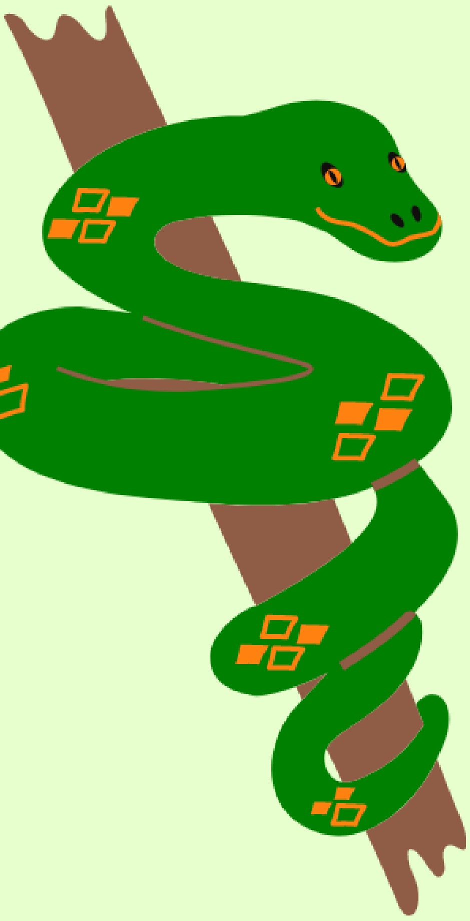
learnbyexample

UNDERSTANDING PYTHON RE(GEX)?



✓ 300+ examples

✓ 100+ exercises

Sundeep Agarwal

Table of contents

Preface

Scripting and automation tasks often need to extract particular portions of text from input data or modify them from one format to another. This book will help you understand Regular Expressions, a mini-programming language for all sorts of text processing needs.

This book heavily leans on examples to present features of regular expressions one by one. It is recommended that you manually type each example and experiment with them. Understanding both the nature of input strings and the output produced is essential. As an analogy, consider learning to drive a car — no matter how much you read about them or listen to explanations, you'd need practical experience to become proficient.

Prerequisites

You should be familiar with programming basics. You should also have a working knowledge of Python syntax and features like string formats, string methods and list comprehensions.

You are also expected to get comfortable with reading manuals, searching online, visiting external links provided for further reading, tinkering with illustrated examples, asking for help when you are stuck and so on. In other words, be proactive and curious instead of just consuming the content passively.

If you have prior experience with a programming language but not Python, see my curated list of learning resources before starting this book.

Conventions

- The examples presented here have been tested with **Python version 3.11.1** and includes features not available in earlier versions.
- Code snippets shown are copy pasted from the Python REPL shell and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines have been added to improve readability. Error messages are shortened. `import` statements are skipped after initial use. And so on.
- Unless otherwise noted, all examples and explanations are meant for **ASCII** characters.
- External links are provided throughout the book for you to explore certain topics in more depth.
- The py_regular_expressions repo has all the code snippets and exercises used in the book. Solutions file is also provided. If you are not familiar with the `git` command, click the **Code** button on the webpage to get the files.

Acknowledgements

- Python documentation — manuals and tutorials
- /r/learnpython/, /r/Python/ and /r/regex/ — helpful forums for beginners and experienced programmers alike
- stackoverflow — for getting answers to pertinent questions on Python and regular expressions
- tex.stackexchange — for help on pandoc and `tex` related questions
- canva — cover image
- Warning and Info icons by Amada44 under public domain
- oxipng, pngquant and svgcleaner — optimizing images

- David Cortesi for helpful feedback on both the technical content and grammar issues
- **Kye** and gmovchan for spotting a typo
- **Hugh**'s email exchanges helped me significantly to improve the presentation of concepts and exercises
- Christopher Patti for reviewing the book, providing feedback and brightening the day with kind words
- Users **73tada**, **DrBobHope**, **nlomb** and others for feedback in this reddit thread

Special thanks to Al Sweigart. His Automate the Boring Stuff book was instrumental for me to get started with Python.

Feedback and Errata

I would highly appreciate if you'd let me know how you felt about this book. It could be anything from a simple thank you, pointing out a typo, mistakes in code snippets, which aspects of the book worked for you (or didn't!) and so on. Reader feedback is essential and especially so for self-published authors.

You can reach me via:

- Issue Manager: https://github.com/learnbyexample/py_regular_expressions/issues
- E-mail: learnbyexample.net@gmail.com
- Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a lazy being who prefers to work just enough to support his modest lifestyle. He accumulated vast wealth working as a Design Engineer at Analog Devices and retired from the corporate world at the ripe age of twenty-eight. Unfortunately, he squandered his savings within a few years and had to scramble trying to earn a living. Against all odds, selling programming ebooks saved his lazy self from having to look for a job again. He can now afford all the fantasy ebooks he wants to read and spends unhealthy amount of time browsing the internet.

When the creative muse strikes, he can be found working on yet another programming ebook (which invariably ends up having at least one example with regular expressions). Researching materials for his ebooks and everyday social media usage drowned his bookmarks, so he maintains curated resource lists for sanity sake. He is thankful for free learning resources and open source tools. His own contributions can be found at https://github.com/learnbyexample.

List of books: https://learnbyexample.github.io/books/

License

Book version

4.1

See Version_changes.md to track changes across book versions.

Why is it needed?

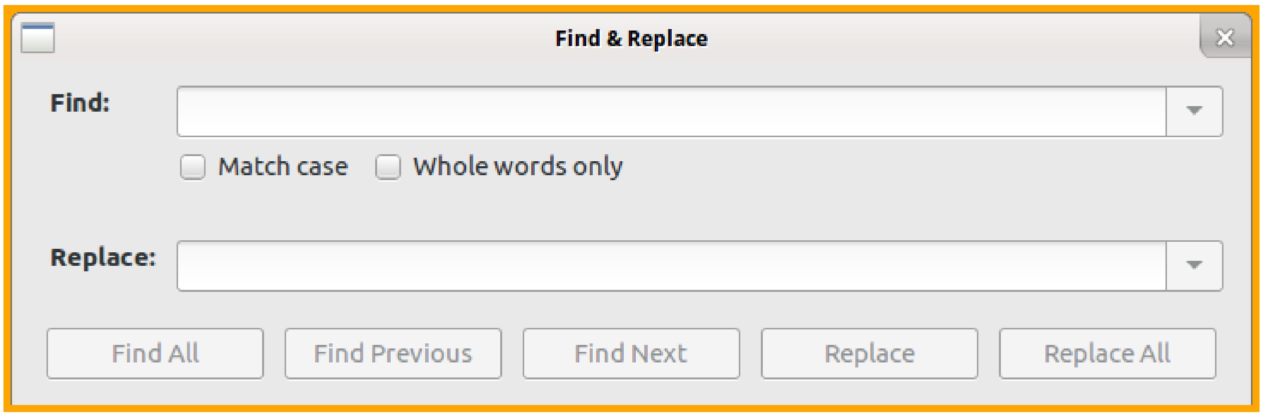
Regular Expressions is a versatile tool for text processing. You'll find them included as part of the standard library of most programming languages that are used for scripting purposes. If not, you can usually find a third-party library. Syntax and features of regular expressions vary from language to language. Python's syntax is similar to that of Perl language, but there are significant feature differences.

