learnbyexample

Ruby One-Liners Guide

✓ 200+ examples

✓ 50+ exercises

Sundeep Agarwal

Table of contents

Preface

As per ruby-lang.org, Ruby is based on programming languages like Perl, Smalltalk, Eiffel, Ada, and Lisp. This book focuses on using Ruby from the command line, similar to Perl one-liners usage.

You'll learn about various command line options and Ruby features that make it possible to write compact CLI scripts. Learning to use Ruby from the command line will also allow you to construct solutions where Ruby is just another tool in the shell ecosystem.

Prerequisites

You should be comfortable with programming basics and have prior experience working with Ruby. You should know concepts like blocks, be familiar with string/array/hash/enumerable methods, regular expressions etc. You can check out my free ebook Understanding Ruby Regexp if you wish to learn regular expressions in depth.

You should also be familiar with command line usage in a Unix-like environment. You should be comfortable with concepts like file redirection and command pipelines. Knowing the basics of the `grep` , `sed` and `awk` commands will come in handy as well.

Conventions

- The examples presented here have been tested with Ruby version **3.3.0** and includes features not available in earlier versions.
- Code snippets are copy pasted from the `GNU bash` shell and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines to improve readability, only `real` time shown for speed comparisons, output skipped/modified for certain commands and so on.
- External links are provided throughout the book for you to explore certain topics in more depth.
- The learn_ruby_oneliners repo has all the code snippets and files used in examples, exercises and other details related to the book. 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
- stackoverflow — for getting answers to pertinent questions on Ruby and related commands
- tex.stackexchange — for help on pandoc and `tex` related questions
- /r/ruby/ — helpful forum
- canva — cover image
- oxipng, pngquant and svgcleaner — optimizing images
- Warning and Info icons by Amada44 under public domain
- **KOTP** for spotting grammatical mistakes

A heartfelt thanks to all my readers. Your valuable support has significantly eased my financial concerns and allows me to continue working on programming ebooks.

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/learn_ruby_oneliners/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

2.0

See Version_changes.md to track changes across book versions.

One-liner introduction

This chapter will give an overview of Ruby syntax for command line usage. You'll see examples to understand what kind of problems are typically suited for one-liners.

Why use Ruby for one-liners?

I'll assume that you are already familiar with use cases where the command line is more productive compared to GUI. See also this series of articles titled Unix as IDE.

A shell utility like Bash provides built-in commands and scripting features to easily solve and automate various tasks. External commands like `grep` , `sed` , `awk` , `sort` , `find` , `parallel` , etc help to solve a wide variety of text processing tasks. These tools are often combined to work together along with shell features like pipelines, wildcards and loops. You can use Ruby as an alternative to such external tools and also complement them for some use cases.

Here are some sample text processing tasks that you can solve using Ruby one-liners. Options and related details will be explained later.

```
# retain only the first copy of duplicated lines
ruby -e 'puts readlines.uniq' *.txt

# retain only the first copy of duplicated lines,
# using the second field as the comparison criteria
ruby -e 'puts readlines.uniq {_1.split[1]}' *.txt

# extract only URLs
# uses a third-party library CommonRegexRuby
ruby -rcommonregex -ne 'puts CommonRegex.get_links($_)' *.md
```

Here are some questions that I've answered with simpler Ruby solution compared to other CLI tools:

- stackoverflow: merge duplicate key values while preserving order
- unix.stackexchange: pair each line of file

The selling point of Ruby over tools like `grep` , `sed` and `awk` includes feature rich regular expression engine and standard/third-party modules. Another advantage is that Ruby is more portable, given the many differences between GNU, BSD and other such implementations. The main disadvantage is that Ruby is likely to be verbose and slower for features that are supported out of the box by those tools.

Installation and Documentation

See ruby-lang.org for instructions on installing Ruby.

Visit ruby-doc.org for documentation.

Command line options

Use `ruby -h` to get a list of command line options, along with a brief description.

Option	Description
`-0[octal]`	specify record separator (`\0` , if no argument)
`-a`	autosplit mode with `-n` or `-p` (splits `$_` into `$F`)
`-c`	check syntax only
`-Cdirectory`	cd to directory before executing your script
`-d`	set debugging flags (set `$DEBUG` to true)
`-e 'command'`	one line of script. Several `-e` ’s allowed. Omit [programfile]
`-Eex[:in]`	specify the default external and internal character encodings
`-Fpattern`	`split()` pattern for autosplit (`-a`)
`-i[extension]`	edit `ARGV` files in place (make backup if extension supplied)
`-Idirectory`	specify `$LOAD_PATH` directory (may be used more than once)
`-l`	enable line ending processing
`-n`	assume `'while gets(); ... end'` loop around your script
`-p`	assume loop like `-n` but print line also like `sed`
`-rlibrary`	require the library before executing your script
`-s`	enable some switch parsing for switches after script name
`-S`	look for the script using PATH environment variable
`-v`	print the version number, then turn on verbose mode
`-w`	turn warnings on for your script
`-W[level=2|:category]`	set warning level; 0=silence, 1=medium, 2=verbose
`-x[directory]`	strip off text before #!ruby line and perhaps cd to directory
`--jit`	enable JIT for the platform, same as `--rjit` (experimental)
`--rjit`	enable pure-Ruby JIT compiler (experimental)
`-h`	show this message, `--help` for more info

This chapter will show examples with `-e` , `-n` , `-p` and `-a` options. Some more options will be covered in later chapters, but not all of them are discussed in this book.

Executing Ruby code

If you want to execute a Ruby program file, one way is to pass the filename as argument to the `ruby` command.

```
$ echo 'puts "Hello Ruby"' > hello.rb
$ ruby hello.rb
Hello Ruby
```

For short programs, you can also directly pass the code as an argument to the `-e` option.

```
$ ruby -e 'puts "Hello Ruby"'
Hello Ruby

# multiple statements can be issued separated by ;
$ ruby -e 'x=25; y=12; puts x**y'
59604644775390625
# or use -e option multiple times
$ ruby -e 'x=25' -e 'y=12' -e 'puts x**y'
59604644775390625
```

Filtering

Ruby one-liners can be used for filtering lines matched by a regular expression (regexp), similar to the `grep` , `sed` and `awk` commands. And similar to many command line utilities, Ruby can accept input from both stdin and file arguments.

```
# sample stdin data
$ printf 'gate\napple\nwhat\nkite\n'
gate
apple
what
kite

# print lines containing 'at'
# same as: grep 'at' and sed -n '/at/p' and awk '/at/'
$ printf 'gate\napple\nwhat\nkite\n' | ruby -ne 'print if /at/'
gate
what

# print lines NOT containing 'e'
# same as: grep -v 'e' and sed -n '/e/!p' and awk '!/e/'
$ printf 'gate\napple\nwhat\nkite\n' | ruby -ne 'print if !/e/'
what
```

By default, `grep` , `sed` and `awk` automatically loop over the input content line by line (with newline character as the default line separator). To do so with Ruby, you can use the `-n` and `-p` options. As seen before, the `-e` option accepts code as a command line argument. Many shortcuts are available to reduce the amount of typing needed.

In the above examples, a regular expression (defined by the pattern between a pair of forward slashes) has been used to filter the input. When the input string isn't specified in a conditional context (for example: `if`), the test is performed against the global variable `$_` , which has the contents of the current input line (the correct term would be input **record**, as discussed in the Record separators chapter). To summarize, in a conditional context:

- `/regexp/` is a shortcut for `$_ =~ /regexp/`
- `!/regexp/` is a shortcut for `$_ !~ /regexp/`

`$_` is also the default argument for the `print` method, which is why it is generally preferred in one-liners over the `puts` method. More such defaults that apply to the `print` method will be discussed later.

