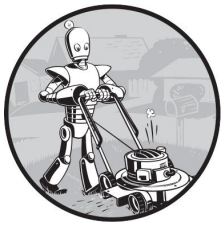
9

TEXT PATTERN MATCHING WITH REGULAR EXPRESSIONS



You may be familiar with the process of searching for text by pressing CTRL-F and entering the words you're looking for. *Regular expressions* go one step further: they allow you to specify a pattern of text to search for. You may not know a business's exact phone number, but if you live in the United States or Canada, you know it will consist of a three-digit area code, followed by a hyphen, then three more digits, another hyphen, and four more digits. This is how you, as a human, know a phone number when you see it: 415-555-1234 is a phone number, but $4,155,551,234 is not.

We recognize all sorts of other text patterns every day: email addresses have @ symbols in the middle, US Social Security numbers have nine digits and two hyphens, website URLs often have periods and forward slashes, news headlines use title case, and social media hashtags begin with # and contain no spaces, to give some examples.

Regular expressions are helpful, but few nonprogrammers know about them, even though most modern text editors and word processors have find-and-replace features that can search based on regular expressions. Regular expressions are huge time-savers, not just for software users but also for programmers. In fact, in the *Guardian* article "Here's What ICT Should Really Teach Kids: How to Do Regular Expressions," tech writer Cory Doctorow argues that we should be teaching regular expressions before we teach programming:

> Knowing [regular expressions] can mean the difference between solving a problem in 3 steps and solving it in 3,000 steps. When you're a nerd, you forget that the problems you solve with a couple

keystrokes can take other people days of tedious, error-prone work to slog through.

In this chapter, you'll start by writing a program to find text patterns *without* using regular expressions and then learn how to use regular expressions to make the code simpler. I'll show you basic matching with regular expressions, then move on to some more powerful features, such as string substitution and creating your own character classes. You'll also learn how to use the Humre module, which offers plain-English substitutes for regular expressions' cryptic symbol-based syntax.

Finding Text Patterns Without Regular Expressions

Say you want to find a US phone number in a string; you're looking for three numbers, a hyphen, three numbers, a hyphen, and four numbers. Here's an example: 415-555-4242.

Let's write a function named `is_phone_number()` to check whether a string matches this pattern and return either `True` or `False`. Open a new file editor tab and enter the following code, then save the file as *isPhoneNumber.py*:

```
def is_phone_number(text):

❶ if len(text) != 12:  # Phone numbers have
exactly 12 characters.
        return False
    for i in range(0, 3):  # The first three
characters must be numbers.
      ❷ if not text[i].isdecimal():
            return False
  ❸ if text[3] != '-':  # The fourth
character must be a dash.
        return False
    for i in range(4, 7): # The next three
characters must be numbers.
      ❹ if not text[i].isdecimal():
            return False
  ❺ if text[7] != '-':  # The eighth
```

```
    character must be a dash.
            return False
        for i in range(8, 12):  # The next four
    characters must be numbers.
            ❻ if not text[i].isdecimal():
                return False
        ❼ return True

    print('Is 415-555-4242 a phone number?',
    is_phone_number('415-555-4242'))
    print(is_phone_number('415-555-4242'))
    print('Is Moshi moshi a phone number?',
    is_phone_number('Moshi moshi'))
    print(is_phone_number('Moshi moshi'))
```

When this program is run, the output looks like this:

```
Is 415-555-4242 a phone number?
True
Is Moshi moshi a phone number?
False
```

The `is_phone_number()` function has code that does several checks to determine whether the string in `text` is a valid phone number. If any of these checks fail, the function returns `False`. First, the code checks that the string is exactly 12 characters long ❶. Then, it checks that the area code (that is, the first three characters in `text`) consists of only numeric characters ❷ by calling the `isdecimal()` string method. The rest of the function checks that the string follows the pattern of a phone number: the number must have the first hyphen after the area code ❸, three more numeric characters ❹, another hyphen ❺, and finally, four more numeric characters ❻. If the program execution manages to get past all the checks, it returns `True` ❼.

Calling `is_phone_number()` with the argument `'415-555-4242'` will return `True`. Calling

`is_phone_number()` with `'Moshi moshi'` will return `False`; the first test fails because `'Moshi moshi'` is not 12 characters long.

If you wanted to find a phone number within a larger string, you would have to add even more code to locate the pattern. Replace the last four `print()` function calls in *isPhoneNumber.py* with the following:

```
message = 'Call me at 415-555-1011 tomorrow.
415-555-9999 is my office.'
for i in range(len(message)):
  ❶ segment = message[i:i+12]
  ❷ if is_phone_number(segment):
        print('Phone number found: ' +
segment)
print('Done')
```

When this program is run, the output will look like this:

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

On each iteration of the `for` loop, a new segment of 12 characters from `message` is assigned to the variable `segment` ❶. For example, on the first iteration, `i` is `0`, and `segment` is assigned `message[0:12]` (that is, the string `'Call me at 4'`). On the next iteration, `i` is `1`, and `segment` is assigned `message[1:13]` (the string `'all me at 41'`). In other words, on each iteration of the `for` loop, `segment` takes on the following values

```
'Call me at 4'
'all me at 41'
'll me at 415'
'l me at 415-'
```

and so on, until its last value is `'s my office.'`

The loop's code passes `segment` to `is_phone_number()` to check whether it matches the phone number pattern ❷, and if so, it prints the segment. Once it has finished going through `message`, we print `Done`.

While the string in `message` is short in this example, the program would run in less than a second even if it were millions of characters long. A similar program that finds phone numbers using regular expressions would also run in less than a second; however, regular expressions make writing these programs much quicker.

Finding Text Patterns with Regular Expressions

The previous phone number–finding program works, but it uses a lot of code to do something limited. The `is_phone_number()` function is 17 lines but can find only one phone number format. What about a phone number formatted like 415.555.4242 or (415) 555-4242? And what if the phone number had an extension, like 415-555-4242 x99? The `is_phone_number()` function would fail to find them. You could add yet more code for these additional patterns, but there is an easier way to tackle the problem.

Regular expressions, called *regexes* for short, are a sort of mini language that describes a pattern of text. For example, the characters `\d` in a regex stand for a decimal numeral between 0 and 9. Python uses the regex string `r'\d\d\d-\d\d\d-\d\d\d\d'` to match the same text pattern the previous `is_phone_number()` function did: a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers. Any other string would not match the `r'\d\d\d-\d\d\d-\d\d\d\d'` regex.

Regular expressions can be much more sophisticated than this one. For example, adding a numeral, such as 3, in curly brackets (`{3}`) after a pattern is like saying, "Match this pattern three times." So the slightly shorter regex `r'\d{3}-\d{3}-\d{4}'` also matches the phone number pattern.

Note that we often write regex strings as raw strings, with the `r` prefix. This is useful, as regex strings often have backslashes. Without using raw strings, we would have to enter expressions such as `'\\d'`.

Before we cover all of the details of regular expression syntax, let's go over how to use them in Python. We'll stick with the example regular expression string `r'\d{3}-\d{3}-\d{4}'` used to find US phone numbers in a text string `'My number is 415-555-4242'`. The general process of using regular expressions in Python involves four steps:

1. Import the `re` module.
2. Pass the regex string to `re.compile()` to get a `Pattern` object.
3. Pass the text string to the `Pattern` object's `search()` method to get a `Match` object.
4. Call the `Match` object's `group()` method to get the string of the matched text.