The `str` class comes loaded with variety of methods to deal with text. So, what's so special about regular expressions and why would you need it? For learning and understanding purposes, one can view regular expressions as a mini-programming language specialized for text processing. Parts of a regular expression can be saved for future use, analogous to variables and functions. There are ways to perform AND, OR, NOT conditionals. Operations similar to the `range()` function, string repetition operator and so on.

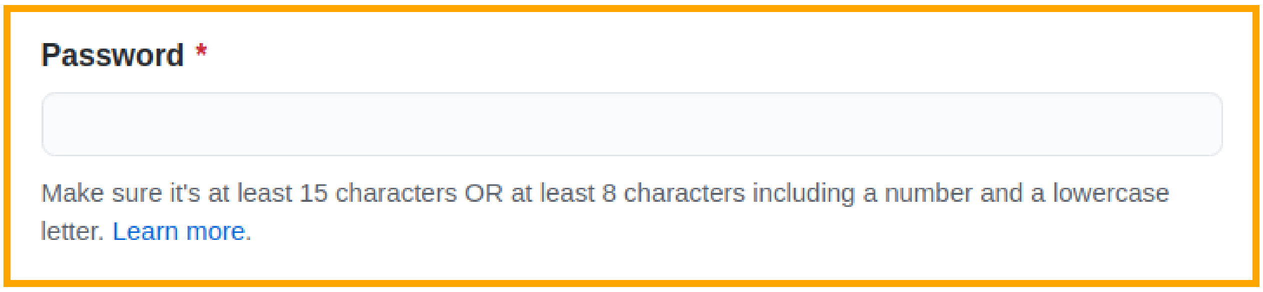
Here are some common use cases:

- Sanitizing a string to ensure that it satisfies a known set of rules. For example, to check if a given string matches password rules.
- Filtering or extracting portions on an abstract level like alphabets, digits, punctuation and so on.
- Qualified string replacement. For example, at the start or the end of a string, only whole words, based on surrounding text, etc.

You are likely to be familiar with graphical search and replace tools, like the screenshot shown below from LibreOffice Writer. **Match case**, **Whole words only**, **Replace** and **Replace All** are some of the basic features supported by regular expressions.



Another real world use case is password validation. The screenshot below is from GitHub sign up page. Performing multiple checks like **string length** and the **type of characters allowed** is another core feature of regular expressions.


Password *

Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. Learn more.

Here are some articles on regular expressions to know about its history and the type of problems it is suited for.

- The true power of regular expressions — it also includes a nice explanation of what *regular* means in this context
- softwareengineering: Is it a must for every programmer to learn regular expressions?
- softwareengineering: When you should NOT use Regular Expressions?
- codinghorror: Now You Have Two Problems — demystifies the often (mis)quoted meme
- wikipedia: Regular expression — this article includes discussion on regular expressions as a formal language as well as details about various implementations

How this book is organized

This book introduces concepts one by one and exercises at the end of chapters will require only the features introduced until that chapter. Each concept is accompanied by plenty of examples to cover multiple problems and corner cases. As mentioned before, it is highly recommended that you follow along the examples by typing out the code snippets manually. It is important to understand both the nature of the sample input string as well as the actual programming command used. There are two interlude chapters that give an overview of useful tools and some more resources are collated in the final chapter.

By the end of the book, you should be comfortable with both writing and reading regular expressions, how to debug them and know when to *avoid* them.

re introduction

This chapter gives an introduction to the `re` module. This module is part of the standard library. For some examples, the equivalent normal string method is also shown for comparison. This chapter just focuses on the basics of using functions from the `re` module. Regular expression features will be covered from the next chapter onwards.

re module documentation

It is always a good idea to know where to find the documentation. The default offering for Python regular expressions is the `re` standard library module. Visit docs.python: re for information on available methods, syntax, features, examples and more. Here's a quote:

> A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression

re.search()

Normally you'd use the `in` operator to test whether a string is part of another string or not. For regular expressions, use the `re.search()` function whose argument list is shown below.

```
re.search(pattern, string, flags=0)
```

The first argument is the RE pattern you want to test against the input string, which is the second argument. `flags` is optional, it helps to change the default behavior of RE patterns.

As a good practice, always use **raw strings** to construct the RE pattern. This will become clearer in later chapters. Here are some examples to get started.

```
>>> sentence = 'This is a sample string'

# check if 'sentence' contains the given search string
>>> 'is' in sentence
True
>>> 'xyz' in sentence
False

# need to load the re module before use
>>> import re

# check if 'sentence' contains the pattern described by the RE argument
>>> bool(re.search(r'is', sentence))
True
>>> bool(re.search(r'xyz', sentence))
False
```

Before using the `re` module, you need to `import` it. Further example snippets will assume that this module is already loaded. The return value of the `re.search()` function is a `re.Match` object when a match is found and `None` otherwise (note that I treat `re` as a word, not as `r` and `e` separately, hence the use of *a* instead of *an*). More details about the `re.Match` object will be discussed in the Working with matched portions chapter. For presentation purposes, the examples will use the `bool()` function to show `True` or `False` depending on whether the RE pattern matched or not.

Here's an example with the `flags` optional argument. By default, the pattern will match the input string case sensitively. By using the `re.I` flag, you can match case insensitively. See the Flags chapter for more details.

```
>>> sentence = 'This is a sample string'

>>> bool(re.search(r'this', sentence))
False

# re.IGNORECASE (or re.I) is a flag to enable case insensitive matching
>>> bool(re.search(r'this', sentence, flags=re.I))
True
```

re.search() in conditional expressions

As Python evaluates `None` as `False` in boolean context, `re.search()` can be used directly in conditional expressions. See also docs.python: Truth Value Testing.

```
>>> sentence = 'This is a sample string'
>>> if re.search(r'ring', sentence):
...     print('mission success')
...
mission success

>>> if not re.search(r'xyz', sentence):
...     print('mission failed')
...
mission failed
```

Here are some examples with list comprehensions and generator expressions:

```
>>> words = ['cat', 'attempt', 'tattle']

>>> [w for w in words if re.search(r'tt', w)]
['attempt', 'tattle']
>>> all(re.search(r'at', w) for w in words)
True
>>> any(re.search(r'stat', w) for w in words)
False
```

re.sub()

For normal search and replace, you'd use the `str.replace()` method. For regular expressions, use the `re.sub()` function, whose argument list is shown below.

```
re.sub(pattern, repl, string, count=0, flags=0)
```