See ruby-doc: Pre-Defined Global Variables for documentation on `$_` , `$&` , etc.

Here's an example with file input instead of stdin.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14
```

```
# same as: grep -oE '[0-9]+$' table.txt
# digits at the end of lines
$ ruby -ne 'puts $& if /\d+$/' table.txt
42
7
14

# digits at the end of lines that are not preceded by -
$ ruby -ne 'puts $& if /(?<!-)\d+$/' table.txt
42
14
```

The example_files directory has all the files used in the examples (like `table.txt` in the above illustration).

Substitution

Use the `sub` and `gsub` methods for search and replace requirements. By default, these methods operate on `$_` when the input string isn't provided. For these examples, the `-p` option is used instead of `-n`, so that the value of `$_` is automatically printed after processing each input line.

```
# for each input line, change only the first ':' to '-'
# same as: sed 's/:/-/' and awk '{sub(/:/, "-")} 1'
$ printf '1:2:3:4\na:b:c:d\n' | ruby -pe 'sub(/:/, "-")'
1-2:3:4
a-b:c:d

# for each input line, change all ':' to '-'
# same as: sed 's/:/-/g' and awk '{gsub(/:/, "-")} 1'
$ printf '1:2:3:4\na:b:c:d\n' | ruby -pe 'gsub(/:/, "-")'
1-2-3-4
a-b-c-d
```

You might wonder how `$_` is modified without the use of `!` methods. The reason is that these methods are part of Kernel (see ruby-doc: Kernel for details) and are available only when the `-n` and `-p` options are used.

- `sub(/regexp/, repl)` is a shortcut for `$_.sub(/regexp/, repl)` and `$_` will be updated if the substitution succeeds
- `gsub(/regexp/, repl)` is a shortcut for `$_.gsub(/regexp/, repl)` and `$_` gets updated if the substitution succeeds

This book assumes that you are already familiar with regular expressions. If not, you can check out my free ebook Understanding Ruby Regexp.

Field processing

Consider the sample input file shown below with fields separated by a single space character.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14
```

Here are some examples that are based on specific fields rather than the entire line. The `-a` option will cause the input line to be split based on whitespaces and the array contents can be accessed using the `$F` global variable. Leading and trailing whitespaces will be suppressed, so there's no possibility of empty fields. More details will be discussed in the Default field separation section.

```
# print the second field of each input line
# same as: awk '{print $2}' table.txt
$ ruby -ane 'puts $F[1]' table.txt
bread
cake
banana

# print lines only if the last field is a negative number
# same as: awk '$NF<0' table.txt
$ ruby -ane 'print if $F[-1].to_f < 0' table.txt
blue cake mug shirt -7

# change 'b' to 'B' only for the first field
# same as: awk '{gsub(/b/, "B", $1)} 1' table.txt
$ ruby -ane '$F[0].gsub!(/b/, "B"); puts $F * " "' table.txt
Brown bread mat hair 42
Blue cake mug shirt -7
yellow banana window shoes 3.14
```

BEGIN and END

You can use a `BEGIN{}` block when you need to execute something before the input is read and an `END{}` block to execute something after all of the input has been processed.

```
# same as: awk 'BEGIN{print "---"} 1; END{print "%%%"}'
# note the use of ; after the BEGIN block
$ seq 4 | ruby -pe 'BEGIN{puts "---"}; END{puts "%%%"}'
---
1
2
3
4
%%%
```

ENV hash

When it comes to automation and scripting, you'd often need to construct commands that can accept input from users, use data from files and the output of a shell command and so on. As mentioned before, this book assumes `bash` as the shell being used. To access environment variables of the shell, you can use the special hash variable `ENV` with the name of the environment variable as a string key.

```
# existing environment variable
# output shown here is for my machine, would differ for you
$ ruby -e 'puts ENV["HOME"]'
/home/learnbyexample
$ ruby -e 'puts ENV["SHELL"]'
/bin/bash

# defined along with the command
# note that the variable definition is placed before the command
$ word='hello' ruby -e 'puts ENV["word"]'
hello
# the characters are preserved as is
$ ip='hi\nbye' ruby -e 'puts ENV["ip"]'
hi\nbye
```

Here's another example when a regexp is passed as an environment variable content.

```
$ cat word_anchors.txt
sub par
spar
apparent effort
two spare computers
cart part tart mart

# assume 'r' is a shell variable containing user provided regexp
$ r='\Bpar\B'
$ rgx="$r" ruby -ne 'print if /#{ENV["rgx"]}/' word_anchors.txt
apparent effort
two spare computers
```

You can also make use of the `-s` option to assign a global variable.

```
$ r='\Bpar\B'
$ ruby -sne 'print if /#{$rgx}/' -- -rgx="$r" word_anchors.txt
apparent effort
two spare computers
```

As an example, see my repo ch: command help for a practical shell script, where commands are constructed dynamically.

Executing external commands

You can call external commands using the `system` Kernel method. See ruby-doc: system for documentation.

```
$ ruby -e 'system("echo Hello World")'
Hello World

$ ruby -e 'system("wc -w <word_anchors.txt")'
12

$ ruby -e 'system("seq -s, 10 > out.txt")'
$ cat out.txt
1,2,3,4,5,6,7,8,9,10
```

Return value of `system` or the global variable `$?` can be used to act upon the exit status of the command issued.

```
$ ruby -e 'es=system("ls word_anchors.txt"); puts es'
word_anchors.txt
true

$ ruby -e 'system("ls word_anchors.txt"); puts $?'
word_anchors.txt
pid 6087 exit 0

$ ruby -e 'system("ls xyz.txt"); puts $?'
ls: cannot access 'xyz.txt': No such file or directory
pid 6164 exit 2
```

To save the result of an external command, use backticks or `%x` .

```
$ ruby -e 'words = `wc -w <word_anchors.txt`; puts words'
12

$ ruby -e 'nums = %x/seq 3/; print nums'
1
2
3
```

See also stackoverflow: difference between exec, system and %x() or backticks.

Summary

This chapter introduced some of the common options for Ruby CLI usage, along with some of the typical text processing examples. While specific purpose CLI tools like `grep` , `sed` and `awk` are usually faster, Ruby has a much more extensive standard library and ecosystem. And you do not have to learn a lot if you are already comfortable with Ruby but not familiar with those CLI tools. The next section has a few exercises for you to practice the CLI options and text processing use cases.

Exercises

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

The exercises directory has all the files used in this section.

1) For the input file `ip.txt`, display all lines containing `is`.

```
$ cat ip.txt
Hello World
How are you
This game is good
Today is sunny
12345
You are funny

##### add your solution here
This game is good
Today is sunny
```

2) For the input file `ip.txt`, display the first field of lines *not* containing `y`. Consider space as the field separator for this file.

```
##### add your solution here
Hello
This
12345
```

3) For the input file `ip.txt`, display all lines containing no more than 2 fields.

```
##### add your solution here
Hello World
12345
```

4) For the input file `ip.txt`, display all lines containing `is` in the second field.

```
##### add your solution here
Today is sunny
```

5) For each line of the input file `ip.txt`, replace the first occurrence of `o` with `0`.

```
##### add your solution here
Hell0 World
H0w are you
This game is g0od
T0day is sunny
12345
Y0u are funny
```

6) For the input file `table.txt`, calculate and display the product of numbers in the last field of each line. Consider space as the field separator for this file.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

##### add your solution here
-923.1600000000001
```

7) Append `.` to all the input lines for the given stdin data.

```
$ printf 'last\nappend\nstop\ntail\n' | ##### add your solution here
last.
append.
stop.
tail.
```

8) Use contents of the `s` variable to display matching lines from the input file `ip.txt`. Assume that `s` doesn't have any regexp metacharacters. Construct the solution such that there's at least one word character immediately preceding the contents of the `s` variable.

