learnbyexample

UNDERSTANDING RUBY REGEXP

- ✓ 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. Make an effort to understand the sample input as well as the solution presented and check if the output changes (or not!) when you alter some part of the input and the command. 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 Ruby syntax and features like string formats, string and Enumerable methods.

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 new to Ruby, check out the official quickstart guide.

Conventions

- The examples presented here have been tested with **Ruby version 3.3.0** and includes features not available in earlier versions.
- Code snippets shown are copy pasted from the `irb --simple-prompt` 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. `nil` return value is not shown for `puts` statements. Error messages are shortened. 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 Ruby_Regexp repo has all the code snippets used in examples, exercises and other details related to 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

- ruby-lang documentation — manuals and tutorials
- /r/ruby/ and /r/regex/ — helpful forum for beginners and experienced programmers alike
- stackoverflow — for getting answers to pertinent questions on Ruby 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
- gmovchan for spotting a typo
- **KOTP** for spotting grammatical mistakes

Special thanks to Allen Downey. An attempt at translating his book Think Python to Think Ruby gave me the confidence to publish my own book.

Feedback and Errata

I would highly appreciate it 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/Ruby_Regexp/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

3.0

See Version_changes.md to track changes across book versions.

Why is it needed?

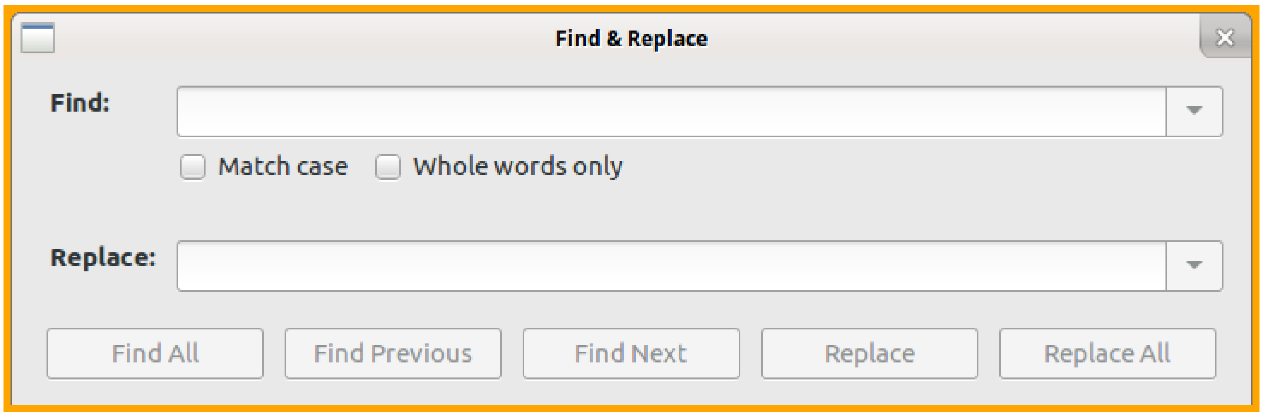
Regular Expressions is a versatile tool for text processing. You'll find them included as part of the standard library of most programming languages that are used for scripting purposes. If not, you can usually find a third-party library. Syntax and features of regular expressions vary from language to language. Ruby's offering is based upon the Onigmo regular expressions library.

The `String` 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 and string repetition operators and so on.

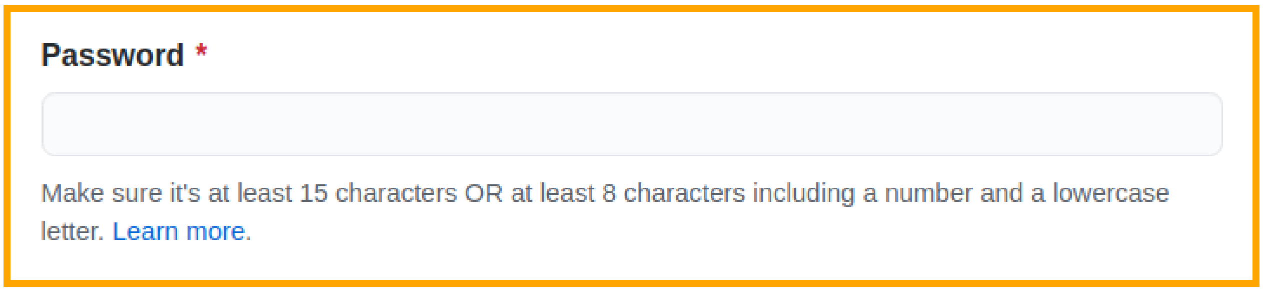
Here are some common use cases:

- 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.

