Chapter 12. Wrangle and Mangle Data

If you torture the data enough, nature will always confess.

—Ronald Coase

Up to this point, we've talked mainly about the Python language itself—its data types, code structures, syntax, and so on. The rest of this book is about application of these to real-world problems.

In this chapter, you'll learn many practical techniques for taming data. Sometimes, this is called *data munging*, or the more businesslike *ETL* (extract/transform/load) of the database world. Although programming books usually don't cover the topic explicitly, programmers spend a lot of time trying to mold data into the right shape for their purposes.

The specialty called *data science* has become very popular in the past few years. A *Harvard Business Review* article called data scientist the "sexiest job of the 21st century." If this meant in demand and well paying, then okay, but there's also more than enough drudgery. Data science goes beyond the ETL requirements of databases, often involving *machine learning* to unearth insights that were not visible to human eyes.

I'll start with basic data formats and then work up to the most useful new tools for data science.

Data formats fall roughly into two categories: *text* and *binary*. Python *strings* are used for text data, and this chapter includes string information that we've skipped so far:

- *Unicode* characters
- *Regular expression* pattern matching.

Then, we jump to binary data, and two more of Python's built-in types:

- *Bytes* for immutable eight-bit values

- *Bytearrays* for mutable ones

Text Strings: Unicode

You saw the basics of Python strings in Chapter 5. Now it's time to really dig into Unicode.

Python 3 strings are Unicode character sequences, not byte arrays. This is, by far, the single largest language change from Python 2.

All of the text examples in this book thus far have been plain old ASCII (American Standard Code for Information Interchange). ASCII was defined in the 1960s, before mullets roamed the earth. Computers then were the size of refrigerators, and only slightly smarter.

The basic unit of computer storage is the *byte*, which can store 256 unique values in its eight *bits*. For various reasons, ASCII used only seven bits (128 unique values): 26 uppercase letters, 26 lowercase letters, 10 digits, some punctuation symbols, some spacing characters, and some nonprinting control codes.

Unfortunately, the world has more letters than ASCII provides. You could have a hot dog at a diner, but never a Gewürztraminer[1] at a café. Many attempts have been made to cram more letters and symbols into eight bits, and you'll see them at times. Just a couple of those include:

- *Latin-1*, or *ISO 8859-1*
- Windows code page *1252*

Each of these uses all eight bits, but even that's not enough, especially when you need non-European languages. *Unicode* is an ongoing international standard to define the characters of all the world's languages, plus symbols from mathematics and other fields. And emojis!

> *Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.*
>
> —The Unicode Consortium

The Unicode Code Charts page has links to all the currently defined character sets with images. The latest version (12.0) defines more than 137,000 characters, each with a unique name and identification number. Python 3.8 handles all of these. The characters are divided into eight-bit sets called *planes*. The first 256 planes are the *basic multilingual planes*. See the Wikipedia page about Unicode planes for details.

Python 3 Unicode Strings

If you know the Unicode ID or name for a character, you can use it in a Python string. Here are some examples:

- A `\u` followed by *four* hex numbers[2] specifies a character in one of Unicode's 256 basic multilingual planes. The first two are the plane number (`00` to `FF`), and the next two are the index of the character within the plane. Plane `00` is good old ASCII, and the character positions within that plane are the same as ASCII.
- For characters in the higher planes, we need more bits. The Python escape sequence for these is `\U` followed by *eight* hex characters; the leftmost ones need to be `0`.
- For all characters, `\N{`*name*`}` lets you specify it by its standard *name*. The Unicode Character Name Index page lists these.

The Python `unicodedata` module has functions that translate in both directions:

- `lookup()` —Takes a case-insensitive name and returns a Unicode character
- `name()` —Takes a Unicode character and returns an uppercase name

In the following example, we'll write a test function that takes a Python Unicode character, looks up its name, and looks up the character again

from the name (it should match the original character):

```
>>> def unicode_test(value):
...     import unicodedata
...     name = unicodedata.name(value)
...     value2 = unicodedata.lookup(name)
...     print('value="%s", name="%s", value2="%s"' % (value, name, value2))
...
```

Let's try some characters, beginning with a plain ASCII letter:

```
>>> unicode_test('A')
value="A", name="LATIN CAPITAL LETTER A", value2="A"
```

ASCII punctuation:

```
>>> unicode_test('$')
value="$", name="DOLLAR SIGN", value2="$"
```

A Unicode currency character:

```
>>> unicode_test('\u00a2')
value="¢", name="CENT SIGN", value2="¢"
```

Another Unicode currency character:

```
>>> unicode_test('\u20ac')
value="€", name="EURO SIGN", value2="€"
```

The only problem you could potentially run into is limitations in the font you're using to display text. Few fonts have images for all Unicode characters, and might display some placeholder character for missing ones. For instance, here's the Unicode symbol for `SNOWMAN`, like symbols in dingbat fonts:

```
>>> unicode_test('\u2603')
```

```
value="☃", name="SNOWMAN", value2="☃"
```

Suppose that we want to save the word `café` in a Python string. One way is to copy and paste it from a file or website and hope that it works:

```
>>> place = 'café'
>>> place
'café'
```

This worked because I copied and pasted from a source that used UTF-8 encoding (which we look at in a few pages) for its text.

