Leonardo Giordani

Beginner's Guide to the

Unix Terminal

Learn bash and the Unix core utilities in a relaxed way

Beginner's Guide to the Unix Terminal

Learn bash and the Unix core utilities in a relaxed way

Leonardo Giordani

This book is for sale at http://leanpub.com/beginners-guide-to-the-unix-terminal

This version was published on 2020-05-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface

> – *Begin.*
> The Princess Bride (1987)

Who this book is for

This book is meant for anyone who wants to be more productive in a Unix environment such as Linux or Mac OS. This is an introductory textbook, so the reader is supposed to have no previous knowledge of the subject matter, and they should expect to learn the most important commands and concepts they might be using daily.

In recent years the gap between developers and system administrators has been fortunately reduced in various ways, all under the umbrella of the devops philosophy, so this book will be useful to those developers that have to start getting in touch with the "bare metal". These developers might find in the book a gradual learning curve and a quick reference for the most common commands of a Unix environment.

On the other hand, there are people (like the author of this book) who are interested in the low-level side of applications development and want to pursue a career as system administrators or architects. For these people, this book can be a good starting point, where they can be introduced to tools and concepts they will use every single day. They will probably soon feel the need of something more complete and deeper, and there are tons of books and resources that can quench the thirst for knowledge, but I hope they will enjoy the first steps that they will move together with me.

Why this book exists

When I decided to write this book I was working on a series of short workshops for my junior colleagues. I hadn't worked with juniors for many years, and while I was perfectly conscious that some people didn't grow up with a command line, practically speaking I was assuming that they knew what I did, which is probably a mistake that many advanced programmers make.

So, I decided to run some sessions in which we discovered the bash shell, starting from the simplest commands up to some proper scripting. I was rewarded (and I still am) by a lot of satisfied smiles when they finally got the right sequence of pipes or when they discovered regular expressions. From those sessions, and from the notes that I wrote to prepare them, I had the idea of a book that could

be easily followed, maybe in one's spare time, as nowadays it's always very complex for people to find time to study big tomes.

I usually avoid trying to be funny when writing or speaking in English; it's not my native language, so I lack the vocabulary and the comedic timing. But a thing that I learned many years ago is that a laugh helps people to remember what was said, so this time I decided to try with a light-hearted approach. I hope this will ease the read and make your journey into the Unix world more comfortable.

If it doesn't work, I apologise in advance. you can always go and get some ancient tome of dark magic like "The Unix Bible". I'm pretty sure you will remember for years the moment in which you'll manage to conjure some sort of multi-tentacular Lovecraftian horror, but I'm not sure this will have a significant impact on your career. If we exclude the fact that it will probably put an end to it.

Why you should read this book

If you are a programmer, nowadays you probably got in touch with Open Source software, with version control system like Git, with cloud services like AWS, with some dynamic language like Python or Ruby. What do all these things have in common? Many things probably, but one of them is definitely that they are rooted in the Unix world. [1]

That of Unix is a rich world, and this book will not even scratch the surface. but one of the most powerful things Unix systems have is the command line. Yes, in 2019, with AIs constantly processing everything we search and do online, mobile phones with perfect user interfaces, and proper virtual reality around the corner, the good old green-on-black terminal still has something to say. More than something!

The command line is a tool that I think will never get too old to be useful. I am happy when the GUI of some program is simple and well designed, but there is simply nothing that can give you more power than the command line.

So, long story short, you should definitely learn at least the fundamentals of the command line!

Who should not read the book

To put it simply, this is a primer, so if you are an advanced Unix user you should stop reading here. This book is not for you. If you decide to go on, however, please consider that this wants to be an introduction to Unix for people who never heard of it. Yeah, I know it sounds incredible that people can use a computer without a terminal and that they can survive day by day thinking that pipes convey water and that sockets can be found on walls, but this is why I believe a book like this can be useful.