```
$ s='is'

##### add your solution here
This game is good
```

9) Use `system` to display the contents of the filename present in the second field of the given input line. Consider space as the field separator.

```
$ s='report.log ip.txt sorted.txt'
$ echo "$s" | ##### add your solution here
Hello World
How are you
This game is good
Today is sunny
12345
You are funny

$ s='power.txt table.txt'
$ echo "$s" | ##### add your solution here
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14
```

Line processing

Now that you are familiar with basic Ruby CLI usage, this chapter will dive deeper into line processing examples. You'll learn various ways for matching lines based on regular expressions, fixed string matching, line numbers, etc. You'll also see how to group multiple statements and learn about the control flow keywords `next` and `exit` .

The example_files directory has all the files used in the examples.

Regexp based filtering

As mentioned before, in a conditional context:

- `/regexp/` is a shortcut for `$_ =~ /regexp/`
- `!/regexp/` is a shortcut for `$_ !~ /regexp/`

Here are some examples:

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

$ ruby -ne 'print if /-|ow\b/' table.txt
blue cake mug shirt -7
yellow banana window shoes 3.14

$ ruby -ne 'print if !/[ksy]/' table.txt
brown bread mat hair 42
```

But, this is not applicable for all types of expressions. For example:

```
# /at$/ will be 'true' as it is treated as just a Regexp object here
$ printf 'gate\napple\nwhat\n' | ruby -ne '/at$/ && print'
gate
apple
what

# same as: ruby -ne 'print if /at$/'
$ printf 'gate\napple\nwhat\n' | ruby -ne '$_ =~ /at$/ && print'
what
```

If required, you can also use different delimiters with `%r` . See ruby-doc: %r Regexp Literals for details.

```
$ cat paths.txt
/home/joe/report.log
/home/ram/power.log
/home/rambo/errors.log
```

```
# leaning toothpick syndrome
$ ruby -ne 'print if /\/home\/ram\//' paths.txt
/home/ram/power.log

$ ruby -ne 'print if %r{/home/ram/}' paths.txt
/home/ram/power.log

$ ruby -ne 'print if !%r#/home/ram/#' paths.txt
/home/joe/report.log
/home/rambo/errors.log
```

Extracting matched portions

You can use regexp related global variables to extract only the matching portions. Consider this input file.

```
$ cat ip.txt
it is a warm and cozy day
listen to what I say
go play in the park
come back before the sky turns dark

There are so many delights to cherish
Apple, Banana and Cherry
Bread, Butter and Jelly
Try them all before you perish
```

Here are some examples of extracting only the matched portions.

```
# note that this will print only the first match for each input line
$ ruby -ne 'puts $& if /\b[a-z]\w*[ty]\b/' ip.txt
it
what
play
sky
many

# extract the capture group portions
$ ruby -ne 'puts "#{$1}::#{$2}" if /(\b[bdp]\w+).*(\b[a-f]\w+)/i' ip.txt
back::dark
delights::cherish
Banana::Cherry
Bread::and
```

See the Working with matched portions chapter from my ebook for examples that use the `match` method and regexp global variables.

match? method

As seen in the previous section, using `$_ =~ /regexp/` also sets global variables. If you just need a `true` or `false` result, using the `match?` method is better suited for performance reasons. The difference would be more visible for large input files.

```
# same result as: ruby -ne 'print if /[AB]|the\b/'
$ ruby -ne 'print if $_.match?(/[AB]|the\b/)' ip.txt
go play in the park
come back before the sky turns dark
Apple, Banana and Cherry
Bread, Butter and Jelly
```

Transliteration

The transliteration method `tr` helps you perform transformations character-wise. See ruby-doc: tr for documentation.

```
# rot13
$ echo 'Uryyb Jbeyq' | ruby -pe '$_.tr!("a-zA-Z", "n-za-mN-ZA-M")'
Hello World

# ^ at the start of the first argument complements the specified characters
$ echo 'apple:123:banana' | ruby -pe '$_.tr!("^0-9\n", "-")'
------123-------

# an empty second argument deletes the specified characters
$ echo 'apple:123:banana' | ruby -pe '$_.tr!("^0-9\n", "")'
123

# if the second list is shorter than the number of characters in the first list,
# the last character in the second list will be used to fill the gaps
$ s='orange apple appleseed cab'
$ echo "$s" | ruby -pe 'gsub(/\b(?!apple\b)\w++/) {$&.tr("a-z", "1-9")}'
991975 apple 199959554 312
```

You can use the `tr_s` method to squeeze repeated characters.

```
$ echo 'APPLESEED gobbledygook' | ruby -pe '$_.tr_s!("a-zA-Z", "a-zA-Z")'
APLESED gobledygok

# transliteration as well as squeeze
$ echo 'APPLESEED gobbledygook' | ruby -pe '$_.tr_s!("A-Z", "a-z")'
aplesed gobbledygook
```

Conditional substitution

These examples combine line filtering and substitution in different ways. As noted before, the `sub` and `gsub` Kernel methods update `$_` if the substitution succeeds and always return the value of `$_` .

```
# change commas to hyphens if the input line does NOT contain '2'
# prints all input lines even if the substitution fails
$ printf '1,2,3,4\na,b,c,d\n' | ruby -pe 'gsub(/,/, "-") if !/2/'
1,2,3,4
a-b-c-d

# perform substitution only for the filtered lines
# prints filtered input lines, even if the substitution fails
$ ruby -ne 'print gsub(/ark/, "[\\0]") if /the/' ip.txt
go play in the p[ark]
come back before the sky turns d[ark]
Try them all before you perish

# print only if the substitution succeeds
# $_.gsub! is required for this scenario
$ ruby -ne 'print if $_.gsub!(/\bw\w*t\b/, "{\\0}")' ip.txt
listen to {what} I say
```

Multiple conditions

It is good to remember that Ruby is a programming language. You can make use of control structures and combine multiple conditions using logical operators, methods like `all?` , `any?` , etc. You don't have to create a single complex regexp.

```
$ ruby -ne 'print if /ark/ && !/sky/' ip.txt
go play in the park

$ ruby -ane 'print if /\bthe\b/ || $F.size == 5' ip.txt
listen to what I say
go play in the park
come back before the sky turns dark
```

next

When the `next` statement is executed, rest of the code will be skipped and the next input line will be fetched for processing. It doesn't affect the `BEGIN` and `END` blocks as they are outside the file content loop.

```
$ ruby -ne '(puts "%% #{$_}"; next) if /\bpar/;
            puts /s/ ? "X" : "Y"' word_anchors.txt
%% sub par
X
Y
X
%% cart part tart mart
```

`()` is used in the above example to group multiple statements to be executed for a single `if` condition. You'll see more such examples in the coming chapters.

