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. Everything you thought you knew about strings is wrong.

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.

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."

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 Nth character of a string in constant time, because the Nth character starts at the 4×Nth 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 Nth 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 Nth 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.

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.

FORMATTING STRINGS

Strings can be defined with either single or double quotes.
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.                          ②

    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

    '''
      ③
    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


    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.

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'.

{0} is replaced by the 1st format() argument. {1} is replaced by the 2nd.

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

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.

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.

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 re-
sult of years of scientif-
ic 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.

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.

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.

skip over this code listing

[hide] [open in new window]

```
>>> by = b'abcd\x65'
>>> barr = bytearray(by)  ①
>>> barr
bytearray(b'abcde')
>>> len(barr)             ②
5
>>> barr[0] = 102         ③
>>> barr
bytearray(b'fbcde')
```

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\xb7\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.

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 -*-
```

DIVING IN

Getting a small bit of text out of a large block of text is a challenge. In Python, strings have methods for searching and replacing: index(), find(), split(), count(), replace(), &c. But these methods are limited to the simplest of cases. For example, the index() method looks for a single, hard-coded substring, and the search is always case-sensitive. To do case-insensitive searches of a string s, you must call s.lower() or s.upper() and make sure your search strings are the appropriate case to match. The replace() and split() methods have the same limitations.

If your goal can be accomplished with string methods, you should use them. They're fast and simple and easy to read, and there's a lot to be said for fast, simple, readable code. But if you find yourself using a lot of different string functions with if statements to handle special cases, or if you're chaining calls to split() and join() to slice-and-dice your strings, you may need to move up to regular expressions.

Regular expressions are a powerful and (mostly) standardized way of searching, replacing, and parsing text with complex patterns of characters. Although the regular expression syntax is tight and unlike normal code, the result can end up being more readable than a hand-rolled solution that uses a long chain of string functions. There are even ways of embedding comments within regular expressions, so you can include fine-grained documentation within them.


If you've used regular expressions in other languages (like Perl, JavaScript, or PHP), Python's syntax will be very familiar. Read the summary of the re module to get an overview of the available functions and their arguments.

CASE STUDY: STREET ADDRESSES

This series of examples was inspired by a real-life problem I had in my day job several years ago, when I needed to scrub and standardize street addresses exported from a legacy system before importing them into a newer system. (See, I don't just make this stuff up; it's actually useful.) This example shows how I approached the problem.

skip over this code listing

[hide] [open in new window]

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')                    ①
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')                    ②
```

```
'100 NORTH BRD. RD.'

>>> s[:-4] + s[-4:].replace('ROAD', 'RD.')  ③

'100 NORTH BROAD RD.'

>>> import re                                ④

>>> re.sub('ROAD$', 'RD.', s)               ⑤

'100 NORTH BROAD RD.'
```

1. My goal is to standardize a street address so that 'ROAD' is always abbreviated as 'RD.'. At first glance, I thought this was simple enough that I could just use the string method replace(). After all, all the data was already uppercase, so case mismatches would not be a problem. And the search string, 'ROAD', was a constant. And in this deceptively simple example, s.replace() does indeed work.
2. Life, unfortunately, is full of counterexamples, and I quickly discovered this one. The problem here is that 'ROAD' appears twice in the address, once as part of the street name 'BROAD' and once as its own word. The replace() method sees these two occurrences and blindly replaces both of them; meanwhile, I see my addresses getting destroyed.
3. To solve the problem of addresses with more than one 'ROAD' substring, you could resort to something like this: only search and replace 'ROAD' in the last four characters of the address (s[-4:]), and leave the string alone (s[:-4]). But you can see that this is already getting unwieldy. For example, the pattern is dependent on the length of the string you're replacing. (If you were replacing 'STREET' with 'ST.', you would need to use s[:-6] and s[-6:].replace(...).) Would you like to come back in six months and debug this? I know I wouldn't.
4. It's time to move up to regular expressions. In Python, all functionality related to regular expressions is contained in the re module.
5. Take a look at the first parameter: 'ROAD$'. This is a simple regular expression that matches 'ROAD' only when it occurs at the end of a string. The $ means "end of the string." (There is a corresponding character, the caret ^, which means "beginning of the string.") Using the re.sub() function, you search the string s for the regular expression 'ROAD$' and replace it with 'RD.'. This matches the ROAD at the end of the string s, but does not

match the ROAD that's part of the word BROAD, because that's in the middle of s.