[1] Sure, you can have all these things in your Windows laptop (Windows is not a Unix-like operating system), but I can't recommend it. I mean, you bought a water pistol, and you want to use it with real bullets. Good luck!

To be fair, if you want to introduce a novice to Unix in 2019 you have to face 50 years of history and a legacy of countless choices that sound perfectly reasonable to people who grew up into them. There are at least two widespread implementations of it, Linux and Mac OS X, the first of which comes in dozens of different flavours (distributions) and the second of which behaves in all sorts of unexpected ways.

So, I had to compromise. In particular, dear expert nitpicker, you will notice that I use the terms "Unix", "Unix shell", "bash", "terminal" as synonyms. And believe me, I cringe every time I do it. But, to be honest, this book wants to be a primer for people who don't know anything about what is arguably a niche subject. And my experience tells me that the best way to dishearten someone is to make them feel stupid, flooding the conversation or the lesson with tons of terms they never heard and that they do not understand. Sure, this makes you shine, but what's the point?

So, sorry. I know the nomenclature in the book is all but accurate, and I hope that novices will come back to this book after some years, smiling at the levity with which I desecrated the holy ground of Unix.

The prompt

The command line always prints a fixed string in front of each line when waiting for input. This is aptly called *prompt*. The prompt is highly customisable, as it is an invaluable source of information to have handy. My prompt, for example, typically contains the full path of the directory I'm in at the moment so that I have a clear idea of my position at any time. Many people add the current time, or the Git branch they are in at the moment. Since the prompt changes from system to system, and from user to user, it is custom to represent it with a dollar sign `$`. So this

```
$ command --option value --another-option another-value
```

means that you are supposed to type the string from `command`, as the `$` is already there in some form on your terminal.

I will often show the output of the command as it appears on the terminal, just under the prompt, like this

```
$ command --option value --another-option another-value
output line 1
output line 2
output line 3
output line 4
output line 5
```

I will include the output of the command when it is relevant for the explanation and generally when it is not too long. The idea of the book is that the reader should test the commands on their terminal, so I consider it pointless to fill pages with output that nobody will ever read. When I need to truncate the output I will include an ellipsis between square brackets as in the following example

```
$ command --option value --another-option another-value
output line 1
output line 2
output line 3
[...]
```

If you fancy some historical information, the dollar sign used for the prompt was apparently first introduced in Version 7 Unix, released in 1979 by Bell Labs, as that was the first version to be shipped with the Bourne shell. So much for long-lasting choices!

Discrimination of fruit-labelled products

Mac OS is the Unix-like operating system with the largest desktop user base among programmers, as many have an Apple notebook as their primary device. I'm not one of them, and I was very surprised to find out that the shell provided by Mac OS is only partially compatible with the traditional bash shell provided by Linux distributions. In particular, the core utilities of Mac OS are not the GNU ones, so many command line options are not present, or different.

As I want to publish this book before I retire, I will go for the simplest solution. I won't provide examples or explanations for the Mac OS terminal, I'm sorry. If you are using a Mac, either for your personal choice or because of some external constraint, you will have to look for some specific solutions online.

For the time being, I will provide instructions to run a Linux machine in a Docker container, so everybody can follow the lessons and enjoy the book. Maybe in the future I will provide some coverage of non-GNU versions of the standard utilities, but now this is too complex a task for me.

Despite the mocking tone, I'm really sorry I can't provide instructions for every case! The good news is that, while the single commands have different options the overall concepts are the same, so reading the book and learning how to use commands, how to find help on them and how to compose them in scripts will be extremely useful even for Mac OS users.

Why this book comes for free

The first reason I started writing a technical blog was to share with others my discoveries, and to save them the hassle of going through processes I had already cracked. Moreover, I always enjoy the fact that explaining something forces me to better understand that topic, and writing requires even more study to get things clear in my mind, before attempting to introduce other people to the subject.