The first argument is the RE pattern to match against the input string, which is the third argument. The second argument specifies the string which will replace the portions matched by the RE pattern. `count` and `flags` are optional arguments.

```
>>> greeting = 'Have a nice weekend'

# replace all occurrences of 'e' with 'E'
# same as: greeting.replace('e', 'E')
>>> re.sub(r'e', 'E', greeting)
'HavE a nicE wEEkEnd'

# replace first two occurrences of 'e' with 'E'
# same as: greeting.replace('e', 'E', 2)
>>> re.sub(r'e', 'E', greeting, count=2)
'HavE a nicE weekend'
```

A common mistake, not specific to `re.sub()` , is forgetting that strings are immutable in Python.

```
>>> word = 'cater'
# this will return a string object, won't modify the 'word' variable
>>> re.sub(r'cat', 'wag', word)
'wager'
>>> word
'cater'

# need to explicitly assign the result if 'word' has to be changed
>>> word = re.sub(r'cat', 'wag', word)
>>> word
'wager'
```

Compiling regular expressions

Regular expressions can be compiled using the `re.compile()` function, which gives back a `re.Pattern` object.

```
re.compile(pattern, flags=0)
```

The top level `re` module functions are all available as methods for such objects. Compiling a regular expression is useful if the RE has to be used in multiple places or called upon multiple times inside a loop (speed benefit).

By default, Python maintains a small list of recently used RE, so the speed benefit doesn't apply for trivial use cases. See also stackoverflow: Is it worth using re.compile?

```
>>> pet = re.compile(r'dog')
>>> type(pet)
```

```
<class 're.Pattern'>

# note that 'search' is called upon 'pet' which is a 're.Pattern' object
# since 'pet' has the RE information, you only need to pass the input string
>>> bool(pet.search('They bought a dog'))
True
>>> bool(pet.search('A cat crossed their path'))
False

# replace all occurrences of 'dog' with 'cat'
>>> pet.sub('cat', 'They bought a dog')
'They bought a cat'
```

Some of the methods available for compiled patterns also accept more arguments than those available for the top level functions of the `re` module. For example, the `search()` method on a compiled pattern has two optional arguments to specify the **start** and **end** index positions. Similar to the `range()` function and slicing notation, the ending index has to be specified `1` greater than the desired index.

```
Pattern.search(string[, pos[, endpos]])
```

Note that there's no `flags` option as that has to be specified with `re.compile()` .

```
>>> sentence = 'This is a sample string'
>>> word = re.compile(r'is')

# search for 'is' starting from the 5th character
>>> bool(word.search(sentence, 4))
True

# search for 'is' starting from the 7th character
>>> bool(word.search(sentence, 6))
False

# search for 'is' from the 3rd character to the 4th character
>>> bool(word.search(sentence, 2, 4))
True
```

bytes

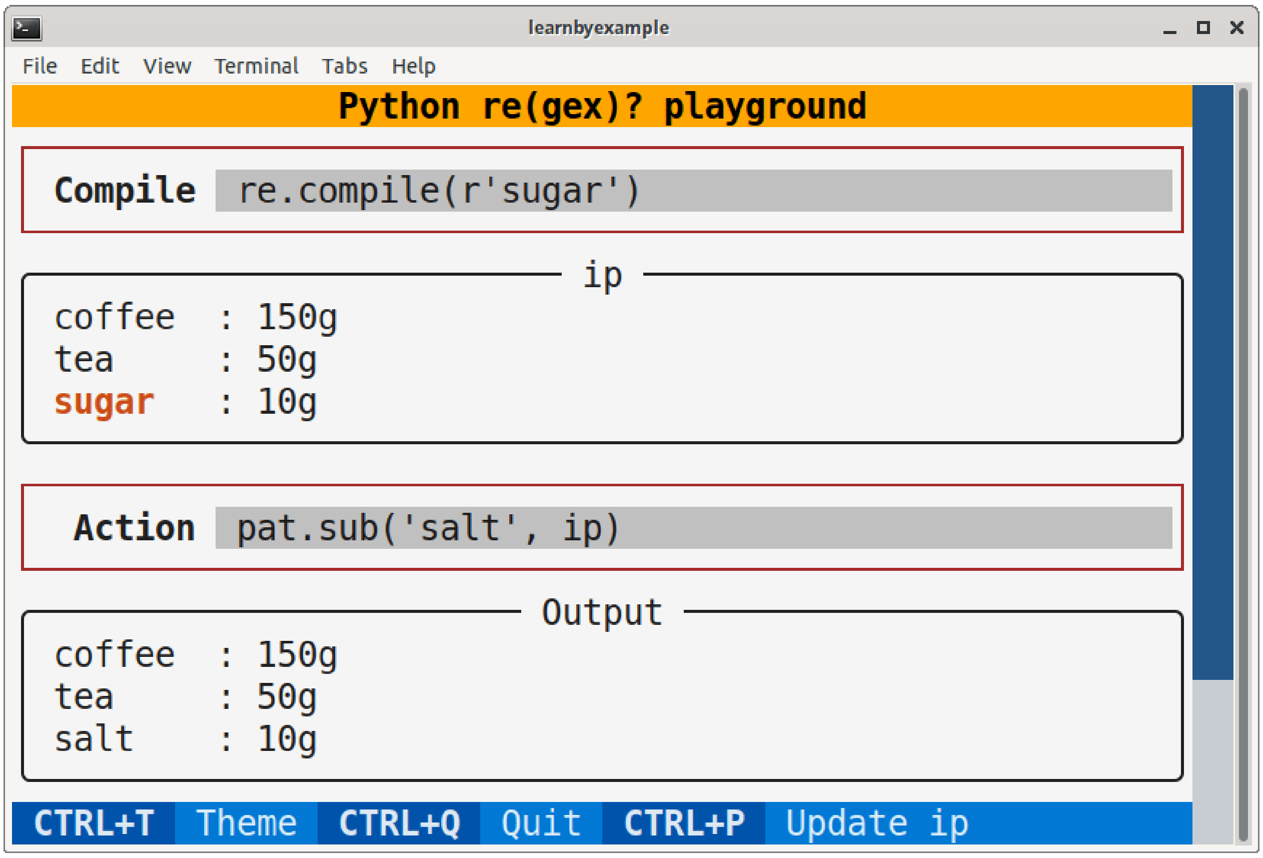
To work with the `bytes` data type, the RE must be specified as `bytes` as well. Similar to the `str` RE, use **raw** format to construct a `bytes` RE.

```
>>> byte_data = b'This is a sample string'

# error message truncated for presentation purposes
>>> re.search(r'is', byte_data)
TypeError: cannot use a string pattern on a bytes-like object

# use rb'..' for constructing bytes pattern
```

```
>>> bool(re.search(rb'is', byte_data))
True
>>> bool(re.search(rb'xyz', byte_data))
False
```

re(gex)? playground

To make it easier to experiment, I wrote an interactive TUI app. See PyRegexPlayground repo for installation instructions and usage guide. A sample screenshot is shown below:



```
re.compile(r'sugar')
```

```
coffee   : 150g
tea      : 50g
sugar    : 10g
```

```
pat.sub('salt', ip)
```

```
coffee   : 150g
tea      : 50g
salt     : 10g
```