^ matches the start of a string. $ matches the end of a string.

Continuing with my story of scrubbing addresses, I soon discovered that the previous example, matching 'ROAD' at the end of the address, was not good enough, because not all addresses included a street designation at all. Some addresses simply ended with the street name. I got away with it most of the time, but if the street name was 'BROAD', then the regular expression would match 'ROAD' at the end of the string as part of the word 'BROAD', which is not what I wanted.

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)    ①
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)    ②
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)    ③
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s)   ④
'100 BROAD RD. APT 3'
```

1)What I really wanted was to match 'ROAD' when it was at the end of the string and it was its own word (and not a part of some larger word). To express this in a regular expression, you use \b, which means "a word boundary must occur right here." In Python, this is complicated by the fact that the '\' character in a string must itself be escaped. This is sometimes referred to as the backslash plague, and it is one reason why regular expressions are easier in Perl than in Python. On the down side, Perl mixes regular expressions with other syntax, so if you have a bug, it may be hard to tell whether it's a bug in syntax or a bug in your

regular expression.

2)To work around the backslash plague, you can use what is called a raw string, by prefixing the string with the letter r. This tells Python that nothing in this string should be escaped; '\t' is a tab character, but r'\t' is really the backslash character \ followed by the letter t. I recommend always using raw strings when dealing with regular expressions; otherwise, things get too confusing too quickly (and regular expressions are confusing enough already).

3)*sigh* Unfortunately, I soon found more cases that contradicted my logic. In this case, the street address contained the word 'ROAD' as a whole word by itself, but it wasn't at the end, because the address had an apartment number after the street designation. Because 'ROAD' isn't at the very end of the string, it doesn't match, so the entire call to re.sub() ends up replacing nothing at all, and you get the original string back, which is not what you want.

4)To solve this problem, I removed the $ character and added another \b. Now the regular expression reads "match 'ROAD' when it's a whole word by itself anywhere in the string," whether at the end, the beginning, or somewhere in the middle.

CASE STUDY: ROMAN NUMERALS

You've most likely seen Roman numerals, even if you didn't recognize them. You may have seen them in copyrights of old movies and television shows ("Copyright MCMXLVI" instead of "Copyright 1946"), or on the dedication walls of libraries or universities ("established MDCCCLXXXVIII" instead of "established 1888"). You may also have seen them in outlines and bibliographical references. It's a system of representing numbers that really does date back to the ancient Roman empire (hence the name).

In Roman numerals, there are seven characters that are repeated and combined in various ways to represent numbers.

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

The following are some general rules for constructing Roman numerals:

- Sometimes characters are additive. I is 1, II is 2, and III is 3. VI is 6 (literally, "5 and 1"), VII is 7, and VIII is 8.
- The tens characters (I, X, C, and M) can be repeated up to three times. At 4, you need to subtract from the next highest fives character. You can't represent 4 as IIII; instead, it is represented

as IV (“1 less than 5”). 40 is written as XL (“10 less than 50”), 41 as XLI, 42 as XLII, 43 as XLIII, and then 44 as XLIV (“10 less than 50, then 1 less than 5”).

- Sometimes characters are… the opposite of additive. By putting certain characters before others, you subtract from the final value. For example, at 9, you need to subtract from the next highest tens character: 8 is VIII, but 9 is IX (“1 less than 10”), not VIIII (since the I character can not be repeated four times). 90 is XC, 900 is CM.
- The fives characters can not be repeated. 10 is always represented as X, never as VV. 100 is always C, never LL.
- Roman numerals are read left to right, so the order of characters matters very much. DC is 600; CD is a completely different number (400, “100 less than 500”). CI is 101; IC is not even a valid Roman numeral (because you can't subtract 1 directly from 100; you would need to write it as XCIX, “10 less than 100, then 1 less than 10”).

CHECKING FOR THOUSANDS

What would it take to validate that an arbitrary string is a valid Roman numeral? Let’s take it one digit at a time. Since Roman numerals are always written highest to lowest, let’s start with the highest: the thousands place. For numbers 1000 and higher, the thousands are represented by a series of M characters.