Much of what I know comes from personal investigations, but without the work of people who shared their knowledge for free I would not have been able to make much progress. The Free Software Movement didn't start with Internet, and I got a taste of it during the 80s and 90s, but the

World Wide Web undeniably gave an impressive boost to the speed and quality of this knowledge sharing.

So this book is together with other things that I wrote, a way to say thanks to everybody gave their time to write blog posts, free books, software, and to organise conferences, groups, meetups. This is why I teach people at conferences, this is why I write a technical blog, this is the reason behind this book.

That said, if you want to acknowledge the effort with money, feel free. Anyone who publishes a book or travels to conferences incurs expenses, and any help is welcome. However the best thing you can do is to become part of this process of shared knowledge; experiment, learn and share what you learn.

Submitting issues or patches

This book is not a collaborative effort. It is the product of my work, and it expresses my personal view on some topics, and also follows my way of teaching. Both can definitely be improved, and they might also be wrong, so I am open to suggestions, and I will gladly receive any report about mistakes or any request for clarifications. Feel free to use the GitHub Issues of the book repository[2]. I will answer or fix issues as soon as possible, and if needed I will publish a new version of the book with the correction. Thanks!

About the author

My name is Leonardo Giordani, I was born in Italy in 1977. That year gave to the world also Star Wars, bash, Apple][, BSD, Dire Straits, The Silmarillion. I'm interested in operating systems and computer languages, photography, fantasy and science fiction, video and board games, guitar playing, and (too) many other things.

I studied and used several programming languages, from the Z80 and x86 Assembly to Python and Scala. I love mathematics and cryptography. I'm mainly interested in open source software, and I like both the theoretical and practical aspects of computer science.

For 13 years I was a C/Python programmer and devops for a satellite imagery company and I am currently infrastructure engineer at WeGotPOP[3], a UK company based in London and New York that creates innovative software for film productions.

In 2013 I started publishing some technical thoughts on my blog, The Digital Cat[4]. In 2018 I published my first book with Leanpub, Clean Architectures in Python[5], which at the time of writing has been already downloaded more than 10,000 times.

[2] https://github.com/bgutbook/bgutbook/issues
[3] https://www.wegotpop.com
[4] http://thedigitalcatonline.com
[5] https://leanpub.com/clean-architectures-in-python/

Acknowledgments

- Ken Thompson, Dennis Ritchie, Brian Kernighan and others for writing the Unix operating system.
- Linus Torvalds and all the kernel developers for writing the Linux kernel.
- Stephen Bourne for his Bourne shell, Brian Fox for the Bourne-again shell, and many other people contributing to these amazing tools.
- Richard Stallman and all the people working on the GNU project for porting the Unix utilities.
- Machtelt Garrels for her amazing Bash Guide for Beginners. It saved my day many times.
- Łukasz Dziedzic, who developed the free "Lato" font (Latofonts[6]), used for the cover.

The cover photograph is by pxhere[7]. A detail of the photo has been extracted and edited. As a commuter in London, I spend hours in stations and on public transport, and even these places can reveal a secret beauty. This book was mostly written in the morning on busses of the line 26, if there are more typos than words now you know the reason.

Sources

The sources of the knowledge I tried to distil in this book are countless. Many people and the books they wrote, forums, Stack Overflow answers, a lot of experiments, a couple of hard disk partitions zapped trying to understand how the filesystem works, and a decent number of years spent automating processes. While all these are too many to be properly mentioned I can definitely provide the source of the log file used in the examples and the exercises. It is a simplified version of an Apache log file that can be downloaded here[8]

[6] http://www.latofonts.com
[7] https://pxhere.com/en/photo/175470
[8] https://github.com/elastic/examples/blob/master/Common%20Data%20Formats/apache_logs/apache_logs

Setup

> – *This is after the preface but before chapter one*
> An American Werewolf in London (1981)

Example files

Many examples and exercises in the book use text files that are available on the GitHub page of the book. To access them you just need to clone the repository.

```
$ git clone https://github.com/bgutbook/bgutbook_files.git
```

If you don't know how to use Git you can find many resources online, or you can download a zipped version of the repository on the main page of the repository.