Cheatsheet and Summary

Note	Description
docs.python: re	Python standard module for regular expressions
`re.search()`	Check if the given pattern is present anywhere in the input string
	`re.search(pattern, string, flags=0)`
	Output is a `re.Match` object, usable in conditional expressions
	raw strings preferred to define RE
	Additionally, Python maintains a small cache of recent RE
`re.sub()`	search and replace using RE
	`re.sub(pattern, repl, string, count=0, flags=0)`
`re.compile()`	Compile a pattern for reuse, output is a `re.Pattern` object
	`re.compile(pattern, flags=0)`
`rb'pat'`	Use byte pattern for byte input
`re.IGNORECASE` or `re.I`	flag to ignore case while matching

This chapter introduced the `re` module, which is part of the standard library. Functions `re.search()` and `re.sub()` were discussed as well as how to compile RE using the `re.compile()` function. The RE pattern is usually defined using raw strings. For byte input, the pattern has to be of byte type too. Although the `re` module is good enough for most use cases, there are situations where you need to use the third-party `regex` module. To avoid mixing up features, a separate chapter is dedicated for the regex module at the end of this book.

The next section has exercises to test your understanding of the concepts introduced in this chapter. Please do solve them before moving on to the next chapter.

Exercises

Try to solve exercises in every chapter using only the features discussed until that chapter. Some of the exercises will be easier to solve with techniques presented in later chapters, but the aim of these exercises is to explore the features presented so far.

All the exercises are also collated together in one place at Exercises.md. For solutions, see Exercise_solutions.md.

a) Check whether the given strings contain `0xB0`. Display a boolean result as shown below.

```
>>> line1 = 'start address: 0xA0, func1 address: 0xC0'
>>> line2 = 'end address: 0xFF, func2 address: 0xB0'

>>> bool(re.search(r'', line1))     ##### add your solution here
False
>>> bool(re.search(r'', line2))     ##### add your solution here
True
```

b) Replace all occurrences of `5` with `five` for the given string.

```
>>> ip = 'They ate 5 apples and 5 oranges'

>>> re.sub()        ##### add your solution here
'They ate five apples and five oranges'
```

c) Replace only the first occurrence of `5` with `five` for the given string.

```
>>> ip = 'They ate 5 apples and 5 oranges'

>>> re.sub()       ##### add your solution here
'They ate five apples and 5 oranges'
```

d) For the given list, filter all elements that do *not* contain `e`.

```
>>> items = ['goal', 'new', 'user', 'sit', 'eat', 'dinner']

>>> [w for w in items if not re.search()]        ##### add your solution here
['goal', 'sit']
```

e) Replace all occurrences of `note` irrespective of case with `X` .

```
>>> ip = 'This note should not be NoTeD'

>>> re.sub()        ##### add your solution here
'This X should not be XD'
```

f) Check if `at` is present in the given byte input data.

```
>>> ip = b'tiger imp goat'

>>> bool(re.search())     ##### add your solution here
True
```

g) For the given input string, display all lines not containing `start` irrespective of case.

```
>>> para = '''good start
... Start working on that
... project you always wanted
... stars are shining brightly
... hi there
... start and try to
... finish the book
... bye'''

>>> pat = re.compile()      ##### add your solution here
>>> for line in para.split('\n'):
...     if not pat.search(line):
...         print(line)
...
project you always wanted
stars are shining brightly
hi there
finish the book
bye
```

h) For the given list, filter all elements that contain either `a` or `w` .

```
>>> items = ['goal', 'new', 'user', 'sit', 'eat', 'dinner']

##### add your solution here
>>> [w for w in items if re.search() or re.search()]
['goal', 'new', 'eat']
```

i) For the given list, filter all elements that contain both `e` and `n` .

```
>>> items = ['goal', 'new', 'user', 'sit', 'eat', 'dinner']

##### add your solution here
>>> [w for w in items if re.search() and re.search()]
['new', 'dinner']
```

j) For the given string, replace `0xA0` with `0x7F` and `0xC0` with `0x1F` .

```
>>> ip = 'start address: 0xA0, func1 address: 0xC0'

##### add your solution here
'start address: 0x7F, func1 address: 0x1F'
```

Anchors

Now that you're familiar with RE syntax and couple of `re` module functions, the next step is to know about the special features of regular expressions. In this chapter, you'll be learning about qualifying a pattern. Instead of matching anywhere in the given input string, restrictions can be specified. For now, you'll see the ones that are already part of the `re` module. In later chapters, you'll learn how to define custom rules.

These restrictions are made possible by assigning special meaning to certain characters and escape sequences. The characters with special meaning are known as **metacharacters** in regular expressions parlance. In case you need to match those characters literally, you need to escape them with a `\` character (discussed in the Escaping metacharacters chapter).

String anchors

This restriction is about qualifying a RE to match only at the start or the end of an input string. These provide functionality similar to the `str` methods `startswith()` and `endswith()`. First up, the escape sequence `\A` which restricts the matching to the start of string.

```
# \A is placed as a prefix to the search term
>>> bool(re.search(r'\Acat', 'cater'))
True
>>> bool(re.search(r'\Acat', 'concatenation'))
False

>>> bool(re.search(r'\Ahi', 'hi hello\ntop spot'))
True
>>> bool(re.search(r'\Atop', 'hi hello\ntop spot'))
False
```

To restrict the matching to the end of string, `\Z` is used.

```
# \Z is placed as a suffix to the search term
>>> bool(re.search(r'are\Z', 'spare'))
True
>>> bool(re.search(r'are\Z', 'nearest'))
False

>>> words = ['surrender', 'unicorn', 'newer', 'door', 'erase', 'eel', 'pest']
>>> [w for w in words if re.search(r'er\Z', w)]
['surrender', 'newer']
>>> [w for w in words if re.search(r't\Z', w)]
['pest']
```

You can emulate string concatenation operations by using the anchors by themselves as a pattern.

```
# insert text at the start of a string
>>> re.sub(r'\A', 're', 'live')
'relive'
>>> re.sub(r'\A', 're', 'send')
'resend'
```

```
# appending text
>>> re.sub(r'\Z', 'er', 'cat')
'cater'
>>> re.sub(r'\Z', 'er', 'hack')
'hacker'
```

⚠ Use the optional start and end index arguments of the `Pattern.search()` method with caution. They are not equivalent to string slicing. For example, specifying a greater than `0` start index when using `\A` is always going to return `False`. This is because, as far as the `search()` method is concerned, only the search space has been narrowed — the anchor positions haven't changed. When slicing is used, you are creating an entirely new string object with new anchor positions.

```
>>> word_pat = re.compile(r'\Aat')

>>> bool(word_pat.search('cater', 1))
False
>>> bool(word_pat.search('cater'[1:]))
True
```

re.fullmatch()