```
>>> import re

>>> pattern = '^M?M?M?$'          ①


>>> re.search(pattern, 'M')       ②

<_sre.SRE_Match object at 0106FB58>

>>> re.search(pattern, 'MM')      ③

<_sre.SRE_Match object at 0106C290>

>>> re.search(pattern, 'MMM')     ④

<_sre.SRE_Match object at 0106AA38>

>>> re.search(pattern, 'MMMM')    ⑤


>>> re.search(pattern, '')        ⑥
```

```
<_sre.SRE_Match object at 0106F4A8>
```

1)This pattern has three parts. ^ matches what follows only at the beginning of the string. If this were not specified, the pattern would match no matter where the M characters were, which is not what you want. You want to make sure that the M characters, if they're there, are at the beginning of the string. M? optionally matches a single M character. Since this is repeated three times, you're matching anywhere from zero to three M characters in a row. And $ matches the end of the string. When combined with the ^ character at the beginning, this means that the pattern must match the entire string, with no other characters before or after the M characters.
2)The essence of the re module is the search() function, that takes a regular expression (pattern) and a string ('M') to try to match against the regular expression. If a match is found, search() returns an object which has various methods to describe the match; if no match is found, search() returns None, the Python null value. All you care about at the moment is whether the pattern matches, which you can tell by just looking at the return value of search(). 'M' matches this regular expression, because the first optional M matches and the second and third optional M characters are ignored.

```
3)'MM' matches because the first and second optional M characters match and the third M is ignored.
5)'MMM' matches because all three M characters match.
6)'MMMM' does not match. All three M characters match, but then the regular expression insists

on the string ending (because of the $ character), and the string doesn't end yet

(because of the fourth M). So search() returns None.
Interestingly, an empty string also matches this regular expression, since all the M

characters are optional.
```

CHECKING FOR HUNDREDS

? makes a pattern optional.

The hundreds place is more difficult than the thousands, because there are several mutually exclusive ways it could be expressed, depending on its value.

- 100 = C
- 200 = CC
- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

So there are four possible patterns:

- CM
- CD
- Zero to three C characters (zero if the hundreds place is 0)
- D, followed by zero to three C characters
- The last two patterns can be combined:
- an optional D, followed by zero to three C characters
- This example shows how to validate the hundreds place of a Roman numeral.

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$'  ①
>>> re.search(pattern, 'MCM')             ②
<_sre.SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')              ③
<_sre.SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')          ④
<_sre.SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')            ⑤
>>> re.search(pattern, '')                ⑥
<_sre.SRE_Match object at 01071D98>
```

1)This pattern starts out the same as the previous one, checking for the beginning of the string (^), then the thousands place (M?M?M?). Then it has the new part, in parentheses, which defines a set of three

mutually exclusive patterns, separated by vertical bars: CM, CD, and D?C?C?C? (which is an optional D followed by zero to three optional C characters). The regular expression parser checks for each of these patterns in order (from left to right), takes the first one that matches, and ignores the rest.

2)'MCM' matches because the first M matches, the second and third M characters are ignored, and the CM matches (so the CD and D?C?C?C? patterns are never even considered). MCM is the Roman numeral representation of 1900.