   In the interactive shell, these steps look like this:

```
>>> import re
>>> phone_num_pattern_obj =
re.compile(r'\d{3}-\d{3}-\d{4}')
>>> match_obj =
phone_num_pattern_obj.search('My number is
415-555-4242.')
>>> match_obj.group()
'415-555-4242'
```

All regex functions in Python are in the `re` module. Most of the examples in this chapter will require the `re` module, so remember to import it at the beginning of the program. Otherwise, you'll get a `NameError: name 're' is not defined` error message. As with importing any module, you need to import it only once per program or interactive shell session.

Passing the regular expression string to `re.compile()` returns a `Pattern` object. You only need to compile the `Pattern` object once; after that, you can call the `Pattern` object's `search()` method for as many different text strings as you want.

A `Pattern` object's `search()` method searches the string it is passed for any matches to the regex. The `search()` method will return `None` if the regex pattern isn't found in the string. If the pattern *is* found, the `search()` method returns a `Match` object, which will have a `group()` method that returns a string of the matched text.

NOTE

While I encourage you to enter the example code into the interactive shell, you could also make use of web-based regular expression testers, which can show you exactly how a regex matches a piece of text that you enter. I recommend the testers at https://pythex.org *and* https://regex101.com. *Different programming languages have slightly different*

regular expression syntax, so be sure to select the "Python" flavor on these websites.

The Syntax of Regular Expressions

Now that you know the basic steps for creating and finding regular expression objects using Python, you're ready to learn the full range of regular expression syntax. In this section, you'll learn how to group regular expression elements together with parentheses, escape special characters, match several alternative groups with the pipe character, and return all matches with the `findall()` method.

Grouping with Parentheses

Say you want to separate one smaller part of the matched text, such as the area code, from the rest of the phone number (to, for example, perform some operation on it). Adding parentheses will create *groups* in the regex string: `r'(\d\d\d)-(\d\d\d-\d\d\d\d)'`. Then, you can use the `group()` method of `Match` objects to grab the matching text from just one group.

The first set of parentheses in a regex string will be group `1`. The second set will be group `2`. By passing the integer `1` or `2` to the `group()` method, you can grab different parts of the matched text. Passing `0` or nothing to the `group()` method will return the entire matched text. Enter the following into the interactive shell:

```
>>> import re
>>> phone_re = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phone_re.search('My number is 415-555-4242.')
>>>
mo.group(1)  # Returns the first group of the matched text
'415'
>>> mo.group(2)  # Returns the second group of the matched text
'555-4242'
>>> mo.group(0)  # Returns the full matched text
```

```
'415-555-4242'
>>> mo.group()  # Also returns the full
matched text
'415-555-4242'
```

If you would like to retrieve all the groups at once, use the `groups()` method (note the plural form in the name):

```
>>> mo.groups()
('415', '555-4242')
>>> area_code, main_number = mo.groups()
>>> print(area_code)
415
>>> print(main_number)
555-4242
```

Because `mo.groups()` returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable, as in the previous `area_code, main_number = mo.groups()` line.

Using Escape Characters

Parentheses create groups in regular expressions and are not interpreted as part of the text pattern. So, what do you do if you need to match a parenthesis in your text? For instance, maybe the phone numbers you are trying to match have the area code set in parentheses: `'(415) 555-4242'`.

In this case, you need to escape the `(` and `)` characters with a backslash. The `\(` and `\)` escaped parentheses will be interpreted as part of the pattern you are matching. Enter the following into the interactive shell:

```
>>> pattern = re.compile(r'(\(\d\d\d\))
(\d\d\d-\d\d\d\d)')
>>> mo = pattern.search('My phone number is
(415) 555-4242.')
>>> mo.group(1)
```

```
'(415)'
>>> mo.group(2)
'555-4242'
```

The `\(` and `\)` escape characters in the raw string passed to `re.compile()` will match actual parenthesis characters. In regular expressions, the following characters have special meanings:

```
$ () * + - . ? [\] ^ {|}
```

If you want to detect these characters as part of your text pattern, you need to escape them with a backslash:

```
\$ \(\) \* \+ \- \. \? \[\\ \] \^ \{\| \}
```

Always double-check that you haven't mistaken escaped parentheses `\(` and `\)` for unescaped parentheses `(` and `)` in a regular expression. If you receive an error message about "missing)" or "unbalanced parenthesis," you may have forgotten to include the closing unescaped parenthesis for a group, like in this example:

```
>>> import re
>>> re.compile(r'(\(Parentheses\)')
Traceback (most recent call last):
--snip--
re.error: missing), unterminated subpattern
at position 0
```

The error message tells you that there is an opening parenthesis at index `0` of the `r'(\(Parentheses\)'` string that is missing its corresponding closing parenthesis. Using the Humre module described later in this chapter helps prevent these kinds of typos.

Matching Characters from Alternate Groups

The `|` character is called a *pipe*, and it's used as the *alternation operator* in regular expressions. You can use it anywhere you want to match one of multiple expressions. For example, the regular expression `r'Cat|Dog'` will match either `'Cat'` or `'Dog'`.

You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings `'Caterpillar'`, `'Catastrophe'`, `'Catch'`, or `'Category'`. Since all of these strings start with `Cat`, it would be nice if you could specify that prefix only once. You can do this by using the pipe within parentheses to separate the possible suffixes. Enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'Cat(erpillar|astrophe|ch|egory)')
>>> match = pattern.search('Catch me if you can.')
>>> match.group()
'Catch'
>>> match.group(1)
'ch'
```

The method call `match.group()` returns the full matched text `'Catch'`, while `match.group(1)` returns just the part of the matched text inside the first parentheses group, `'ch'`. By using the pipe character and grouping parentheses, you can specify several alternative patterns you would like your regex to match.

If you need to match an actual pipe character, escape it with a backslash, like `\|`.

Returning All Matches

In addition to a `search()` method, `Pattern` objects have a `findall()` method. While `search()` will return a `Match` object of the *first* matched text in the searched string, the `findall()` method will return the strings of *every* match in the searched string.

There is one detail you need to keep in mind when using `findall()`. The method returns a list of strings *as long as there are*

no groups in the regular expression. Enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'\d{3}-\d{3}-
\d{4}')  # This regex has no groups.
>>>
pattern.findall('Cell: 415-555-9999 Work:
212-555-0000')
['415-555-9999', '212-555-0000']
```

If there *are* groups in the regular expression, then `findall()` will return a list of tuples. Each tuple represents a single match, and the tuple has strings for each group in the regex. To see this behavior in action, enter the following into the interactive shell (and notice that the regular expression being compiled now has groups in parentheses):

```
>>> import re
>>> pattern = re.compile(r'(\d{3})-(\d{3})-
(\d{4})')  # This regex has groups.
>>>
pattern.findall('Cell: 415-555-9999 Work:
212-555-0000')
[('415', '555', '9999'), ('212', '555',
'0000')]
```

Also keep in mind that `findall()` doesn't overlap matches. For example, matching three numbers with the regex string `r'\d{3}'` matches the first three numbers in `'1234'` but not the last three:

```
>>> import re
>>> pattern = re.compile(r'\d{3}')
>>> pattern.findall('1234')
['123']
>>> pattern.findall('12345')
```

```
['123']
>>> pattern.findall('123456')
['123', '456']
```

Because the first three digits in `'1234'` have been matched as `'123'`, the digits `'234'` won't be included in further matches, even though they fit the `r'\d{3}'` pattern.

