
>>> import humansize
>>> import sys
>>> '1MB = 1000{0.modules[humansize].SUFFIXES[1000][0]}'.format(sys)
'1MB = 1000KB'
```

Here's how it works:


- The `sys` module holds information about the currently running Python instance. Since you just imported it, you can pass the `sys` module itself as an argument to the `format()` method. So the replacement field `{0}` refers to the `sys` module.
- `sys.modules` is a dictionary of all the modules that have been imported in this Python instance. The keys are the module names as strings; the values are the module objects themselves. So the replacement field `{0.modules}` refers to the dictionary of imported modules.
- `sys.modules['humansize']` is the `humansize` module which you just imported. The replacement field `{0.modules[humansize]}` refers to the `humansize` module. Note the slight difference in syntax here. In real Python code, the keys of the `sys.modules` dictionary are strings; to refer to them, you need to put quotes around the module name (e.g. `'humansize'`). But within a replacement field, you skip the quotes around the dictionary key name (e.g. `humansize`). To quote [PEP 3101: Advanced String Formatting](#), “The rules for parsing an item key are very simple. If it starts with a digit, then it is treated as a number, otherwise it is used as a string.”
- `sys.modules['humansize'].SUFFIXES` is the dictionary defined at the top of the `humansize` module. The replacement field `{0.modules[humansize].SUFFIXES}` refers to that dictionary.
- `sys.modules['humansize'].SUFFIXES[1000]` is a list of SI suffixes: `['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']`. So the replacement field `{0.modules[humansize].SUFFIXES[1000]}` refers to that list.
- `sys.modules['humansize'].SUFFIXES[1000][0]` is the first item of the list of SI suffixes: `'KB'`. Therefore, the complete replacement field `{0.modules[humansize].SUFFIXES[1000][0]}` is replaced by the two-character string `KB`.

#### 4.4.2. FORMAT SPECIFIERS #

But wait! There's more! Let's take another look at that strange line of code from `humansize.py`:

```
if size < multiple:
    return '{0:.1f} {1}'.format(size, suffix)
```

`{1}` is replaced with the second argument passed to the `format()` method, which is `suffix`. But what is `{0:.1f}`? It's two things: `{0}`, which you recognize, and `:.1f`, which you don't. The second half (including and after the colon) defines the *format specifier*, which further refines how the replaced variable should be formatted.

 Format specifiers allow you to munge the replacement text in a variety of useful ways, like the `printf()` function in C. You can add zero- or space-padding, align strings, control decimal precision, and even convert numbers to hexadecimal.

Within a replacement field, a colon (:) marks the start of the format specifier. The format specifier “.1” means “round to the nearest tenth” (i.e. display only one digit after the decimal point). The format specifier “f” means “fixed-point number” (as opposed to exponential notation or some other decimal representation). Thus, given a size of 698.24 and suffix of 'GB', the formatted string would be '698.2 GB', because 698.24 gets rounded to one decimal place, then the suffix is appended after the number.

```
>>> '{0:.1f} {1}'.format(698.24, 'GB')
'698.2 GB'
```

For all the gory details on format specifiers, consult the [Format Specification Mini-Language](#) in the official Python documentation.

\*  
\*\*

## 4.5. OTHER COMMON STRING METHODS #

Besides formatting, strings can do a number of other useful tricks.

```
>>> s = '''Finished files are the re- ①
... sult of years of scientif-
... ic study combined with the
... experience of years.'''
>>> s.splitlines() ②
['Finished files are the re-',
 'sult of years of scientif-',
 'ic study combined with the',
 'experience of years.']
>>> print(s.lower()) ③
```

finished files are the result of years of scientific study combined with the experience of years.

```
>>> s.lower().count('f') ④
6
```

- ① You can input multiline strings in the Python interactive shell. Once you start a multiline string with triple quotation marks, just hit ENTER and the interactive shell will prompt you to continue the string. Typing the closing triple quotation marks ends the string, and the next ENTER will execute the command (in this case, assigning the string to `s`).
- ② The `splitlines()` method takes one multiline string and returns a list of strings, one for each line of the original. Note that the carriage returns at the end of each line are not included.
- ③ The `lower()` method converts the entire string to lowercase. (Similarly, the `upper()` method converts a string to uppercase.)
- ④ The `count()` method counts the number of occurrences of a substring. Yes, there really are six “f”s in that sentence!

Here’s another common case. Let’s say you have a list of key-value pairs in the form

`key1=value1&key2=value2`, and you want to split them up and make a dictionary of the form `{key1: value1, key2: value2}`.