- Regexp introduction
- Anchors
- Alternation and Grouping
- Escaping metacharacters
- Dot metacharacter and Quantifiers
- Interlude: Tools for debugging and visualization
- Working with matched portions
- Character class
- Groupings and backreferences
- Interlude: Common tasks
- Lookarounds
- Modifiers
- Unicode
- Further Reading

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.

Regexp introduction

In this chapter, you'll get to know how to declare and use regexps. For some examples, the equivalent normal string method is also shown for comparison. The main focus will be to get you comfortable with syntax and text processing examples. Three methods will be introduced in this chapter. The `match?` method to search if the input contains a string and the `sub` and `gsub` methods to substitute a portion of the input with something else. Regular expression features will be covered from the next chapter onwards.

This book will use the terms **regular expressions** and **regexp** interchangeably.

Regexp documentation

It is always a good idea to know where to find the documentation. Visit ruby-doc: Regexp for information on `Regexp` class, available methods, syntax, features, examples and more. Here's a quote from an older version of the documentation:

> Regular expressions (*regexps*) are patterns which describe the contents of a string. They're used for testing whether a string contains a given pattern, or extracting the portions that match. They are created with the `/pat/` and `%r{pat}` literals or the `Regexp.new` constructor.

match? method

First up, a simple example to test whether a string is part of another string or not. Normally, you'd use the `include?` method and pass a string as an argument. For regular expressions, use the `match?` method and enclose the search string within `//` delimiters (regexp literal).

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

# check if 'sentence' contains the given search string
>> sentence.include?('is')
=> true
>> sentence.include?('z')
=> false

# check if 'sentence' matches the pattern described by the regexp argument
>> sentence.match?(/is/)
=> true
>> sentence.match?(/z/)
=> false

# unlike the include? method, match? is commutative
>> /ring/.match?(sentence)
=> true
```

The `match?` method accepts an optional second argument which specifies the index to start searching from.

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

>> sentence.match?(/is/, 2)
=> true
>> sentence.match?(/is/, 6)
=> false
```

Regexp modifiers

Some of the regular expressions functionality is enabled by passing modifiers, represented by an alphabet character. Modifiers are similar to command-line options, for example `grep -i` will perform case insensitive matching. These will be discussed in detail in the Modifiers chapter. Here's an example for the `i` modifier.

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

>> sentence.match?(/this/)
=> false

# 'i' modifier enables case insensitive matching
>> sentence.match?(/this/i)
=> true
```

Regexp literal reuse and interpolation

The regexp literal can be saved in a variable. This helps to improve code clarity, pass around as method arguments, enable reuse, etc.

```
>> pet = /dog/i
>> pet
=> /dog/i

>> 'They bought a Dog'.match?(pet)
=> true
>> 'A cat crossed their path'.match?(pet)
=> false
```

Similar to double quoted string literals, you can use interpolation and escape sequences within a regexp literal. See ruby-doc: Strings for syntax details on string escape sequences. Regexp literals have their own special escapes, which will be discussed in the Escape sequences section.