How can we specify that final `é` character? If you look at the character index for E, you see that the name `E WITH ACUTE, LATIN SMALL LETTER` has the value `00E9`. Let's check with the `name()` and `lookup()` functions that we were just playing with. First give the code to get the name:

```
>>> unicodedata.name('\u00e9')
'LATIN SMALL LETTER E WITH ACUTE'
```

Next, give the name to look up the code:

```
>>> unicodedata.lookup('E WITH ACUTE, LATIN SMALL LETTER')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: "undefined character name 'E WITH ACUTE, LATIN SMALL LETTER'"
```

NOTE

The names listed on the Unicode Character Name Index page were reformatted to make them sort nicely for display. To convert them to their real Unicode names (the ones that Python uses), remove the comma and move the part of the name that was after the comma to the beginning. Accordingly, change `E WITH ACUTE, LATIN SMALL LETTER` to `LATIN SMALL LETTER E WITH ACUTE`:

```
>>> unicodedata.lookup('LATIN SMALL LETTER E WITH ACUTE')
'é'
```

Now we can specify the string `café` by code or by name:

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> place = 'caf\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> place
'café'
```

In the preceding snippet, we inserted the é directly in the string, but we can also build a string by appending:

```
>>> u_umlaut = '\N{LATIN SMALL LETTER U WITH DIAERESIS}'
>>> u_umlaut
'ü'
>>> drink = 'Gew' + u_umlaut + 'rztraminer'
>>> print('Now I can finally have my', drink, 'in a', place)
Now I can finally have my Gewürztraminer in a café
```

The string `len()` function counts Unicode *characters*, not bytes:

```
>>> len('$')
1
>>> len('\U0001f47b')
1
```

NOTE

If you know the Unicode numeric ID, you can use the standard `ord()` and `chr()` functions to quickly convert between integer IDs and single-character Unicode strings:

```
>>> chr(233)
'é'
>>> chr(0xe9)
'é'
>>> chr(0x1fc6)
'ῆ'
```

UTF-8

You don't need to worry about how Python stores each Unicode character when you do normal string processing.

However, when you exchange data with the outside world, you need a couple of things:

- A way to *encode* character strings to bytes
- A way to *decode* bytes to character strings

If there were fewer than 65,536 characters in Unicode, we could stuff each Unicode character ID into two bytes. Unfortunately, there are more. We could encode every ID into four bytes, but that would increase the memory and disk storage space needs for common text strings by four times.

Ken Thompson and Rob Pike, whose names will be familiar to Unix developers, designed the *UTF-8* dynamic encoding scheme one night on a placemat in a New Jersey diner. It uses one to four bytes per Unicode character:

- One byte for ASCII
- Two bytes for most Latin-derived (but not Cyrillic) languages
- Three bytes for the rest of the basic multilingual plane

- Four bytes for the rest, including some Asian languages and symbols

UTF-8 is the standard text encoding in Python, Linux, and HTML. It's fast, complete, and works well. If you use UTF-8 encoding throughout your code, life will be much easier than trying to hop in and out of various encodings.

NOTE

If you create a Python string by copying and pasting from another source such as a web page, be sure the source is encoded in the UTF-8 format. It's *very* common to see text that was encoded as Latin-1 or Windows 1252 copied into a Python string, which causes an exception later with an invalid byte sequence.

Encode

You *encode* a *string* to *bytes*. The string `encode()` function's first argument is the encoding name. The choices include those presented in Table 12-1.

Table 12-1. Encodings

Encoding name	Description
`'ascii'`	Good old seven-bit ASCII
`'utf-8'`	Eight-bit variable-length encoding, and what you almost always want to use
`'latin-1'`	Also known as ISO 8859-1
`'cp-1252'`	A common Windows encoding
`'unicode-escape'`	Python Unicode literal format, `\u`*xxxx* or `\U`*xxxxxxxx*

You can encode anything as UTF-8. Let's assign the Unicode string `'\u2603'` to the name `snowman`:

```
>>> snowman = '\u2603'
```

`snowman` is a Python Unicode string with a single character, regardless of how many bytes might be needed to store it internally:

```
>>> len(snowman)
1
```

Next, let's encode this Unicode character to a sequence of bytes:

```
>>> ds = snowman.encode('utf-8')
```

As I mentioned earlier, UTF-8 is a variable-length encoding. In this case, it used three bytes to encode the single `snowman` Unicode character:

```
>>> len(ds)
3
>>> ds
b'\xe2\x98\x83'
```

Now, `len()` returns the number of bytes (3) because `ds` is a `bytes` variable.

You can use encodings other than UTF-8, but you'll get errors if the Unicode string can't be handled by the encoding. For example, if you use the `ascii` encoding, it will fail unless your Unicode characters happen to be valid ASCII characters, as well:

```
>>> ds = snowman.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\u2603'
in position 0: ordinal not in range(128)
```

The `encode()` function takes a second argument to help you avoid encoding exceptions. Its default value, which you can see in the previous example, is `'strict'`; it raises a `UnicodeEncodeError` if it sees a non-ASCII character. There are other encodings. Use `'ignore'` to throw away anything that won't encode:

```
>>> snowman.encode('ascii', 'ignore')
b''
```

Use `'replace'` to substitute `?` for unknown characters:

```
>>> snowman.encode('ascii', 'replace')
b'?'
```

Use `'backslashreplace'` to produce a Python Unicode character string, like `unicode-escape`:

```
>>> snowman.encode('ascii', 'backslashreplace')
b'\\u2603'
```

You would use this if you needed a printable version of the Unicode escape sequence.

Use `'xmlcharrefreplace'` to make HTML-safe strings:

```
>>> snowman.encode('ascii', 'xmlcharrefreplace')
b'&#9731;'
```

I provide more details on HTML conversion in "HTML Entities".

Decode

We *decode* byte strings to Unicode text strings. Whenever we get text from some external source (files, databases, websites, network APIs, and so on), it's encoded as byte strings. The tricky part is knowing which encoding was actually used, so we can *run it backward* and get Unicode strings.

The problem is that nothing in the byte string itself says what encoding was used. I mentioned the perils of copying and pasting from websites earlier. You've probably visited websites with odd characters where plain old ASCII characters should be.

Let's create a Unicode string called `place` with the value `'café'`:

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> type(place)
<class 'str'>
```

Encode it in UTF-8 format in a `bytes` variable called `place_bytes`:

```
>>> place_bytes = place.encode('utf-8')
>>> place_bytes
b'caf\xc3\xa9'
>>> type(place_bytes)
<class 'bytes'>
```

Notice that `place_bytes` has five bytes. The first three are the same as ASCII (a strength of UTF-8), and the final two encode the `'é'`. Now let's decode that byte string back to a Unicode string:

```
>>> place2 = place_bytes.decode('utf-8')
>>> place2
'café'
```

This worked because we encoded to UTF-8 and decoded from UTF-8.
What if we told it to decode from some other encoding?

```
>>> place3 = place_bytes.decode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3:
ordinal not in range(128)
```

The ASCII decoder threw an exception because the byte value `0xc3` is illegal in ASCII. There are some 8-bit character set encodings in which values between 128 (hex `80`) and 255 (hex `FF`) are legal but not the same as UTF-8:

```
>>> place4 = place_bytes.decode('latin-1')
>>> place4
'cafÃ©'
>>> place5 = place_bytes.decode('windows-1252')
>>> place5
'cafÃ©'
```

Urk.

The moral of this story: whenever possible, use UTF-8 encoding. It works, is supported everywhere, can express every Unicode character, and is quickly decoded and encoded.

NOTE

Even though you can specify any Unicode character, that doesn't mean that your computer will display all of them. That depends on the *font* that you're using, which may display nothing or some fill-in image for many characters. Apple created the Last Resort Font for the Unicode Consortium, and uses it in its own operating systems. This Wikipedia page has a few more details. Another font with everything between `\u0000` and `\uffff`, and a few more, is Unifont.

HTML Entities

Python 3.4 added another way to convert to and from Unicode but using HTML *character entities*.[3] This may be easier to use than looking up Unicode names, especially if you're working on the web:

```
>>> import html
>>> html.unescape("&egrave;")
'è'
```

This conversion also works with numbered entities, decimal or hex:

```
>>> import html
>>> html.unescape("&#233;")
'é'
>>> html.unescape("&#xe9;")
'é'
```

You can even import the named entity translations as a dictionary and do the conversion yourself. Drop the initial `'&'` for the dictionary key (you can also drop the final `';'`, but it seems to work either way):

```
>>> from html.entities import html5
>>> html5["egrave"]
'è'
>>> html5["egrave;"]
'è'
```

To go the other direction (from a single Python Unicode character to an HTML entity name), first get the decimal value of the character with `ord()`:

```
>>> import html
>>> char = '\u00e9'
>>> dec_value = ord(char)
>>> html.entities.codepoint2name[dec_value]
'eacute'
```

For Unicode strings with more than one character, use this two-step conversion:

```
>>> place = 'caf\u00e9'
>>> byte_value = place.encode('ascii', 'xmlcharrefreplace')
>>> byte_value
b'caf&#233;'
>>> byte_value.decode()
'caf&#233;'
```

The expression `place.encode('ascii', 'xmlcharrefreplace')` returned ASCII characters but as type `bytes` (because it *en*coded). The following `byte_value.decode()` is needed to convert `byte_value` to an HTML-compatible string.

Normalization

Some Unicode characters can be represented by more than one Unicode encoding. They'll look the same, but won't compare the same because they have different internal byte sequences. For example, take the acute accented `'é'` in `'café'`. Let's make a single-character `'é'` in multiple ways:

```
>>> eacute1 = 'é'                                # UTF-8, pasted
>>> eacute2 = '\u00e9'                           # Unicode code point
>>> eacute3 = \                                  # Unicode name
...     '\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> eacute4 = chr(233)                           # decimal byte value
>>> eacute5 = chr(0xe9)                          # hex byte value
>>> eacute1, eacute2, eacute3, eacute4, eacute5
('é', 'é', 'é', 'é', 'é')
>>> eacute1 == eacute2 == eacute3 == eacute4 == eacute5
True
```