```
>>> query = 'user=pilgrim&database=master&password=PapayaWhip'
>>> a_list = query.split('&') ①
>>> a_list
['user=pilgrim', 'database=master', 'password=PapayaWhip']
>>> a_list_of_lists = [v.split('=', 1) for v in a_list if '=' in v] ②
>>> a_list_of_lists
[['user', 'pilgrim'], ['database', 'master'], ['password', 'PapayaWhip']]
>>> a_dict = dict(a_list_of_lists) ③
>>> a_dict
{'password': 'PapayaWhip', 'user': 'pilgrim', 'database': 'master'}
```

- ① The `split()` string method has one required argument, a delimiter. The method splits a string into a list of strings based on the delimiter. Here, the delimiter is an ampersand character, but it could be anything.

- ② Now we have a list of strings, each with a key, followed by an equals sign, followed by a value. We can use a list comprehension to iterate over the entire list and split each string into two strings based on the first equals sign. The optional second argument to the `split()` method is the number of times you want to split. 1 means “only split once,” so the `split()` method will return a two-item list. (In theory, a value could contain an equals sign too. If you just used `'key=value=foo'.split('=')`, you would end up with a three-item list `['key', 'value', 'foo']`.)
- ③ Finally, Python can turn that list-of-lists into a dictionary simply by passing it to the `dict()` function.



The previous example looks a lot like parsing query parameters in a URL, but real-life URL parsing is actually more complicated than this. If you're dealing with URL query parameters, you're better off using the `urllib.parse.parse_qs()` function, which handles some non-obvious edge cases.

#### 4.5.1. SLICING A STRING #

Once you've defined a string, you can get any part of it as a new string. This is called *slicing* the string. Slicing strings works exactly the same as slicing lists, which makes sense, because strings are just sequences of characters.

```
>>> a_string = 'My alphabet starts where your alphabet ends.'
>>> a_string[3:11]           ①
'alphabet'
>>> a_string[3:-3]          ②
'alphabet starts where your alphabet en'
>>> a_string[0:2]           ③
'My'
>>> a_string[:18]           ④
'My alphabet starts'
>>> a_string[18:]           ⑤
' where your alphabet ends.'
```

- ① You can get a part of a string, called a “slice”, by specifying two indices. The return value is a new string containing all the characters of the string, in order, starting with the first slice index.

- ② Like slicing lists, you can use negative indices to slice strings.
- ③ Strings are zero-based, so `a_string[0:2]` returns the first two items of the string, starting at `a_string[0]`, up to but not including `a_string[2]`.
- ④ If the left slice index is 0, you can leave it out, and 0 is implied. So `a_string[:18]` is the same as `a_string[0:18]`, because the starting 0 is implied.
- ⑤ Similarly, if the right slice index is the length of the string, you can leave it out. So `a_string[18:]` is the same as `a_string[18:44]`, because this string has 44 characters. There is a pleasing symmetry here. In this 44-character string, `a_string[:18]` returns the first 18 characters, and `a_string[18:]` returns everything but the first 18 characters. In fact, `a_string[:n]` will always return the first `n` characters, and `a_string[n:]` will return the rest, regardless of the length of the string.

\*  
\*\*

## 4.6. STRINGS VS. BYTES #

Bytes are bytes; characters are an abstraction. An immutable sequence of Unicode characters is called a *string*. An immutable sequence of numbers-between-0-and-255 is called a *bytes* object.

```
>>> by = b'abcd\x65' ①
>>> by
b'abcde'
>>> type(by) ②
<class 'bytes'>
>>> len(by) ③
5
>>> by += b'\xff' ④
>>> by
b'abcde\xff'
>>> len(by) ⑤
6
>>> by[0] ⑥
97
```

```
>>> by[0] = 102 ⑦
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'bytes' object does not support item assignment
```

- ① To define a bytes object, use the b'' "byte literal" syntax. Each byte within the byte literal can be an ASCII character or an encoded hexadecimal number from \x00 to \xff (0–255).
- ② The type of a bytes object is bytes.
- ③ Just like lists and strings, you can get the length of a bytes object with the built-in len() function.
- ④ Just like lists and strings, you can use the + operator to concatenate bytes objects. The result is a new bytes object.
- ⑤ Concatenating a 5-byte bytes object and a 1-byte bytes object gives you a 6-byte bytes object.
- ⑥ Just like lists and strings, you can use index notation to get individual bytes in a bytes object. The items of a string are strings; the items of a bytes object are integers. Specifically, integers between 0–255.
- ⑦ A bytes object is immutable; you can not assign individual bytes. If you need to change individual bytes, you can either use string slicing and concatenation operators (which work the same as strings), or you can convert the bytes object into a bytearray object.