```
>> "cat\tdog".match?(/\t/)
=> true
>> "cat\tdog".match?(/\a/)
=> false

>> greeting = 'hi'
>> /#{greeting} there/
=> /hi there/
>> /#{greeting.upcase} there/
```

```
=> /HI there/
>> /#{2**4} apples/
=> /16 apples/
```

sub and gsub methods

For search and replace requirements, use the `sub` or `gsub` methods. The `sub` method will replace only the first occurrence of the match, whereas `gsub` will replace all the occurrences. The regexp pattern to match against the input string has to be passed as the first argument. The second argument specifies the string to replace the portions matched by the pattern.

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

# replace the first occurrence of 'e' with 'E'
>> greeting.sub(/e/, 'E')
=> "HavE a nice weekend"

# replace all occurrences of 'e' with 'E'
>> greeting.gsub(/e/, 'E')
=> "HavE a nicE wEEkEnd"
```

Use the `sub!` and `gsub!` methods for in-place substitution.

```
>> word = 'cater'

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

# this will modify the 'word' variable itself
>> word.sub!(/cat/, 'wag')
=> "wager"
>> word
=> "wager"
```

Regexp operators

Ruby also provides operators for regexp matching.

- `=~` match operator returns the index of the first match and `nil` if a match is not found
- `!~` match operator returns `true` if the input string *doesn't* contain the given regexp and `false` otherwise
- `===` match operator returns `true` or `false` similar to the `match?` method

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

# can also use: /is/ =~ sentence
>> sentence =~ /is/
=> 2
```

```
>> sentence =~ /z/
=> nil

# can also use: /z/ !~ sentence
>> sentence !~ /z/
=> true
>> sentence !~ /is/
=> false
```

Just like the `match?` method, both `=~` and `!~` can be used in a conditional statement.

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

>> puts 'hi' if sentence =~ /is/
hi

>> puts 'oh' if sentence !~ /z/
oh
```

The `===` operator comes in handy with Enumerable methods like `grep` , `grep_v` , `all?` , `any?` , etc.

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

# regexp literal has to be on LHS and input string on RHS
>> /is/ === sentence
=> true
>> /z/ === sentence
=> false

>> words = %w[cat attempt tattle]
>> words.grep(/tt/)
=> ["attempt", "tattle"]
>> words.all?(/at/)
=> true
>> words.none?(/temp/)
=> false
```

A key difference from the `match?` method is that these operators will also set regexp related global variables.

Cheatsheet and Summary

Note	Description
ruby-doc: Regexp	Ruby Regexp documentation
Onigmo doc	Onigmo library documentation
`/pat/` or `%r{pat}`	regexp literal interpolation and escape sequences can also be used
`var = /pat/`	save regexp literals in a variable

Note	Description
`/pat1#{expr}pat2/`	use the result of an expression to build regexps
`s.match?(/pat/)`	check if string `s` matches the pattern `/pat/`
	returns `true` or `false`
`s.match?(/pat/, 3)`	optional 2nd argument changes the starting index of search
`/pat/i`	modifier `i` matches alphabets case insensitively
`s.sub(/pat/, 'replace')`	search and replace the first matching occurrence
	use `gsub` to replace all occurrences
	use `sub!` and `gsub!` for in-place substitutions
`s =~ /pat/` or `/pat/ =~ s`	returns the index of the first match or `nil`
`s !~ /pat/` or `/pat/ !~ s`	returns `true` if no match, `false` otherwise
`/pat/ === s`	returns `true` or `false` similar to `match?`
	these operators will also set regexp global variables

This chapter introduced the `Regexp` class and the methods `match?`, `sub` and `gsub` were discussed. You also learnt how to save and reuse regexp literals, how to specify modifiers and how to use regexp operators.

You might wonder why there are so many ways to test a matching condition with regexps. The most common approach is to use the `match?` method in a conditional statement. If you need the position of match, use the `=~` operator or the `index` method. The `===` operator is usually relevant in Enumerable methods. Usage of global variables will be covered in later chapters. The `=~` and `!~` operators are also prevalent in command-line usage (see my Ruby One-Liners Guide for examples).

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 the 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 the 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.

1) 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'

>> line1.match?()       ##### add your solution here
=> false
>> line2.match?()       ##### add your solution here
=> true
```

2) Check if the given input strings contain `two` irrespective of case.

```
>> s1 = 'Their artwork is exceptional'
>> s2 = 'one plus tw0 is not three'
>> s3 = 'TRUSTWORTHY'

>> pat1 = //        ##### add your solution here

>> pat1.match?(s1)
=> true
>> pat1.match?(s2)
=> false
>> pat1.match?(s3)
=> true
```