Try a few sanity checks:

```
>>> import unicodedata
>>> unicodedata.name(eacute1)
'LATIN SMALL LETTER E WITH ACUTE'
>>> ord(eacute1)              # as a decimal integer
233
>>> 0xe9                      # Unicode hex integer
233
```

Now let's make an accented `e` by combining a plain `e` with an acute accent:

```
>>> eacute_combined1 = "e\u0301"
```

```
>>> eacute_combined2 = "e\N{COMBINING ACUTE ACCENT}"
>>> eacute_combined3 = "e" + "\u0301"
>>> eacute_combined1, eacute_combined2, eacute_combined3
('é', 'é', 'é'))
>>> eacute_combined1 == eacute_combined2 == eacute_combined3
True
>>> len(eacute_combined1)
2
```

We built a Unicode character from two characters, and it looks the same as the original `'é'`. But as they say on Sesame Street, one of these things is not like the other:

```
>>> eacute1 == eacute_combined1
False
```

If you had two different Unicode text strings from different sources, one using `eacute1` and another `eacute_combined1`, they would appear the same, but would mysteriously not act the same.

You can fix this with the `normalize()` function in the `unicodedata` module:

```
>>> import unicodedata
>>> eacute_normalized = unicodedata.normalize('NFC', eacute_combined1)
>>> len(eacute_normalized)
1
>>> eacute_normalized == eacute1
True
>>> unicodedata.name(eacute_normalized)
'LATIN SMALL LETTER E WITH ACUTE'
```

That `'NFC'` means *normal form, composed.*

For More Information

If you would like to learn more about Unicode, these links are particularly helpful:

- Unicode HOWTO
- Pragmatic Unicode
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

Text Strings: Regular Expressions

Chapter 5 discussed simple string operations. Armed with that introductory information, you've probably used simple "wildcard" patterns on the command line, such as the UNIX command `ls *.py`, which means *list all filenames ending in .py*.

It's time to explore more complex pattern matching by using *regular expressions*. These are provided in the standard module `re`, which we'll import. You define a string *pattern* that you want to match, and the *source* string to match against. For simple matches, usage looks like this:

```
>>> import re
>>> result = re.match('You', 'Young Frankenstein')
```

Here, `'You'` is the *pattern* we're looking for, and `'Young Frankenstein'` is the *source* (the string we want to search). `match()` checks whether the *source* begins with the *pattern*.

For more complex matches, you can *compile* your pattern first to speed up the match later:

```
>>> import re
>>> youpattern = re.compile('You')
```

Then, you can perform your match against the compiled pattern:

```
>>> import re
>>> result = youpattern.match('Young Frankenstein')
```

NOTE

Because this is a common Python gotcha, I'll say it again here: `match()` only matches a pattern starting at the *beginning* of the source. `search()` matches a pattern *anywhere* in the source.

`match()` is not the only way to compare the pattern and source. Here are several other methods you can use (we discuss each in the following sections):

- `search()` returns the first match, if any.
- `findall()` returns a list of all non-overlapping matches, if any.
- `split()` splits *source* at matches with *pattern* and returns a list of the string pieces.
- `sub()` takes another *replacement* argument, and changes all parts of *source* that are matched by *pattern* to *replacement*.

NOTE

Most of the regular expression examples here use ASCII, but Python's string functions, including regular expressions, work with any Python string and any Unicode characters.

Find Exact Beginning Match with match()

Does the string `'Young Frankenstein'` begin with the word `'You'`? Here's some code with comments:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.match('You', source)  # match starts at the beginning of source
>>> if m:  # match returns an object; do this to see what matched
...     print(m.group())
...
You
>>> m = re.match('^You', source) # start anchor does the same
>>> if m:
```

```
...     print(m.group())
...
You
```

How about `'Frank'`?

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.match('Frank', source)
>>> if m:
...     print(m.group())
...
```

This time, `match()` returned nothing, so the `if` did not run the `print` statement.

As I mentioned in "New: I Am the Walrus", in Python 3.8 you can shorten this example with the so-called *walrus operator*:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> if m := re.match('Frank', source):
...     print(m.group())
...
```

Okay, now let's use `search()` to see whether `'Frank'` is anywhere in the source string:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.search('Frank', source)
>>> if m:
...      print(m.group())
...
Frank
```

Let's change the pattern and try a beginning match with `match()` again:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.match('.*Frank', source)
>>> if m:  # match returns an object
...     print(m.group())
...
Young Frank
```

Here's a brief explanation of how our new `'.*Frank'` pattern works:

- `.` means *any single character*.
- `*` means *zero or more of the preceding thing*. Together, `.*` mean *any number of characters* (even zero).
- `Frank` is the phrase that we wanted to match, somewhere.

`match()` returned the string that matched `.*Frank`: `'Young Frank'`.

Find First Match with search()

You can use `search()` to find the pattern `'Frank'` anywhere in the source string `'Young Frankenstein'`, without the need for the `.*` wildcards:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.search('Frank', source)
>>> if m:  # search returns an object
...     print(m.group())
...
Frank
```

Find All Matches with findall()

The preceding examples looked for one match only. But what if you want to know how many instances of the single-letter string `'n'` are in the string?