Linux users

If you are a Linux user you are basically already set up. The only thing you need to do is to download the test files and make sure that you run examples and exercises inside the cloned directory.

Extract the archive in a directory of choice and reach it from your terminal. If you are using a graphical environment like Gnome or KDE you can usually find a "Open terminal here" command in the file browser. Otherwise you can just fire up a terminal (Konsole for KDE, GNOME terminal for GNOME, or any other terminal emulator) and move to that directory with the following command

```
$ cd path/of/the/directory
```

Where `path/of/the/directory` is the full path of the repository that you cloned in the previous section. Congratulations, you are ready to go!

Mac OS users / Windows users

I will cover only Linux in this book, at least in the initial versions.

Linux is the operating system used in the vast majority of cloud solutions, and is the de facto most widespread Unix on servers, and this is a good reason to cover it.

I'm using Linux on my personal computers as well, but the majority of programmers use a Mac or a Windows machine. The main reason for not covering Mac OS is that its default terminal is not 100% compatible with the Linux counterparts (as Mac doesn't use the GNU utilities). I don't have a Mac and even if I could cover all the different versions of the utilities, I think this would be daunting for beginners. The main reason for not covering Windows is that it is not even a Unix. I know that recent versions include some sort of Linux layer, but again I don't have time and interest in those systems.

So if you are using Mac OS X or Windows, you will have to run Linux in a Docker container. I will provide here two different solutions, the first one is simple but will be missing some parts of the system, while the second is a bit more complex but complete. Please note that in either case you need to have Docker installed in your system, refer to the Docker documentation[9] for instructions on how to do it.

You can come back at any time and run the more complex solution if you initially decided to go for the simple one and you want to try the more complete version.

Simple solution

You can run the default Ubuntu image in Docker. This image has been stripped to be smaller, so you will miss some of the documentation that comes with a Linux system, called "man pages". We will use them in the first lesson and I encourage the reader to use them to learn how to use commands, so if you go for this solution you will have to refer to the online man pages every time you need help on a command. I will mention the online resources in the relevant chapter of the book, so don't worry now.

To run the Ubuntu image first clone the repository with the example files, enter that directory and run Docker. The commands are

```
$ git clone https://github.com/bgutbook/bgutbook_files.git
$ cd bgutbook_files
$ docker run -t -i -v $(pwd):/opt/bgutbook_files ubuntu /bin/bash
root@1d888c92ee76:/#
```

As you can see the prompt changes, which tells you that you are inside the Linux container. Please note that the number that follows the `@` sign in the prompt is the ID of the running Docker container, so it will be different for you. At that point move to the directory that contains the example files inside the container

[9] https://docs.docker.com/install/

```
root@1d888c92ee76:/# cd /opt/bgutbook_files/
root@1d888c92ee76:/opt/bgutbook_files#
```

From here you are ready to start reading the book and run the examples. Whenever you want to exit you just need to type `exit` and Enter. When you want to go on you just need to follow again the process, skipping the `git clone` part. That is

```
$ cd bgutbook_files
$ docker run -t -i -v $(pwd):/opt/bgutbook_files ubuntu /bin/bash
root@1d888c92ee76:/# cd /opt/bgutbook_files/
root@1d888c92ee76:/opt/bgutbook_files#
```

Advanced solution

If you want to have a Ubuntu container with the man pages you need to create a new Docker image. I provide a configuration for such a machine as a `Dockerfile` in the `bgutbook_ubuntu` repository.

First of all clone the repository

```
$ git clone https://github.com/bgutbook/bgutbook_ubuntu.git
```

Then enter the directory and build the Docker image

```
$ cd bgutbook_ubuntu
$ docker build -t bgutbook_ubuntu .
```

Now Docker will create an image called `bgutbook_ubuntu` and store it in your system. At this point you are ready to run it whenever you want to follow along the examples of the book.

