

CHAPTER 5. REGULAR EXPRESSIONS

“Some people, when confronted with a problem, think “I know, I’ll use regular expressions.” Now they have two problems.”


— Jamie Zawinski

DIVING IN

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

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

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

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



5.2. CASE STUDY: STREET ADDRESSES

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

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.') ①
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.') ②
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.') ③
'100 NORTH BROAD RD.'
>>> import re ④
>>> re.sub('ROAD$', 'RD.', s) ⑤
'100 NORTH BROAD RD.'
```

- ① My goal is to standardize a street address so that 'ROAD' is always abbreviated as 'RD.'. At first glance, I thought this was simple enough that I could just use the string method `replace()`. After all, all the data was already uppercase, so case mismatches would not be a problem. And the search string, 'ROAD', was a constant. And in this deceptively simple example, `s.replace()` does indeed work.
- ② Life, unfortunately, is full of counterexamples, and I quickly discovered this one. The problem here is that 'ROAD' appears twice in the address, once as part of the street name 'BROAD' and once as its own word. The `replace()` method sees these two occurrences and blindly replaces both of them; meanwhile, I see my addresses getting destroyed.
- ③ To solve the problem of addresses with more than one 'ROAD' substring, you could resort to something like this: only search and replace 'ROAD' in the last four characters of the address (`s[-4:]`), and leave the string alone (`s[:-4]`). But you can see that this is already getting unwieldy. For example, the pattern is dependent on the length of the string you're replacing. (If you were replacing 'STREET' with 'ST.', you

would need to use `s[:-6]` and `s[:-6:].replace(...)`. Would you like to come back in six months and debug this? I know I wouldn't.

- ④ It's time to move up to regular expressions. In Python, all functionality related to regular expressions is contained in the `re` module.
- ⑤ Take a look at the first parameter: `'ROAD$'`. This is a simple regular expression that matches `'ROAD'` only when it occurs at the end of a string. The `$` means “end of the string.” (There is a corresponding character, the caret `^`, which means “beginning of the string.”) Using the `re.sub()` function, you search the string `s` for the regular expression `'ROAD$'` and replace it with `'RD.'`. This matches the `ROAD` at the end of the string `s`, but does *not* match the `ROAD` that's part of the word `BROAD`, because that's in the middle of `s`.

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

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

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

- ① What I *really* wanted was to match `'ROAD'` when it was at the end of the string *and* it was its own word

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

- ② To work around the backslash plague, you can use what is called a *raw string*, by prefixing the string with the letter `r`. This tells Python that nothing in this string should be escaped; `'\t'` is a tab character, but `r'\t'` is really the backslash character `\` followed by the letter `t`. I recommend always using raw strings when dealing with regular expressions; otherwise, things get too confusing too quickly (and regular expressions are confusing enough already).
- ③ **sigh** Unfortunately, I soon found more cases that contradicted my logic. In this case, the street address contained the word `'ROAD'` as a whole word by itself, but it wasn’t at the end, because the address had an apartment number after the street designation. Because `'ROAD'` isn’t at the very end of the string, it doesn’t match, so the entire call to `re.sub()` ends up replacing nothing at all, and you get the original string back, which is not what you want.
- ④ To solve this problem, I removed the `$` character and added another `\b`. Now the regular expression reads “match `'ROAD'` when it’s a whole word by itself anywhere in the string,” whether at the end, the beginning, or somewhere in the middle.

*
**

5.3. CASE STUDY: ROMAN NUMERALS

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

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

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

The following are some general rules for constructing Roman numerals:

- Sometimes characters are additive. I is 1, II is 2, and III is 3. VI is 6 (literally, “5 and 1”), VII is 7, and VIII is 8.
- The tens characters (I, X, C, and M) can be repeated up to three times. At 4, you need to subtract from the next highest fives character. You can't represent 4 as IIII; instead, it is represented as IV (“1 less than 5”). 40 is written as XL (“10 less than 50”), 41 as XLI, 42 as XLII, 43 as XLIII, and then 44 as XLIV (“10 less than 50, then 1 less than 5”).
- Sometimes characters are... the opposite of additive. By putting certain characters before others, you subtract from the final value. For example, at 9, you need to subtract from the next highest tens character: 8 is VIII, but 9 is IX (“1 less than 10”), not VIIII (since the I character can not be repeated four times). 90 is XC, 900 is CM.
- The fives characters can not be repeated. 10 is always represented as X, never as VV. 100 is always C, never LL.
- Roman numerals are read left to right, so the order of characters matters very much. DC is 600; CD is a completely different number (400, “100 less than 500”). CI is 101; IC is not even a valid Roman numeral (because you can't subtract 1 directly from 100; you would need to write it as XCIX, “10 less than 100, then 1 less than 10”).

5.3.1. CHECKING FOR THOUSANDS

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