Qualifier Syntax: What Characters to Match

Regular expressions are split into two parts: the *qualifiers* that dictate what characters you are trying to match followed by the *quantifiers* that dictate how many characters you are trying to match. In the `r'\d{3}-\d{3}-\d{4}'` phone number regex string example we've been using, the `r'\d'` and `'-'` parts are qualifiers and the `'{3}'` and `'{4}'` are quantifiers. Let's now examine the syntax of qualifiers.

Using Character Classes and Negative Character Classes

Although you can define a single character to match, as we've done in the previous examples, you can also define a set of characters to match inside square brackets. This set is called a *character class*. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase. It's the equivalent of writing `a|e|i|o|u|A|E|I|O|U`, but it's easier to type. Enter the following into the interactive shell:

```
>>> import re
>>> vowel_pattern =
re.compile(r'[aeiouAEIOU]')
>>> vowel_pattern.findall('RoboCop eats BABY
FOOD.')
['o', 'o', 'o', 'e', 'a', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class `[a-zA-Z0-9]` will match all lowercase letters, uppercase letters, and numbers.

Note that, inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape characters such as parentheses inside the square brackets if you want to match literal parentheses. For example, the character class `[()]` will match either an open or close parenthesis. You do not need to write this as `[\(\)]`.

By placing a caret character (`^`) just after the character class's opening bracket, you can make a *negative character class*. A negative character class will match all the characters that are *not* in the character class. For example, enter the following into the interactive shell:

```
>>> import re
>>> consonant_pattern =
re.compile(r'[^aeiouAEIOU]')
>>> consonant_pattern.findall('RoboCop eats
BABY FOOD.')
['R', 'b', 'C', 'p', ' ', 't', 's', ' ', 'B',
'B', 'Y', ' ', 'F', 'D', '.']
```

Now, instead of matching every vowel, we're matching every character that isn't a vowel. Keep in mind that this includes spaces, newlines, punctuation characters, and numbers.

Using Shorthand Character Classes

In the earlier phone number regex example, you learned that `\d` could stand for any numeric digit. That is, `\d` is shorthand for the regular expression `0|1|2|3|4|5|6|7|8|9` or `[0-9]`. There are many such *shorthand character classes*, as shown in Table 9-1.

Table 9-1: Shorthand Codes for Common Character Classes

Shorthand character class	Represents ...
`\d`	Any numeric digit from 0 to 9.
`\D`	Any character that is *not* a numeric digit from 0 to 9.
`\w`	Any letter, numeric digit, or the underscore character. (Think of this as matching "word" characters.)
`\W`	Any character that is *not* a letter, numeric digit, or the underscore character.

Shorthand character class	Represents ...
`\s`	Any space, tab, or newline character. (Think of this as matching "space" characters.)
`\S`	Any character that is *not* a space, tab, or newline character.

Note that while `\d` matches digits and `\w` matches digits, letters, and the underscore, there is no shorthand character class that matches only letters. Though you can use the `[a-zA-Z]` character class, this character class won't match accented letters or non-Roman alphabet letters such as `'é'`. Also, remember to use raw strings to escape the backslash: `r'\d'`.

For example, enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'\d+\s\w+')
>>> pattern.findall('12 drummers, 11 pipers,
10 lords, 9 ladies, 8 maids,
7 swans, 6 geese, 5 rings, 4 birds, 3 hens, 2
doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9
ladies', '8 maids', '7 swans', '
6 geese', '5 rings', '4 birds', '3 hens', '2
doves', '1 partridge']
```

The regular expression `\d+\s\w+` will match text that has one or more numeric digits (`\d+`), followed by a whitespace character (`\s`), followed by one or more letter/digit/underscore characters (`\w+`). The `findall()` method returns all matching strings of the regular expression pattern in a list.

Matching Everything with the Dot Character

The `.` (or *dot*) character in a regular expression string matches any character except for a newline. For example, enter the following into the interactive shell:

```
>>> import re
>>> at_re = re.compile(r'.at')
>>> at_re.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Remember that the dot character will match just one character, which is why the text `flat` in the previous example matched only `lat`. To match an actual period, escape the dot with a backslash: `\.`

Being Careful What You Match For

The best and worst thing about regular expressions is that they will match exactly what you ask for. Here are some common points of confusion regarding character classes:

- The `[A-Z]` or `[a-z]` character class matches uppercase or lowercase letters, respectively, but not both. You need to use `[A-Za-z]` to match both cases.
- The `[A-Za-z]` character class matches only plain, unaccented letters. For example, the regex string `r'First Name: ([A-Za-z]+)'` would match "First Name: " followed by a group of one or more unaccented letters. But singer Sinéad O'Connor's first name would match up to the *é* only, and the group would be set to `'Sin'`.
- The `\w` character class matches all letters, including accented letters and characters from other alphabets. But it also matches numbers and the underscore character, so the regex string `r'First Name: (\w+)'` may match more than you intended.
- The `\w` character class matches all letters, but the regex string `r'Last Name: (\w+)'` would capture Sinéad O'Connor's last name only up until the apostrophe character. This means the group would capture her last name as `'O'`.
- Straight and smart quote characters (`' " ‘ ’ “ ”`) are considered completely different from each other and must be specified separately.

Real-world data is complicated. Even if your program manages to capture Sinéad O'Connor's name, it could fail with Jean-Paul Sartre's name because of the hyphen.

Of course, when software declares a name to be invalid input, it is the software, and not the name, that has a bug; people's names cannot be

invalid. You can learn more about this issue from Patrick McKenzie's article "Falsehoods Programmers Believe About Names" at *https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/*. This article spawned a genre of similar "falsehoods programmers believe" pieces about how software mishandles dates, time zones, currencies, postal addresses, genders, airport codes, and love. Watch Carina C. Zona's 2015 PyCon talk on the topic, "Schemas for the Real World," at *https://youtu.be/PYYfVqtcWQY*.

Quantifier Syntax: How Many Qualifiers to Match

In a regular expression string, quantifiers follow qualifier characters to dictate how many of them to match. For example, in the phone number regex considered earlier, the `{3}` follows the `\d` to match exactly three digits. If there is no quantifier following a qualifier, the qualifier must appear exactly once: you can think of `r'\d'` as being the same as `r'\d{1}'`.

Matching an Optional Pattern

Sometimes you may want to match a pattern only optionally. That is, the regex should match zero or one of the preceding qualifiers. The `?` character flags the preceding qualifier as optional. For example, enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'42!?')
>>> pattern.search('42!')
<re.Match object; span=(0, 3), match='42!'>
>>> pattern.search('42')
<re.Match object; span=(0, 2), match='42'>
```

The `?` part of the regular expression means that the pattern `!` is optional. So it matches both `42!` (with the exclamation mark) and `42` (without it).