3)'MD' matches because the first M matches, the second and third M characters are ignored, and the D?C?C?C? pattern matches D (each of the three C characters are optional and are ignored). MD is the Roman numeral representation of 1500.

4)'MMMCCC' matches because all three M characters match, and the D?C?C?C? pattern matches CCC (the D is optional and is ignored). MMMCCC is the Roman numeral representation of 3300.

4)'MCMC' does not match. The first M matches, the second and third M characters are ignored, and the CM matches, but then the $ does not match because you're not at the end of the string yet (you still have an unmatched C character). The C does not match as part of the D?C?C?C? pattern, because the mutually exclusive CM pattern has already matched.

5)Interestingly, an empty string still matches this pattern, because all the M characters are optional and ignored, and the empty string matches the D?C?C?C? pattern where all the characters are optional and ignored.

Whew! See how quickly regular expressions can get nasty? And you've only covered the thousands and hundreds places of Roman numerals. But if you followed all that, the tens and ones places are easy, because they're exactly the same pattern. But let's look at another way to express the pattern.

USING THE {N,M} SYNTAX

{1,4} matches between 1 and 4 occurrences of a pattern.

In the previous section, you were dealing with a pattern where the same character could be repeated up to three times. There is another way to express this in regular expressions, which some people find more readable. First look at the method we already used in the previous example.

skip over this code listing

[hide] [open in new window]

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')      ①

<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MM')     ②

<_sre.SRE_Match object at 0x008EEB48>
```

```
>>> re.search(pattern, 'MMM')    ③

<_sre.SRE_Match object at 0x008EE090>

>>> re.search(pattern, 'MMMM')   ④

>>>
```

1)This matches the start of the string, and then the first optional M, but not the second and third M (but that's okay because they're optional), and then the end of the string.
2)This matches the start of the string, and then the first and second optional M, but not the third M (but that's okay because it's optional), and then the end of the string.
3)This matches the start of the string, and then all three optional M, and then the end of the string.
4)This matches the start of the string, and then all three optional M, but then does not match the end of the string (because there is still one unmatched M), so the pattern does not match and returns None.

```
>>> pattern = '^M{0,3}$'         ①

>>> re.search(pattern, 'M')      ②

<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'MM')     ③

<_sre.SRE_Match object at 0x008EE090>

>>> re.search(pattern, 'MMM')    ④

<_sre.SRE_Match object at 0x008EEDA8>

>>> re.search(pattern, 'MMMM')   ⑤

>>>
```

1)This pattern says: "Match the start of the string, then anywhere from zero to three M characters, then the

end of the string.” The 0 and 3 can be any numbers; if you want to match at least one but no more than three M characters, you could say M{1,3}.

2)This matches the start of the string, then one M out of a possible three, then the end of the string.

3)This matches the start of the string, then two M out of a possible three, then the end of the string.

4)This matches the start of the string, then three M out of a possible three, then the end of the string.

5)This matches the start of the string, then three M out of a possible three, but then does not match the end of the string. The regular expression allows for up to only three M characters before the end of the string, but you have four, so the pattern does not match and returns None.

CHECKING FOR TENS AND ONES

Now let’s expand the Roman numeral regular expression to cover the tens and ones place. This example shows the check for tens.

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')      ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')       ②
<_sre.SRE_Match object at 0x008EEB48>
```

```
>>> re.search(pattern, 'MCMLX')      ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')    ④
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX')   ⑤
>>>
```

1)This matches the start of the string, then the first optional M, then CM, then XL, then the end of the string. Remember, the (A|B|C) syntax means “match exactly one of A, B, or C”. You match XL, so you ignore the XC and L?X?X?X? choices, and then move on to the end of the string. MCMXL is the Roman numeral representation of 1940.

2)This matches the start of the string, then the first optional M, then CM, then L?X?X?X?. Of the L?X?X?X?, it matches the L and skips all three optional X characters. Then you move to the end of the string. MCML is the Roman numeral representation of 1950.