Combining both the start and end string anchors, you can restrict the matching to the whole string. The effect is similar to comparing strings using the `==` operator.

```
>>> word_pat = re.compile(r'\Acat\Z')

>>> bool(word_pat.search('cat'))
True
>>> bool(word_pat.search('concatenation'))
False
```

You can also use the `re.fullmatch()` function to ensure the pattern matches only the whole input string and not just a part of the input. This may not seem useful with features introduced so far, but when you have a complex RE pattern with multiple alternatives, this function is quite handy. The argument list is same as the `re.search()` function.

```
re.fullmatch(pattern, string, flags=0)
```

```
>>> word_pat = re.compile(r'cat', flags=re.I)

>>> bool(word_pat.fullmatch('Cat'))
True
>>> bool(word_pat.fullmatch('Scatter'))
False
```

Line anchors

A string input may contain single or multiple lines. The newline character `\n` is considered as the line separator. There are two line anchors. `^` metacharacter for matching the start of line and `$` for matching the end of line. If there are no newline characters in the input string, these will behave exactly the same as `\A` and `\Z` respectively.

```
>>> pets = 'cat and dog'

>>> bool(re.search(r'^cat', pets))
True
>>> bool(re.search(r'^dog', pets))
False

>>> bool(re.search(r'dog$', pets))
True
>>> bool(re.search(r'^dog$', pets))
False
```

By default, the input string is considered as a single line, even if multiple newline characters are present. In such cases, the `$` metacharacter can match both the end of string and just before `\n` if it is the last character. However, `\Z` will always match the end of string, irrespective of the characters present.

```
>>> greeting = 'hi there\nhave a nice day\n'

>>> bool(re.search(r'day$', greeting))
True
>>> bool(re.search(r'day\n$', greeting))
True

>>> bool(re.search(r'day\Z', greeting))
False
>>> bool(re.search(r'day\n\Z', greeting))
True
```

To indicate that the input string should be treated as multiple lines, you need to enable the `re.MULTILINE` flag (`re.M` for short).

```
# check if any line in the string starts with 'top'
>>> bool(re.search(r'^top', 'hi hello\ntop spot', flags=re.M))
True

# check if any line in the string ends with 'ar'
>>> bool(re.search(r'ar$', 'spare\npar\ndare', flags=re.M))
True

# filter all elements having lines ending with 'are'
>>> elements = ['spare\ntool', 'par\n', 'dare']
>>> [e for e in elements if re.search(r'are$', e, flags=re.M)]
['spare\ntool', 'dare']
```

```
# check if any whole line in the string is 'par'
>>> bool(re.search(r'^par$', 'spare\npar\ndare', flags=re.M))
True
```

Just like string anchors, you can use the line anchors by themselves as a pattern.

```
>>> ip_lines = 'catapults\nconcatenate\ncat'
>>> print(re.sub(r'^', '* ', ip_lines, flags=re.M))
* catapults
* concatenate
* cat

>>> print(re.sub(r'$', '.', ip_lines, flags=re.M))
catapults.
concatenate.
cat.
```

⚠ If you are dealing with Windows OS based text files, you may have to convert `\r\n` line endings to `\n` first. Python functions and methods make it easier to handle such situations. For example, you can specify which line ending to use for the `open()` function, the `split()` string method handles all whitespaces by default and so on. Or, you can handle `\r` as an optional character with quantifiers (see the Dot metacharacter and Quantifiers chapter for details).

Word anchors

The third type of restriction is word anchors. Alphabets (irrespective of case), digits and the underscore character qualify as word characters. You might wonder why there are digits and underscores as well, why not just alphabets? This comes from variable and function naming conventions — typically alphabets, digits and underscores are allowed. So, the definition is more oriented to programming languages than natural ones.

The escape sequence `\b` denotes a word boundary. This works for both the start and end of word anchoring. Start of word means either the character prior to the word is a non-word character or there is no character (start of string). Similarly, end of word means the character after the word is a non-word character or no character (end of string). This implies that you cannot have word boundary `\b` without a word character.

```
>>> words = 'par spar apparent spare part'

# replace 'par' irrespective of where it occurs
>>> re.sub(r'par', 'X', words)
'X sX apXent sXe Xt'
# replace 'par' only at the start of word
>>> re.sub(r'\bpar', 'X', words)
'X spar apparent spare Xt'
# replace 'par' only at the end of word
>>> re.sub(r'par\b', 'X', words)
'X sX apparent spare part'
```

```
# replace 'par' only if it is not part of another word
>>> re.sub(r'\bpar\b', 'X', words)
'X spar apparent spare part'
```

Using word boundary as a pattern by itself can yield creative solutions:

```
# space separated words to double quoted csv
# note the use of 'replace' string method for normal string replacement
# 'translate' method can also be used
>>> words = 'par spar apparent spare part'
>>> print(re.sub(r'\b', '"', words).replace(' ', ','))
"par","spar","apparent","spare","part"

>>> re.sub(r'\b', ' ', '-----hello-----')
'----- hello -----'

# make a programming statement more readable
# shown for illustration purpose only, won't work for all cases
>>> re.sub(r'\b', ' ', 'output=num1+35*42/num2')
' output = num1 + 35 * 42 / num2 '
# excess space at start/end of string can be stripped off
# later you'll learn how to add a qualifier so that strip is not needed
>>> re.sub(r'\b', ' ', 'output=num1+35*42/num2').strip()
'output = num1 + 35 * 42 / num2'
```

The word boundary has an opposite anchor too. `\B` matches wherever `\b` doesn't match. This duality will be seen with some other escape sequences too. Negative logic is handy in many text processing situations. But use it with care, you might end up matching things you didn't intend!

```
>>> words = 'par spar apparent spare part'

# replace 'par' if it is not at the start of word
>>> re.sub(r'\Bpar', 'X', words)
'par sX apXent sXe part'
# replace 'par' at the end of word but not the whole word 'par'
>>> re.sub(r'\Bpar\b', 'X', words)
'par sX apparent spare part'
# replace 'par' if it is not at the end of word
>>> re.sub(r'par\B', 'X', words)
'par spar apXent sXe Xt'
# replace 'par' if it is surrounded by word characters
>>> re.sub(r'\Bpar\B', 'X', words)
'par spar apXent sXe part'
```

Here are some standalone pattern usage to compare and contrast the two word anchors.

```
>>> re.sub(r'\b', ':', 'copper')
':copper:'
>>> re.sub(r'\B', ':', 'copper')
'c:o:p:p:e:r'
```

```
>>> re.sub(r'\b', ' ', '-----hello-----')
'----- hello -----'
>>> re.sub(r'\B', ' ', '-----hello-----')
' - - - - -h e l l o- - - - - '
```

Cheatsheet and Summary

Note	Description
`\A`	restricts the match to the start of string
`\Z`	restricts the match to the end of string
`re.fullmatch()`	ensures pattern matches the entire input string
	`re.fullmatch(pattern, string, flags=0)`
`\n`	line separator, dos-style files may need special attention
metacharacter	characters with special meaning in RE
`^`	restricts the match to the start of line
`$`	restricts the match to the end of line
`re.MULTILINE` or `re.M`	flag to treat input as multiline string
`\b`	restricts the match to the start and end of words
	word characters: alphabets, digits, underscore
`\B`	matches wherever `\b` doesn't match

In this chapter, you've begun to see building blocks of regular expressions and how they can be used in interesting ways. But at the same time, regular expression is but another tool in the land of text processing. Often, you'd get simpler solution by combining regular expressions with other string methods and expressions. Practice, experience and imagination would help you construct creative solutions. In the coming chapters, you'll see examples for anchors in combination with other features.