As you're beginning to see, regular expression syntax's reliance on symbols and punctuation makes it tricky to read: the `?` question mark has meaning in regex syntax, but the `!` exclamation mark doesn't. So `r'42!?'` means `'42'` optionally followed by a `'!'`, but `r'42?!'` means `'4'` optionally followed by `'2'` followed by `'!'`:

```
>>> import re
>>> pattern = re.compile(r'42?!')
>>> pattern.search('42!')
<re.Match object; span=(0, 3), match='42!'>
>>> pattern.search('4!')
<re.Match object; span=(0, 2), match='4!'>
>>> pattern.search('42') == None  # No match
True
```

To make multiple characters optional, place them in a group and put the `?` after the group. In the earlier phone number example, you can use `?` to make the regex look for phone numbers that either do or do not have an area code. Enter the following into the interactive shell:

```
>>> pattern = re.compile(r'(\d{3}-)?\d{3}-
\d{4}')
>>> match1 = pattern.search('My number is
415-555-4242')
>>> match1.group()
'415-555-4242'

>>> match2 = pattern.search('My number is
555-4242')
>>> match2.group()
'555-4242'
```

You can think of the `?` as saying, "Match zero or one of the group preceding this question mark."

If you need to match an actual question mark character, escape it with `\?`.

Matching Zero or More Qualifiers

The `*` (called the *star* or *asterisk*) means "match zero or more." In other words, the qualifier that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again. Take a look at the following example:

```
>>> import re
>>> pattern = re.compile('Eggs(and spam)*')
>>> pattern.search('Eggs')
<re.Match object; span=(0, 4), match='Eggs'>
>>> pattern.search('Eggs and spam')
<re.Match object; span=(0, 13), match='Eggs
and spam'>
>>> pattern.search('Eggs and spam and spam')
<re.Match object; span=(0, 22), match='Eggs
and spam and spam'>
>>> pattern.search('Eggs and spam and spam
and spam')
<re.Match object; span=(0, 31), match='Eggs
and spam and spam and spam'>
```

While the `'Eggs'` part of the string must appear once, there can be any number of `' and spam'` following it, including zero instances.

If you need to match an actual star character, prefix the star in the regular expression with a backslash, `\*`.

Matching One or More Qualifiers

While `*` means "match zero or more," the + (or *plus*) means "match one or more." Unlike the star, which does not require its qualifier to appear in the matched string, the plus requires the qualifier preceding it to appear *at least once*. It is not optional. Enter the following into the interactive shell, and compare it with the star regexes in the previous section:

```
>>> pattern = re.compile('Eggs(and spam)+')
>>> pattern.search('Eggs and spam')
<re.Match object; span=(0, 13), match='Eggs
and spam'>
>>> pattern.search('Eggs and spam and spam')
<re.Match object; span=(0, 22), match='Eggs
and spam and spam'>
>>> pattern.search('Eggs and spam and spam
```

```
and spam')
<re.Match object; span=(0, 31), match='Eggs
and spam and spam and spam'>
```

The regex `'Eggs(and spam)+'` will not match the string `'Eggs'`, because the plus sign requires at least one `' and spam'`.

You'll often use parentheses in your regex strings to group together qualifiers so that a quantifier can apply to the entire group. For example, you could match any combination of dots and dashes of Morse code with `r'(\.|\-)+'` (though this expression would also match invalid Morse code combinations).

If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: `\+`.

Matching a Specific Number of Qualifiers

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex `(Ha){3}` will match the string `'HaHaHa'` but not `'HaHa'`, since the latter has only two repeats of the `(Ha)` group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex `(Ha){3,5}` will match `'HaHaHa'`, `'HaHaHaHa'`, and `'HaHaHaHaHa'`.

You can also leave out the first or second number in the curly brackets to keep the minimum or maximum unbounded. For example, `(Ha){3,}` will match three or more instances of the `(Ha)` group, while `(Ha){,5}` will match zero to five instances. Curly brackets can help make your regular expressions shorter. These two regular expressions match identical patterns:

```
(Ha){3}
HaHaHa
```

So do these two regular expressions:

```
(Ha){3,5}
(HaHaHa)|(HaHaHaHa)|(HaHaHaHaHa)
```

Enter the following into the interactive shell:

```
>>> import re
>>> haRegex = re.compile(r'(Ha){3}')
>>> match1 = haRegex.search('HaHaHa')
>>> match1.group()
'HaHaHa'

>>> match = haRegex.search('HaHa')
>>> match == None
True
```

Here, `(Ha){3}` matches `'HaHaHa'` but not `'Ha'`. Because it doesn't match `'HaHa'`, `search()` returns `None`.

The syntax of the curly bracket quantifier is similar to Python's slice syntax (such as `'Hello, world!'[3:5]`, which evaluates to `'lo'`). But there are key differences. In the regex quantifier, the two numbers are separated by a comma and not a colon. Also, the second number in the quantifier is inclusive: `'(Ha){3,5}'` matches up to *and including* five instances of the `'(Ha)'` qualifier.

Greedy and Non-greedy Matching

Because `(Ha){3,5}` can match three, four, or five instances of `Ha` in the string `'HaHaHaHaHa'`, you may wonder why the `Match` object's call to `group()` in the previous curly bracket example returns `'HaHaHaHaHa'` instead of the shorter possibilities. After all, `'HaHaHa'` and `'HaHaHaHa'` are also valid matches of the regular expression `(Ha){3,5}`.

Python's regular expressions are *greedy* by default, which means that in ambiguous situations, they will match the longest string possible. The *non-greedy* (also called *lazy*) version of the curly brackets, which matches the shortest string possible, must follow the closing curly bracket with a question mark.

Enter the following into the interactive shell, and notice the difference between the greedy and non-greedy forms of the curly brackets searching the same string:

```
>>> import re
>>> greedy_pattern = re.compile(r'(Ha){3,5}')
>>> match1 =
greedy_pattern.search('HaHaHaHaHa')
>>> match1.group()
'HaHaHaHaHa'

>>> lazy_pattern = re.compile(r'(Ha){3,5}?')
>>> match2 =
lazy_pattern.search('HaHaHaHaHa')
>>> match2.group()
'HaHaHa'
```

Note that the question mark can have two meanings in regular expressions: declaring a lazy match or declaring an optional qualifier. These meanings are entirely unrelated.

It's worth pointing out that, technically, you could get by without using the optional `?` quantifier, or even the `*` and + quantifiers:

- The `?` quantifier is the same as `{0,1}`.
- The `*` quantifier is the same as `{0,}`.
- The + quantifier is the same as `{1,}`.

However, the `?`, `*`, and + quantifiers are common shorthand.

Matching Everything

Sometimes you may want to match everything and anything. For example, say you want to match the string `'First Name:'`, followed by any and all text, followed by `'Last Name:'` and any text once again. You can use the dot-star (`.*`) to stand in for that "anything." Remember that the dot character means "any single character except the newline," and the star character means "zero or more of the preceding character."

Enter the following into the interactive shell:

```
>>> import re
>>> name_pattern = re.compile(r'First Name:
(.*) Last Name: (.*)')
```

```
>>> name_match = name_pattern.search('First
Name: Al Last Name: Sweigart')
>>> name_match.group(1)
'Al'
>>> name_match.group(2)
'Sweigart'
```

The dot-star uses greedy mode: it will always try to match as much text as possible. To match any and all text in a non-greedy or lazy fashion, use the dot, star, and question mark (`.*?`). As when it's used with curly brackets, the question mark tells Python to match in a non-greedy way.