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.findall('n', source)
>>> m   # findall returns a list
['n', 'n', 'n', 'n']
>>> print('Found', len(m), 'matches')
Found 4 matches
```

How about `'n'` followed by any character?

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.findall('n.', source)
>>> m
['ng', 'nk', 'ns']
```

Notice that it did not match that final `'n'`. We need to say that the character after `'n'` is optional, with `?`:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.findall('n.?', source)
>>> m
['ng', 'nk', 'ns', 'n']
```

Split at Matches with split()

The next example shows you how to split a string into a list by a pattern rather than a simple string (as the normal string `split()` method would do):

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.split('n', source)
>>> m    # split returns a list
['You', 'g Fra', 'ke', 'stei', '']
```

Replace at Matches with sub()

This is like the string `replace()` method, but for patterns rather than literal strings:

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.sub('n', '?', source)
>>> m   # sub returns a string
'You?g Fra?ke?stei?'
```

Patterns: Special Characters

Many descriptions of regular expressions start with all the details of how to define them. I think that's a mistake. Regular expressions are a not-so-little language in their own right, with too many details to fit in your head at once. They use so much punctuation that they look like cartoon characters swearing.

With these expressions (`match()`, `search()`, `findall()`, and `sub()`) under your belt, let's get into the details of building them. The patterns you make apply to any of these functions.

You've seen the basics:

- Literal matches with any nonspecial characters
- Any single character except `\n` with `.`
- Any number of the preceding character (including zero) with `*`
- Optional (zero or one) of the preceding character with `?`

First, special characters are shown in Table 12-2.

Table 12-2. Special characters

Pattern	Matches
`\d`	A single digit
`\D`	A single nondigit
`\w`	An alphanumeric character
`\W`	A non-alphanumeric character
`\s`	A whitespace character
`\S`	A nonwhitespace character
`\b`	A word boundary (between a `\w` and a `\W`, in either order)
`\B`	A nonword boundary

The Python `string` module has predefined string constants that we can use for testing. Let's use `printable`, which contains 100 printable ASCII characters, including letters in both cases, digits, space characters, and punctuation:

```
>>> import string
>>> printable = string.printable
>>> len(printable)
100
>>> printable[0:50]
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN'
>>> printable[50:]
'OPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

Which characters in `printable` are digits?

```
>>> re.findall('\d', printable)
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Which characters are digits, letters, or an underscore?

```
>>> re.findall('\w', printable)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b',
'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z', '_']
```

Which are spaces?

```
>>> re.findall('\s', printable)
[' ', '\t', '\n', '\r', '\x0b', '\x0c']
```

In order, those were: plain old space, tab, newline, carriage return, vertical tab, and form feed.

Regular expressions are not confined to ASCII. A `\d` will match whatever Unicode calls a digit, not just ASCII characters `'0'` through `'9'`. Let's add two non-ASCII lowercase letters from FileFormat.info:

In this test, we'll throw in the following:

- Three ASCII letters
- Three punctuation symbols that should *not* match a `\w`
- A Unicode *LATIN SMALL LETTER E WITH CIRCUMFLEX* (\u00ea)
- A Unicode *LATIN SMALL LETTER E WITH BREVE* (\u0115)

```
>>> x = 'abc' + '-/*' + '\u00ea' + '\u0115'
```

As expected, this pattern found only the letters:

```
>>> re.findall('\w', x)
['a', 'b', 'c', 'ê', 'ĕ']
```

Patterns: Using Specifiers

Now let's make "punctuation pizza," using the main pattern specifiers for regular expressions, which are presented in Table 12-3.

In the table, *expr* and the other italicized words mean any valid regular expression.

Table 12-3. Pattern specifiers

Pattern	Matches
`abc`	Literal `abc`
`(` *expr* `)`	*expr*
expr1 `|` *expr2*	*expr1* or *expr2*
`.`	Any character except `\n`
`^`	Start of source string
`$`	End of source string
prev `?`	Zero or one *prev*
prev `*`	Zero or more *prev*, as many as possible
prev `*?`	Zero or more *prev*, as few as possible
prev `+`	One or more *prev*, as many as possible
prev `+?`	One or more *prev*, as few as possible
prev `{` *m* `}`	*m* consecutive *prev*
prev `{` *m*, *n* `}`	*m* to *n* consecutive *prev*, as many as possible
prev `{` *m*, *n* `}?`	*m* to *n* consecutive *prev*, as few as possible
`[` *abc* `]`	`a` or `b` or `c` (same as `a|b|c`)
`[^` *abc* `]`	*not* (`a` or `b` or `c`)
prev `(?=` *next* `)`	*prev* if followed by *next*

Pattern	Matches
prev `(?!` *next* `)`	*prev* if *not* followed by *next*
`(?<=` *prev* `)` *next*	*next* if preceded by *prev*
`(?<!` *prev* `)` *next*	*next* if *not* preceded by *prev*

Your eyes might cross permanently when trying to read these examples. First, let's define our source string:

```
>>> source = '''I wish I may, I wish I might
... Have a dish of fish tonight.'''
```

Now we apply different regular expression pattern strings to try to match something in the `source` string.

NOTE

In the following examples, I use plain quoted strings for the patterns. A little later in this section I show how a raw pattern string (`r` before the initial quote) helps avoid some conflicts between Python's normal string escapes and regular expression ones. So, to be safest, the first argument in all the following examples should actually be a raw string.

First, find `wish` anywhere:

```
>>> re.findall('wish', source)
['wish', 'wish']
```

Next, find `wish` or `fish` anywhere:

```
>>> re.findall('wish|fish', source)
['wish', 'wish', 'fish']
```

Find `wish` at the beginning:

```
>>> re.findall('^wish', source)
[]
```

Find `I wish` at the beginning:

```
>>> re.findall('^I wish', source)
['I wish']
```

Find `fish` at the end:

```
>>> re.findall('fish$', source)
[]
```

Finally, find `fish tonight.` at the end:

```
>>> re.findall('fish tonight.$', source)
['fish tonight.']
```

The characters `^` and `$` are called *anchors*: `^` anchors the search to the beginning of the search string, and `$` anchors it to the end. `.$` matches any character at the end of the line, including a period, so that worked. To be more precise, we should escape the dot to match it literally:

```
>>> re.findall('fish tonight\.$', source)
['fish tonight.']
```

Begin by finding `w` or `f` followed by `ish`:

```
>>> re.findall('[wf]ish', source)
['wish', 'wish', 'fish']
```

Find one or more runs of `w`, `s`, or `h`:

```
>>> re.findall('[wsh]+', source)
['w', 'sh', 'w', 'sh', 'h', 'sh', 'sh', 'h']
```

Find `ght` followed by a non-alphanumeric:

```
>>> re.findall('ght\W', source)
['ght\n', 'ght.']
```

Find `I` followed by `wish`:

```
>>> re.findall('I (?=wish)', source)
['I ', 'I ']
```

And last, `wish` preceded by `I`:

```
>>> re.findall('(?<=I) wish', source)
[' wish', ' wish']
```

I mentioned earlier that there are a few cases in which the regular expression pattern rules conflict with the Python string rules. The following pattern should match any word that begins with `fish`:

```
>>> re.findall('\bfish', source)
[]
```

Why doesn't it? As is discussed in Chapter 5, Python employs a few special *escape characters* for strings. For example, `\b` means backspace in strings, but in the mini-language of regular expressions it means the beginning of a word. Avoid the accidental use of escape characters by using Python's *raw strings* when you define your regular expression string. Always put an `r` character before your regular expression pattern string, and Python escape characters will be disabled, as demonstrated here:

```
>>> re.findall(r'\bfish', source)
['fish']
```

Patterns: Specifying match() Output

When using `match()` or `search()`, all matches are returned from the result object `m` as `m.group()`. If you enclose a pattern in parentheses, the match will be saved to its own group, and a tuple of them will be available as `m.groups()`, as shown here:

```
>>> m = re.search(r'(. dish\b).*(\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
```

If you use this pattern `(?P< name > expr )`, it will match *expr*, saving the match in group *name*:

```
>>> m = re.search(r'(?P<DISH>. dish\b).*(?P<FISH>\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
>>> m.group('DISH')
'a dish'
>>> m.group('FISH')
'fish'
```

Binary Data

Text data can be challenging, but binary data can be, well, interesting. You need to know about concepts such as *endianness* (how your computer's processor breaks data into bytes) and *sign bits* for integers. You might need to delve into binary file formats or network packets to extract or even change data. This section shows you the basics of binary data wrangling in Python.

bytes and bytearray

Python 3 introduced the following sequences of eight-bit integers, with possible values from 0 to 255, in two types:

- *bytes* is immutable, like a tuple of bytes
- *bytearray* is mutable, like a list of bytes

Beginning with a list called `blist`, this next example creates a `bytes` variable called `the_bytes` and a `bytearray` variable called `the_byte_array`:

```
>>> blist = [1, 2, 3, 255]
>>> the_bytes = bytes(blist)
>>> the_bytes
b'\x01\x02\x03\xff'
>>> the_byte_array = bytearray(blist)
>>> the_byte_array
bytearray(b'\x01\x02\x03\xff')
```

NOTE

The representation of a `bytes` value begins with a `b` and a quote character, followed by hex sequences such as `\x02` or ASCII characters, and ends with a matching quote character. Python converts the hex sequences or ASCII characters to little integers, but shows byte values that are also valid ASCII encodings as ASCII characters:

```
>>> b'\x61'
b'a'
```

```
>>> b'\x01abc\xff'
b'\x01abc\xff'
```

This next example demonstrates that you can't change a `bytes` variable:

```
>>> blist = [1, 2, 3, 255]
>>> the_bytes = bytes(blist)
>>> the_bytes[1] = 127
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

But a `bytearray` variable is mellow and mutable:

```
>>> blist = [1, 2, 3, 255]
>>> the_byte_array = bytearray(blist)
>>> the_byte_array
bytearray(b'\x01\x02\x03\xff')
>>> the_byte_array[1] = 127
>>> the_byte_array
bytearray(b'\x01\x7f\x03\xff')
```

Each of these would create a 256-element result, with values from 0 to 255:

```
>>> the_bytes = bytes(range(0, 256))
>>> the_byte_array = bytearray(range(0, 256))
```

When printing `bytes` or `bytearray` data, Python uses `\x` *xx* for non-printable bytes and their ASCII equivalents for printable ones (plus some common escape characters, such as `\n` instead of `\x0a`). Here's the printed representation of `the_bytes` (manually reformatted to show 16 bytes per line):

```
>>> the_bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f
\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~\x7f
\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f
\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf
\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf
```

```
\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf
\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf
\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef
\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff'
```

This can be confusing, because they're bytes (teeny integers), not characters.