```
>>> import re
>>> pattern = '^M?M?M?M?$' ①
```

```

>>> re.search(pattern, 'M')      ②
<_sre.SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')     ③
<_sre.SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')    ④
<_sre.SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM')   ⑤
>>> re.search(pattern, '')       ⑥
<_sre.SRE_Match object at 0106F4A8>

```

- ① This pattern has three parts. `^` matches what follows only at the beginning of the string. If this were not specified, the pattern would match no matter where the `M` characters were, which is not what you want. You want to make sure that the `M` characters, if they're there, are at the beginning of the string. `M?` optionally matches a single `M` character. Since this is repeated three times, you're matching anywhere from zero to three `M` characters in a row. And `$` matches the end of the string. When combined with the `^` character at the beginning, this means that the pattern must match the entire string, with no other characters before or after the `M` characters.
- ② The essence of the `re` module is the `search()` function, that takes a regular expression (pattern) and a string ('M') to try to match against the regular expression. If a match is found, `search()` returns an object which has various methods to describe the match; if no match is found, `search()` returns `None`, the Python null value. All you care about at the moment is whether the pattern matches, which you can tell by just looking at the return value of `search()`. 'M' matches this regular expression, because the first optional `M` matches and the second and third optional `M` characters are ignored.
- ③ 'MM' matches because the first and second optional `M` characters match and the third `M` is ignored.
- ④ 'MMM' matches because all three `M` characters match.
- ⑤ 'MMMM' does not match. All three `M` characters match, but then the regular expression insists on the string ending (because of the `$` character), and the string doesn't end yet (because of the fourth `M`). So `search()` returns `None`.
- ⑥ Interestingly, an empty string also matches this regular expression, since all the `M` characters are optional.

5.3.2. CHECKING FOR HUNDREDS

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

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

*? makes a
pattern
optional.*

So there are four possible patterns:

- CM
- CD
- Zero to three C characters (zero if the hundreds place is 0)
- D, followed by zero to three C characters

The last two patterns can be combined:

- an optional D, followed by zero to three C characters

This example shows how to validate the hundreds place of a Roman numeral.

```
>>> import re

>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$' ①

>>> re.search(pattern, 'MCM') ②
<_sre.SRE_Match object at 01070390>

>>> re.search(pattern, 'MD') ③
<_sre.SRE_Match object at 01073A50>

>>> re.search(pattern, 'MMMCCC') ④
<_sre.SRE_Match object at 010748A8>

>>> re.search(pattern, 'MCMC') ⑤

>>> re.search(pattern, '') ⑥
<_sre.SRE_Match object at 01071D98>
```

- ① This pattern starts out the same as the previous one, checking for the beginning of the string (^), then the thousands place (M?M?M?). Then it has the new part, in parentheses, which defines a set of three mutually exclusive patterns, separated by vertical bars: CM, CD, and D?C?C?C? (which is an optional D followed by zero to three optional C characters). The regular expression parser checks for each of these patterns in order (from left to right), takes the first one that matches, and ignores the rest.
- ② 'MCM' matches because the first M matches, the second and third M characters are ignored, and the CM matches (so the CD and D?C?C?C? patterns are never even considered). MCM is the Roman numeral representation of 1900.
- ③ 'MD' matches because the first M matches, the second and third M characters are ignored, and the D?C?C?C? pattern matches D (each of the three C characters are optional and are ignored). MD is the Roman numeral representation of 1500.
- ④ 'MMMCCC' matches because all three M characters match, and the D?C?C?C? pattern matches CCC (the D is optional and is ignored). MMMCCC is the Roman numeral representation of 3300.
- ⑤ 'MCMC' does not match. The first M matches, the second and third M characters are ignored, and the CM matches, but then the \$ does not match because you're not at the end of the string yet (you still have an unmatched C character). The C does *not* match as part of the D?C?C?C? pattern, because the mutually exclusive CM pattern has already matched.
- ⑥ Interestingly, an empty string still matches this pattern, because all the M characters are optional and ignored, and the empty string matches the D?C?C?C? pattern where all the characters are optional and ignored.

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



5.4. USING THE {n,m} SYNTAX

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