3)This matches the start of the string, then the first optional M, then CM, then the optional L and the first optional X, skips the second and third optional X, then the end of the string. MCMLX is the Roman numeral representation of 1960.

4)This matches the start of the string, then the first optional M, then CM, then the optional L and all three optional X characters, then the end of the string. MCMLXXX is the Roman numeral representation of 1980.

5)This matches the start of the string, then the first optional M, then CM, then the optional L and all three optional X characters, then fails to match the end of the string because there is still one more X unaccounted for. So the entire pattern fails to match, and returns None. MCMLXXXX is not a valid Roman numeral.

(A|B) matches either pattern A or pattern B, but not both.

The expression for the ones place follows the same pattern. I'll spare you the details and show you the end result.

```
>>> pattern =
'^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

So what does that look like using this alternate {n,m} syntax? This example shows the new syntax.

```
>>> pattern =
'^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV')                    ①
<_sre.SRE_Match object at 0x008EEB48>
```

```
>>> re.search(pattern, 'MMDCLXVI')          ②

<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'MMMDCCCLXXXVIII')   ③

<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'I')                 ④

<_sre.SRE_Match object at 0x008EEB48>
```

1)This matches the start of the string, then one of a possible three M characters, then D?C{0,3}. Of that, it matches the optional D and zero of three possible C characters. Moving on, it matches L?X{0,3} by matching the optional L and zero of three possible X characters. Then it matches V?I{0,3} by matching the optional V and zero of three possible I characters, and finally the end of the string. MDLV is the Roman numeral representation of 1555.

2)This matches the start of the string, then two of a possible three M characters, then the D?C{0,3} with a D and one of three possible C characters; then L?X{0,3} with an L and one of three possible X characters; then V?I{0,3} with a V and one of three possible I characters; then the end of the string. MMDCLXVI is the Roman numeral representation of 2666.

3)This matches the start of the string, then three out of three M characters, then D?C{0,3} with a D and

three out of three C characters; then L?X{0,3} with an L and three out of three X characters; then V?I{0,3} with a V and three out of three I characters; then the end of the string. MMMDCCCLXXXVIII is the Roman numeral representation of 3888, and it's the longest Roman numeral you can write without extended syntax.

4)Watch closely. (I feel like a magician. "Watch closely, kids, I'm going to pull a rabbit out of my hat.")

This matches the start of the string, then zero out of three M, then matches D?C{0,3} by skipping the optional D and matching zero out of three C, then matches L?X{0,3} by skipping the optional L and matching zero out of three X, then matches V?I{0,3} by skipping the optional V and matching one out of three I. Then the end of the string. Whoa.

If you followed all that and understood it on the first try, you're doing better than I did. Now imagine trying to understand someone else's regular expressions, in the middle of a critical function of a large program. Or even imagine coming back to your own regular expressions a few months later. I've done it, and it's not a pretty sight.

Now let's explore an alternate syntax that can help keep your expressions maintainable.

VERBOSE REGULAR EXPRESSIONS

So far you've just been dealing with what I'll call "compact" regular expressions. As you've seen, they are difficult to read, and even if you figure out what one does, that's no guarantee that you'll be able to understand it six months later. What you really need is inline documentation.

Python allows you to do this with something called verbose regular expressions. A verbose regular expression is different from a compact regular expression in two ways:

- Whitespace is ignored. Spaces, tabs, and carriage returns are not matched as spaces, tabs, and carriage returns. They're not matched at all. (If you want to match a space in a verbose regular expression, you'll need to escape it by putting a backslash in front of it.)
- Comments are ignored. A comment in a verbose regular expression is just like a comment in Python code: it starts with a # character and goes until the end of the line. In this case it's a comment within a multi-line string instead of within your source code, but it works the same way.

This will be more clear with an example. Let's revisit the compact regular expression you've been working with, and make it a verbose regular expression. This example shows how.

