Sample chapters

Python re(gex)?

a magical tool for text processing

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 learn Regular Expressions, a mini-programming language for all sorts of text processing needs.

The 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 sample input string and the output produced is essential. As an analogy, consider learning to drive a bike or a car - no matter how much you read about them or listen to explanations, you need to practice a lot and infer your own conclusions. Should you feel that copy-paste is ideal for you, code snippets are available chapter wise on GitHub.

The examples presented here have been tested with **Python version 3.7.1** and may include features not available in earlier versions. Unless otherwise noted, all examples and explanations are meant for ASCII characters only. The examples are copy pasted from Python REPL shell, but modified slightly for presentation purposes (like adding comments and blank lines, shortened error messages, skipping import statements, etc).

Prerequisites

Prior experience working with Python, should know concepts like string formats, string methods, list comprehension and so on.

If you have prior experience with a programming language, but new to Python, check out my GitHub repository on Python Basics before starting this book.

Acknowledgements

- Python documentation - manuals and tutorials
- /r/learnpython/ - helpful forum 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
- Cover image: draw.io, tree icon by Gopi Doraisamy under Creative Commons Attribution 3.0 Unported and wand icon by roundicons.com
- Warning and Info icons by Amada44 under public domain
- softwareengineering.stackexchange and skolakoda for programming quotes
- David Cortesi for helpful feedback on both the technical content and grammar issues
- **Kye** for spotting a typo

Special thanks to Al Sweigart, for introducing me to Python with his awesome automatetheboringstuff book and video course.

Feedback and Errata

I would highly appreciate if you'd let me know how you felt about this book, it would help to improve this book as well as my future attempts. Also, please do let me know if you spot any error or typo.

Issue Manager: https://github.com/learnbyexample/py_regular_expressions/issues

Goodreads: https://www.goodreads.com/book/show/47142552-python-re-gex

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a freelance trainer, author and mentor. His previous experience includes working as a Design Engineer at Analog Devices for more than 5 years. You can find his other works, primarily focused on Linux command line, text processing, scripting languages and curated lists, at https://github.com/learnbyexample. He has also been a technical reviewer for Command Line Fundamentals book and video course published by Packt.

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

License

Book version

2.1

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 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 in itself, 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 range function, string repetition operator and so on.

Here's 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, numbers, 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.

Further Reading

- The true power of regular expressions - it also includes a nice explanation of what *regular* means
- 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
- wikipedia: Regular expression - this article includes discussion on regular expressions as a formal language as well as details on various implementations

Regular Expression modules

In this chapter, you'll get an introduction to two regular expression modules. For some examples, the equivalent normal string method is shown for comparison. Regular expression features will be covered next chapter onwards.

re module

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

First up, a simple example to test whether a string is part of another string or not. Normally, you'd use the `in` operator. For regular expressions, use the `re.search` function. Pass the RE as first argument and string to test against as second argument. As a good practice, always use **raw strings** to construct the RE, unless other formats are required (will become clearer in coming chapters).

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

# check if 'sentence' contains the given string argument
>>> '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 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 the module is already loaded. The return value of `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 a later chapter. For presentation purposes, the examples will use `bool` function to show `True` or `False` depending on whether the RE pattern matched or not.

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's some generator expression examples.

```
>>> 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
```

Compiling regular expressions

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

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

>>> bool(pet.search('They bought a dog'))
True
>>> bool(pet.search('A cat crossed their path'))
False
```

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

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

```
# search for 'is' starting from 5th character of 'sentence' variable
>>> bool(word.search(sentence, 4))
True
>>> bool(word.search(sentence, 6))
False

# search for 'is' between 3rd and 4th characters
>>> bool(word.search(sentence, 2, 4))
True
```

bytes

To work with `bytes` data type, the RE must be of `bytes` data as well. Similar to `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

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

regex module

The third party `regex` module (https://pypi.org/project/regex/) is backward-compatible with the standard `re` module. The `regex` module also offers advanced features like those found in Perl regular expressions.

To install the module from command line, you can use either of these depending on your usage:

- `pip install regex` in a virtual environment
- `python3.7 -m pip install --user regex` for system wide accessibility

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

>>> bool(regex.search(r'is', sentence))
True
>>> bool(regex.search(r'xyz', sentence))
False
```

By default, `regex` module uses `VERSION0` which is compatible with the `re` module. `VERSION1` includes more features and its behavior may differ from the `re` module. Details will be discussed later.

Cheatsheet and Summary

Note	Description
docs.python: re	Python standard module for regular expressions
pypi: regex	3rd party module, compatible with `re` , has advanced features
`re.search(r'pat', s)`	Check if given pattern is present anywhere in input string
	Output is a `re.Match` object, usable in conditional expressions
	r-strings preferred to define RE
	Additionally, Python maintains a small cache of recent RE
`re.compile(r'pat')`	Compile a pattern for reuse, output is a `re.Pattern` object
`re.search(rb'pat', s)`	Use byte pattern for byte input

You might wonder why two regular expression modules are being presented in this book. The `re` module is good enough for most usecases. But if text processing occupies a large share of your work, the extra features of `regex` module would certainly come in handy. It would also make it easier to adapt from/to other programming languages. You can also consider always using the `regex` module for your project instead of having to decide which one to use depending on features required.