exit

The `exit` method will cause the Ruby script to terminate immediately. This is useful to avoid processing unnecessary input content after a termination condition is reached.

```
# quits after an input line containing 'say' is found
$ ruby -ne 'print; exit if /say/' ip.txt
it is a warm and cozy day
listen to what I say

# the matching line won't be printed in this case
$ ruby -pe 'exit if /say/' ip.txt
it is a warm and cozy day
```

Use `tac` to get all lines starting from the last occurrence of the search string in the entire file.

```
$ tac ip.txt | ruby -ne 'print; exit if /an/' | tac
Bread, Butter and Jelly
Try them all before you perish
```

You can optionally provide a status code as an argument to the `exit` method.

```
$ printf 'sea\neat\ndrop\n' | ruby -ne 'print; exit(2) if /at/'
sea
eat
$ echo $?
2
```

Any code in the `END` block will still be executed before exiting. This doesn't apply if `exit` was called from the `BEGIN` block.

```
$ ruby -pe 'exit if /cake/' table.txt
brown bread mat hair 42

$ ruby -pe 'exit if /cake/; END{puts "bye"}' table.txt
brown bread mat hair 42
bye

$ ruby -pe 'BEGIN{puts "hi"; exit; puts "hello"}; END{puts "bye"}' table.txt
hi
```

⚠ Be careful if you want to use `exit` with multiple input files, as Ruby will stop even if there are other files remaining to be processed.

Line number based processing

Line numbers can also be specified as a matching criteria by using the `$.` global variable.

```
# print only the third line
$ ruby -ne 'print if $. == 3' ip.txt
go play in the park
```

```
# print the second and sixth lines
$ ruby -ne 'print if $. == 2 || $. == 6' ip.txt
listen to what I say
There are so many delights to cherish

# transliterate only the second line
$ printf 'gates\nnot\nused\n' | ruby -pe '$_.tr!("a-z", "*") if $. == 2'
gates
***
used

# print from a particular line number to the end of the input
$ seq 14 25 | ruby -ne 'print if $. >= 10'
23
24
25
```

The global variable `$<` contains the file handle for the current file input being processed. Use the `eof` method to check for the end of the file condition. See ruby-doc: eof for documentation. You can also use `ARGF` instead of `$<` here, see the ARGV and ARGF section for details.

```
# same as: tail -n1 ip.txt
$ ruby -ne 'print if $<.eof' ip.txt
Try them all before you perish

$ ruby -ne 'puts "#{$.}:#{$_}" if $<.eof' ip.txt
9:Try them all before you perish

# multiple file example
# same as: tail -q -n1 ip.txt table.txt
$ ruby -ne 'print if $<.eof' ip.txt table.txt
Try them all before you perish
yellow banana window shoes 3.14
```

For large input files, use the `exit` method to avoid processing unnecessary input lines.

```
$ seq 3542 4623452 | ruby -ne '(print; exit) if $. == 2452'
5993

$ seq 3542 4623452 | ruby -ne 'print if $. == 250; (print; exit) if $. == 2452'
3791
5993

# here is a sample time comparison
$ time seq 3542 4623452 | ruby -ne '(print; exit) if $. == 2452' > f1
real    0m0.055s
$ time seq 3542 4623452 | ruby -ne 'print if $. == 2452' > f2
real    0m1.130s
$ rm f1 f2
```

Flip-Flop operator

You can use the Flip-Flop operator to select between a pair of matching conditions like line numbers and regexp. See ruby-doc: Flip-Flop for documentation.

```
# the range is automatically compared against $. in this context
$ seq 14 25 | ruby -ne 'print if 3..5'
16
17
18

# 'print if 3...5' gives the same result as above,
# you can use the include? method to exclude the end range
$ seq 14 25 | ruby -ne 'print if (3...5).include?($.)'
16
17

# the range is automatically compared against $_ in this context
# note that all the matching ranges are printed
$ ruby -ne 'print if /to/../pl/' ip.txt
listen to what I say
go play in the park
There are so many delights to cherish
Apple, Banana and Cherry
```

See the Records bounded by distinct markers section for an alternate solution.

Line numbers and regexp filtering can be mixed.

```
$ ruby -ne 'print if 6../utter/' ip.txt
There are so many delights to cherish
Apple, Banana and Cherry
Bread, Butter and Jelly

# same logic as: ruby -pe 'exit if /\bba/'
# inefficient, but this will work for multiple file inputs
$ ruby -ne 'print if !(/\bba/..$<.eof)' ip.txt table.txt
it is a warm and cozy day
listen to what I say
go play in the park
brown bread mat hair 42
blue cake mug shirt -7
```

Both conditions can match the same line too! Also, if the second condition doesn't match, lines starting from the first condition to the last line of the input will be matched.

```
# 'and' matches the 7th line
$ ruby -ne 'print if 7../and/' ip.txt
Apple, Banana and Cherry
# 'and' will be tested against 8th line onwards
$ ruby -ne 'print if 7.../and/' ip.txt
```

```
Apple, Banana and Cherry
Bread, Butter and Jelly

# there's a line containing 'Banana' but the matching pair isn't found
# so, all lines till the end of the input is printed
$ ruby -ne 'print if /Banana/../XYZ/' ip.txt
Apple, Banana and Cherry
Bread, Butter and Jelly
Try them all before you perish
```

Working with fixed strings

To match strings literally, use the `include?` method for line filtering. Use string argument instead of regexp for fixed string matching with substitution methods.

```
$ printf 'int a[5]\nfig\n1+4=5\n' | ruby -ne 'print if /a[5]/'
$ printf 'int a[5]\nfig\n1+4=5\n' | ruby -ne 'print if $_.include?("a[5]")'
int a[5]

$ printf 'int a[5]\nfig\n1+4=5\n' | ruby -pe 'sub(/a[5]/, "b")'
int a[5]
fig
1+4=5
$ printf 'int a[5]\nfig\n1+4=5\n' | ruby -pe 'sub("a[5]", "b")'
int b
fig
1+4=5
```

The above examples use double quotes for the string argument, which allows escape sequences like `\t` , `\n` , etc and interpolation with `#{}` . This isn't the case with single quoted string values. Using single quotes within the script from the command line requires messing with shell metacharacters. So, use `%q` instead or pass the fixed string to be matched as an environment variable.

```
# double quotes allow escape sequences and interpolation
$ ruby -e 'a=5; puts "value of a:\t#{a}"'
value of a:     5

# use %q as an alternate to specify single quoted strings
$ echo 'int #{a}' | ruby -ne 'print if $_.include?(%q/#{a}/)'
int #{a}
$ echo 'int #{a}' | ruby -pe 'sub(%q/#{a}/, "b")'
int b

# or pass the string as an environment variable
$ echo 'int #{a}' | s='#{a}' ruby -ne 'print if $_.include?(ENV["s"])'
int #{a}
# \\ is special within single quotes, so ENV is the better choice here
$ echo 'int #{a\\}' | s='#{a\\}' ruby -pe 'sub(ENV["s"], "b")'
int b
```

To provide a fixed string in the replacement section, environment variables comes in handy again. Need to use block form, since `\` is special in the replacement section.

```
# \\ will be treated as \ and \0 will backreference the entire matched portion
$ echo 'int a' | s='x\\y\0z' ruby -pe 'sub(/a/, ENV["s"])'
int x\yaz

# use block form to avoid such issues
$ echo 'int a' | s='x\\y\0z' ruby -pe 'sub(/a/) {ENV["s"]}'
int x\\y\0z
```

