

## Pattern Matching with regular expressions

You may be familiar with searching for text by pressing ctrl-F and typing in the words you're looking for. *Regular expressions* go one step further: They allow you to specify a *pattern*

of text to search for. In this chapter, you'll start by writing a program to find text patterns *without* using regular expressions and then see how to use regular expressions to make the code much less bloated

### finding Patterns of text without regular expressions

Say you want to find a phone number in a string. You know the pattern: three numbers, a hyphen, three numbers, a hyphen, and four numbers. Here's an example: 415-555-4242.

Let's use a function named `isPhoneNumber()` to check whether a string matches this pattern, returning either `True` or `False`.

```
def isPhoneNumber(text):
    if len(text) != 12:
        return False
    for i in range(0, 3):
        if not text[i].isdecimal():
            return False
    if text[3] != '-':
        return False
    for i in range(4, 7):
        if not text[i].isdecimal():
            return False
    if text[7] != '-':
        return False
    for i in range(8, 12):
        if not text[i].isdecimal():
            return False

    return True

print('415-555-4242 is a phone number:')
print(isPhoneNumber('415-555-4242'))
print('Moshi moshi is a phone number:')
print(isPhoneNumber('Moshi moshi'))
```

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

```
415-555-4242 is a phone number:
True
Moshi moshi is a phone number:
```

False

The `isPhoneNumber()` function has code that does several checks to see 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. Then it checks that the area code (that is, the first three characters in text) consists of only numeric characters. 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, three more numeric characters, then another hyphen, and finally four more numbers. If the program execution manages to get past all the checks, it returns True

Calling `isPhoneNumber()` with the argument '415-555-4242' will return True. Calling `isPhoneNumber()` with 'Moshi moshi' will return False; the first test fails because 'Moshi moshi' is not 12 characters long.

You would have to add even more code to find this pattern of text in a larger string. 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)):
    chunk = message[i:i+12]
    if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)
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 chunk of 12 characters from message is assigned to the variable `chunk`. For example, on the first iteration, `i` is 0, and `chunk` is assigned `message[0:12]` (that is, the string 'Call me at 4'). On the next iteration, `i` is 1, and `chunk` is assigned `message[1:13]` (the string 'all me at 41').

You pass `chunk` to `isPhoneNumber()` to see whether it matches the phone number pattern, and if so, you print the chunk.

Continue to loop through message, and eventually the 12 characters in `chunk` will be a phone number. The loop goes through the entire string, testing each 12-character piece and printing any chunk it finds that satisfies `isPhoneNumber()`. Once we're done going through message, we print Done.

While the string in message is short in this example, it could be millions of characters long and the program would still run in less than a second. A similar program that finds phone numbers using regular expressions would also run in less than a second, but regular expressions make it quicker to write these programs.

## **finding Patterns of text with regular expressions**

The previous phone number-finding program works, but it uses a lot of code to do something limited: The `isPhoneNumber()` function is 17 lines but can find only one pattern of phone numbers. What about a phone number formatted like 415.555.4242 or (415) 555-4242? What if the phone number had an extension, like 415-555-4242 x99? The `isPhoneNumber()`

function would fail to validate them. You could add yet more code for these additional patterns, but there is an easier way.

Regular expressions, called *regexes* for short, are descriptions for a pattern of text. For example, a `\d` in a regex stands for a digit character—that is, any single numeral 0 to 9. The regex `\d\d\d-\d\d\d-\d\d\d\d` is used by Python to match the same text the previous `isPhoneNumber()` function did: a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers. Any other string would not match the `\d\d\d-\d\d\d-\d\d\d\d` regex.

But regular expressions can be much more sophisticated. For example, adding a 3 in curly brackets (`{3}`) after a pattern is like saying, “Match this pattern three times.” So the slightly shorter regex `\d{3}-\d{3}-\d{4}` also matches the correct phone number format.

## Creating Regex Objects

All the regex functions in Python are in the `re` module. Enter the following into the interactive shell to import this module:

```
>>> import re
```

To create a Regex object that matches the phone number pattern, enter the following into the interactive shell. (Remember that `\d` means “a digit character” and `\d\d\d-\d\d\d-\d\d\d\d` is the regular expression for the correct phone number pattern.)

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Now the `phoneNumRegex` variable contains a Regex object.