Exercises

Refer to exercises folder for input files required to solve the exercises.

a) For the given input file, print all lines containing the string `two`

```
# note that the expected output shown here is wrapped to fit pdf width
>>> filename = 'programming_quotes.txt'
>>> word = re.compile()     ##### add your solution here
>>> with open(filename, 'r') as ip_file:
...     for ip_line in ip_file:
...         if word.search(ip_line):
...             print(ip_line, end='')
...
"Some people, when confronted with a problem, think - I know, I'll use regular
expressions. Now they have two problems" by Jamie Zawinski
"So much complexity in software comes from trying to make one thing do two
things" by Ryan Singer
```

b) For the given input string, print all lines NOT containing the string `2`

```
>>> purchases = '''\
... apple 24
... mango 50
... guava 42
... onion 31
... water 10'''
>>> num = re.compile()      ##### add your solution here
>>> for line in purchases.split('\n'):
```

```
...     if not num.search(line):
...         print(line)
...
mango 50
onion 31
water 10
```

Anchors

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 `re` module. In later chapters, you'll learn how to define your own rules for restriction.

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 `\` (discussed in a later 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 pattern
>>> 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 pattern
>>> bool(re.search(r'are\Z', 'spare'))
True
>>> bool(re.search(r'are\Z', 'nearest'))
False

>>> words = ['surrender', 'unicorn', 'newer', 'door', 'empty', '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']
```

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

```
>>> word_pat = re.compile(r'\Acat\Z')
>>> bool(word_pat.search('cat'))
True
>>> bool(word_pat.search('cater'))
False
```

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

Use the optional start and end index arguments for `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 is narrowed and the anchor positions haven't changed. When slicing is used, you are creating an entirely new string object with its own anchor positions.

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

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

The `re.sub` function performs search and replace operation similar to the normal `replace` string method. Metacharacters and escape sequences differ between search and replacement sections. It will be discussed separately in later chapters, for now only normal strings will be used for replacements. You can emulate string concatenation operations by using the anchors by themselves as a pattern.

```
# insert text at the start of a string
# first argument to re.sub is the search RE
# second argument is the replacement value
# third argument is the string value to be acted upon
>>> re.sub(r'\A', r're', 'live')
'relive'
>>> re.sub(r'\A', r're', 'send')
'resend'

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

⚠ 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 'word' variable
>>> re.sub(r'\Acat', r'hack', word)
'hacker'
>>> word
'cater'

# need to explicitly assign the result if 'word' has to be changed
>>> word = re.sub(r'\Acat', r'hack', word)
>>> word
'hacker'
```

Line anchors

A string input may contain single or multiple lines. The newline character `\n` is used 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 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 the last newline character. However, `\Z` will always match the end of string, irrespective of what characters are 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 use the `re.MULTILINE` flag (or, `re.M` short form). The `flags` optional argument will be covered in more detail later.

```
# 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 complete 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.

```
# note that there is no \n at the end of this input string
>>> ip_lines = 'catapults\nconcatenate\ncat'
>>> print(re.sub(r'^', r'* ', ip_lines, flags=re.M))
* catapults
* concatenate
* cat

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

> ⚠ If you are dealing with Windows OS based text files, you'll have to convert `\r\n` line endings to `\n` first. Which is easily handled by many of the Python functions and methods. For example, you can specify which line ending to use for `open` function, the `split` string method handles all whitespaces by default and so on. Or, you can handle `\r` as optional character with quantifiers (covered later).

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

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

You can get lot more creative with using word boundary as a pattern by itself:

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

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

# make a programming statement more readable
# shown for illustration purpose only, won't work for all cases
>>> re.sub(r'\b', r' ', 'foo_baz=num1+35*42/num2')
' foo_baz = 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', r' ', 'foo_baz=num1+35*42/num2').strip()
'foo_baz = 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 start of word
>>> re.sub(r'\Bpar', r'X', words)
'par sX apXent sXe part'
# replace 'par' at end of word but not whole word 'par'
>>> re.sub(r'\Bpar\b', r'X', words)
'par sX apparent spare part'
# replace 'par' if it is not end of word
>>> re.sub(r'par\B', r'X', words)
'par spar apXent sXe Xt'
# replace 'par' if it is surrounded by word characters
>>> re.sub(r'\Bpar\B', r'X', words)
'par spar apXent sXe part'
```

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

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

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

Cheatsheet and Summary

Note	Description
`\A`	restricts the match to start of string
`\Z`	restricts the match to end of string
`re.sub(r'pat', r'replace', s)`	search and replace
`\n`	line separator, dos-style files need special attention
metacharacter	characters with special meaning in RE
`^`	restricts the match to start of line
`$`	restricts the match to end of line
`re.MULTILINE` or `re.M`	flag to treat input as multiline string
`\b`	restricts the match to start/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 comprehensions. Practice, experience and imagination would help you construct creative solutions. In coming chapters, you'll see more applications of anchors as well as the `\G` anchor which is best understood in combination with other regular expression features.