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

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

>> ip.gsub(//, 'five')      ##### add your solution here
=> "They ate five apples and five oranges"
```

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

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

>> ip.sub(//, 'five')       ##### add your solution here
=> "They ate five apples and 5 oranges"
```

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

```
>> items = %w[goal new user sit eat dinner]

>> items.grep_v(//)     ##### add your solution here
=> ["goal", "sit"]
```

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

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

>> ip.gsub(//, 'X')     ##### add your solution here
=> "This X should not be XD"
```

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

```
'> purchases = %q{items qty
'> apple 24
'> mango 50
'> guava 42
'> onion 31
>> water 10}

>> num = //     ##### add your solution here
>> puts purchases.each_line.grep_v(num)
items qty
mango 50
onion 31
water 10
```

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

```
>> items = %w[goal new user sit eat dinner]

>> items.filter { }     ##### add your solution here
=> ["goal", "new", "eat"]
```

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

```
>> items = %w[goal new user sit eat dinner]

>> items.filter { }     ##### add your solution here
=> ["new", "dinner"]
```

10) 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"
```

11) Find the starting index of the first occurrence of `is` for the given input string.

```
>> ip = 'match this after the history lesson'

##### add your solution here
=> 8
```

Anchors

Now that you're familiar with regexp syntax and some of the methods, 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 regular expression features. 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 regexp parlance. In case you need to match those characters literally, you need to escape them with a `\` character (discussed in the Escaping with backslash section).

String anchors

This restriction is about qualifying a regexp to match only at the start or end of an input string. These provide functionality similar to the string methods `start_with?` and `end_with?`. There are three different escape sequences related to string level regexp anchors. First up is `\A` which restricts the matching to the start of string.

```
# \A is placed as a prefix to the search term
>> 'cater'.match?(/\Acat/)
=> true
>> 'concatenation'.match?(/\Acat/)
=> false

>> "hi hello\ntop spot".match?(/\Ahi/)
=> true
>> "hi hello\ntop spot".match?(/\Atop/)
=> false
```

To restrict the match to the end of string, `\z` (lowercase `z`) is used.

```
# \z is placed as a suffix to the search term
>> 'spare'.match?(/are\z/)
=> true
>> 'nearest'.match?(/are\z/)
=> false

>> words = %w[surrender unicorn newer door empty eel pest]
>> words.grep(/er\z/)
=> ["surrender", "newer"]
>> words.grep(/t\z/)
=> ["pest"]
```

There is another end of string anchor `\Z` (uppercase). It is similar to `\z` but if newline is the last character, then `\Z` allows matching just before the newline character.

```
# same result for both \z and \Z
# as there is no newline character at the end of string
>> "spare\ndare".sub(/are\z/, 'X')
=> "spare\ndX"
```

```
>> "spare\ndare".sub(/are\Z/, 'X')
=> "spare\ndX"

# different results as there is a newline character at the end of string
>> "spare\ndare\n".sub(/are\z/, 'X')
=> "spare\ndare\n"
>> "spare\ndare\n".sub(/are\Z/, 'X')
=> "spare\ndX\n"
```

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

```
>> 'cat'.match?(/\Acat\z/)
=> true
>> 'cater'.match?(/\Acat\z/)
=> false
>> 'concatenation'.match?(/\Acat\z/)
=> false
```

The anchors can be used by themselves as a pattern. Helps to insert text at the start or end of string, emulating string concatenation operations. These might not feel like useful capability, but combined with other regexp features they become quite a handy tool.

```
>> 'live'.sub(/\A/, 're')
=> "relive"
>> 'send'.sub(/\A/, 're')
=> "resend"

>> 'cat'.sub(/\z/, 'er')
=> "cater"
>> 'hack'.sub(/\z/, 'er')
=> "hacker"
```

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'

>> pets.match?(/^cat/)
=> true
>> pets.match?(/^dog/)
=> false

>> pets.match?(/dog$/)
=> true
>> pets.match?(/^dog$/)
=> false
```

Here are some multiline examples to distinguish line anchors from string anchors.