Enter the following into the interactive shell to see the difference between the greedy and non-greedy expressions:

```
>>> import re
>>> lazy_pattern = re.compile(r'<.*?>')
>>> match1 = lazy_pattern.search('<To serve
man> for dinner.>')
>>> match1.group()
'<To serve man>'

>>> greedy_re = re.compile(r'<.*>')
>>>
match2 = greedy_re.search('<To serve man> for
dinner.>')
>>> match2.group()
'<To serve man> for dinner.>'
```

Both regexes roughly translate to "Match an opening angle bracket, followed by anything, followed by a closing angle bracket." But the string `'<To serve man> for dinner.>'` has two possible matches for the closing angle bracket. In the non-greedy version of the regex, Python matches the shortest possible string: `'<To serve man>'`. In the greedy version, Python matches the longest possible string: `'<To serve man> for dinner.>'`.

Matching Newline Characters

The dot in `.*` will match everything except a newline. By passing `re.DOTALL` as the second argument to `re.compile()`, you can make the dot character match *all* characters, including the newline character.

Enter the following into the interactive shell:

```
>>> import re
>>> no_newline_re = re.compile('.*')
>>> no_newline_re.search('Serve the public
trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newline_re = re.compile('.*', re.DOTALL)
>>> newline_re.search('Serve the public
trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the
innocent.\nUphold the law.'
```

The regex `no_newline_re`, which did not have `re.DOTALL` passed to the `re.compile()` call that created it, will match everything only up to the first newline character, whereas `newline_re`, which *did* have `re.DOTALL` passed to `re.compile()`, matches everything. This is why the `newline_re.search()` call matches the full string, including its newline characters.

Matching at the Start and End of a String

You can use the caret symbol (`^`) at the start of a regex to indicate that a match must occur at the *beginning* of the searched text. Likewise, you can put a dollar sign (`$`) at the end of the regex to indicate that the string must *end* with this regex pattern. And you can use the `^` and `$` together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

For example, the `r'^Hello'` regular expression string matches strings that begin with `'Hello'`. Enter the following into the interactive shell:

```
>>> import re
>>> begins_with_hello = re.compile(r'^Hello')
>>> begins_with_hello.search('Hello, world!')
<re.Match object; span=(0, 5), match='Hello'>
>>> begins_with_hello.search('He said
"Hello."') == None
True
```

The `r'\d$'` regular expression string matches strings that end with a numeric character between 0 and 9. Enter the following into the interactive shell:

```
>>> import re
>>> ends_with_number = re.compile(r'\d$')
>>> ends_with_number.search('Your number is
42')
<re.Match object; span=(16, 17), match='2'>
>>> ends_with_number.search('Your number is
forty two.') == None
True
```

The `r'^\d+$'` regular expression string matches strings that both begin and end with one or more numeric characters. Enter the following into the interactive shell:

```
>>> import re
>>> whole_string_is_num = re.compile(r'^\d+
$')
>>> whole_string_is_num.search('1234567890')
<re.Match object; span=(0, 10),
match='1234567890'>
```

```
>>> whole_string_is_num.search('12345xyz67890
') == None
True
```

The last two `search()` calls in the previous interactive shell example demonstrate how the entire string must match the regex if `^` and `$` are used. (I always confuse the meanings of these two symbols, so I use the mnemonic "carrots cost dollars" to remind myself that the caret comes first and the dollar sign comes last.)

You can also use `\b` to make a regex pattern match only on a *word boundary*: the start of a word, end of a word, or both the start and end of a word. In this case, a "word" is a sequence of letters separated by non-letter characters. For example, `r'\bcat.*?\b'` matches a word that begins with `'cat'` followed by any other characters up to the next word boundary:

```
>>> import re
>>> pattern = re.compile(r'\bcat.*?\b')
>>>
pattern.findall('The cat found a catapult
catalog in the catacombs.')
['cat', 'catapult', 'catalog', 'catacombs']
```

The `\B` syntax matches anything that is not a word boundary:

```
>>> import re
>>> pattern = re.compile(r'\Bcat\B')
>>> pattern.findall('certificate')  # Match
['cat']
>>> pattern.findall('catastrophe')  # No
match
[]
```

It is useful for finding matches in the middle of a word.

A REVIEW OF REGEX SYMBOLS

This chapter has covered a lot of notation so far, so here's a quick review of what you've learned about basic regular expression syntax:

- The `?` matches zero or one instance of the preceding qualifier.
- The `*` matches zero or more instances of the preceding qualifier.
- The `+` matches one or more instances of the preceding qualifier.
- The `{n}` matches exactly *n* instances of the preceding qualifier.
- The `{n,}` matches *n* or more instances of the preceding qualifier.
- The `{,m}` matches 0 to *m* instances of the preceding qualifier.
- The `{n,m}` matches at least *n* and at most *m* instances of the preceding qualifier.
- `{n,m}?` or `*?` or `+?` performs a non-greedy match of the preceding qualifier.
- `^spam` means the string must begin with *spam*.
- `spam$` means the string must end with *spam*.
- The `.` matches any character, except newline characters.
- The `\d`, `\w`, and `\s` match a digit, word, or space character, respectively.
- The `\D`, `\W`, and `\S` match anything except a digit, word, or space character, respectively. `[abc]` matches any character between the square brackets (such as *a*, *b*, or *c*).
- `[^abc]` matches any character that isn't between the square brackets.
- `(Hello)` groups `'Hello'` together as a single qualifier.

Case-Insensitive Matching

Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> import re
>>> pattern1 = re.compile('RoboCop')
```

```
>>> pattern2 = re.compile('ROBOCOP')
>>> pattern3 = re.compile('robOcop')
>>> pattern4 = re.compile('RobocOp')
```

But sometimes you care only about matching the letters, and aren't worried about whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`. Enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'robocop', re.I)
>>>
pattern.search('RoboCop is part man, part
machine, all cop.').group()
'RoboCop'

>>> pattern.search('ROBOCOP protects the
innocent.').group()
'ROBOCOP'

>>> pattern.search('Have you seen
robocop?').group()
'robocop'
```

The regular expression now matches strings with any casing.

Substituting Strings

Regular expressions don't merely find text patterns; they can also substitute new text in place of those patterns. The `sub()` method for `Pattern` objects accepts two arguments. The first is a string that should replace any matches. The second is the string of the regular expression. The `sub()` method returns a string with the substitutions applied.

For example, enter the following into the interactive shell to replace secret agents' names with `CENSORED`:

```
>>> import re
>>> agent_pattern = re.compile(r'Agent \w+')
>>> agent_pattern.sub('CENSORED', 'Agent
Alice contacted Agent Bob.')
'CENSORED contacted CENSORED.'
```

Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to `sub()`, you can include `\1`, `\2`, `\3`, and so on, to mean "Enter the text of group `1`, `2`, `3`, and so on, in the substitution." This syntax is called a *back reference*.