```
>>> pattern = '''
    ^                   # beginning of string
    M{0,3}              # thousands - 0 to 3 Ms
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3
Cs),
                        #            or 500-800 (D, followed by 0 to 3
Cs)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                        #        or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                        #        or 5-8 (V, followed by 0 to 3 Is)
    $                   # end of string
    '''

>>> re.search(pattern, 'M', re.VERBOSE)                    ①

<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)            ②
```

```
<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'MMMDCCCLXXXVIII', re.VERBOSE)    ③

<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'M')                              ④
```

1)The most important thing to remember when using verbose regular expressions is that you need to pass an extra argument when working with them: re.VERBOSE is a constant defined in the re module that signals that the pattern should be treated as a verbose regular expression. As you can see, this pattern has quite a bit of whitespace (all of which is ignored), and several comments (all of which are ignored). Once you ignore the whitespace and the comments, this is exactly the same regular expression as you saw in the previous section, but it's a lot more readable.

2)This matches the start of the string, then one of a possible three M, then CM, then L and three of a possible thr

3)This matches the start of the string, then three of a possible three M, then D and three of a possible three C, three of a possible three I, then the end of the string.

4)This does not match. Why? Because it doesn't have the re.VERBOSE flag, so the re.search function is treating the pattern as a compact regular expression, with significant whitespace and literal hash marks.

Python can't auto-detect whether a regular expression is verbose or not. Python assumes every regular

expression is compact unless you explicitly state that it is verbose.

CASE STUDY: PARSING PHONE NUMBERS

\d matches any numeric digit (0–9). \D matches anything but digits.

So far you've concentrated on matching whole patterns. Either the pattern matches, or it doesn't. But regular expressions are much more powerful than that. When a regular expression does match, you can pick out specific pieces of it. You can find out what matched where.

This example came from another real-world problem I encountered, again from a previous day job. The problem: parsing an American phone number. The client wanted to be able to enter the number free-form (in a single field), but then wanted to store the area code, trunk, number, and optionally an extension separately in the company's database. I scoured the Web and found many examples of regular expressions that purported to do this, but none of them were permissive enough.

Here are the phone numbers I needed to be able to accept:

- `800-555-1212`
- `800 555 1212`
- `800.555.1212`
- `(800) 555-1212`
- `1-800-555-1212`
- `800-555-1212-1234`
- `800-555-1212x1234`
- `800-555-1212 ext. 1234`
- `work 1-(800) 555.1212 #1234`

Quite a variety! In each of these cases, I need to know that the area code was 800, the trunk was 555, and the rest of the phone number was 1212. For those with an extension, I need to know that the extension was 1234.

Let's work through developing a solution for phone number parsing. This example shows the first step.

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$')  ①

>>> phonePattern.search('800-555-1212').groups()             ②
```

```
('800', '555', '1212')

>>> phonePattern.search('800-555-1212-1234')                    ③

>>> phonePattern.search('800-555-1212-1234').groups()           ④

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```

1)Always read regular expressions from left to right. This one matches the beginning of the string, and then (\d{3}). What's \d{3}? Well, \d means "any numeric digit" (0 through 9). The {3} means "match exactly three numeric digits"; it's a variation on the {n,m} syntax you saw earlier. Putting it all in parentheses means "match exactly three numeric digits, and then remember them as a group that I can ask for later". Then match a literal hyphen. Then match another group of exactly three digits. Then another literal hyphen. Then another group of exactly four digits. Then match the end of the string.

2)To get access to the groups that the regular expression parser remembered along the way, use the groups() method on the object that the search() method returns. It will return a tuple of however many groups were defined in the regular expression. In this case, you defined three groups, one with three digits, one with three digits, and one with four digits.

3)This regular expression is not the final answer, because it doesn't handle a phone number with an extension on the end. For that, you'll need to expand the regular expression.

4)And this is why you should never "chain" the search() and groups() methods in production code. If the search() method returns no matches, it returns None, not a regular expression match object.

Calling None.groups() raises a perfectly obvious exception: None doesn't have a groups() method. (Of course, it's slightly less obvious when you get this exception from deep within your code. Yes, I speak from experience here.)

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$')  ①
>>> phonePattern.search('800-555-1212-1234').groups()              ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')                       ③
>>>
>>> phonePattern.search('800-555-1212')                            ④
>>>
```

1)This regular expression is almost identical to the previous one. Just as before, you match the beginning

of the string, then a remembered group of three digits, then a hyphen, then a remembered group of three digits, then a hyphen, then a remembered group of four digits. What's new is that you then match another hyphen, and a remembered group of one or more digits, then the end of the string.

