learnbyexample

Ruby one-liners cookbook

```
ruby -ne 'puts $& if /\d+$/'

'puts readlines.uniq {_1.split[2]}'

'ip=Nokogiri.XML(ARGF);
 puts ip.xpath("//greeting/@type")'
```

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 book on Ruby Regexp if you wish to learn regular expressions in depth.

You should also have prior experience working with command line and `bash` shell and be familiar with concepts like file redirection, command pipeline and so on.

Conventions

- The examples presented here have been tested with **Ruby version 2.7.1** and includes features not available in earlier versions.
- Code snippets shown are copy pasted from **bash** 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, only `real` time is shown for speed comparisons and so on.
- External links are provided for further reading throughout the book. Not necessary to immediately visit them. They have been chosen with care and would help, especially during re-reads.
- The learn_ruby_oneliners repo has all the code snippets and files used in examples and exercises and other details related to the book. If you are not familiar with `git` command, click the **Code** button on the webpage to get the files.

Acknowledgements

- ruby-lang documentation — manuals and tutorials
- /r/ruby/ — helpful forum for beginners and experienced programmers alike
- stackoverflow — for getting answers to pertinent questions on Ruby, one-liners, etc
- tex.stackexchange — for help on `pandoc` and `tex` related questions
- LibreOffice Draw — cover image
- Warning and Info icons by Amada44 under public domain
- softwareengineering.stackexchange and skolakoda for programming quotes

A heartfelt thanks to all my readers. Your valuable support has significantly eased my financial concerns and allows me to continue writing books.

Feedback and Errata

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

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

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

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

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

License

Book version

1.0

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 and some examples to show what kind of problems are typically suited for one-liners.

Why use Ruby for one-liners?

I assume you are already familiar with use cases where 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 make it easier to solve and automate various tasks. External *nix commands like `grep`, `sed`, `awk`, `sort`, `find`, `parallel` etc can be combined to work with each other. Depending upon your familiarity with those tools, you can either use `ruby` as a single replacement or complement them for specific use cases.

Here's some one-liners (options will be explained later):

- `ruby -e 'puts readlines.uniq' *.txt` — retain only one copy if lines are duplicated from the given list of input file(s)
- `ruby -e 'puts readlines.uniq {|s| s.split[1]}' *.txt` — retain only first copy of duplicate lines using second field as duplicate criteria
- `ruby -rcommonregex -ne 'puts CommonRegex.get_links($_)' *.md` — extract only the URLs, using a third-party CommonRegexRuby library
- stackoverflow: merge duplicate key values while preserving order — a recent Q&A that I answered with a simpler `ruby` solution compared to `awk`

The main advantage of `ruby` over tools like `grep`, `sed` and `awk` includes feature rich regular expression engine, standard library and third-party libraries. If you don't already know the syntax and idioms for `sed` and `awk`, learning command line options for `ruby` would be the easier option. The main disadvantage is that `ruby` is likely to be slower compared to those tools.

Command line options

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

Option	Description
`-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 with default options (experimental)
`--jit-[option]`	enable JIT with an option (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 regexp, similar to `grep`, `sed` and `awk`. 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 all 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 all 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` will automatically loop over input content line by line (with `\n` as the line distinguishing character). The `-n` or `-p` option will enable this feature for `ruby`. As seen before, the `-e` option accepts code as 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 global variable `$_`, which has the contents of the input line (the correct term would be input **record**, see 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 `print` method, which is why it is generally preferred in one-liners over `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
$ ruby -ne 'puts $& if /\d+$/' table.txt
42
7
14
```


The learn_ruby_oneliners repo has all the files used in examples.

Substitution

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

```
$ # for each input line, change only 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 `-n` or `-p` options are used.

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

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

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's some examples that is based on specific field rather than the entire line. The `-a` option will cause the input line to be split based on whitespaces and the field contents can be accessed using `$F` global variable. Leading and trailing whitespaces will be suppressed and won't result in empty fields. More details is discussed in 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 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 input is read and a `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 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 user, file, output of a shell command, etc. As mentioned before, this book assumes `bash` as the shell being used. To access environment variables of the shell, you can call 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 ruby command
$ # note that the variable is placed before the shell command
$ word='hello' ruby -e 'puts ENV["word"]'
hello
```

```
$ # the input 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 that has to be passed to the ruby command
$ r='\Bpar\B'
$ rgx="$r" ruby -ne 'print if /#{ENV["rgx"]}/' 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 global variable `$?` can be used to act upon exit status of 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 typical cli 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 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


Exercise related files are available from exercises folder of learn_ruby_oneliners repo.