```

>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')      ①
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MM')     ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMM')    ③
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM')   ④
>>>

```

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

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

```

>>> pattern = '^M{0,3}$'      ①
>>> re.search(pattern, 'M')    ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM')   ③
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM')  ④
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM') ⑤
>>>

```

- ① This pattern says: “Match the start of the string, then anywhere from zero to three M characters, then the end of the string.” The 0 and 3 can be any numbers; if you want to match at least one but no more than three M characters, you could say `M{1,3}`.
- ② This matches the start of the string, then one M out of a possible three, then the end of the string.
- ③ This matches the start of the string, then two M out of a possible three, then the end of the string.
- ④ This matches the start of the string, then three M out of a possible three, then the end of the string.
- ⑤ This matches the start of the string, then three M out of a possible three, but then *does not match* the end of the string. The regular expression allows for up to only three M characters before the end of the string, but you have four, so the pattern does not match and returns None.

5.4.1. CHECKING FOR TENS AND ONES

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

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')      ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')        ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX')        ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')      ④
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX')     ⑤
>>>
```

- ① This matches the start of the string, then the first optional M, then CM, then XL, then the end of the string. Remember, the `(A|B|C)` syntax means “match exactly one of A, B, or C”. You match XL, so you ignore the XC and `L?X?X?X?` choices, and then move on to the end of the string. MCMXL is the Roman numeral representation of 1940.
- ② This matches the start of the string, then the first optional M, then CM, then `L?X?X?X?`. Of the `L?X?X?X?`, it matches the L and skips all three optional X characters. Then you move to the end of the string. MCML is the Roman numeral representation of 1950.
- ③ This matches the start of the string, then the first optional M, then CM, then the optional L and the first

optional X, skips the second and third optional X, then the end of the string. MCMLX is the Roman numeral representation of 1960.

- ④ This matches the start of the string, then the first optional M, then CM, then the optional L and all three optional X characters, then the end of the string. MCMLXXX is the Roman numeral representation of 1980.
- ⑤ This matches the start of the string, then the first optional M, then CM, then the optional L and all three optional X characters, then *fails to match* the end of the string because there is still one more X unaccounted for. So the entire pattern fails to match, and returns None. MCMLXXXX is not a valid Roman numeral.

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

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

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

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

- ① This matches the start of the string, then one of a possible three M characters, then D?C{0,3}. Of that, it matches the optional D and zero of three possible C characters. Moving on, it matches L?X{0,3} by matching the optional L and zero of three possible X characters. Then it matches V?I{0,3} by matching the optional V and zero of three possible I characters, and finally the end of the string. MDLV is the Roman numeral representation of 1555.
- ② This matches the start of the string, then two of a possible three M characters, then the D?C{0,3} with a D and one of three possible C characters; then L?X{0,3} with an L and one of three possible X

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

characters; then $V?I\{0,3\}$ with a V and one of three possible I characters; then the end of the string. MMDCLXVI is the Roman numeral representation of 2666.

- ③ This matches the start of the string, then three out of three M characters, then $D?C\{0,3\}$ with a D and three out of three C characters; then $L?X\{0,3\}$ with an L and three out of three X characters; then $V?I\{0,3\}$ with a V and three out of three I characters; then the end of the string. MMMDCCCLXXXVIII is the Roman numeral representation of 3888, and it's the longest Roman numeral you can write without extended syntax.
- ④ Watch closely. (I feel like a magician. “Watch closely, kids, I’m going to pull a rabbit out of my hat.”) This matches the start of the string, then zero out of three M, then matches $D?C\{0,3\}$ by skipping the optional D and matching zero out of three C, then matches $L?X\{0,3\}$ by skipping the optional L and matching zero out of three X, then matches $V?I\{0,3\}$ by skipping the optional V and matching one out of three I. Then the end of the string. Whoa.