2)The groups() method now returns a tuple of four elements, since the regular expression now defines four groups to remember.

3)Unfortunately, this regular expression is not the final answer either, because it assumes that the different parts of the phone number are separated by hyphens. What if they're separated by spaces, or commas, or dots? You need a more general solution to match several different types of separators.

4)Oops! Not only does this regular expression not do everything you want, it's actually a step backwards, because now you can't parse phone numbers without an extension. That's not what you wanted at all; if the extension is there, you want to know what it is, but if it's not there, you still want to know what the different parts of the main number are.

The next example shows the regular expression to handle separators between the different parts of the phone number.

```
>>> phonePattern =
re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$')  ①

>>> phonePattern.search('800 555 1212 1234').groups()  ②
```

```
('800', '555', '1212', '1234')

>>> phonePattern.search('800-555-1212-1234').groups()   ③

('800', '555', '1212', '1234')

>>> phonePattern.search('80055512121234')               ④

>>>

>>> phonePattern.search('800-555-1212')                 ⑤

>>>
```

1)Hang on to your hat. You're matching the beginning of the string, then a group of three digits, then \D+. What the heck is that? Well, \D matches any character except a numeric digit, and + means "1 or more". So \D+ matches one or more characters that are not digits. This is what you're using instead of a literal hyphen, to try to match different separators.

2)Using \D+ instead of - means you can now match phone numbers where the parts are separated by spaces instead of hyphens.

3)Of course, phone numbers separated by hyphens still work too.

4)Unfortunately, this is still not the final answer, because it assumes that there is a separator at all. What if the phone number is entered without any spaces or hyphens at all?

5)Oops! This still hasn't fixed the problem of requiring extensions. Now you have two problems, but you can solve both of them with the same technique.

The next example shows the regular expression for handling phone numbers without separators.

```
>>> phonePattern =
re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')  ①

>>> phonePattern.search('80055512121234').groups()       ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups()  ③
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()        ④
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234')           ⑤

>>>
```

1)The only change you've made since that last step is changing all the + to *. Instead of \D+ between the parts of the phone number, you now match on \D*. Remember that + means "1 or more"? Well, * means "zero or more". So now you should be able to parse phone numbers even when there is no

separator character at all.

2)Lo and behold, it actually works. Why? You matched the beginning of the string, then a remembered group of three digits (800), then zero non-numeric characters, then a remembered group of three digits (555), then zero non-numeric characters, then a remembered group of four digits (1212), then zero non-numeric characters, then a remembered group of an arbitrary number of digits (1234), then the end of the string.

3)Other variations work now too: dots instead of hyphens, and both a space and an x before the extension.

4)Finally, you've solved the other long-standing problem: extensions are optional again. If no extension is found, the groups() method still returns a tuple of four elements, but the fourth element is just an empty string.

5)I hate to be the bearer of bad news, but you're not finished yet. What's the problem here? There's an extra character before the area code, but the regular expression assumes that the area code is the first thing at the beginning of the string. No problem, you can use the same technique of "zero or more non-numeric characters" to skip over the leading characters before the area code.

The next example shows how to handle leading characters in phone numbers.

```
>>> phonePattern =
re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')  ①
```

```
>>> phonePattern.search('(800)5551212 ext. 1234').groups()
②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()
③
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234')
④
>>>
```

1)This is the same as in the previous example, except now you're matching \D*, zero or more non-numeric characters, before the first remembered group (the area code). Notice that you're not remembering these non-numeric characters (they're not in parentheses). If you find them, you'll just skip over them and then start remembering the area code whenever you get to it.