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

a) 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
```

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

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

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

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

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

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

e) For each line of the input file `ip.txt`, replace 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
```

f) 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
```

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

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

h) Use contents of `s` variable to display all 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 `s` variable.

```
$ s='is'

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

i) Use `system` to display contents of filename present in second field (space separated) of the given input line.

```
$ 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 `ruby` cli usage, this chapter will dive deep 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 control flow keywords `next` and `exit` .

Regexp based filtering

As mentioned before, in a conditional context:

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

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` . Quoting from ruby-doc: Percent Strings:

> If you are using `(` , `[` , `{` , `<` you must close it with `)` , `]` , `}` , `>` respectively. You may use most other non-alphanumeric characters for percent string delimiters such as `%` , `|` , `^` , etc.

```
$ cat paths.txt
/foo/a/report.log
/foo/y/power.log
/foo/abc/errors.log

$ ruby -ne 'print if /\/foo\/a\//' paths.txt
/foo/a/report.log

$ ruby -ne 'print if %r{/foo/a/}' paths.txt
/foo/a/report.log

$ ruby -ne 'print if !%r#/foo/a/#' paths.txt
/foo/y/power.log
/foo/abc/errors.log
```

Extracting matched portions

You can use regexp related global variables to extract only the matching portions instead of filtering entire matching line. Consider this input file.

```
$ cat programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan

Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

Here's some examples with regexp global variables.

```
$ # note that this will print only the first match for each input line
$ ruby -ne 'puts $& if /\bt\w*[et]\b/' programming_quotes.txt
twice
the
that

$ # extract only capture group portions
$ ruby -ne 'puts $~.captures * "::" if /not (.+)y(.+)/i' programming_quotes.txt
smart enough to debug it b:: Brian W. Kernighan
affect the way ::ou think about programming,
worth knowing b:: Alan Perlis
```

See Working with matched portions chapter from my book for examples with `match` method and regexp global variables.

match? method

As seen in previous section, using `$_ =~ /regexp/` also sets global variables. If you just need `true` or `false` result, using `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 /on\b/'
$ ruby -ne 'print if $_.match?(/on\b/)' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan
There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

tr method

The transliteration method `tr` allows you to specify per character transformation rule. See ruby-doc: tr for documentation.

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

$ # use ^ at start of first argument to complement specified characters
$ # note that input doesn't have newline character here
$ printf 'foo:123:baz' | ruby -ne 'puts $_.tr("^0-9", "-")'
----123----

$ # use empty second argument to delete specified characters
$ printf 'foo:123:baz' | ruby -ne 'puts $_.tr("^0-9", "")'
123
```

Conditional substitution

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

```
$ # change commas to hyphens if the input line does NOT contain '2'
$ # prints all input lines even if 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

$ # prints filtered input lines even if substitution fails
$ # for example, the 2nd output line doesn't match 'by'
$ ruby -ne 'print gsub(/by/, "**") if /not/' programming_quotes.txt
** definition, not smart enough to debug it ** Brian W. Kernighan
A language that does not affect the way you think about programming,
is not worth knowing ** Alan Perlis

$ # print only if substitution succeeded
$ # $_.gsub! is required for this scenario
$ ruby -ne 'print if $_.gsub!(/1/, "one")' programming_quotes.txt
naming things, and off-by-one errors by Leon Bambrick
```

Multiple conditions

It is good to remember that Ruby is a programming language. You have control structures and you can 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 /not/ && !/it/' programming_quotes.txt
A language that does not affect the way you think about programming,
```

```
is not worth knowing by Alan Perlis

$ ruby -ane 'print if /twice/ || $F.size > 12' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Some people, when confronted with a problem, think - I know, I will
```

next

When `next` is executed, rest of the code will be skipped and the next input line will be fetched for processing. It doesn't affect `BEGIN` or `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
```

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

exit

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

```
$ # quits after an input line containing 'you' is found
$ ruby -ne 'print; exit if /you/' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,

$ # matching line won't be printed in this case
$ ruby -pe 'exit if /you/' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
```

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

```
$ tac programming_quotes.txt | ruby -ne 'print; exit if /not/' | tac
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

You can optionally provide a status code along with the `exit` method.