Convert Binary Data with struct

As you've seen, Python has many tools for manipulating text. Tools for binary data are much less prevalent. The standard library contains the `struct` module, which handles data similar to *structs* in C and C++. Using `struct`, you can convert binary data to and from Python data structures.

Let's see how this works with data from a PNG file—a common image format that you'll see along with GIF and JPEG files. We'll write a small program that extracts the width and height of an image from some PNG data.

We'll use the O'Reilly logo—the little bug-eyed tarsier shown in Figure 12-1.

Figure 12-1. The O'Reilly tarsier

The PNG file for this image is available on Wikipedia. I don't show how to read files until Chapter 14, so I downloaded this file, wrote a little program to print its values as bytes, and just typed the values of the first 30 bytes into a Python `bytes` variable called `data` for the example that follows. (The PNG format specification says that the width and height are stored within the first 24 bytes, so we don't need more than that for now.)

```
>>> import struct
```

```
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> if data[:8] == valid_png_header:
...     width, height = struct.unpack('>LL', data[16:24])
...     print('Valid PNG, width', width, 'height', height)
... else:
...     print('Not a valid PNG')
...
Valid PNG, width 154 height 141
```

Here's what this code does:

- `data` contains the first 30 bytes from the PNG file. To fit on the page, I joined two byte strings with `+` and the continuation character (`\`).
- `valid_png_header` contains the eight-byte sequence that marks the start of a valid PNG file.
- `width` is extracted from bytes 16–19, and `height` from bytes 20–23.

The `>LL` is the format string that instructs `unpack()` how to interpret its input byte sequences and assemble them into Python data types. Here's the breakdown:

- The `>` means that integers are stored in *big-endian* format.
- Each `L` specifies a four-byte unsigned long integer.

You can examine each four-byte value directly:

```
>>> data[16:20]
b'\x00\x00\x00\x9a'
>>> data[20:24]0x9a
b'\x00\x00\x00\x8d'
```

Big-endian integers have the most significant bytes to the left. Because the width and height are each less than 255, they fit into the last byte of each sequence. You can verify that these hex values match the expected decimal values:

```
>>> 0x9a
154
>>> 0x8d
141
```

When you want to go in the other direction and convert Python data to bytes, use the `struct pack()` function:

```
>>> import struct
>>> struct.pack('>L', 154)
b'\x00\x00\x00\x9a'
>>> struct.pack('>L', 141)
b'\x00\x00\x00\x8d'
```

Tables 12-4 and 12-5 show the format specifiers for `pack()` and `unpack()`.

The endian specifiers go first in the format string.

Table 12-4. Endian specifiers

Specifier	Byte order
<	Little endian
>	Big endian

Table 12-5. Format specifiers

Specifier	Description	Bytes
`x`	Skip a byte	1
`b`	Signed byte	1
`B`	Unsigned byte	1
`h`	Signed short integer	2
`H`	Unsigned short integer	2
`i`	Signed integer	4
`I`	Unsigned integer	4
`l`	Signed long integer	4
`L`	Unsigned long integer	4
`Q`	Unsigned long long integer	8
`f`	Single-precision float	4
`d`	Double-precision float	8
`p`	*count* and characters	1 + *count*
`s`	Characters	*count*

The type specifiers follow the endian character. Any specifier may be preceded by a number that indicates the *`count`*; `5B` is the same as `BBBBB`.

You can use a *`count`* prefix instead of `>LL`:

```
>>> struct.unpack('>2L', data[16:24])
(154, 141)
```

We used the slice `data[16:24]` to grab the interesting bytes directly. We could also use the `x` specifier to skip the uninteresting parts:

```
>>> struct.unpack('>16x2L6x', data)
(154, 141)
```

This means:

- Use big-endian integer format (`>`)
- Skip 16 bytes (`16x`)
- Read eight bytes—two unsigned long integers (`2L`)
- Skip the final six bytes (`6x`)

Other Binary Data Tools

Some third-party open source packages offer the following, more-declarative ways of defining and extracting binary data:

- bitstring
- construct
- hachoir
- binio
- kaitai struct

Appendix B has details on how to download and install external packages such as these. For the next example, you need to install `construct`. Here's all you need to do:

```
$ pip install construct
```

Here's how to extract the PNG dimensions from our `data` bytestring by using `construct`:

```
>>> from construct import Struct, Magic, UBInt32, Const, String
>>> # adapted from code at https://github.com/construct
>>> fmt = Struct('png',
...     Magic(b'\x89PNG\r\n\x1a\n'),
...     UBInt32('length'),
...     Const(String('type', 4), b'IHDR'),
...     UBInt32('width'),
...     UBInt32('height')
...     )
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> result = fmt.parse(data)
>>> print(result)
Container:
    length = 13
    type = b'IHDR'
    width = 154
    height = 141
>>> print(result.width, result.height)
154, 141
```