```
>>> by = b'abcd\x65'
```

```
>>> barr = bytearray(by) ①
```

```
>>> barr
```

```
bytearray(b'abcde')
```

```
>>> len(barr) ②
```

```
5
```

```
>>> barr[0] = 102 ③
```

```
>>> barr
```

```
bytearray(b'fbcd')
```

- ① To convert a bytes object into a mutable bytearray object, use the built-in bytearray() function.
- ② All the methods and operations you can do on a bytes object, you can do on a bytearray object too.
- ③ The one difference is that, with the bytearray object, you can assign individual bytes using index notation. The assigned value must be an integer between 0–255.

The one thing you *can never do* is mix bytes and strings.

```

>>> by = b'd'
>>> s = 'abcde'
>>> by + s                                     ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
>>> s.count(by)                                ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly
>>> s.count(by.decode('ascii'))               ③
1

```

- ① You can't concatenate bytes and strings. They are two different data types.
- ② You can't count the occurrences of bytes in a string, because there are no bytes in a string. A string is a sequence of characters. Perhaps you meant "count the occurrences of the string that you would get after decoding this sequence of bytes in a particular character encoding"? Well then, you'll need to say that explicitly. Python 3 won't implicitly convert bytes to strings or strings to bytes.
- ③ By an amazing coincidence, this line of code says "count the occurrences of the string that you would get after decoding this sequence of bytes in this particular character encoding."

And here is the link between strings and bytes: bytes objects have a `decode()` method that takes a character encoding and returns a string, and strings have an `encode()` method that takes a character encoding and returns a bytes object. In the previous example, the decoding was relatively straightforward — converting a sequence of bytes in the ASCII encoding into a string of characters. But the same process works with any encoding that supports the characters of the string — even legacy (non-Unicode) encodings.

```

>>> a_string = '深入 Python'                 ①
>>> len(a_string)
9
>>> by = a_string.encode('utf-8')             ②
>>> by
b'\xe6\xe7\xb1\xe5\x85\xa5 Python'
>>> len(by)
13

```



```

>>> by = a_string.encode('gb18030') ③
>>> by
b'\xc9\xee\xc8\xeb Python'
>>> len(by)
11
>>> by = a_string.encode('big5') ④
>>> by
b'\xb2`\xa4J Python'
>>> len(by)
11
>>> roundtrip = by.decode('big5') ⑤
>>> roundtrip
'深入 Python'
>>> a_string == roundtrip
True

```

- ① This is a string. It has nine characters.
- ② This is a bytes object. It has 13 bytes. It is the sequence of bytes you get when you take `a_string` and encode it in UTF-8.
- ③ This is a bytes object. It has 11 bytes. It is the sequence of bytes you get when you take `a_string` and encode it in GB18030.
- ④ This is a bytes object. It has 11 bytes. It is an *entirely different sequence of bytes* that you get when you take `a_string` and encode it in Big5.
- ⑤ This is a string. It has nine characters. It is the sequence of characters you get when you take `by` and decode it using the Big5 encoding algorithm. It is identical to the original string.

\*  
\*\*

## 4.7. POSTSCRIPT: CHARACTER ENCODING OF PYTHON SOURCE CODE #

Python 3 assumes that your source code — i.e. each `.py` file — is encoded in UTF-8.



In Python 2, the default encoding for `.py` files was `ASCII`. In Python 3, the default encoding is `UTF-8`.

If you would like to use a different encoding within your Python code, you can put an encoding declaration on the first line of each file. This declaration defines a `.py` file to be `windows-1252`:

```
# -*- coding: windows-1252 -*-
```

Technically, the character encoding override can also be on the second line, if the first line is a `UNIX`-like hash-bang command.

```
#!/usr/bin/python3
```

```
# -*- coding: windows-1252 -*-
```

For more information, consult PEP 263: Defining Python Source Code Encodings.



## 4.8. FURTHER READING #

On Unicode in Python:

- Python Unicode HOWTO
- What's New In Python 3: Text vs. Data Instead Of Unicode vs. 8-bit
- PEP 261 explains how Python handles astral characters outside of the Basic Multilingual Plane (i.e. characters whose ordinal value is greater than 65535)

On Unicode in general:

- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
- On the Goodness of Unicode

- On Character Strings
- Characters vs. Bytes

On character encoding in other formats:

- Character encoding in XML
- Character encoding in HTML

On strings and string formatting:

- string — Common string operations
- Format String Syntax
- Format Specification Mini-Language
- PEP 3101: Advanced String Formatting



© 2001–|| Mark Pilgrim