### Matching Regex Objects

A Regex 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 is not found in the string. If the pattern is found, the `search()` method returns a `Match` object. Match objects have a `group()` method that will return the actual matched text from the searched string.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
```

Phone number found: 415-555-4242

Here, we pass our desired pattern to `re.compile()` and store the resulting Regex object in `phoneNumRegex`. Then we call `search()` on `phoneNumRegex` and pass `search()` the string we want to search for a match. The result of the search gets stored in the variable `mo`. In this example, we know that our pattern will be found in the string, so we know that a `Match` object will be returned. Knowing that `mo` contains a `Match` object and not the null value `None`, we can call `group()` on `mo` to return the match. Writing `mo.group()` inside our print statement displays the whole match, 415-555-4242.

## Review of Regular Expression Matching

While there are several steps to using regular expressions in Python, each step is fairly simple.

1. Import the regex module with `import re`.
2. Create a Regex object with the `re.compile()` function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object's `search()` method. This returns a Match object.
4. Call the Match object's `group()` method to return a string of the actual matched text.

## More Pattern matching with regular expressions

### *Grouping with Parentheses*

Say you want to separate the area code from the rest of the phone number. Adding parentheses will create *groups* in the regex: `(\d\d\d)-(\d\d\d-\d\d\d\d)`. Then you can use the `group()` match object method 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()` match object method, you can grab different parts of the matched text. Passing 0 or nothing to the `group()` method will return the entire matched text.

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2) '555-4242'
>>> mo.group(0) '415-555-4242'
>>> mo.group() '415-555-4242'
```

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

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()

>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Since `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 `areaCode, mainNumber = mo.groups()` line.

Parentheses have a special meaning in regular expressions, but 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. In this case, you need to escape the `(` and `)` characters with a backslash.

```
>>> phoneNumRegex = re.compile(r'(\d\d\d) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.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.

### *Matching Multiple Groups with the Pipe*

The `|` character is called a *pipe*. You can use it anywhere you want to match one of many expressions. For example, the regular expression `r'Batman|Tina Fey'` will match either 'Batman' or 'Tina Fey'.

When *both* Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object.

```
>>> heroRegex = re.compile(r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman'
>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

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 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'. Since all these strings start with Bat, it would be nice if you could specify that prefix only once. This can be done with parentheses.

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1) 'mobile'
```

The method call `mo.group()` returns the full matched text 'Batmobile', while `mo.group(1)` returns just the part of the matched text inside the first parentheses group, 'mobile'. 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 `\\`.

## Optional Matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match whether or not that bit of text is there. The `?` character flags the group that precedes it as an optional part of the pattern. For example

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

The `(wo)?` part of the regular expression means that the pattern `wo` is an optional group. The regex will match text that has zero instances or one instance of `wo` in it. This is why the regex matches both 'Batwoman' and 'Batman'.

Using the earlier phone number example, you can make the regex look for phone numbers that do or do not have an area code.

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'
>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.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 with the Star

The `*` (called the *star* or *asterisk*) means “match zero or more”—the group that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again. Let’s look at the Batman example again.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

```
>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

For 'Batman', the (wo)\* part of the regex matches zero instances of wo in the string; for 'Batwoman', the (wo)\* matches one instance of wo; and for 'Batwowowowoman', (wo)\* matches four instances of wo.

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

### ***Matching One or More with the Plus***

While \* means “match zero or more,” the + (or *plus*) means “match one or more.” Unlike the star, which does not require its group to appear in the matched string, the group preceding a plus must 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:

```
>>> batRegex = re.compile(r'Bat(wo)+man')

>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'

>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

The regex Bat(wo)+man will not match the string 'The Adventures of Batman' because at least one wo is required by the plus sign.

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

## Matching Specific Repetitions with Curly Brackets

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 it will not match '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 leave 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}
(Ha)(Ha)(Ha)
```

And these two regular expressions also match identical patterns:

```
(Ha){3,5}
((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))
```

Example:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'
>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

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

## Greedy and nongreedy matching

Python's regular expressions are *greedy* by default, which means that in ambiguous situations they will match the longest string possible. The *non-greedy* version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

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

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'
```



```
>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

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

### the findall() method

In addition to the search() method, Regex objects also 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. To see how search() returns a Match object only on the first instance of matching text, enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

On the other hand, findall() will not return a Match object but a list of strings—*as long as there are no groups in the regular expression*. Each string in the list is a piece of the searched text that matched the regular expression.