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

```
eat
$ echo $?
2
```


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 numbers

Line numbers can also be used as a filtering criteria. It can be accessed using the `$.` global variable.

```
$ # print only the 3rd line
$ ruby -ne 'print if $. == 3' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan

$ # print 2nd and 5th line
$ ruby -ne 'print if $. == 2 || $. == 5' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
Some people, when confronted with a problem, think - I know, I will

$ # substitution only on 2nd line
$ printf 'gates\nnot\nused\n' | ruby -pe 'gsub(/t/, "*") if $. == 2'
gates
no*
used

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

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

```
$ ruby -ne 'print if $<.eof' programming_quotes.txt
naming things, and off-by-1 errors by Leon Bambrick

$ ruby -ne 'puts "#{$.}:#{$_}" if $<.eof' programming_quotes.txt
12:naming things, and off-by-1 errors by Leon Bambrick

$ # same as: tail -q -n1 programming_quotes.txt table.txt
$ ruby -ne 'print if $<.eof' programming_quotes.txt table.txt
naming things, and off-by-1 errors by Leon Bambrick
yellow banana window shoes 3.14
```

You can use Flip-Flop operator to select between pair of line numbers. See ruby-doc: Flip-Flop for syntax details.

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

For large input files, use `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    0m0.068s
$ time seq 3542 4623452 | ruby -ne 'print if $. == 2452' > f2
real    0m1.158s
```

Fixed string matching

To match strings literally, use the `include?` method instead of regular expressions.

```
$ echo 'int a[5]' | ruby -ne 'print if /a[5]/'
$ echo 'int a[5]' | ruby -ne 'print if $_.include?("a[5]")'
int a[5]
```

The above example uses 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 command line requires messing with shell metacharacters. So, use `%q` instead or pass the fixed string to be matched as an environment variable, which can be accessed via the `ENV` hash.

```
$ # 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 string
$ echo 'int #{a}' | ruby -ne 'print if $_.include?(%q/#{a}/)'
int #{a}
```

```
$ # or pass the string as environment variable
$ echo 'int #{a}' | s='#{a}' ruby -ne 'print if $_.include?(ENV["s"])'
int #{a}
```

Use `start_with?` and `end_with?` methods to restrict the fixed string matching to the start or end of the input line. The line content in `$_` variable contains the `\n` line ending character as well. You can either use `chomp` method explicitly or use the `-l` command line option, which will be discussed in detail in Record separators chapter. For now, it is enough to know that `-l` will remove the line ending from `$_` 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 line
$ s='a+b' ruby -ne 'print if $_.start_with?(ENV["s"])' eqns.txt
a+b,pi=3.14,5e12

$ # end of 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 `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 <= numeric 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
```

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, `ip.txt.orig` will be the backup filename.

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

$ # no output on 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. Using `-i` option on its own will not create backups.

Be careful though, as changes made cannot be undone. In such cases, test the command with sample input before using `-i` option on 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(/an/, "AN")' fruits.txt
$ cat fruits.txt
bANANa
papaya
mANgo
```

Summary

This chapter showed various examples of processing only lines of interest instead of 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 using `()` for compact cli usage. `next` and `exit` are often needed to control the flow of code. The `-i` option is handy for in-place editing.

Exercises

a) Remove only the third line of given input.

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

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

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

c) 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 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
```

d) 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
```

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

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

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

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

g) 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 whole word `is` will not be present more than once in a single line.

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

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

```
$ s='start address: 0xA0, func1 address: 0xC0'

$ echo "$s" | ##### add your solution here
start address: 0x7F, func1 address: 0x1F
```

i) 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
```

j) 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
```

k) 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 all input lines 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
```

l) 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
```

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.