Use the `start_with?` and `end_with?` methods to restrict the matching to the start or end of the input line. The line content in the `$_` variable contains the `\n` line ending character as well. You can either use the `chomp` method explicitly or use the `-l` command line option (which will be discussed in detail in the Record separators chapter). For now, it is enough to know that `-l` will remove the line separator and add it back when `print` is used.

```
$ cat eqns.txt
a=b,a-b=c,c*d
a+b,pi=3.14,5e12
i*(t+9-g)/8,4-a+b

# start of the line
$ s='a+b' ruby -ne 'print if $_.start_with?(ENV["s"])' eqns.txt
a+b,pi=3.14,5e12

# end of the line
# -l option is needed here to remove \n from $_
$ s='a+b' ruby -lne 'print if $_.end_with?(ENV["s"])' eqns.txt
i*(t+9-g)/8,4-a+b
```

Use the `index` method if you need more control over the location of the matching strings. You can use either the return value (which gives you the index of the matching string) or use the optional second argument to specify an offset to start searching. See ruby-doc: index for details.

```
# same as: $_.include?("a+b")
$ ruby -ne 'print if $_.index("a+b")' eqns.txt
a+b,pi=3.14,5e12
i*(t+9-g)/8,4-a+b

# same as: $_.start_with?("a+b")
$ ruby -ne 'print if $_.index("a+b")==0' eqns.txt
a+b,pi=3.14,5e12

# since 'index' returns 'nil' if there's no match,
# you need some more processing for < or <= comparison
$ ruby -ne '$i = $_.index("="); print if $i && $i < 6' eqns.txt
a=b,a-b=c,c*d

# for > or >= comparison, use the optional second argument
```

```
$ s='a+b' ruby -ne 'print if $_.index(ENV["s"], 1)' eqns.txt
i*(t+9-g)/8,4-a+b
```

If you need to match the entire input line or a particular field, you can use the comparison operators.

```
$ printf 'a.b\na+b\n' | ruby -lne 'print if /^a.b$/'
a.b
a+b
$ printf 'a.b\na+b\n' | ruby -lne 'print if $_ == %q/a.b/'
a.b

$ printf '1 a.b\n2 a+b\n' | ruby -lane 'print if $F[1] != %q/a.b/'
2 a+b
```

In-place file editing

You can use the `-i` option to write back the changes to the input file instead of displaying the output on terminal. When an extension is provided as an argument to `-i`, the original contents of the input file gets preserved as per the extension given. For example, if the input file is `ip.txt` and `-i.orig` is used, the backup file will be named as `ip.txt.orig`.

```
$ cat colors.txt
deep blue
light orange
blue delight

# no output on the terminal as -i option is used
# space is NOT allowed between -i and the extension
$ ruby -i.bkp -pe 'sub(/blue/, "-green-")' colors.txt
# changes are written back to 'colors.txt'
$ cat colors.txt
deep -green-
light orange
-green- delight

# original file is preserved in 'colors.txt.bkp'
$ cat colors.txt.bkp
deep blue
light orange
blue delight
```

Multiple input files are treated individually and the changes are written back to respective files.

```
$ cat t1.txt
have a nice day
bad morning
what a pleasant evening
$ cat t2.txt
worse than ever
```

```
too bad

$ ruby -i.bkp -pe 'sub(/bad/, "good")' t1.txt t2.txt
$ ls t?.*
t1.txt  t1.txt.bkp  t2.txt  t2.txt.bkp

$ cat t1.txt
have a nice day
good morning
what a pleasant evening
$ cat t2.txt
worse than ever
too good
```

Sometimes backups are not desirable. In such cases, you can use the `-i` option without an argument. Be careful though, as changes made cannot be undone. It is recommended to test the command with sample inputs before applying the `-i` option on the actual file. You could also use the option with backup, compare the differences with a `diff` program and then delete the backup.

```
$ cat fruits.txt
banana
papaya
mango

$ ruby -i -pe 'gsub(/(..)\1/) {$&.upcase}' fruits.txt
$ cat fruits.txt
bANANa
PAPAya
mango
```

Summary

This chapter showed various examples of processing only the lines of interest instead of the entire input file. Filtering can be specified using a regexp, fixed string, line number or a combination of them. You also saw how to combine multiple statements inside `()` for compact CLI usage. The `next` and `exit` methods are useful to control the flow of code. The `-i` option is handy for in-place editing.

Exercises

The exercises directory has all the files used in this section.

1) For the given input, display except the third line.

```
$ seq 34 37 | ##### add your solution here
34
35
37
```

2) Display only the fourth, fifth, sixth and seventh lines for the given input.

```
$ seq 65 78 | ##### add your solution here
68
69
70
71
```

3) For the input file `ip.txt`, replace all occurrences of `are` with `are not` and `is` with `is not` only from line number **4** till the end of file. Also, only the lines that were changed should be displayed in the output.

```
$ cat ip.txt
Hello World
How are you
This game is good
Today is sunny
12345
You are funny

##### add your solution here
Today is not sunny
You are not funny
```

4) For the given stdin, display only the first three lines. Avoid processing lines that are not relevant.

```
$ seq 14 25 | ##### add your solution here
14
15
16
```

5) For the input file `ip.txt`, display all lines from the start of the file till the first occurrence of `game`.

```
##### add your solution here
Hello World
How are you
This game is good
```

6) For the input file `ip.txt`, display all lines that contain `is` but not `good`.

```
##### add your solution here
Today is sunny
```

7) For the input file `ip.txt`, extract the word before the whole word `is` as well as the word after it. If such a match is found, display the two words around `is` in reversed order. For example, `hi;1 is--234 bye` should be converted to `234:1`. Assume that the whole word `is` will not be present more than once in a single line.

```
##### add your solution here
good:game
sunny:Today
```

8) For the input file `hex.txt`, replace all occurrences of `0xA0` with `0x50` and `0xFF` with `0x7F`.

```
$ cat hex.txt
start: 0xA0, func1: 0xA0
end: 0xFF, func2: 0xB0
restart: 0xA010, func3: 0x7F

##### add your solution here
start: 0x50, func1: 0x50
end: 0x7F, func2: 0xB0
restart: 0x5010, func3: 0x7F
```

9) For the input file `text.txt`, replace all occurrences of `in` with `an` and write back the changes to `text.txt` itself. The original contents should get saved to `text.txt.orig`.

```
$ cat text.txt
can ran want plant
tin fin fit mine line

##### add your solution here

$ cat text.txt
can ran want plant
tan fan fit mane lane
$ cat text.txt.orig
can ran want plant
tin fin fit mine line
```

10) For the input file `text.txt`, replace all occurrences of `an` with `in` and write back the changes to `text.txt` itself. Do not create backups for this exercise. Note that you should have solved the previous exercise before starting this one.

```
$ cat text.txt
can ran want plant
tan fan fit mane lane

##### add your solution here

$ cat text.txt
cin rin wint plint
tin fin fit mine line
$ diff text.txt text.txt.orig
1c1
< cin rin wint plint
---
> can ran want plant
```

11) Find the starting index of first occurrence of `is` or `the` or `was` or `to` for each input line of the file `idx.txt`. Assume that every input line will match at least one of these terms.

```
$ cat idx.txt
match after the last newline character
and then you want to test
this is good bye then
you were there to see?

##### add your solution here
12
4
2
9
```

12) Display all lines containing `[4]*` for the given stdin data.

```
$ printf '2.3/[4]*6\n2[4]5\n5.3-[4]*9\n' | ##### add your solution here
2.3/[4]*6
5.3-[4]*9
```