Enter the following into the interactive shell:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.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 found match, and its items are the

matched strings for each group in the regex. To see findall() in action, enter the following into the interactive shell (notice that the regular expression being compiled now has groups in parentheses):

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '1122'), ('212', '555', '0000')]
```

To summarize what the findall() method returns, remember the following:

1. When called on a regex with no groups, such as `\d\d\d-\d\d\d-\d\d\d\d`, the method findall() returns a list of string matches, such as `['415-555-9999', '212-555-0000']`.

2. When called on a regex that has groups, such as `(\d\d\d)-(\d\d\d)-(\d\d\d\d)`, the method `findall()` returns a list of tuples of strings (one string for each group), such as `[('415', '555', '1122'), ('212', '555', '0000')]`.

## 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)`. There are many such *shorthand character classes*, as shown in Table 7-1.

**Table 7-1:** Shorthand Codes for Common Character Classes

Shorthand character class	Represents
<code>\d</code>	Any numeric digit from 0 to 9.
<code>\D</code>	Any character that is <i>not</i> a numeric digit from 0 to 9.
<code>\w</code>	Any letter, numeric digit, or the underscore character. (Think of this as matching “word” characters.)
<code>\W</code>	Any character that is <i>not</i> a letter, numeric digit, or the underscore character.
<code>\s</code>	Any space, tab, or newline character. (Think of this as matching “space” characters.)
<code>\S</code>	Any character that is <i>not</i> a space, tab, or newline.

Character classes are nice for shortening regular expressions. The character class `[0-5]` will match only the numbers 0 to 5; this is much shorter than typing `(0|1|2|3|4|5)`.

For example

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.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 regex pattern in a list.

## Making your own character classes

There are times when you want to match a set of characters but the shorthand character classes (`\d`, `\w`, `\s`, and so on) are too broad. You can define your own character class using square brackets. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase. Enter the following into the interactive shell:

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
```

```
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', '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 the `.`, `*`, `?`, or `()` characters with a preceding backslash. For example, the character class `[0-5.]` will match digits 0 to 5 and a period. You do not need to write it as `[0-5\.]`.

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,

```
>>> consonantRegex = re.compile(r'^aeiouAEIOU')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', ' ', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

Now, instead of matching every vowel, we're matching every character that isn't a vowel.

## The caret and dollar Sign characters

You can also 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 (`$`) at the end of the regex to indicate the string must *end* with this regex pattern. And you can use the `^` and `$` 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:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')

<_sre.SRE_Match object; span=(0, 5), match='Hello'>

>>> beginsWithHello.search('He said hello.') == None
```

True

The `r'\d$'` regular expression string matches strings that end with a numeric character from 0 to 9

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')

<_sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.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:

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

>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12 34567890') == 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.

use the mnemonic “Carrots cost dollars” to remind that the caret comes first and the dollar sign comes last. To avoid confusion

### The wildcard character

The `.` (or *dot*) character in a regular expression is called a *wildcard* and will match any character except for a newline. For example

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.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 match for the text `flat` in the previous example matched only `lat`. To match an actual dot, escape the dot with a backslash: `\.`

### Matching Everything with Dot-Star

Sometimes you will 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 then followed by anything 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:

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

>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')

>>> mo.group(1)
'Al'

>>> mo.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 *nongreedy* fashion, use the dot, star, and question mark (`.*`). Like with curly brackets, the question mark tells Python to match in a nongreedy way. Enter the following into the interactive shell to see the difference between the greedy and nongreedy versions:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'
>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.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 nongreedy 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 Newlines with the Dot Character*

The dot-star 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:

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

The regex `noNewlineRegex`, 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 `newlineRegex`, which *did* have `re.DOTALL` passed to `re.compile()`, matches everything. This is why the `newlineRegex.search()` call matches the full string, including its newline characters.

## review of regex Symbols

This chapter covered a lot of notation, so here's a quick review of what you learned:

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

## case-insensitive matching

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

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

But sometimes you care only about matching the letters without worrying 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:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'
>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'
>>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
'robocop'
```

## Substituting Strings with the sub() method

Regular expressions can not only find text patterns but can also substitute new text in place of those patterns. The sub() method for Regex objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. The sub() method returns a string with the substitutions applied.

For example,

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
```

'CENSORED gave the secret documents to CENSORED.'

Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to sub(), you can type \1, \2, \3, and so on, to mean “Enter the text of group 1, 2, 3, and so on, in the substitution.”

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

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob was a double agent.')
```

A\*\*\*\* told C\*\*\*\* that E\*\*\*\* knew B\*\*\*\* was a double agent.'