Exit the directory with the `Dockerfile`, clone the repository with the example files, enter that directory and run Docker. The commands are

```
$ cd ..
$ git clone https://github.com/bgutbook/bgutbook_files.git
$ cd bgutbook_files
$ docker run -t -i -v $(pwd):/opt/bgutbook_files bgutbook_ubuntu /bin/bash
root@1d888c92ee76:/#
```

As you can see the prompt changes, which tells you that you are inside the Linux container. Please note that the number that follows the `@` sign in the prompt is the ID of the running Docker container, so it will be different for you. At that point move to the directory that contains the example files inside the container

```
root@1d888c92ee76:/# cd /opt/bgutbook_files/
root@1d888c92ee76:/opt/bgutbook_files#
```

From here you are ready to start reading the book and run the examples. Whenever you want to exit you just need to type `exit` and Enter. When you want to go on you just need to follow again the process, skipping the `git clone` part. That is

```
$ cd bgutbook_files
$ docker run -t -i -v $(pwd):/opt/bgutbook_files bgutbook_ubuntu /bin/bash
root@1d888c92ee76:/# cd /opt/bgutbook_files/
root@1d888c92ee76:/opt/bgutbook_files#
```

Part 1 - Basic commands

Day 1 - Getting help

> – *Help me, Obi-Wan Kenobi. You are my only hope.*
> Star Wars (1977)

The most important part of a software is the documentation. Sadly, often our software doesn't have good documentation, as "the fun" is in the code. We mostly enjoy developing a program that does something, not writing boring text. Fortunately for us, the Unix system and the people working in that environment have a great culture of documentation, and so the first thing you have to learn is how to find and read help on the commands you will learn.

Welcome to the Unix manual pages, or *man pages* for short. Whenever you need help on a command you can run

```
$ man <command>
```

and enter a text-only (but very powerful, mind it) help system. To learn how to navigate the system let's get help on a rather simple command that we will learn shortly, `echo`.

```
$ man echo
```

If you are running Linux in a Docker container and you followed the simple solution you will not have man pages installed in your system, so you need to read them at https://manpages.ubuntu.com/and the search form at the top right of the page.

First of all, let me teach you some of the basic fundamentals of man page driving. You can move through the documentation with your arrow keys, down and up, and with the page down and page up keys to quickly scroll the whole page. Please note that, since manual pages are read-only, there is no cursor, just a current line, which is the topmost one.

A nice help text at the bottom of the screen lets you know that this it

```
Manual page echo(1) line 1/68 51% (press h for help or q to quit)
```

Quitting programs can be surprisingly complex at times (ask any `vi` novice user), but `man` exits to the system with a simple `q` for `quit`. Try it, then enter the manual page for `echo` again. Phew! At least we won't be trapped here forever.

The help system is not always that straightforward, though. There are commands to learn there as well. So, how do we get help… on the help? Well, as you can see from the bottom line while `q` stands for `quit`, `h` stands for `help`. So far so good, sooner or later there will be an `a` that stands for `zoo`, but the basic commands are apparently in good terms with the alphabet.

If you press `h`, then, the screen changes and shows the meta-help, which is the help on the help. I'm not going through it, you can read it later. For the time being, be aware that if you don't remember a command or a shortcut, you can ask the system itself before reading the whole Stack Overflow knowledge base looking for the answer.

There is one important action that is worth learning now, namely searching for strings. Manual pages can be very long, and sometimes very boring, endlessly discussing details that are apparently irrelevant, while you absolutely need *that* switch now (most often because your system is crumbling to pieces, users are calling infuriated, but you still want to have the list of running processes ordered by CPU usage, because it's neat).

So, to seach for a string in a manual page you tap `/` (note the bottom line waiting for your input) and type the string that you are looking for. Try `escape`, remember to hit `ENTER` after you typed the word. Well, you should have been gratified by the highlighting of the searched words, which is a great visual help. You can now move forward through the results hitting `n` (`n`ext), and backwards hitting `N` (`N`ot the next one, the previous one).