```
# check if any line in the string starts with 'top'
>> "hi hello\ntop spot".match?(/^top/)
=> true

# check if any line in the string ends with 'er'
>> "spare\npar\nera\ndare".match?(/er$/)
=> false

# filter lines ending with 'are'
>> "spare\npar\ndare".each_line.grep(/are$/)
=> ["spare\n", "dare"]

# check if any whole line in the string is 'par'
>> "spare\npar\ndare".match?(/^par$/)
=> true
```

Just like string anchors, you can use the line anchors by themselves as a pattern. `gsub` and `puts` will be used here to better illustrate the transformation. The `gsub` method returns an Enumerator if you don't specify a replacement string nor pass a block. That paves a way to use all those wonderful Enumerator and Enumerable methods.

```
>> str = "catapults\nconcatenate\ncat"

>> puts str.gsub(/^/, '1: ')
1: catapults
1: concatenate
1: cat
>> puts str.gsub(/^/).with_index(1) { "#{_2}: " }
1: catapults
2: concatenate
3: cat

>> puts str.gsub(/$/, '.')
catapults.
concatenate.
cat.
```

If there is a newline character at the end of the input string, there is an additional end of line match but no additional start of line match.

```
>> puts "1\n2\n".gsub(/^/, 'fig ')
fig 1
fig 2
>> puts "1\n\n".gsub(/^/, 'fig ')
fig 1
fig

# note the number of lines in the output
>> puts "1\n2\n".gsub(/$/, ' banana')
1 banana
```

```
2 banana
 banana
>> puts "1\n\n".gsub(/$/, ' banana')
1 banana
 banana
 banana
```

⚠️ If you are dealing with Windows OS based text files, you'll have to convert the `\r\n` line endings to `\n` first. Which is easily handled by many of the Ruby methods. For example, you can specify the line ending to use for the `File.open` method, 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 Greedy quantifiers section for examples).

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 boundaries without a word character.

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

# replace 'par' irrespective of where it occurs
>> words.gsub(/par/, 'X')
=> "X sX apXent sXe Xt"

# replace 'par' only at the start of word
>> words.gsub(/\bpar/, 'X')
=> "X spar apparent spare Xt"

# replace 'par' only at the end of word
>> words.gsub(/par\b/, 'X')
=> "X sX apparent spare part"

# replace 'par' only if it is not part of another word
>> words.gsub(/\bpar\b/, 'X')
=> "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 the 'tr' string method
```

```
>> words = 'par spar apparent spare part'
>> puts words.gsub(/\b/, '"').tr(' ', ',')
"par","spar","apparent","spare","part"

>> '-----hello-----'.gsub(/\b/, ' ')
=> "----- hello -----"

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

Opposite Word anchors

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

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

```
>> 'copper'.gsub(/\b/, ':')
=> ":copper:"
>> 'copper'.gsub(/\B/, ':')
=> "c:o:p:p:e:r"

>> '-----hello-----'.gsub(/\b/, ' ')
=> "----- hello -----"
>> '-----hello-----'.gsub(/\B/, ' ')
=> " - - - - -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
`\Z`	restricts the match to end or just before a newline at the end of string
`\n`	line separator
	DOS-style files need special attention
metacharacter	characters with special meaning in regexp
`^`	restricts the match to the start of line
`$`	restricts the match to the end of line
`\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. At the same time, regular expression is but another tool for text processing problems. Often, you'd get simpler solution by combining regular expressions with other string and Enumerable methods. Practice, experience and imagination would help you in constructing creative solutions. In the coming chapters, you'll see more applications of anchors as well as the `\G` anchor which is best understood in combination with other regexp features.

Exercises

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

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

>> pat =        ##### add your solution here

>> pat.match?(line1)
=> true
>> pat.match?(line2)
=> false
>> pat.match?(line3)
=> true
>> pat.match?(line4)
=> false
```

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

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

>> words.gsub()     ##### add your solution here
=> "bred brown spread credible brown."
```

3) For the given input array, filter elements that contain `42` surrounded by word characters.