## managing complex regexes

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 by telling the re.compile() function to ignore whitespace and comments inside the regular expression string. This “verbose mode” can be enabled by passing the variable re.VERBOSE as the second argument to re.compile().

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

```
phoneRegex = 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 with comments like this:

```
phoneRegex = re.compile(r'(\d{3})|(\d{3}))?
(\s|-|\.)?
\d{3}
# area code
# separator
# first 3 digits
# separator
# last 4 digits
(\s|-|\.)')
```

```
\d{4}
(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 regular Python code: The # symbol and everything after it to 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 it's easier to read.

### 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.

So if you want a regular expression that's case-insensitive *and* includes newlines to match the dot character, you would form your re.compile() call like this:

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

All three options for the second argument will look like this:

```
>>> someRegexValue = 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 <http://nostarch.com/automatestuff/> 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: Phone number and email Address extractor

Say you have 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 and email address extractor will need to do the following:

- Get the text off 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 the other 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. Each step is fairly manageable and 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*:

```
#!/ python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.

import pyperclip, re

phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?           # area code
    (\s|-|\.)?                  # separator
    (\d{3})                      # first 3 digits
    (\s|-|\.)                   # separator
    (\d{4})                     # last 4 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 the area code group is followed 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 a space (`\s`), hyphen (`-`), or period (`.`), so these parts should also be joined by 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.

## **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:

```
#!/ python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.
import pyperclip, re
phoneRegex = re.compile(r'''

--snip--

# Create email regex.
emailRegex = re.compile(r'''(
[a-zA-Z0-9._%+-]+      # username
@                      # @ symbol
[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 is 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. The domain name has a slightly less permissive character class with only letters, numbers, periods, and hyphens: `[a-zA-Z0-9.-]`. And last will be the “dot-com” part (technically known as the *top-level domain*), which can really be dot-anything. This is between two and four characters.

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

## **Step 3: Find All Matches in the Clipboard Text**

Now that you have 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:

```
#!/ python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.
import pyperclip, re
phoneRegex = re.compile(r'''

--snip--

# Find matches in clipboard text.
text = str(pyperclip.paste())
matches = []
for groups in phoneRegex.findall(text):
    phoneNum = ''.join([groups[1], groups[3], groups[5]])
    if groups[8] != '':
        phoneNum += 'x' + groups[8]
    matches.append(phoneNum)
for groups in emailRegex.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, you'll store the matches in a list variable named `matches`. It starts off as an empty list, and a couple for loops. For the email addresses, you append group 0 of each match. 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 `phoneNum` variable contains a string built from groups 1, 3, 5, and 8 of the matched text. (These groups are the area code, first three digits, last four digits, and extension.)

#### ***Step 4: Join the Matches into a String for the Clipboard***

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 call the `join()` method on `matches`.

To make it easier to see that the program is working, let's print any matches you find to the terminal. And if no phone numbers or email addresses were found, the program should tell the user this.

Make your program look like the following:

```

#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.
--snip--
for groups in emailRegex.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.')

```

### ***Running the Program***

For an example, open your web browser to the No Starch Press contact page at <http://www.nostarch.com/contactus.htm>, 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 will look something like this:

```

Copied to clipboard:
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
help@nostarch.com

```

### ***Ideas for Similar Programs***

Identifying patterns of text (and possibly substituting them with the `sub()` method) has many different potential applications.

- Find website URLs that begin with *http://* or *https://*.
- Clean up dates in different date formats (such as 3/14/2015, 03-14-2015, and 2015/3/14) by replacing them with dates in a single, standard format.
- Remove sensitive information such as Social Security or credit card numbers.
- Find common typos such as multiple spaces between words, accidentally repeated words, or multiple exclamation marks at the end of sentences. Those are annoying!!

### **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 precise patterns of characters you

are looking for. In fact, some word processing and spreadsheet applications provide find-and-replace features that allow you to search using regular expressions.

The `re` module that comes with Python lets you compile Regex objects. These values 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.

There's a bit more to regular expression syntax than is described in this chapter. You can find out more in the official Python documentation at <http://docs.python.org/3/library/re.html>. The tutorial website <http://www.regular-expressions.info/> is also a useful resource.

Now that you have expertise manipulating and matching strings, it's time to dive into how to read from and write to files on your computer's hard drive.