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

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



5.5. VERBOSE REGULAR EXPRESSIONS

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

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

- Whitespace is ignored. Spaces, tabs, and carriage returns are not matched as spaces, tabs, and carriage returns. They’re not matched at all. (If you want to match a space in a verbose regular expression, you’ll

need to escape it by putting a backslash in front of it.)

- Comments are ignored. A comment in a verbose regular expression is just like a comment in Python code: it starts with a # character and goes until the end of the line. In this case it's a comment within a multi-line string instead of within your source code, but it works the same way.

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

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

>>> re.search(pattern, 'M', re.VERBOSE)           ①
<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)  ②
<_sre.SRE_Match object at 0x008EEB48>

>>> re.search(pattern, 'MMMDCCLXXXVIII', re.VERBOSE) ③
<_sre.SRE_Match object at 0x008EEB48>

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

- ① The most important thing to remember when using verbose regular expressions is that you need to pass an extra argument when working with them: `re.VERBOSE` is a constant defined in the `re` module that signals that the pattern should be treated as a verbose regular expression. As you can see, this pattern has quite a bit of whitespace (all of which is ignored), and several comments (all of which are ignored). Once you ignore the whitespace and the comments, this is exactly the same regular expression as you saw in the previous section, but it's a lot more readable.
- ② This matches the start of the string, then one of a possible three M, then CM, then L and three of a possible three X, then IX, then the end of the string.

- ③ This matches the start of the string, then three of a possible three M, then D and three of a possible three C, then L and three of a possible three X, then V and three of a possible three I, then the end of the string.
- ④ This does not match. Why? Because it doesn't have the `re.VERBOSE` flag, so the `re.search` function is treating the pattern as a compact regular expression, with significant whitespace and literal hash marks. Python can't auto-detect whether a regular expression is verbose or not. Python assumes every regular expression is compact unless you explicitly state that it is verbose.

*
**

5.6. CASE STUDY: PARSING PHONE NUMBERS

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

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

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

- 800-555-1212
- 800 555 1212

\d matches

*any numeric
digit (0–9).*

*\D matches
anything but
digits.*

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

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

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

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

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

- ① Always read regular expressions from left to right. This one matches the beginning of the string, and then `(\d{3})`. What's `\d{3}`? Well, `\d` means “any numeric digit” (0 through 9). The `{3}` means “match exactly three numeric digits”; it's a variation on the `{n,m}` syntax you saw earlier. Putting it all in parentheses means “match exactly three numeric digits, *and then remember them as a group that I can ask for later*”. Then match a literal hyphen. Then match another group of exactly three digits. Then another literal hyphen. Then another group of exactly four digits. Then match the end of the string.
- ② To get access to the groups that the regular expression parser remembered along the way, use the `groups()` method on the object that the `search()` method returns. It will return a tuple of however many groups were defined in the regular expression. In this case, you defined three groups, one with three digits, one with three digits, and one with four digits.
- ③ This regular expression is not the final answer, because it doesn't handle a phone number with an extension on the end. For that, you'll need to expand the regular expression.

- ④ And this is why you should never “chain” the `search()` and `groups()` methods in production code. If the `search()` method returns no matches, it returns `None`, not a regular expression match object. Calling `None.groups()` raises a perfectly obvious exception: `None` doesn’t have a `groups()` method. (Of course, it’s slightly less obvious when you get this exception from deep within your code. Yes, I speak from experience here.)

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

- ① This regular expression is almost identical to the previous one. Just as before, you match the beginning of the string, then a remembered group of three digits, then a hyphen, then a remembered group of three digits, then a hyphen, then a remembered group of four digits. What’s new is that you then match another hyphen, and a remembered group of one or more digits, then the end of the string.
- ② The `groups()` method now returns a tuple of four elements, since the regular expression now defines four groups to remember.
- ③ Unfortunately, this regular expression is not the final answer either, because it assumes that the different parts of the phone number are separated by hyphens. What if they’re separated by spaces, or commas, or dots? You need a more general solution to match several different types of separators.
- ④ Oops! Not only does this regular expression not do everything you want, it’s actually a step backwards, because now you can’t parse phone numbers *without* an extension. That’s not what you wanted at all; if the extension is there, you want to know what it is, but if it’s not there, you still want to know what the different parts of the main number are.