For example, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could use the regex `Agent (\w)\w*` and pass `r'\1****'` as the first argument to `sub()`:

```
>>> import re
>>> agent_pattern = re.compile(r'Agent (\w)
\w*')
>>>
agent_pattern.sub(r'\1****', 'Agent Alice
contacted Agent Bob.')
'A**** contacted B****.'
```

The `\1` in the regular expression string is replaced by whatever text was matched by group `1`—that is, the `(\w)` group of the regular expression.

Managing Complex Regexes with Verbose Mode

Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this complexity by telling the `re.compile()` function to ignore whitespace and comments inside the regular expression string. Enable this "verbose mode" by passing the variable `re.VERBOSE` as the second argument to `re.compile()`.

Now, instead of a hard-to-read regular expression like this

```
pattern = re.compile(r'((\d{3}|\(\d{3}\))?
(\s|-|\.)?\d{3}(\s|-
|\.)\d{4}(\s*(ext|x|ext\.)\s*\d{2,5})?)')
```

you can spread the regular expression over multiple lines and use comments to label its components, like this:

```
pattern = re.compile(r'''(
    (\d{3}|\(\d{3}\))?  # Area code
    (\s|-|\.)?  # Separator
    \d{3}  # First three digits
    (\s|-|\.)  # Separator
    \d{4}  # Last four digits
    (\s*(ext|x|ext\.)\s*\d{2,5})?  #
Extension
    )''', re.VERBOSE)
```

Note how the previous example uses the triple-quote syntax (`'''`) to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.

The comment rules inside the regular expression string are the same as for regular Python code: the `#` symbol and everything after it until the end of the line are ignored. Also, the extra spaces inside the multiline string for the regular expression are not considered part of the text pattern to be matched. This lets you organize the regular expression so that it's easier to read.

While verbose mode makes your regex strings more readable, I advise you to instead use the Humre module, covered later in this chapter, to improve the readability of your regular expressions.

Combining re.IGNORECASE, re.DOTALL, and re.VERBOSE

What if you want to use `re.VERBOSE` to write comments in your regular expression, but also want to use `re.IGNORECASE` to ignore capitalization? Unfortunately, the `re.compile()` function takes only a single value as its second argument.

You can get around this limitation by combining the `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE` variables using the pipe character (`|`), which in this context is known as the *bitwise or* operator. For example, if you want a regular expression that is case-insensitive *and* includes newlines to match the dot character, you would form your `re.compile()` call like this:

```
>>> some_regex = re.compile('foo',
re.IGNORECASE | re.DOTALL)
```

Including all three options in the second argument looks like this:

```
>>> some_regex = re.compile('foo',
re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

This syntax is a little old-fashioned and originates from early versions of Python. The details of the bitwise operators are beyond the scope of this book, but check out the resources at *https://nostarch.com/automate-boring-stuff-python-3rd-edition* for more information. You can also pass other options for the second argument; they're uncommon, but you can read more about them in the resources too.

Project 3: Extract Contact Information from Large Documents

Say you've been given the boring task of finding every phone number and email address in a long web page or document. If you manually scroll through the page, you might end up searching for a long time. But if you had a program that could search the text in your clipboard for phone numbers and email addresses, you could simply press CTRL-A to select all the text, press CTRL-C to copy it to the clipboard, and then run your program. It could replace the text on the clipboard with just the phone numbers and email addresses it finds.

Whenever you're tackling a new project, it can be tempting to dive right into writing code. But more often than not, it's best to take a step back and consider the bigger picture. I recommend first drawing up a high-level plan for what your program needs to do. Don't think about the actual code yet; you can worry about that later. Right now, stick to broad strokes.

For example, your phone number and email address extractor will need to do the following:

- Get the text from the clipboard.
- Find all phone numbers and email addresses in the text.
- Paste them onto the clipboard.

Now you can start thinking about how this might work in code. The code will need to do the following:

- Use the `pyperclip` module to copy and paste strings.
- Create two regexes, one for matching phone numbers and one for matching email addresses.
- Find all matches (not just the first match) of both regexes.
- Neatly format the matched strings into a single string to paste.
- Display some kind of message if no matches were found in the text.

This list is like a road map for the project. As you write the code, you can focus on each of these steps separately, and each step should seem fairly manageable. They're also expressed in terms of things you already know how to do in Python.

Step 1: Create a Regex for Phone Numbers

First, you have to create a regular expression to search for phone numbers. Create a new file, enter the following, and save it as *phoneAndEmail.py*:

```
import pyperclip, re

phone_re = re.compile(r'''(
    (\d{3}|\(\d{3}\))?  # Area code
    (\s|-|\.)?  # Separator
    (\d{3})  # First three digits
    (\s|-|\.)  # Separator
    (\d{4})  # Last four digits
    (\s*(ext|x|ext\.)\s*(\d{2,5}))?  #
Extension
    )''', re.VERBOSE)

# TODO: Create email regex.
```

```
# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

The `TODO` comments are just a skeleton for the program. They'll be replaced as you write the actual code.

The phone number begins with an *optional* area code, so we follow the area code group with a question mark. Since the area code can be just three digits (that is, `\d{3}`) *or* three digits within parentheses (that is, `\(\d{3}\)`), you should have a pipe joining those parts. You can add the regex comment `# Area code` to this part of the multiline string to help you remember what `(\d{3}|\(\d{3}\))?` is supposed to match.

The phone number separator character can be an *optional* space (`\s`), hyphen (`-`), or period (`.`), so we should also join these parts using pipes. The next few parts of the regular expression are straightforward: three digits, followed by another separator, followed by four digits. The last part is an optional extension made up of any number of spaces followed by `ext`, `x`, or `ext.`, followed by two to five digits.

NOTE

It's easy to get mixed up when writing regular expressions that contain groups with parentheses `()` and escaped parentheses `\(\)`. Remember to double-check that you're using the correct syntax if you get a "missing), unterminated subpattern" error message.

Step 2: Create a Regex for Email Addresses

You will also need a regular expression that can match email addresses. Make your program look like the following:

```
import pyperclip, re

phone_re = re.compile(r'''(
--snip--

# Create email regex.
email_re = re.compile(r'''(
```

```
    ❶ [a-zA-Z0-9._%+-]+  # Username
    ❷ @  # @ symbol
    ❸ [a-zA-Z0-9.-]+  # Domain name
      (\.[a-zA-Z]{2,4})  # Dot-something
      )''', re.VERBOSE)

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

The username part of the email address ❶ consists of one or more characters that can be any of the following: lowercase and uppercase letters, numbers, a dot, an underscore, a percent sign, a plus sign, or a hyphen. You can put all of these into a character class: `[a-zA-Z0-9._%+-]`.

The domain and username are separated by an `@` symbol ❷. The domain name ❸ has a slightly less permissive character class, with only letters, numbers, periods, and hyphens: `[a-zA-Z0-9.-]`. Last is the "dot-com" part (technically known as the *top-level domain*), which can really be dot-anything.

The format for email addresses has a lot of weird rules. This regular expression won't match every possible valid email address, but it will match almost any typical email address you'll encounter.

Step 3: Find All Matches in the Clipboard Text

Now that you've specified the regular expressions for phone numbers and email addresses, you can let Python's `re` module do the hard work of finding all the matches on the clipboard. The `pyperclip.paste()` function will get a string value of the text on the clipboard, and the `findall()` regex method will return a list of tuples.

Make your program look like the following:

```
import pyperclip, re

phone_re = re.compile(r'''(
--snip--
```

```
    # Find matches in clipboard text.
    text = str(pyperclip.paste())

❶ matches = []
❷ for groups in phone_re.findall(text):
        phone_num = '-'.join([groups[1],
    groups[3], groups[5]])
        if groups[6] != '':
            phone_num += ' x' + groups[6]
        matches.append(phone_num)
❸ for groups in email_re.findall(text):
        matches.append(groups[0])

    # TODO: Copy results to the clipboard.
```

There is one tuple for each match, and each tuple contains strings for each group in the regular expression. Remember that group `0` matches the entire regular expression, so the group at index `0` of the tuple is the one you are interested in.

As you can see at ❶, you'll store the matches in a list variable named `matches`. It starts off as an empty list and a couple of `for` loops. For the email addresses, you append group `0` of each match ❸. For the matched phone numbers, you don't want to just append group `0`. While the program *detects* phone numbers in several formats, you want the phone number appended to be in a single, standard format. The `phone_num` variable contains a string built from groups `1`, `3`, `5`, and `6` of the matched text ❷. (These groups are the area code, first three digits, last four digits, and extension.)

Step 4: Join the Matches into a String

Now that you have the email addresses and phone numbers as a list of strings in `matches`, you want to put them on the clipboard. The `pyperclip.copy()` function takes only a single string value, not a list of strings, so you must call the `join()` method on `matches`.

Make your program look like the following:

```
import pyperclip, re

phone_re = re.compile(r'''(
--snip--
for groups in email_re.findall(text):
    matches.append(groups[0])

# Copy results to the clipboard.
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email
addresses found.')
```

To make it easier to see that the program is working, we also print any matches you find to the terminal window. If no phone numbers or email addresses were found, the program tells the user this.

To test your program, open your web browser to the No Starch Press contact page at *https://nostarch.com/contactus* press CTRL-A to select all the text on the page, and press CTRL-C to copy it to the clipboard. When you run this program, the output should look something like this:

```
Copied to clipboard:
800-555-7240
415-555-9900
415-555-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
info@nostarch.com
```

You can modify this script to search for mailing addresses, social media handles, and many other types of text patterns.

Ideas for Similar Programs

Identifying patterns of text (and possibly substituting them with the `sub()` method) has many different potential applications. For example, you could do the following:

- Find website URLs that begin with *http://* or *https://*.
- Clean up dates in different date formats (such as 3/14/2030, 03-14-2030, and 2030/3/14) by replacing them with dates in a single, standard format.
- Remove sensitive information such as Social Security numbers or credit card numbers.
- Find common typos, such as multiple spaces between words, accidentally accidentally repeated words, or multiple exclamation marks at the ends of sentences. Those are annoying!!

Humre: A Module for Human-Readable Regexes

Code is read far more often than it's written, so it's important for your code to be readable. But the punctuation-dense syntax of regular expressions can be hard for even experienced programmers to read. To solve this, the third-party Humre Python module takes the good ideas of verbose mode even further by using human-readable, plain-English names to create readable regex code. You can install Humre by following the instructions in Appendix A.

Let's go back to the `r'\d{3}-\d{3}-\d{4}'` phone number example from the beginning of this chapter. The functions and constants in Humre can produce the same regex string with plain English:

```
>>> from humre import *
>>> phone_regex = exactly(3, DIGIT) + '-' +
exactly(3, DIGIT) + '-' + exactly(4, DIGIT)
>>> phone_regex
'\\d{3}-\\d{3}-\\d{4}'
```

Humre's constants (like `DIGIT`) contain strings, and Humre's functions (like `exactly()`) return strings. Humre doesn't replace the

re module. Rather, it produces regex strings that can be passed to re.compile():

```
>>> import re
>>> pattern = re.compile(phone_regex)
>>> pattern.search('My number is
415-555-4242')
<re.Match object; span=(13, 25),
match='415-555-4242'>
```

Humre has constants and functions for each feature of regular expression syntax. You can then concatenate the constants and returned strings like any other string. For example, here are Humre's constants for the shorthand character classes:

- DIGIT and NONDIGIT represent r'\d' and r'\D', respectively.
- WORD and NONWORD represent r'\w' and r'\W', respectively.
- WHITESPACE and NONWHITESPACE represent r'\s' and r'\S', respectively.

A common source of regex bugs is forgetting which characters need to be escaped. You can use Humre's constants instead of typing the escaped character yourself. For example, say you want to match a single-digit floating-point number with one digit after the decimal point, like '0.9' or '4.5'. However, if you use the regex string r'\d.\d', you might not realize that the dot matches a period (as in '4.5') but also matches any other character (as in '4A5').

Instead, use Humre's PERIOD constant, which contains the string r'\.'. The expression DIGIT + PERIOD + DIGIT evaluates to r'\d\.\d' and makes it much more obvious what the regex intends to match.

The following Humre constants exist for escaped characters:

```
PERIOD      OPEN_PAREN  OPEN_BRACKET     PIPE
DOLLAR_SIGN CLOSE_PAREN CLOSE_BRACKET    CARET
QUESTION_MARK   ASTERISK     OPEN_BRACE  TILDE
HASHTAG      PLUS         CLOSE_BRACE
AMPERSAND    MINUS        BACKSLASH
```

There are also constants for `NEWLINE`, `TAB`, `QUOTE`, and `DOUBLE_QUOTE`. Back references from `r'\1'` to `r'\99'` are represented as `BACK_1` to `BACK_99`.

However, you'll make the largest readability gains by using Humre's functions. Table 9-2 shows these functions and their equivalent regular expression syntax.

Table 9-2: Humre Functions

Humre function	Regex string
`group('A')`	`r'(A)'`
`optional('A')`	`r'A?'`
`either('A', 'B', 'C')`	`r'A|B|C'`
`exactly(3, 'A')`	`'A{3}'`
`between(3, 5, 'A')`	`'A{3,5}'`
`at_least(3, 'A')`	`'A{3,}'`
`at_most(3, 'A')`	`'A{,3}'`
`chars('A-Z')`	`'[A-Z]'`
`nonchars('A-Z')`	`'[^A-Z]'`
`zero_or_more('A')`	`'A*'`
`zero_or_more_lazy('A')`	`'A*?'`
`one_or_more('A')`	`'A+'`
`one_or_more_lazy('A')`	`'A+?'`
`starts_with('A')`	`'^A'`
`ends_with('A')`	`'A$'`
`starts_and_ends_with('A')`	`'^A$'`
`named_group('name', 'A')`	`'(?P<name>A)'`

Humre also has several convenience functions that combine common pairs of function calls. For example, instead of using `optional(group('A'))` to create `'(A)?'`, you can simply call `optional_group('A')`. Table 9-3 has the full list of Humre convenience functions.

Table 9-3: Humre Convenience Functions

Convenience function	Function equivalent	Regex string
`optional_group('A')`	`optional(group('A'))`	`'(A)?'`
`group_either('A')`	`group(either('A', 'B', 'C'))`	`'(A|B|C)'`

Convenience function	Function equivalent	Regex string
`exactly_group(3, 'A')`	`exactly(3, group('A'))`	`'(A){3}'`
`between_group(3, 5, 'A')`	`between(3, 5, group('A'))`	`'(A){3,5}'`
`at_least_group (3, 'A')`	`at_least(3, group('A'))`	`'(A){3,}'`
`at_most_group (3, 'A')`	`at_most(3, group('A'))`	`'(A){,3}'`
`zero_or_more_group('A')`	`zero_or_more(group('A'))`	`'(A)*'`
`zero_or_more_lazy_group('A')`	`zero_or_more_lazy(group('A'))`	`'(A)*?'`
`one_or_more_group('A')`	`one_or_more(group('A'))`	`'(A)+'`
`one_or_more_lazy_group('A')`	`one_or_more_lazy(group('A'))`	`'(A)+?'`

All of Humre's functions except `either()` and `group_either()` allow you to pass multiple strings to automatically join them. This means that calling `group(DIGIT, PERIOD, DIGIT)` produces the same regex string as `group(DIGIT + PERIOD + DIGIT)`. They both return the regex string `r'(\d\.\d)'`.

Finally, Humre has constants for common regex patterns:

`ANY_SINGLE` The `.` pattern that matches any single character (except newlines)

`ANYTHING_LAZY` The lazy `.*?` zero or more pattern

`ANYTHING_GREEDY` The greedy `.*` zero or more pattern

`SOMETHING_LAZY` The lazy `.+?` one or more pattern

`SOMETHING_GREEDY` The greedy `.+` one or more pattern

The readability of regex written with Humre becomes more obvious when you consider large, complicated regular expressions. Let's rewrite the phone number regex from the previous phone number extractor project using Humre:

```
import re
from humre import *
phone_regex = group(
    optional_group(either(exactly(3, DIGIT),
# Area code
                          OPEN_PAREN +
```

```
exactly(3, DIGIT) + CLOSE_PAREN)),
    optional(group_either(WHITESPACE, '-',
PERIOD)),  # Separator
    group(exactly(3, DIGIT)),  # First three
digits
    group_either(WHITESPACE, '-', PERIOD),  #
Separator
    group(exactly(4, DIGIT)),  # Last four
digits
    optional_group(  # Extension
      zero_or_more(WHITESPACE),
      group_either('ext', 'x', r'ext\.'),
      zero_or_more(WHITESPACE),
      group(between(2, 5, DIGIT))
      )
    )

pattern = re.compile(phone_regex)
match = pattern.search('My number is
415-555-1212.')
print(match.group())
```

When you run this program, the output is this:

```
415-555-1212
```

This code is much more verbose than even the verbose mode regex. It helps to import Humre using the `from humre import *` syntax so that you don't need to put `humre.` before every function and constant. But the length of the code doesn't matter as much as the readability.

You can switch your existing regular expressions to Humre code by calling the `humre.parse()` function, which returns a string of Python source code:

```
>>> import humre
>>> humre.parse(r'\d{3}-\d{3}-\d{4}')
"exactly(3, DIGIT) + '-' + exactly(3, DIGIT)
+ '-' + exactly(4, DIGIT)"
```

When combined with a modern editor such as PyCharm or Visual Studio Code, Humre offers several further advantages:

- You can indent your code to make it obvious which parts of the regex contain which other parts.
- Your editor's parentheses matching works.
- Your editor's syntax highlighting works.
- Your editor's linter and type hints tool picks up typos.
- Your editor's autocomplete fills in the function and constant names.
- Humre handles raw strings and escaping for you.
- You can put Python comments alongside your Humre code.
- Typos cause more helpful error messages.

Many experienced programmers will object to using anything other than the standard, complicated, unreadable regular expression syntax. As programmer Peter Bhat Harkins once said, "One of the most irritating things programmers do regularly is feel so good about learning a hard thing that they don't look for ways to make it easy, or even oppose things that would do so."

However, if a co-worker objects to your use of Humre, you can simply print the underlying regex string that your Humre code generates and put it back into your source code. For example, the contents of the `phone_regex` variable in the phone number extractor project are as follows:

```
r'((\d{3}|\(\d{3}\))?(\s|-|\.)?(\d{3})(\s|-|
\.)(\d{4})(\s*(ext|x|ext\.)\s*(\d{2,5}))?)'
```

Your co-worker is welcome to use this regular expression string if they feel it is more appropriate.

Summary

While a computer can search for text quickly, it must be told precisely what to look for. Regular expressions allow you to specify the pattern of characters you are looking for, rather than the exact text itself. In fact,

some word processing and spreadsheet applications provide find-and-replace features that allow you to search using regular expressions. The punctuation-heavy syntax of regular expressions is composed of qualifiers that detail what to match and quantifiers that detail how many to match.

The `re` module that comes with Python lets you compile a regex string into a `Pattern` object. These objects have several methods: `search()`, to find a single match; `findall()`, to find all matching instances; and `sub()`, to do a find-and-replace substitution of text.

You can find out more in the official Python documentation at *https://docs.python.org/3/library/re.html*. Another useful resource is the tutorial website *https://www.regular-expressions.info*. The Humre page on the Python Package Index is *https://pypi.org/project/Humre/*.

Practice Questions

1. What is the function that returns `Regex` objects?
2. Why are raw strings often used when creating `Regex` objects?
3. What does the `search()` method return?
4. How do you get the actual strings that match the pattern from a `Match` object?
5. In the regex created from `r'(\d\d\d)-(\d\d\d-\d\d\d\d)'`, what does group `0` cover? Group `1`? Group `2`?
6. Parentheses and periods have specific meanings in regular expression syntax. How would you specify that you want a regex to match actual parentheses and period characters?
7. The `findall()` method returns a list of strings or a list of tuples of strings. What makes it return one or the other?
8. What does the `|` character signify in regular expressions?
9. What two things does the `?` character signify in regular expressions?
10. What is the difference between the `+` and `*` characters in regular expressions?
11. What is the difference between `{3}` and `{3,5}` in regular expressions?
12. What do the `\d`, `\w`, and `\s` shorthand character classes signify in regular expressions?
13. What do the `\D`, `\W`, and `\S` shorthand character classes signify in regular expressions?
14. What is the difference between the `.*` and `.*?` regular expressions?

15. What is the character class syntax to match all numbers and lowercase letters?
16. How do you make a regular expression case-insensitive?
17. What does the `.` character normally match? What does it match if `re.DOTALL` is passed as the second argument to `re.compile()`?
18. If `num_re = re.compile(r'\d+')`, what will `num_re.sub('X', '12 drummers, 11 pipers, five rings, 3 hens')` return?
19. What does passing `re.VERBOSE` as the second argument to `re.compile()` allow you to do?

Practice Programs

For practice, write programs to do the following tasks.

Strong Password Detection

Write a function that uses regular expressions to make sure the password string it is passed is strong. A strong password has several rules: it must be at least eight characters long, contain both uppercase and lowercase characters, and have at least one digit. Hint: It's easier to test the string against multiple regex patterns than to try to come up with a single regex that can validate all the rules.

Regex Version of the strip() Method

Write a function that takes a string and does the same thing as the `strip()` string method. If no other arguments are passed other than the string to strip, then the function should remove whitespace characters from the beginning and end of the string. Otherwise, the function should remove the characters specified in the second argument to the function.