13) For the given input string, change all lowercase alphabets to `x` only for words starting with `m` .

```
$ s='ma2T3a a2p kite e2e3m meet'

$ echo "$s" | ##### add your solution here
xx2T3x a2p kite e2e3m xxxx
```

14) For the input file `ip.txt` , delete all characters other than lowercase vowels and the newline character. Perform this transformation only between a line containing `you` up to line number `4` (inclusive).

```
##### add your solution here
Hello World
oaeou
iaeioo
oaiu
12345
You are funny
```

15) For the input file `sample.txt` , display from the start of the file till the first occurrence of `are` , excluding the matching line.

```
$ cat sample.txt
Hello World

Good day
How are you

Just do-it
Believe it

Today is sunny
Not a bit funny
```

```
No doubt you like it too

Much ado about nothing
He he he

##### add your solution here
Hello World

Good day
```

16) For the input file `sample.txt`, display from the last occurrence of `do` till the end of the file.

```
##### add your solution here
Much ado about nothing
He he he
```

17) For the input file `sample.txt`, display from the 9th line till a line containing `you`.

```
##### add your solution here
Today is sunny
Not a bit funny
No doubt you like it too
```

18) Display only the odd numbered lines from `ip.txt`.

```
##### add your solution here
Hello World
This game is good
12345
```

19) For the `table.txt` file, print only the line number for lines containing `air` or `win`.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

##### add your solution here
1
3
```

20) For the input file `table.txt`, calculate the sum of numbers in the last column, excluding the second line.

```
##### add your solution here
45.14
```

21) Print the second and fourth line for every block of five lines.

```
$ seq 15 | ##### add your solution here
2
4
7
9
```

```
12
14
```

22) For the input file `ip.txt`, display all lines containing `e` or `u` but not both.

```
##### add your solution here
Hello World
This game is good
Today is sunny
```

Field separators

This chapter will dive deep into field processing. You'll learn how to set input and output field separators, how to use regexps for defining fields and how to work with fixed length fields.

The example_files directory has all the files used in the examples.

Default field separation

The `-a` option splits the input based on one or more sequence of **whitespace** characters. In addition, whitespaces at the start or end of input gets trimmed and won't be part of the field contents. Using `-a` is equivalent to `$F = $_.split` . From ruby-doc: split:

> If `$;` is `nil` (its default value), the split occurs just as if `field_sep` were given as a space character. When `field_sep` is `' '` and `limit` is `nil` , the split occurs at each sequence of whitespace.

```
$ echo '   a   b   c   ' | ruby -ane 'puts $F.size'
3
# note that the leading whitespaces aren't part of the field content
$ echo '   a   b   c   ' | ruby -ane 'puts "(#{$F[0]})"'
(a)
# trailing whitespaces are removed as well
$ echo '   a   b   c   ' | ruby -ane 'puts "(#{$F[-1]})"'
(c)

# here's another example with more whitespace characters thrown in
$ printf '     one \t\f\v two\t\r\tthree \t\r ' | ruby -ane 'puts $F.size'
3
$ printf '     one \t\f\v two\t\r\tthree \t\r ' | ruby -ane 'puts $F[1] + "."'
two.
```

Input field separator

You can use the `-F` command line option to specify a custom field separator. The value passed to this option will be treated as a regexp. Note that the `-a` option is also necessary for `-F` to work.

```
# use ':' as the input field separator
$ echo 'goal:amazing:whistle:kwality' | ruby -F: -ane 'puts $F[0], $F[-1], $F[1]'
goal
kwality
amazing

# use quotes to avoid clashes with shell special characters
$ echo 'one;two;three;four' | ruby -F';' -ane 'puts $F[2]'
three
```

```
$ echo 'load;err_msg--\ant,r2..not' | ruby -F'\W+' -ane 'puts $F[2]'
ant

$ echo 'hi.bye.hello' | ruby -F'\.' -ane 'puts $F[1]'
bye

# count the number of vowels for each input line
$ printf 'COOL\nnice car\n' | ruby -F'(?i)[aeiou]' -ane 'puts $F.size - 1'
2
3
```

Character-wise separation

No need to use field separation to access individual characters. See ruby-doc: Encoding for details on handling different string encodings.

```
$ echo 'apple' | ruby -ne 'puts $_[0]'
a

$ ruby -e 'puts Encoding.default_external'
UTF-8
$ LC_ALL=C ruby -e 'puts Encoding.default_external'
US-ASCII

$ echo 'fox:αλεπού' | ruby -ne 'puts $_[4..5]'
αλ
# use the -E option to explicitly specify external/internal encodings
$ echo 'fox:αλεπού' | ruby -E UTF-8:UTF-8 -ne 'puts $_[4..5]'
αλ
```

Newline character in the last field

If the custom field separator doesn't affect the newline character, then the last element can contain the newline character.

```
# last element will not have the newline character with the -a option
# as leading/trailing whitespaces are trimmed with default split
$ echo 'cat dog' | ruby -ane 'puts "[#{$F[-1]}]"'
[dog]

# last element will have the newline character since the field separator is ':'
$ echo 'cat:dog' | ruby -F: -ane 'puts "[#{$F[-1]}]"'
[dog
]
# unless the input itself doesn't have newline characters
$ printf 'cat:dog' | ruby -F: -ane 'puts "[#{$F[-1]}]"'
[dog]
```

The newline character can also show up as the entire content of the last field.

```
# both the leading and trailing whitespaces are trimmed
$ echo '   a b   c   ' | ruby -ane 'puts $F.size'
3

# leading empty element won't be removed here
# and the last element will have only the newline character as the value
$ echo ':a:b:c:' | ruby -F: -ane 'puts $F.size; puts "[#{$F[-1]}]"'
5
[
]
```

Using the -l option for field splitting

As mentioned before, the `-l` option is helpful if you wish to remove the newline character (more details will be discussed in the Record separators chapter). A side effect of removing the newline character before applying `split` is that the trailing empty fields will also get removed (you can explicitly call the `split` method with `-1` as limit to prevent this).

```
# -l will remove the newline character
$ echo 'cat:dog' | ruby -F: -lane 'puts "[#{$F[-1]}]"'
[dog]
# -l will also cause 'print' to append the newline character
$ echo 'cat:dog' | ruby -F: -lane 'print "[#{$F[-1]}]"'
[dog]

# since the newline character is chomped, last element is empty
# which is then removed due to the default 'split' behavior
$ echo ':a:b:c:' | ruby -F: -lane 'puts $F.size'
4
# explicit call to split with -1 as the limit will preserve the empty element
$ echo ':a:b:c:' | ruby -lane 'puts $_.split(/:/, -1).size'
5
```

Output field separator

There are a few ways to affect the separator to be used while displaying multiple values. The value of the `$,` global variable is used as the separator when multiple arguments are passed to the `print` method. This is usually used in combination with the `-l` option so that a newline character is appended automatically as well. The `join` method also uses `$,` as the default value.

```
$ ruby -lane 'BEGIN{$, = " "}; print $F[0], $F[2]' table.txt
brown mat
blue mug
yellow window
```

The other options include manually building the output string within double quotes. Or, use the `join` method. Note that the `-l` option is used in the examples below as a good practice even when not needed.