Exercises

a) Check if the given strings start with `be` .

```
>>> line1 = 'be nice'
>>> line2 = '"best!"'
>>> line3 = 'better?'
>>> line4 = 'oh no\nbear spotted'

>>> pat = re.compile()       ##### add your solution here

>>> bool(pat.search(line1))
True
>>> bool(pat.search(line2))
False
>>> bool(pat.search(line3))
True
>>> bool(pat.search(line4))
False
```

b) For the given input string, change only the whole word `red` to `brown` .

```
>>> words = 'bred red spread credible red.'

>>> re.sub()      ##### add your solution here
'bred brown spread credible brown.'
```

c) For the given input list, filter all elements that contain `42` surrounded by word characters.

```
>>> words = ['hi42bye', 'nice1423', 'bad42', 'cool_42a', '42fake', '_42_']

>>> [w for w in words if re.search()]    ##### add your solution here
['hi42bye', 'nice1423', 'cool_42a', '_42_']
```

d) For the given input list, filter all elements that start with `den` or end with `ly` .

```
>>> items = ['lovely', '1\ndentist', '2 lonely', 'eden', 'fly\n', 'dent']

>>> [e for e in items if ]         ##### add your solution here
['lovely', '2 lonely', 'dent']
```

e) For the given input string, change whole word `mall` to `1234` only if it is at the start of a line.

```
>>> para = '''\
... (mall) call ball pall
... ball fall wall tall
... mall call ball pall
... wall mall ball fall
... mallet wallet malls
... mall:call:ball:pall'''

>>> print(re.sub())     ##### add your solution here
(mall) call ball pall
ball fall wall tall
1234 call ball pall
wall mall ball fall
mallet wallet malls
1234:call:ball:pall
```

f) For the given list, filter all elements having a line starting with `den` or ending with `ly` .

```
>>> items = ['lovely', '1\ndentist', '2 lonely', 'eden', 'fly\nfar', 'dent']

##### add your solution here
['lovely', '1\ndentist', '2 lonely', 'fly\nfar', 'dent']
```

g) For the given input list, filter all whole elements `12\nthree` irrespective of case.

```
>>> items = ['12\nthree\n', '12\nThree', '12\nthree\n4', '12\nthree']

##### add your solution here
['12\nThree', '12\nthree']
```

h) For the given input list, replace `hand` with `X` for all elements that start with `hand` followed by at least one word character.

```
>>> items = ['handed', 'hand', 'handy', 'un-handed', 'handle', 'hand-2']

##### add your solution here
['Xed', 'hand', 'Xy', 'un-handed', 'Xle', 'hand-2']
```

i) For the given input list, filter all elements starting with `h`. Additionally, replace `e` with `X` for these filtered elements.

```
>>> items = ['handed', 'hand', 'handy', 'unhanded', 'handle', 'hand-2']

##### add your solution here
['handXd', 'hand', 'handy', 'handlX', 'hand-2']
```

Alternation and Grouping

Similar to logical OR, alternation in regular expressions allows you to combine multiple patterns. These patterns can have some common elements between them, in which case grouping helps to form terser expressions. This chapter will also discuss the precedence rules used to determine which alternation wins.

Alternation

A conditional expression combined with logical OR evaluates to `True` if any of the condition is satisfied. Similarly, in regular expressions, you can use the `|` metacharacter to combine multiple patterns to indicate logical OR. The matching will succeed if any of the alternate pattern is found in the input string. These alternatives have the full power of a regular expression, for example they can have their own independent anchors. Here are some examples.

```
# match either 'cat' or 'dog'
>>> pet = re.compile(r'cat|dog')
>>> bool(pet.search('I like cats'))
True
>>> bool(pet.search('I like dogs'))
True
>>> bool(pet.search('I like parrots'))
False

# replace 'cat' at the start of string or 'cat' at the end of word
>>> re.sub(r'\Acat|cat\b', 'X', 'catapults concatenate cat scat cater')
'Xapults concatenate X sX cater'
# replace 'cat' or 'dog' or 'fox' with 'mammal'
>>> re.sub(r'cat|dog|fox', 'mammal', 'cat dog bee parrot fox')
'mammal mammal bee parrot mammal'
```

You might infer from the above examples that there can be cases where many alternations are required. The `join` string method can be used to build the alternation list automatically from an iterable of strings.

```
>>> '|'.join(['car', 'jeep'])
'car|jeep'
>>> words = ['cat', 'dog', 'fox']
>>> '|'.join(words)
'cat|dog|fox'
>>> re.sub('|'.join(words), 'mammal', 'cat dog bee parrot fox')
'mammal mammal bee parrot mammal'
```

⚠ In the above examples, the elements do not contain any special regular expression characters. Handling strings that contain metacharacters will be discussed in the re.escape() section.

If you have thousands of search terms to be matched, using specialized libraries like flashtext is highly recommended instead of regular expressions.

Grouping

Often, there are some common portions among the alternatives. It could be common characters, qualifiers like the anchors and so on. In such cases, you can group them using a pair of parentheses metacharacters. Similar to `a(b+c)d = abd+acd` in maths, you get `a(b|c)d = abd|acd` in regular expressions.

```
# without grouping
>>> re.sub(r'reform|rest', 'X', 'red reform read arrest')
'red X read arX'
# with grouping
>>> re.sub(r're(form|st)', 'X', 'red reform read arrest')
'red X read arX'

# without grouping
>>> re.sub(r'\bpar\b|\bpart\b', 'X', 'par spare part party')
'X spare X party'
# taking out common anchors
>>> re.sub(r'\b(par|part)\b', 'X', 'par spare part party')
'X spare X party'
# taking out common characters as well
# you'll later learn a better technique instead of using empty alternates
>>> re.sub(r'\bpar(|t)\b', 'X', 'par spare part party')
'X spare X party'
```

There are many more uses for grouping than just forming a terser RE. They will be discussed as they become relevant in the coming chapters.