Convert Bytes/Strings with binascii()

The standard `binascii` module has functions to convert between binary data and various string representations: hex (base 16), base 64, uuencoded, and others. For example, in the next snippet, let's print that eight-byte PNG header as a sequence of hex values, instead of the mixture of ASCII and `\x` *xx* escapes that Python uses to display *bytes* variables:

```
>>> import binascii
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> print(binascii.hexlify(valid_png_header))
b'89504e470d0a1a0a'
```

Hey, this thing works backward, too:

```
>>> print(binascii.unhexlify(b'89504e470d0a1a0a'))
b'\x89PNG\r\n\x1a\n'
```

Bit Operators

Python provides bit-level integer operators, similar to those in the C language. Table 12-6 summarizes them and includes examples with the integer variables `x` (decimal `5`, binary `0b0101`) and `y` (decimal `1`, binary `0b0001`).

Table 12-6. Bit-level integer operators

Operator	Description	Example	Decimal result	Binary result
`&`	And	`x & y`	`1`	`0b0001`
`|`	Or	`x | y`	`5`	`0b0101`
`^`	Exclusive or	`x ^ y`	`4`	`0b0100`
`~`	Flip bits	`~x`	`-6`	*binary representation depends on int size*
`<<`	Left shift	`x << 1`	`10`	`0b1010`
`>>`	Right shift	`x >> 1`	`2`	`0b0010`

These operators work something like the set operators in Chapter 8. The `&` operator returns bits that are the same in both arguments, and `|` returns bits that are set in either of them. The `^` operator returns bits that are in one or the other, but not both. The `~` operator reverses all the bits in its single argument; this also reverses the sign because an integer's highest bit indicates its sign (`1` = negative) in *two's complement* arithmetic, used in all modern computers. The `<<` and `>>` operators just move bits to the left or right. A left shift of one bit is the same as multiplying by two, and a right shift is the same as dividing by two.

A Jewelry Analogy

Unicode strings are like charm bracelets, and bytes are like strands of beads.

Coming Up

Next is another practical chapter: how to handle dates and times.

Things to Do

12.1 Create a Unicode string called `mystery` and assign it the value `'\U0001f984'`. Print `mystery` and its Unicode name.

12.2 Encode `mystery`, this time using UTF-8, into the `bytes` variable `pop_bytes`. Print `pop_bytes`.

12.3 Using UTF-8, decode `pop_bytes` into the string variable `pop_string`. Print `pop_string`. Is `pop_string` equal to `mystery`?

12.4 When you're working with text, regular expressions come in very handy. We'll apply them in a number of ways to our featured text sample. It's a poem titled "Ode on the Mammoth Cheese," written by James McIntyre in 1866 in homage to a seven-thousand-pound cheese that was crafted in Ontario and sent on an international tour. If you'd rather not type all of it, use your favorite search engine and cut and paste the words into your Python program, or just grab it from Project Gutenberg. Call the text string `mammoth`.

Example 12-1. mammoth.txt

```
We have seen thee, queen of cheese,
Lying quietly at your ease,
Gently fanned by evening breeze,
Thy fair form no flies dare seize.

All gaily dressed soon you'll go
To the great Provincial show,
To be admired by many a beau
In the city of Toronto.

Cows numerous as a swarm of bees,
Or as the leaves upon the trees,
It did require to make thee please,
And stand unrivalled, queen of cheese.

May you not receive a scar as
We have heard that Mr. Harris
Intends to send you off as far as
The great world's show at Paris.

Of the youth beware of these,
For some of them might rudely squeeze
And bite your cheek, then songs or glees
We could not sing, oh! queen of cheese.

We'rt thou suspended from balloon,
You'd cast a shade even at noon,
Folks would think it was the moon
About to fall and crush them soon.
```

12.5 Import the `re` module to use Python's

regular expression functions. Use the `re.findall()` to print all the words that begin with `c`.

12.6 Find all four-

let-
ter
words
that
be-
gin
with
`c`.

12.7
Find
all
the
words
that
end
with
`r`.

12.8
Find
all
words
that
con-
tain
ex-
actly
three
vow-
els
in
a
row.

12.9
Use

u
n
h
e
x
l
i
f
y
to
con-
vert
this
hex
string
(com-
bined
from
two
strings
to
fit
on
a
page)
to
a
b
y
t
e
s
vari-
able
called
g
i

```
f:

   '4749463839610100010080000000000ffffff21f9' +
   '0401000000002c000000000100010000020144003b'
```

12.10 The bytes in `gif` define a one-pixel transparent GIF file, one of the most common graphics file formats. A legal GIF

starts with the ASCII characters *GIF89a*. Does `gif` match this?

12.11 The pixel width of a GIF is a 16-bit little-endian integer beginning at

byte off- set 6, and the height is the same size, start- ing at off- set 8. Extract and print these val- ues for `gif`. Are they both `1`?

This wine has

an
um-
laut
in
Germany,
but
loses
it
in
Alsace
on
the
way
to
France.

Base
16,
spec-
i-
fied
with
char-
ac-
ters
`0`-
`9`
and
`A`-
`F`.

See
the
HTML5
named-
char-
ac-
ter
ref-

er-
ence
chart.