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

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



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

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

④

```
>>>
```

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

⑤

```
>>>
```

- ① Hang on to your hat. You're matching the beginning of the string, then a group of three digits, then `\D+`. What the heck is that? Well, `\D` matches any character except a numeric digit, and `+` means "1 or more". So `\D+` matches one or more characters that are not digits. This is what you're using instead of a literal hyphen, to try to match different separators.
- ② Using `\D+` instead of `-` means you can now match phone numbers where the parts are separated by spaces instead of hyphens.
- ③ Of course, phone numbers separated by hyphens still work too.
- ④ Unfortunately, this is still not the final answer, because it assumes that there is a separator at all. What if the phone number is entered without any spaces or hyphens at all?
- ⑤ Oops! This still hasn't fixed the problem of requiring extensions. Now you have two problems, but you can solve both of them with the same technique.

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

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

①

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

②

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

```
>>> phonePattern.search('800.555.1212 x1234').groups()
```

③

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

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

④

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

```
>>> phonePattern.search('(800)5551212 x1234')
```

⑤

```
>>>
```

- ① The only change you've made since that last step is changing all the `+` to `*`. Instead of `\D+` between the parts of the phone number, you now match on `\D*`. Remember that `+` means "1 or more"? Well, `*` means "zero or more". So now you should be able to parse phone numbers even when there is no separator character at all.
- ② Lo and behold, it actually works. Why? You matched the beginning of the string, then a remembered

group of three digits (800), then zero non-numeric characters, then a remembered group of three digits (555), then zero non-numeric characters, then a remembered group of four digits (1212), then zero non-numeric characters, then a remembered group of an arbitrary number of digits (1234), then the end of the string.

- ③ Other variations work now too: dots instead of hyphens, and both a space and an x before the extension.
- ④ Finally, you've solved the other long-standing problem: extensions are optional again. If no extension is found, the `groups()` method still returns a tuple of four elements, but the fourth element is just an empty string.
- ⑤ I hate to be the bearer of bad news, but you're not finished yet. What's the problem here? There's an extra character before the area code, but the regular expression assumes that the area code is the first thing at the beginning of the string. No problem, you can use the same technique of "zero or more non-numeric characters" to skip over the leading characters before the area code.

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

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('(800)5551212 ext. 1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ③
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234') ④
>>>
```

- ① This is the same as in the previous example, except now you're matching `\D*`, zero or more non-numeric characters, before the first remembered group (the area code). Notice that you're not remembering these non-numeric characters (they're not in parentheses). If you find them, you'll just skip over them and then start remembering the area code whenever you get to it.
- ② You can successfully parse the phone number, even with the leading left parenthesis before the area code. (The right parenthesis after the area code is already handled; it's treated as a non-numeric separator and matched by the `\D*` after the first remembered group.)
- ③ Just a sanity check to make sure you haven't broken anything that used to work. Since the leading characters are entirely optional, this matches the beginning of the string, then zero non-numeric characters, then a remembered group of three digits (800), then one non-numeric character (the hyphen), then a remembered group of three digits (555), then one non-numeric character (the hyphen), then a

remembered group of four digits (1212), then zero non-numeric characters, then a remembered group of zero digits, then the end of the string.

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

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

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ③
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234').groups() ④
('800', '555', '1212', '1234')
```

- ① Note the lack of `^` in this regular expression. You are not matching the beginning of the string anymore. There's nothing that says you need to match the entire input with your regular expression. The regular expression engine will do the hard work of figuring out where the input string starts to match, and go from there.
- ② Now you can successfully parse a phone number that includes leading characters and a leading digit, plus any number of any kind of separators around each part of the phone number.
- ③ Sanity check. This still works.
- ④ That still works too.

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

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

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

>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups() ①
('800', '555', '1212', '1234')

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

- ① Other than being spread out over multiple lines, this is exactly the same regular expression as the last step, so it's no surprise that it parses the same inputs.
- ② Final sanity check. Yes, this still works. You're done.

*
**

5.7. SUMMARY

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

You should now be familiar with the following techniques:

- `^` matches the beginning of a string.
- `$` matches the end of a string.
- `\b` matches a word boundary.
- `\d` matches any numeric digit.

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

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



© 2001–11 Mark Pilgrim