2)You can successfully parse the phone number, even with the leading left parenthesis before the area code. (The right parenthesis after the area code is already handled; it's treated as a non-numeric separator and matched by the \D* after the first remembered group.)

3)Just a sanity check to make sure you haven't broken anything that used to work. Since the leading

characters are entirely optional, this matches the beginning of the string, then zero non-numeric characters, then a remembered group of three digits (800), then one non-numeric character (the hyphen), then a remembered group of three digits (555), then one non-numeric character (the hyphen), then a remembered group of four digits (1212), then zero non-numeric characters, then a remembered group of zero digits, then the end of the string.

4)This is where regular expressions make me want to gouge my eyes out with a blunt object. Why doesn't this phone number match? Because there's a 1 before the area code, but you assumed that all the leading characters before the area code were non-numeric characters (\D*). Aargh.

Let's back up for a second. So far the regular expressions have all matched from the beginning of the string. But now you see that there may be an indeterminate amount of stuff at the beginning of the string that you want to ignore. Rather than trying to match it all just so you can skip over it, let's take a different approach: don't explicitly match the beginning of the string at all. This approach is shown in the next example.

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')
```

①

```
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()
```

②

```
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()
```

③

```
('800', '555', '1212', '')

>>> phonePattern.search('80055512121234').groups()

④

('800', '555', '1212', '1234')
```

1)Note the lack of ^ in this regular expression. You are not matching the beginning of the string anymore. There's nothing that says you need to match the entire input with your regular expression. The regular expression engine will do the hard work of figuring out where the input string starts to match, and go from there.

2)Now you can successfully parse a phone number that includes leading characters and a leading digit, plus any number of any kind of separators around each part of the phone number.

3)Sanity check. This still works.

4)That still works too.

See how quickly a regular expression can get out of control? Take a quick glance at any of the previous iterations. Can you tell the difference between one and the next?

While you still understand the final answer (and it is the final answer; if you've discovered a case it doesn't handle, I don't want to know about it), let's write it out as a verbose regular expression, before you forget why you made the choices you made.

```
>>> phonePattern = re.compile(r'''
                # don't match beginning of string, number can start
anywhere
    (\d{3})     # area code is 3 digits (e.g. '800')
    \D*         # optional separator is any number of non-digits
    (\d{3})     # trunk is 3 digits (e.g. '555')
    \D*         # optional separator
    (\d{4})     # rest of number is 4 digits (e.g. '1212')
    \D*         # optional separator
    (\d*)       # extension is optional and can be any number of
digits
    $           # end of string
    ''', re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()  ①
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                          ②
('800', '555', '1212', '')
```

1)Other than being spread out over multiple lines, this is exactly the same regular expression as the last step, so it’s no surprise that it parses the same inputs.

2)Final sanity check. Yes, this still works. You're done.

SUMMARY

This is just the tiniest tip of the iceberg of what regular expressions can do. In other words, even though you're completely overwhelmed by them now, believe me, you ain't seen nothing yet.

You should now be familiar with the following techniques:

^ matches the beginning of a string.

$ matches the end of a string.

- \b matches a word boundary.
- \d matches any numeric digit.
- \D matches any non-numeric character.
- x? matches an optional x character (in other words, it matches an x zero or one times).
- x* matches x zero or more times.
- x+ matches x one or more times.
- x{n,m} matches an x character at least n times, but not more than m times.
- (a|b|c) matches exactly one of a, b or c.
- (x) in general is a remembered group. You can get the value of what matched by using the groups() method of the object returned by re.search.

Regular expressions are extremely powerful, but they are not the correct solution for every problem. You should learn enough about them to know when they are appropriate, when they will solve your problems, and when they will cause more problems than they solve.