Exercises

a) For the given url, count the total number of lines that contain `is` or `the` as whole words. **Note** that each `line` in the `for` loop will be of `bytes` data type.

```
>>> import urllib.request
>>> scarlet_pimpernel_link = r'https://www.gutenberg.org/cache/epub/60/pg60.txt'
>>> word1 = re.compile()    ##### add your solution here
>>> word2 = re.compile()    ##### add your solution here
>>> count = 0
>>> with urllib.request.urlopen(scarlet_pimpernel_link) as ip_file:
...     for line in ip_file:
...         if word1.search(line) or word2.search(line):
...             count += 1
...
>>> print(count)
3737
```

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

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

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

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

```
>>> words = ['hi42bye', 'nice1423', 'bad42', 'cool_42a', 'fake4b']

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

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

```
>>> foo = ['lovely', '1 dentist', '2 lonely', 'eden', 'fly away', 'dent']

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

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

```
>>> para = '''\
... ball fall wall tall
... mall call ball pall
... wall mall ball fall'''

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

Alternation and Grouping

Many a times, you'd want to search for multiple terms. In a conditional expression, you can use the logical operators to combine multiple conditions. With regular expressions, the `|` metacharacter is similar to logical OR. The RE will match if any of the expression separated by `|` is satisfied. These can have their own independent anchors as well.

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

# replace either 'cat' at start of string or 'cat' at end of word
>>> re.sub(r'\Acat|cat\b', r'X', 'catapults concatenate cat scat')
'Xapults concatenate X sX'

# replace either 'cat' or 'dog' or 'fox' with 'mammal'
>>> re.sub(r'cat|dog|fox', r'mammal', 'cat dog bee parrot fox')
'mammal mammal bee parrot mammal'
```

You might infer from 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), r'mammal', 'cat dog bee parrot fox')
'mammal mammal bee parrot mammal'
```

Often, there are some common things among the RE alternatives. It could be common characters or qualifiers like the anchors. 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', r'X', 'red reform read arrest')
'red X read arX'
# with grouping
>>> re.sub(r're(form|st)', r'X', 'red reform read arrest')
'red X read arX'

# without grouping
>>> re.sub(r'\bpar\b|\bpart\b', r'X', 'par spare part party')
```

```
'X spare X party'
# taking out common anchors
>>> re.sub(r'\b(par|part)\b', r'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 alternate
>>> re.sub(r'\bpar(|t)\b', r'X', 'par spare part party')
'X spare X party'
```

There's lot more features to grouping than just forming terser RE. For now, this is a good place to show how to incorporate normal strings (could be a variable, result from an expression, etc) while building a regular expression. For example, adding anchors to 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), r'X', 'cater cat concatenate par spare')
'Xer X conXenate X sXe'

# note how raw string is used on either side of concatenation
# avoid f-strings unless you know how to compensate for RE
>>> alt = re.compile(r'\b(' + '|'.join(words) + r')\b')
# only whole words will be replaced now
>>> alt.sub(r'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 examples with `join` method, the string iterable elements do not contain any special regular expression characters. How to deal with strings that have metacharacters will be discussed in a later chapter.

Precedence rules

There's some tricky situations when using alternation. If it is used for testing a match to get `True/False` against a string input, there is no ambiguity. However, for other things 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` object is used in the examples below for illustration.

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

# span shows the start and end+1 index of 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 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', r'X', words, count=1)
'liX elephant are rope not'
>>> re.sub(r'ant|on', r'X', words, count=1)
'liX elephant are rope not'
```

What happens if alternatives match on same index? The precedence is then 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 same
# so, which one gets replaced depends on the order of alternation
>>> re.sub(r'year|years', r'X', mood, count=1)
'best Xs'
>>> re.sub(r'years|year', r'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', r'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', r'X', words)
'eX xerox at mX pXt leXn eye'
# phew, finally this one works as needed
>>> re.sub(r'are|art|ar', r'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(r'X', 'hands handful handed handy')
'Xs X Xed X'
# without sorting, alternation order will come into play
>>> re.sub('|'.join(words), r'X', 'hands handful handed handy')
'Xs Xful Xed Xy'
```

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 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 `|` metacharacter. Which can further be simplified using `()` grouping if the alternations have common aspects. Among the alternations, earliest matching pattern gets precedence. Left to right ordering is used as a tie-breaker if multiple alternations match starting from same location. You also learnt ways to programmatically construct a RE.

Exercises

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

```
>>> foo = ['lovely', '1 dentist', '2 lonely', 'eden', 'fly away', 'dent']

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

b) For the given url, count the total number of lines that contain `removed` or `rested` or `received` or `replied` or `refused` or `retired` as whole words. **Note** that each `line` in the `for` loop will be of `bytes` data type.

```
>>> import urllib.request
>>> scarlet_pimpernel_link = r'https://www.gutenberg.org/cache/epub/60/pg60.txt'
>>> words = re.compile()    ##### add your solution here
>>> count = 0
>>> with urllib.request.urlopen(scarlet_pimpernel_link) as ip_file:
...     for line in ip_file:
...         if words.search(line):
...             count += 1
...
>>> print(count)
83
```