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

>> items.grep()     ##### add your solution here
=> ["hi42bye", "nice1423", "cool_42a", "_42_"]
```

4) For the given input array, filter elements that start with `den` or end with `ly` .

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

>> items.filter { }     ##### add your solution here
=> ["lovely", "2 lonely", "dent"]
```

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

```
'> para = %q{(mall) call ball pall
'> ball fall wall tall
'> mall call ball pall
'> wall mall ball fall
'> mallet wallet malls
>> mall:call:ball:pall}

>> puts para.gsub()     ##### 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
```

6) For the given array, filter elements having a line starting with `den` or ending with `ly` .

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

>> items.filter { }     ##### add your solution here
=> ["lovely", "1\ndentist", "2 lonely", "fly\nfar", "dent"]
```

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

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

>> items.grep()     ##### add your solution here
=> ["12\nThree", "12\nthree"]
```

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

```
>> items = %w[handed hand handy unhanded handle hand-2]

>> items.map { }        ##### add your solution here
=> ["Xed", "hand", "Xy", "unhanded", "Xle", "hand-2"]
```

9) For the given input array, filter all elements starting with `h` . Additionally, replace `e` with

`X` for these filtered elements.

```
>> items = %w[handed hand handy unhanded handle hand-2]

>> items.filter_map { }     ##### add your solution here
=> ["handXd", "hand", "handy", "handlX", "hand-2"]
```

Alternation and Grouping

Many a times, you want to check if the input string matches multiple patterns. For example, whether a product's color is *green* or *blue* or *red*. This chapter will show how to use alternation for such cases. These patterns can have some common elements between them, in which case grouping helps to form terser regexps. 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 conditions 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 patterns 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 = /cat|dog/
>> 'I like cats'.match?(pet)
=> true
>> 'I like dogs'.match?(pet)
=> true
>> 'I like parrots'.match?(pet)
=> false

# replace 'cat' at the start of string or 'cat' at the end of word
>> 'catapults concatenate cat scat cater'.gsub(/\Acat|cat\b/, 'X')
=> "Xapults concatenate X sX cater"

# replace 'cat' or 'dog' or 'fox' with 'mammal'
>> 'cat dog bee parrot fox'.gsub(/cat|dog|fox/, 'mammal')
=> "mammal mammal bee parrot mammal"
```

Regexp.union method

You might infer from the above examples that there can be cases where many alternations are required. The `Regexp.union` method can be used to build the alternation list automatically. It accepts an array as an argument or a list of comma separated arguments.

```
>> Regexp.union('car', 'jeep')
=> /car|jeep/

>> words = %w[cat dog fox]
>> pat = Regexp.union(words)
>> pat
=> /cat|dog|fox/
>> 'cat dog bee parrot fox'.gsub(pat, 'mammal')
=> "mammal mammal bee parrot mammal"
```

In the above examples, the elements do not contain any special regexp characters. Handling strings that contain metacharacters will be discussed in the Regexp.escape method section.

Grouping

Often, there are some common portions among the regexp 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
>> 'red reform read arrest'.gsub(/reform|rest/, 'X')
=> "red X read arX"
# with grouping
>> 'red reform read arrest'.gsub(/re(form|st)/, 'X')
=> "red X read arX"

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

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

Regexp.source method

The `Regexp.source` method helps to interpolate a regexp literal inside another regexp. For example, adding anchors to an alternation list created using the `Regexp.union` method.

```
>> words = %w[cat par]
>> alt = Regexp.union(words)
>> alt
=> /cat|par/
>> alt_w = /\b(#{alt.source})\b/
>> alt_w
=> /\b(cat|par)\b/

>> 'cater cat concatenate par spare'.gsub(alt, 'X')
=> "Xer X conXenate X sXe"
>> 'cater cat concatenate par spare'.gsub(alt_w, 'X')
=> "cater X concatenate X spare"
```

The above example will work without the `Regexp.source` method too, but you'll see that `/\b(#{alt})\b/` gives `/\b((?-mix:cat|par))\b/` instead of `/\b(cat|par)\b/` . Their meaning will be explained in the Modifiers chapter.

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 Ruby, the alternative which matches earliest in the input string gets precedence. The regexp operator `=~` is handy to illustrate this concept.

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

>> words =~ /on/
=> 2
>> words =~ /ant/
=> 10

# starting index of 'on' < index of 'ant' for the given string input
# so 'on' will be replaced irrespective of the order
>> words.sub(/on|ant/, 'X')
=> "liX elephant are rope not"
>> words.sub(/ant|on/, 'X')
=> "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'

>> mood =~ /year/
=> 5
>> mood =~ /years/
=> 5

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

Another example with `gsub` to drive home the issue:

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