For now, this is a good place to show how to incorporate normal strings (from a variable, expression result, etc) while building a regular expression. For example, adding anchors to an alternation list created using the `join` method.

```
>>> words = ['cat', 'par']
>>> '|'.join(words)
'cat|par'
# without word boundaries, any matching portion will be replaced
>>> re.sub('|'.join(words), 'X', 'cater cat concatenate par spare')
'Xer X conXenate X sXe'

# you can also use: alt = re.compile(r'\b(' + '|'.join(words) + r')\b')
>>> alt = re.compile(rf"\b({'|'.join(words)})\b")
# only whole words will be replaced now
>>> alt.sub('X', 'cater cat concatenate par spare')
'cater X concatenate X spare'
```

```
# this is how the above RE looks as a normal string
>>> alt.pattern
'\\b(cat|par)\\b'
>>> alt.pattern == r'\b(cat|par)\b'
True
```

In the above example, you had to concatenate strings to add word boundaries. If you needed to add string anchors so that the pattern only matches whole string, you can use `re.fullmatch()` instead of manually adding the anchors.

```
>>> terms = ['no', 'ten', 'it']
>>> items = ['dip', 'nobody', 'it', 'oh', 'no', 'bitten']

>>> pat = re.compile('|'.join(terms))

# matching only whole elements
>>> [w for w in items if(pat.fullmatch(w))]
['it', 'no']
# matching anywhere
>>> [w for w in items if(pat.search(w))]
['nobody', 'it', 'no', 'bitten']
```

Precedence rules

There are tricky situations when using alternation. There is no ambiguity if it is used to get a boolean result by testing a match against a string input. However, for cases like string replacement, it depends on a few factors. Say, you want to replace either `are` or `spared` — which one should get precedence? The bigger word `spared` or the substring `are` inside it or based on something else?

In Python, the alternative which matches earliest in the input string gets precedence. `re.Match` output is handy to illustrate this concept.

```
>>> words = 'lion elephant are rope not'

# span shows the start and end+1 index of the matched portion
# match shows the text that satisfied the search criteria
>>> re.search(r'on', words)
<re.Match object; span=(2, 4), match='on'>
>>> re.search(r'ant', words)
<re.Match object; span=(10, 13), match='ant'>

# starting index of 'on' < index of 'ant' for the given string input
# so 'on' will be replaced irrespective of order
# count optional argument here restricts no. of replacements to 1
>>> re.sub(r'on|ant', 'X', words, count=1)
'liX elephant are rope not'
>>> re.sub(r'ant|on', 'X', words, count=1)
'liX elephant are rope not'
```

What happens if alternatives have the same starting index? The precedence is left-to-right in the order of declaration.

```
>>> mood = 'best years'
>>> re.search(r'year', mood)
<re.Match object; span=(5, 9), match='year'>
>>> re.search(r'years', mood)
<re.Match object; span=(5, 10), match='years'>

# starting index for 'year' and 'years' will always be the same
# so, which one gets replaced depends on the order of alternation
>>> re.sub(r'year|years', 'X', mood, count=1)
'best Xs'
>>> re.sub(r'years|year', 'X', mood, count=1)
'best X'
```

Another example (without `count` restriction) to drive home the issue:

```
>>> words = 'ear xerox at mare part learn eye'

# this is going to be same as: r'ar'
>>> re.sub(r'ar|are|art', 'X', words)
'eX xerox at mXe pXt leXn eye'

# this is going to be same as: r'are|ar'
>>> re.sub(r'are|ar|art', 'X', words)
'eX xerox at mX pXt leXn eye'

# phew, finally this one works as needed
>>> re.sub(r'are|art|ar', 'X', words)
'eX xerox at mX pX leXn eye'
```

If you do not want substrings to sabotage your replacements, a robust workaround is to sort the alternations based on length, longest first.

```
>>> words = ['hand', 'handy', 'handful']

>>> alt = re.compile('|'.join(sorted(words, key=len, reverse=True)))
>>> alt.pattern
'handful|handy|hand'

>>> alt.sub('X', 'hands handful handed handy')
'Xs X Xed X'

# alternation order will come into play if you don't sort them properly
>>> re.sub('|'.join(words), 'X', 'hands handful handed handy')
'Xs Xful Xed Xy'
```

See also regular-expressions: alternation for more information regarding alternation and precedence rules in various regular expression implementations.

Cheatsheet and Summary

Note	Description
`|`	multiple RE combined as conditional OR
	each alternative can have independent anchors
`'|'.join(iterable)`	programmatically combine multiple RE
`()`	group pattern(s)
`a(b|c)d`	same as `abd|acd`
Alternation precedence	pattern which matches earliest in the input gets precedence
	tie-breaker is left-to-right if patterns have the same starting location
	robust solution: sort the alternations based on length, longest first
	`'|'.join(sorted(iterable, key=len, reverse=True))`

So, this chapter was about specifying one or more alternate matches within the same RE using the `|` metacharacter. Which can further be simplified using `()` grouping if the alternations have common portions. Among the alternations, earliest matching pattern gets precedence. Left-to-right ordering is used as a tie-breaker if multiple alternations have the same starting location. You also learnt ways to programmatically construct a RE.

Exercises

a) For the given list, filter all elements that start with `den` or end with `ly` .

```
>>> items = ['lovely', '1\ndentist', '2 lonely', 'eden', 'fly\n', 'dent']

##### add your solution here
['lovely', '2 lonely', 'dent']
```

b) For the given list, filter all elements having a line starting with `den` or ending with `ly` .

```
>>> items = ['lovely', '1\ndentist', '2 lonely', 'eden', 'fly\nfar', 'dent']

##### add your solution here
['lovely', '1\ndentist', '2 lonely', 'fly\nfar', 'dent']
```

c) For the given strings, replace all occurrences of `removed` or `reed` or `received` or `refused` with `X` .

```
>>> s1 = 'creed refuse removed read'
>>> s2 = 'refused reed redo received'

>>> pat = re.compile()        ##### add your solution here

>>> pat.sub('X', s1)
'cX refuse X read'
>>> pat.sub('X', s2)
'X X redo X'
```

d) For the given strings, replace all matches from the list `words` with `A` .

```
>>> s1 = 'plate full of slate'
>>> s2 = "slated for later, don't be late"
>>> words = ['late', 'later', 'slated']

>>> pat = re.compile()        ##### add your solution here

>>> pat.sub('A', s1)
'pA full of sA'
>>> pat.sub('A', s2)
"A for A, don't be A"
```

e) Filter all whole elements from the input list `items` based on elements listed in `words` .

```
>>> items = ['slate', 'later', 'plate', 'late', 'slates', 'slated ']
>>> words = ['late', 'later', 'slated']

>>> pat = re.compile()       ##### add your solution here

##### add your solution here
['later', 'late']
```

Escaping metacharacters

This chapter will show how to match metacharacters literally. Examples will be discussed for both manually as well as programmatically constructed patterns. You'll also learn about escape sequences supported by the `re` module.