```
$ ruby -lane 'puts "#{$F[0]} #{$F[2]}"' table.txt
brown mat
blue mug
yellow window

$ echo 'Sample123string42with777numbers' | ruby -F'\d+' -lane 'puts $F.join(",")'
Sample,string,with,numbers

$ s='goal:amazing:whistle:kwality'
$ echo "$s" | ruby -F: -lane 'puts $F.values_at(-1, 1, 0).join("-")'
kwality-amazing-goal
# you can also use the '*' operator
$ echo "$s" | ruby -F: -lane '$F.append(42); puts $F * "::"'
goal::amazing::whistle::kwality::42
```

scan method

The `-F` option uses the `split` method to generate the fields. In contrast, the `scan` method allows you to define what should the fields be made up of. The `scan` method does not have the concept of removing empty trailing fields nor does it have the `limit` argument.

```
$ s='Sample123string42with777numbers'
# define fields to be one or more consecutive digits
$ echo "$s" | ruby -ne 'puts $_.scan(/\d+/)[1]'
42

$ s='coat Bin food tar12 best Apple fig_42'
# whole words made up of lowercase alphabets and digits only
$ echo "$s" | ruby -ne 'puts $_.scan(/\b[a-z0-9]+\b/) * ","'
coat,food,tar12,best

$ s='items: "apple" and "mango"'
# get the second double quoted item
$ echo "$s" | ruby -ne 'puts $_.scan(/"[^"]+"/)[1]'
"mango"
# no need to use 'scan' to extract the first matching portion
$ echo "$s" | ruby -ne 'puts $_[/"[^"]+"/]'
"apple"
```

A simple `split` fails for CSV input where fields can contain embedded delimiter characters. For example, a field content `"fox,42"` when `,` is the delimiter.

```
$ s='eagle,"fox,42",bee,frog'

# simply using , as the separator isn't sufficient
$ echo "$s" | ruby -F, -lane 'puts $F[1]'
"fox
```

While the ruby-doc: CSV library should be preferred for robust CSV parsing, regexp is enough for simple formats.

```
$ echo "$s" | ruby -lne 'puts $_.scan(/"[^"]*"|[^,]+/)[1]'
"fox,42"
```

Fixed width processing

The `unpack` method is more than just a different way of using string slicing. It supports various formats and pre-processing, see ruby-doc: Packed Data for details.

In the example below, `a` indicates arbitrary binary string. The optional number that follows indicates length of the field.

```
$ cat items.txt
apple   fig banana
50      10  200

# here field widths have been assigned such that
# extra spaces are placed at the end of each field
$ ruby -ne 'puts $_.unpack("a8a4a6") * ","' items.txt
apple   ,fig ,banana
50      ,10  ,200

$ ruby -ne 'puts $_.unpack("a8a4a6")[1]' items.txt
fig
10
```

You can specify characters to be ignored with `x` followed by an optional length.

```
# first field is 5 characters
# then 3 characters are ignored and 3 characters for the second field
# then 1 character is ignored and 6 characters for the third field
$ ruby -ne 'puts $_.unpack("a5x3a3xa6") * ","' items.txt
apple,fig,banana
50   ,10 ,200
```

Using `*` will cause remaining characters of that particular format to be consumed. Here `Z` is used to process strings that are separated by the ASCII NUL character.

```
$ printf 'banana\x0050\x00' | ruby -ne 'puts $_.unpack("Z*Z*") * ":"'
banana:50

# first field is 5 characters, then 3 characters are ignored
# all the remaining characters are assigned to the second field
$ ruby -ne 'puts $_.unpack("a5x3a*") * ","' items.txt
apple,fig banana
50   ,10  200
```

Unpacking isn't always needed, simple string slicing might suffice.

```
$ echo 'b 123 good' | ruby -ne 'puts $_[2,3]'
123
$ echo 'b 123 good' | ruby -ne 'puts $_[6,4]'
good
```

```
# replacing arbitrary slice
$ echo 'b 123 good' | ruby -lpe '$_[2,3] = "gleam"'
b gleam good
```

Assorted field processing methods

Having seen command line options and features commonly used for field processing, this section will highlight some of the built-in array and Enumerable methods. There are just too many to meaningfully cover them all in detail, so consider this to be just a brief overview of features.

First up, regexp based field selection. `grep(cond)` and `grep_v(cond)` are specialized filter methods that perform `cond === object` test check. See [stackoverflow: What does the === operator do in Ruby?](https://stackoverflow.com/q/3422223/4082052) for more details.

```
$ s='goal:amazing:42:whistle:kwality:3.14'

# fields containing 'in' or 'it' or 'is'
$ echo "$s" | ruby -F: -lane 'puts $F.grep(/i[nts]/) * ":"'
amazing:whistle:kwality

# fields NOT containing a digit character
$ echo "$s" | ruby -F: -lane 'puts $F.grep_v(/\d/) * ":"'
goal:amazing:whistle:kwality

# no more than one field can contain 'r'
$ ruby -lane 'print if $F.grep(/r/).size <= 1' table.txt
blue cake mug shirt -7
yellow banana window shoes 3.14
```

The `map` method transforms each element according to the logic passed to it.

```
$ s='goal:amazing:42:whistle:kwality:3.14'
$ echo "$s" | ruby -F: -lane 'puts $F.map(&:upcase) * ":"'
GOAL:AMAZING:42:WHISTLE:KWALITY:3.14

$ echo '23 756 -983 5' | ruby -ane 'puts $F.map {_1.to_i ** 2} * " "'
529 571536 966289 25

$ echo 'AaBbCc' | ruby -lne 'puts $_.chars.map(&:ord) * " "'
65 97 66 98 67 99

$ echo '3.14,17,6' | ruby -F, -ane 'puts $F.map(&:to_f).sum'
26.14
```

The `filter` method (which has other aliases and opposites too) is handy to construct all kinds of selection conditions. You can combine with `map` by using the `filter_map` method.

```
$ s='hour hand band mat heated pineapple'

$ echo "$s" | ruby -ane 'puts $F.filter {_1[0]!="h" && _1.size<6}'
band
mat
```

```
$ echo "$s" | ruby -ane 'puts $F.filter_map {|w|
                         w.gsub(/[ae]/, "X") if w[0]=="h"}'
hour
hXnd
hXXtXd
```

The `reduce` method can be used to perform an action against all the elements of an array and get a singular value as the result.

```
# sum of input numbers, with initial value of 100
$ echo '3.14,17,6' | ruby -F, -lane 'puts $F.map(&:to_f).reduce(100, :+)'
126.14

# product of input numbers
$ echo '3.14,17,6' | ruby -F, -lane 'puts $F.map(&:to_f).reduce(:*)'
320.28000000000003
# with initial value of 2
$ echo '3.14,17,6' | ruby -F, -lane 'puts $F.reduce(2) {|op,n| op*n.to_f}'
640.5600000000001
```

Here are some examples with the `sort`, `sort_by` and `uniq` methods for arrays and strings.

```
$ s='floor bat to dubious four'
$ echo "$s" | ruby -ane 'puts $F.sort * ":"'
bat:dubious:floor:four:to
$ echo "$s" | ruby -ane 'puts $F.sort_by(&:size) * ":"'
to:bat:four:floor:dubious

# numeric sort example
$ echo '23 756 -983 5' | ruby -lane 'puts $F.sort_by(&:to_i) * ":"'
-983:5:23:756

$ echo 'dragon' | ruby -lne 'puts $_.chars.sort.reverse * ""'
rongda

$ s='try a bad to good i teal by nice how'
# longer words first, ascending alphabetic order as tie-breaker
$ echo "$s" | ruby -ane 'puts $F.sort_by {|w| [-w.size, w]} * ":"'
good:nice:teal:bad:how:try:by:to:a:i

$ s='3,b,a,3,c,d,1,d,c,2,2,2,3,1,b'
# note that the input order of elements is preserved
$ echo "$s" | ruby -F, -lane 'puts $F.uniq * ","'
3,b,a,c,d,1,2
```

Here's an example for sorting in descending order based on header column names.