```
# same as: gsub(/ar/, 'X')
>> words.gsub(/ar|are|art/, 'X')
=> "eX xerox at mXe pXt leXn eye"

# same as: gsub(/are|ar/, 'X')
>> words.gsub(/are|ar|art/, 'X')
=> "eX xerox at mX pXt leXn eye"

# phew, finally this one works as needed
>> words.gsub(/are|art|ar/, 'X')
=> "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 = %w[hand handy handful]

>> alt = Regexp.union(words.sort_by { |w| -w.length })
>> alt
=> /handful|handy|hand/

>> 'hands handful handed handy'.gsub(alt, 'X')
=> "Xs X Xed X"

# alternation order will come into play if you don't sort them properly
>> 'hands handful handed handy'.gsub(Regexp.union(words), 'X')
=> "Xs Xful Xed Xy"
```

Cheatsheet and Summary

Note	Description
`|`	helps to combine multiple patterns as conditional OR
	each alternative can have independent anchors
`Regexp.union(array)`	programmatically combine multiple strings/regexps
`()`	group pattern(s)
`a(b|c)d`	same as `abd|acd`
`/#{pat.source}/`	interpolate a regexp literal inside another regexp
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
	for ex: `Regexp.union(words.sort_by { |w| -w.length })`

So, this chapter was about specifying one or more alternate matches within the same regexp 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 couple of `Regexp` methods that help to programmatically construct a regexp literal.

Exercises

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

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

>> items.grep()      ##### add your solution here
=> ["lovely", "2 lonely", "dent"]
```

2) For the given array, filter elements having a line starting with `den` or ending with `ly` .

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

>> items.grep()      ##### add your solution here
=> ["lovely", "1\ndentist", "2 lonely", "fly\nfar", "dent"]
```

3) 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 =        ##### add your solution here

>> s1.gsub(pat, 'X')
=> "cX refuse X read"
>> s2.gsub(pat, 'X')
=> "X X redo X"
```

4) For the given strings, replace all matches from the array `words` with `A` .

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

>> pat =        ##### add your solution here

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

5) Filter all whole elements from the input array `items` that exactly matches any of the elements present in the array `words` .

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

>> pat =        ##### add your solution here

>> items.grep(pat)
=> ["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 regexp and how they differ from strings.

Escaping with backslash

You have seen a few metacharacters and escape sequences for composing regexp literals. 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 `\\` .

To spice up the examples a bit, block form has been used below to modify the matched portion of string with an expression. In later chapters, you'll see more ways to work directly with the matched portions.

```
# even though ^ is not being used as anchor, it won't be matched literally
>> 'a^2 + b^2 - C*3'.match?(/b^2/)
=> false
# escaping will work
>> 'a^2 + b^2 - C*3'.gsub(/(a|b)\^2/) { _1.upcase }
=> "A^2 + B^2 - C*3"

# match ( or ) literally
>> '(a*b) + c'.gsub(/\(|\)/, '')
=> "a*b + c"

>> '\learn\by\example'.gsub(/\\/, '/')
=> "/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 regexp shenanigans
>> eqn.gsub('(a^b)', 'c')
=> "f*c - 3*c"
```

Regexp.escape method

How to escape all the metacharacters when a regexp is constructed dynamically? Relax, the `Regexp.escape` method has got you covered. No need to manually take care of all the metacharacters or worry about changes in future versions.

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

>> puts Regexp.escape(expr)
\(a\^b\)
```

```
# replace only at the end of string
>> eqn.sub(/#{Regexp.escape(expr)}\z/, 'c')
=> "f*(a^b) - 3*c"
```

The `Regexp.union` method automatically applies escaping for string arguments.