Escaping with backslash

You have seen a few metacharacters and escape sequences that help to compose a RE. To match the metacharacters literally, i.e. to remove their special meaning, prefix those characters with a `\` (backslash) character. To indicate a literal `\` character, use `\\`. This assumes you are using raw strings and not normal strings.

```
# even though ^ is not being used as anchor, it won't be matched literally
>>> bool(re.search(r'b^2', 'a^2 + b^2 - C*3'))
False
# escaping will work
>>> bool(re.search(r'b\^2', 'a^2 + b^2 - C*3'))
True

# match ( or ) literally
>>> re.sub(r'\(|\)', '', '(a*b) + c')
'a*b + c'

# note that the input string is also a raw string here
>>> re.sub(r'\\', '/', r'\learn\by\example')
'/learn/by/example'
```

As emphasized earlier, regular expressions is just another tool to process text. Some examples and exercises presented in this book can be solved using normal string methods as well. It is a good practice to reason out whether regular expressions is needed for a given problem.

```
>>> eqn = 'f*(a^b) - 3*(a^b)'

# straightforward search and replace, no need RE shenanigans
>>> eqn.replace('(a^b)', 'c')
'f*c - 3*c'
```

re.escape()

Okay, what if you have a string variable that must be used to construct a RE — how to escape all the metacharacters? Relax, the `re.escape()` function has got you covered. No need to manually take care of all the metacharacters or worry about changes in future versions.

```
>>> expr = '(a^b)'
# print used here to show results similar to raw string
>>> print(re.escape(expr))
\(a\^b\)

# replace only at the end of string
>>> eqn = 'f*(a^b) - 3*(a^b)'
>>> re.sub(re.escape(expr) + r'\Z', 'c', eqn)
'f*(a^b) - 3*c'
```

Recall that in the Alternation section, `join` was used to dynamically construct RE pattern from an iterable of strings. However, that didn't handle metacharacters. Here are some examples on how you can use `re.escape()` so that the resulting pattern will match the strings from the input iterable literally.

```
# iterable of strings, assume alternation precedence sorting isn't needed
>>> terms = ['a_42', '(a^b)', '2|3']
# using 're.escape' and 'join' to construct the pattern
>>> pat1 = re.compile('|'.join(re.escape(s) for s in terms))
# using only 'join' to construct the pattern
>>> pat2 = re.compile('|'.join(terms))

>>> print(pat1.pattern)
a_42|\(a\^b\)|2\|3
>>> print(pat2.pattern)
a_42|(a^b)|2|3

>>> s = 'ba_423 (a^b)c 2|3 a^b'
>>> pat1.sub('X', s)
'bX3 Xc X a^b'
>>> pat2.sub('X', s)
'bXX (a^b)c X|X a^b'
```

Escape sequences

Certain characters like tab and newline can be expressed using escape sequences as `\t` and `\n` respectively. These are similar to how they are treated in normal string literals. However, `\b` is for word boundaries as seen earlier, whereas it stands for the backspace character in normal string literals.

The full list is mentioned at the end of docs.python: Regular Expression Syntax section as `\a \b \f \n \N \r \t \u \U \v \x \\` . Do read the documentation for details as well as how it differs for byte data.

```
>>> re.sub(r'\t', ':', 'a\tb\tc')
'a:b:c'

>>> re.sub(r'\n', ' ', '1\n2\n3')
'1 2 3'
```

⚠ If an escape sequence is not defined, you'll get an error.

```
>>> re.search(r'\e', 'hello')
re.error: bad escape \e at position 0
```

You can also represent a character using hexadecimal escape of the format `\xNN` where `NN` are exactly two hexadecimal characters. If you represent a metacharacter using escapes, it will be treated literally instead of its metacharacter feature.

```
# \x20 is space character
>>> re.sub(r'\x20', '', 'h e l l o')
'hello'

# \x7c is '|' character
>>> re.sub(r'2\x7c3', '5', '12|30')
'150'
>>> re.sub(r'2|3', '5', '12|30')
'15|50'
```

See ASCII code table for a handy cheatsheet with all the ASCII characters and their hexadecimal representations.

Octal escapes will be discussed in the Backreference section. The Codepoints and Unicode escapes section will discuss escapes for unicode characters using `\u` and `\U` .

Cheatsheet and Summary

Note	Description
`\`	prefix metacharacters with `\` to match them literally
`\\`	to match `\` literally
`re.escape()`	automatically escape all metacharacters
	ex: `'|'.join(re.escape(s) for s in iterable)`
`\t`	escape sequences like those supported in string literals
`\b`	word boundary in RE but backspace in string literals
`\e`	undefined escapes will result in an error
`\xNN`	represent a character using hexadecimal value
`\x7c`	will match `|` literally

This short chapter discussed how to match metacharacters literally. `re.escape()` helps if you are using input strings sourced from elsewhere to build the final RE. You also saw how to use escape sequences to represent characters and how they differ from normal string literals.

Exercises

a) Transform the given input strings to the expected output using the same logic on both strings.

```
>>> str1 = '(9-2)*5+qty/3-(9-2)*7'
>>> str2 = '(qty+4)/2-(9-2)*5+pq/4'

##### add your solution here for str1
'35+qty/3-(9-2)*7'
##### add your solution here for str2
'(qty+4)/2-35+pq/4'
```

b) Replace `(4)\|` with `2` only at the start or end of the given input strings.

```
>>> s1 = r'2.3/(4)\|6 foo 5.3-(4)\|'
>>> s2 = r'(4)\|42 - (4)\|3'
>>> s3 = 'two - (4)\\|\n'

>>> pat = re.compile()        ##### add your solution here

>>> pat.sub('2', s1)
'2.3/(4)\\|6 foo 5.3-2'
>>> pat.sub('2', s2)
'242 - (4)\\|3'
>>> pat.sub('2', s3)
'two - (4)\\|\n'
```

c) Replace any matching element from the list `items` with `X` for given the input strings. Match the elements from `items` literally. Assume no two elements of `items` will result in any matching conflict.

```
>>> items = ['a.b', '3+n', r'x\y\z', 'qty||price', '{n}']
>>> pat = re.compile()      ##### add your solution here

>>> pat.sub('X', '0a.bcd')
'0Xcd'
>>> pat.sub('X', 'E{n}AMPLE')
'EXAMPLE'
>>> pat.sub('X', r'43+n2 ax\y\ze')
'4X2 aXe'
```

d) Replace the backspace character `\b` with a single space character for the given input string.

```
>>> ip = '123\b456'
>>> ip
'123\x08456'
>>> print(ip)
12456

>>> re.sub()        ##### add your solution here
'123 456'
```

e) Replace all occurrences of `\e` with `e` .

```
>>> ip = r'th\er\e ar\e common asp\ects among th\e alt\ernations'

>>> re.sub()     ##### add your solution here
'there are common aspects among the alternations'
```

f) Replace any matching item from the list `eqns` with `X` for given the string `ip` . Match the items from `eqns` literally.

```
>>> ip = '3-(a^b)+2*(a^b)-(a/b)+3'
>>> eqns = ['(a^b)', '(a/b)', '(a^b)+2']
```

```
##### add your solution here

>>> pat.sub('X', ip)
'3-X*X-X+3'
```