```
$ cat marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59
ECE     Om      92
CSE     Amy     67

$ ruby -ane 'idx = $F.each_index.sort {$F[_2] <=> $F[_1]} if $.==1;
             puts $F.values_at(*idx) * "\t"' marks.txt
Name    Marks   Dept
Raj     53      ECE
Joel    72      ECE
Moi     68      EEE
Surya   81      CSE
Tia     59      EEE
Om      92      ECE
Amy     67      CSE
```

The `shuffle` method randomizes the order of elements.

```
$ s='floor bat to dubious four'
$ echo "$s" | ruby -ane 'puts $F.shuffle * ":"'
bat:floor:dubious:to:four

$ echo 'foobar' | ruby -lne 'print $_.chars.shuffle * ""'
bofrao
```

Use the `sample` method to get one or more elements of an array in random order.

```
$ s='hour hand band mat heated pineapple'

$ echo "$s" | ruby -ane 'puts $F.sample'
band
$ echo "$s" | ruby -ane 'puts $F.sample(2)'
pineapple
hand
```

Summary

This chapter discussed various ways in which you can split (or define) the input into fields and manipulate them. Many more examples will be discussed in later chapters.

Exercises

The exercises directory has all the files used in this section.

1) For the input file `brackets.txt` , extract only the contents between `()` or `)(` from each input line. Assume that `()` characters will be present only once every line.

```
$ cat brackets.txt
foo blah blah(ice) 123 xyz$
(almond-pista) choco
yo )yoyo( yo

##### add your solution here
ice
almond-pista
yoyo
```

2) For the input file `scores.csv`, extract `Name` and `Physics` fields in the format shown below.

```
$ cat scores.csv
Name,Maths,Physics,Chemistry
Blue,67,46,99
Lin,78,83,80
Er,56,79,92
Cy,97,98,95
Ort,68,72,66
Ith,100,100,100

##### add your solution here
Name:Physics
Blue:46
Lin:83
Er:79
Cy:98
Ort:72
Ith:100
```

3) For the input file `scores.csv`, display names of those who've scored above `70` in Maths.

```
##### add your solution here
Lin
Cy
Ith
```

4) Display the number of word characters for the given inputs. Word definition here is same as used in regular expressions. Can you construct a solution with `gsub` and one without the substitution functions?

```
# solve using gsub
$ echo 'hi there' | ##### add your solution here
7

# solve without using the substitution functions
$ echo 'u-no;co%."(do_12:as' | ##### add your solution here
12
```

5) For the input file `quoted.txt`, extract the sequence of characters surrounded by double quotes and display them in the format shown below.

```
$ cat quoted.txt
1 "grape" and "mango" and "guava"
("c 1""d""a-2""b")

##### add your solution here
"grape","guava","mango"
"a-2","b","c 1","d"
```

6) Display only the third and fifth characters from each input line.

```
$ printf 'restore\ncat one\ncricket' | ##### add your solution here
so
to
ik
```

7) Transform the given input file `fw.txt` to get the output as shown below. If the second field is empty (i.e. contains only space characters), replace it with `NA`.

```
$ cat fw.txt
1.3  rs   90  0.134563
3.8           6
5.2  ye       8.2387
4.2  kt   32  45.1

##### add your solution here
1.3,rs,0.134563
3.8,NA,6
5.2,ye,8.2387
4.2,kt,45.1
```

8) For the input file `scores.csv`, display the header as well as any row which contains `b` or `t` (irrespective of case) in the first field.

```
##### add your solution here
Name,Maths,Physics,Chemistry
Blue,67,46,99
Ort,68,72,66
Ith,100,100,100
```

9) Extract all whole words containing `42` but not at the edge of a word. Assume a word cannot contain `42` more than once.

```
$ s='hi42bye nice1423 bad42 cool_42a 42fake'
$ echo "$s" | ##### add your solution here
hi42bye
nice1423
cool_42a
```

10) For the input file `scores.csv`, add another column named **GP** which is calculated out of 100 by giving 50% weightage to Maths and 25% each for Physics and Chemistry.

```
##### add your solution here
Name,Maths,Physics,Chemistry,GP
```

```
Blue,67,46,99,69.75
Lin,78,83,80,79.75
Er,56,79,92,70.75
Cy,97,98,95,96.75
Ort,68,72,66,68.5
Ith,100,100,100,100.0
```

11) For the input file `mixed_fs.txt`, retain only the first two fields from each input line. The input and output field separators should be space for first two lines and `,` for the rest of the lines.

```
$ cat mixed_fs.txt
rose lily jasmine tulip
pink blue white yellow
car,mat,ball,basket
light green,brown,black,purple
apple,banana,cherry

##### add your solution here
rose lily
pink blue
car,mat
light green,brown
apple,banana
```

12) For the given space separated numbers, filter only numbers in the range `20` to `1000` (inclusive).

```
$ s='20 -983 5 756 634223 1000'

$ echo "$s" | ##### add your solution here
20 756 1000
```

13) For the given space separated words, randomize the order of characters for each word.

```
$ s='this is a sample sentence'

# sample randomized output shown here, could be different for you
$ echo "$s" | ##### add your solution here
shti si a salemp sneentce
```

14) For the given input file `words.txt`, filter all lines containing characters in ascending and descending order.

```
$ cat words.txt
bot
art
are
boat
toe
flee
reed
```

```
# ascending order
##### add your solution here
bot
art

# descending order
##### add your solution here
toe
reed
```

15) For the given space separated words, extract the three longest words.

```
$ s='I bought two bananas and three mangoes'

$ echo "$s" | ##### add your solution here
mangoes
bananas
bought
```

16) Convert the contents of `split.txt` as shown below.

```
$ cat split.txt
apple,1:2:5,mango
wry,4,look
pencil,3:8,paper

##### add your solution here
apple,1,mango
apple,2,mango
apple,5,mango
wry,4,look
pencil,3,paper
pencil,8,paper
```

17) For the input file `varying_fields.txt`, construct a solution to get the output shown below.

```
$ cat varying_fields.txt
hi,bye,there,was,here,to
1,2,3,4,5

##### add your solution here
hi:bye:to
1:2:5
```

18) The `fields.txt` file has fields separated by the `:` character. Delete `:` and the last field if there is a digit character anywhere before the last field. Solution shouldn't use the substitution functions.

```
$ cat fields.txt
42:cat
twelve:a2b
```

```
we:be:he:0:a:b:bother
apple:banana-42:cherry:
dragon:unicorn:centaur

##### add your solution here
42
twelve:a2b
we:be:he:0:a:b
apple:banana-42:cherry
dragon:unicorn:centaur
```

19) The sample string shown below uses `cat` as the field separator (irrespective of case). Use space as the output field separator and add `42` as the last field.

```
$ s='applecatfigCaT12345cAtbanana'

$ echo "$s" | ##### add your solution here
apple fig 12345 banana 42
```

20) For the input file `sample.txt`, filter lines containing 5 or more lowercase vowels.

```
##### add your solution here
How are you
Believe it
No doubt you like it too
Much ado about nothing
```