```
# array of strings, assume alternation precedence sorting isn't needed
>> terms = %w[a_42 (a^b) 2|3]

>> pat = Regexp.union(terms)
>> pat
=> /a_42|\(a\^b\)|2\|3/

>> 'ba_423 (a^b)c 2|3 a^b'.gsub(pat, 'X')
=> "bX3 Xc X a^b"
```

`Regexp.union` will also take care of mixing string and regexp patterns correctly. The grouping `(?-mix:` seen in the output below will be explained in the Modifiers chapter.

```
>> Regexp.union(/^cat|dog$/, 'a^b')
=> /(?-mix:^cat|dog$)|a\^b/
```

Escaping delimiter

Another character to keep track for escaping is the delimiter used to define the regexp literal. You can also use a different delimiter with `%r` to avoid or minimize escaping. You need not worry about unescaped delimiters inside `#{}` interpolation.

```
>> path = '/home/joe/report/sales/ip.txt'

# \/ is known as 'leaning toothpick syndrome'
>> path.sub(/\A\/home\/joe\//, '~/')
=> "~/report/sales/ip.txt"

# a different delimiter improves readability and reduces typos
>> path.sub(%r#\A/home/joe/#, '~/')
=> "~/report/sales/ip.txt"
```

Escape sequences

In regexp literals, 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 (see ruby-doc: Strings for details). However, escapes like `\b` (word boundary) and `\s` (see the Escape sequence sets section) are different for regexps. Also, octal escapes `\nnn` have to be three digits to avoid conflict with Backreferences.

```
>> "a\tb\tc".gsub(/\t/, ':')
=> "a:b:c"

>> "1\n2\n3".gsub(/\n/, ' ')
=> "1 2 3"
```

If an escape sequence is not defined, it'll match the character that is escaped. For example, `\%` will match `%` and not `\` followed by `%` .

```
>> 'h%x'.match?(/h\%x/)
=> true
>> 'h\%x'.match?(/h\%x/)
=> false

>> 'hello'.match?(/\l/)
=> true
```

If you represent a metacharacter using escapes, it will be treated literally instead of its metacharacter feature.

```
# \x20 is space character in hexadecimal format
>> 'h e l l o'.gsub(/\x20/, '')
=> "hello"
# \053 is + character in octal format
>> 'a+b'.match?(/a\053b/)
=> true

# \x7c is '|' character
>> '12|30'.gsub(/2\x7c3/, '5')
=> "150"
>> '12|30'.gsub(/2|3/, '5')
=> "15|50"
```

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

The Codepoints and Unicode escapes section will discuss escapes for unicode characters.

Cheatsheet and Summary

Note	Description
`\`	prefix metacharacters with `\` to match them literally
`\\`	to match `\` literally
`Regexp.escape(s)`	automatically escape all metacharacters for string `s`
	`Regexp.union` also automatically escapes string arguments
`%r`	helps to avoid/reduce escaping the delimiter character
`\t`	escape sequences like those supported in string literals
	but escapes like `\b` and `\s` have different meaning in regexps
`\%`	undefined escapes will match the character it escapes
`\x7c`	will match `|` literally
	instead of acting as an alternation metacharacter

Exercises

1) 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'

>> str1.gsub()      ##### add your solution here
=> "35+qty/3-(9-2)*7"
>> str2.gsub()      ##### add your solution here
=> "(qty+4)/2-35+pq/4"
```

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

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

>> pat =        ##### add your solution here

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

3) Replace any matching item from the given array with `X` for the given 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', 'x\y\z', 'qty||price', '{n}']

>> pat =        ##### add your solution here

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

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

```
>> ip = "123\b456"
>> puts ip
12456

>> ip.gsub()        ##### add your solution here
=> "123 456"
```

5) Replace all occurrences of `\o` with `o`.

```
>> ip = 'there are c\omm\on aspects am\ong the alternati\ons'

>> ip.gsub()        ##### add your solution here
=> "there are common aspects among the alternations"
```

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

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

>> pat =        ##### add your solution here

>> ip.gsub(pat, 'X')
=> "3-X*X-X+3"
```