Default field separation

By default, the `-a` option splits 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 field contents. Using `-a` is equivalent to `$F = $_.split` . From ruby-doc: split:

> If pattern is a single space, *str* is split on whitespace, with leading and trailing whitespace and runs of contiguous whitespace characters ignored...If pattern is `nil` , the value of `$;` is used. If `$;` is `nil` (which is the default), *str* is split on whitespace as if `' '` were specified.

```
$ echo '   a   b   c   ' | ruby -ane 'puts $F.size'
3
$ # note that leading whitespaces isn't part of field content
$ echo '   a   b   c   ' | ruby -ane 'puts $F[0]'
a
$ # note that trailing whitespaces isn't part of field content
$ 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  ' | ruby -ane 'puts $F.size'
3
$ printf '     one \t\f\v two\t\r\tthree  ' | 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 the option will be treated as a regexp. Note that `-a` option is also necessary for `-F` option to work. Instead of `-F` option, you can also set `$;` to a string or regexp value in the code, but `$;` is deprecated.

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

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

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

 If the custom field separator with `-F` option doesn't affect the newline character, then the last element can contain the newline character.

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

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

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

```
$ # both 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 last element will have newline character
$ echo ':a:b:c:' | ruby -F: -ane 'puts $F.size'
5
```

As mentioned before, the `-l` option is helpful if you wish to remove the newline character (more details will be discussed in Record separators chapter). A side effect of removing the newline character before applying `split` is that a trailing empty field will also get removed (you can explicitly call `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' method to append the newline character
$ echo 'cat:dog' | ruby -F: -lane 'print "[#{$F[-1]}]"'
[dog]

$ # since newline character is chomped, last element is empty
$ # which is then removed due to default 'split' behavior
$ echo ':a:b:c:' | ruby -F: -lane 'puts $F.size'
4
$ # explicit call to split with -1 as 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 `$,` global variable is used as the separator when multiple arguments are passed to the `print` method. This is usually used in combination with `-l` option so that a newline character is appended automatically as well. The `join` method also uses `$,` as the default value. But `$,` is deprecated now.

```
$ ruby -lane 'BEGIN{$, = " "}; print $F[0], $F[2]' table.txt
-e:1: warning: `$,` is deprecated
brown mat
blue mug
yellow window

$ ruby -W:no-deprecated -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 `-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 get field values from input content. In contrast, `scan` method allows you to define what should the fields be made up of. And `scan` method does not have the concept of removing empty trailing fields nor does it have arguments like `limit` .

```
$ s='Sample123string42with777numbers'

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

$ # define fields to be one or more consecutive alphabets
$ echo "$s" | ruby -lne 'puts $_.scan(/[a-z]+/i) * ","'
Sample,string,with,numbers
```

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

While ruby-doc: CSV library should be preferred for robust `csv` parsing, `scan` can be used for simple workarounds.

```
$ 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: unpack 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 optional length.

```
$ # first field is 5 characters
$ # then 3 characters are ignored and 3 characters for second field
$ # then 1 character is ignored and 6 characters for 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 ASCII NUL separated string.

```
$ 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 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's just too many to meaningfully cover them in 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? 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
```

The `map` method helps to transform 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

$ # you can also use numbered parameters: {_1.to_i ** 2}
$ echo '23 756 -983 5' | ruby -ane 'puts $F.map {|n| n.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 {|w| w[0]!="h" && w.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
$ echo '3.14,17,6' | ruby -F, -lane 'puts $F.reduce(1) {|op,n| op*n.to_f}'
320.28000000000003
```

Here's some examples with `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

$ echo 'foobar' | ruby -lne 'puts $_.chars.sort.reverse * ""'
roofba

$ 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 { |a, b|
                [b.size, a] <=> [a.size, b] } * ":"'
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 {|i,j| $F[j] <=> $F[i]} 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 `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. There's many more examples to be discussed related to fields in upcoming chapters.

Exercises

a) 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
```

b) 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
```

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

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

d) 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 substitution functions?

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

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

e) Construct a solution that works for both the given sample inputs and the corresponding output shown.

```
$ s1='1 "grape" and "mango" and "guava"'
$ s2='("a 1""d""c-2""b")'

$ echo "$s1" | ##### add your solution here
"grape","guava","mango"
$ echo "$s2" | ##### add your solution here
"a 1","b","c-2","d"
```

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

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

g) Transform the given input file `fw.txt` to get the output as shown below. If 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
```

h) 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
```

i) Extract all whole words that contains `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
```

j) 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
```

k) For the input file `mixed_fs.txt` , retain only 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
```

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

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

```
$ echo '20 -983 5 756 634223' | ##### add your solution here
20 756
```

m) 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
```

n) 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
```

o) 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
```

p) 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
```