Exercises

Oh yeah, there are exercises for you! My experience as a learner is that many times I read or listen to something, and I cosider it learned, only to discover that when I'm alone after a while I don't remember it very well, or that some parts of what was discussed are still unclear. So, my advice for you is: go through the exercises, even though they might seem trivial to you. Practising is always good, even when you are a master coder. I'm not, and if you are reading this book, probably you aren't either. Which leads to… exercises!

Exercise 1.01

Enter the man page for the `echo` command and locate the `AUTHOR` (actually the authors).

Go to solution

Exercise 1.02

Enter the man page for the `sort` command (we haven't used it yet, but the man page is there). Enter the online help and locate how to `Undo (toggle) search highlighting` with the search

pattern command. Now exit the help, search for `ignore` and practise the search highlighting toggle command.

Go to solution

This is enough for now. You learned how to get help about commands, so from now on you have a tool to become a master of the Unix shell on your own. I still believe it is worth following the book, though, so don't close it now, there is a lot to learn that is not so easily distilled from man pages.

Oh, by the way. You can run `man man`, which despite the name is not a tome about the human condition.

Suggested film for the evening: Manhunter (1986) - A film about a guy who desperately tries to find man pages in a Unix system.

Day 2 - Printing

> *– Let Polly do the printing.*
> Jaws (1975)

No, forget paper and toner cartridges, we are still working with a keyboard and a screen. In this lesson we will learn arguably the most important part of any programming language or computer system, that is how to print strings on the screen. Yes, you heard me, artificial intelligence, the most useful thing a computer can do is still printing something on the screen. The second thing is obviously moving four ghosts in a labyrinth flooded with beads that you are trying to eat.

Jokes aside (just for this paragraph, however), printing strings on the screen is very important, not only because it is the simplest and occasionally the better way to debug programs, but also because it provides a way to give a feedback to the user. And nowadays, with cloud computing and global networks, it's still extremely common for programmers and system administrators to interact with a text terminal. I personally believe that for these categories of computer users text terminals will never be outclassed by graphical user interfaces.

OK, jokes can come back from the naughty step. Let's stop rambling and finally print a string.

```
$ echo <string>
```

Well, if you typed exactly `echo <string>` you probably got an error, because you should replace `<string>` with something that does not include nasty characters like `<` and `>`. Maybe try

```
echo Hello, world!
```

Yes, I know this is plagiarism.

Whatever, the command should have printed the string on the screen, just under the command line. Maybe you noticed that you didn't need to include quotes, so why don't you try to do it? Run

```
echo "hello, world!"
```

and see what happens. The same? Yes, you are right. So, are quotes useless in bash?

Hardly. Quotes are probably one of the worst topics in the whole bash syntax and generally speaking in the world of Unix scripting. Believe me, you will hate quotes at a certain point of your career, but don't despair. I lived to tell the tale, so you will as well. Just be aware for now, that if you want to wish ill upon an enemy system administrator you can say "May your quotes never work". That is worse than a sentient evil computer that wants to rule the world. I know, I read too much science fiction.

Exercises

For today, before you fall asleep bored by my explanations, try these quick exercises.

Exercise 2.01

Print the string “Just a test”

Go to solution

Exercise 2.02

Print a string without the trailing newline (check the manual page)

Go to solution

Exercise 2.03

Run

```
$ echo "First line\nSecond line"
```

What happens? Can you find a way to convert that `\n` into a newline?

Go to solution

As a final remark, to leave you with some food for thought, what happens if you remove the quotes from the solution of the last exercise? Now go and watch a comedy film, or you will have nightmares.

Suggested film for the evening: Catch Me If You Can (2002) - A film about the dangers of becoming too skilled with printing.

Day 3 - Showing the content of a file

> – *So, have a look inside.*
> Enemy Mine (1985)

Do you remember when I said that the second most important thing a computer can do is to move four ghosts in a labyrinth flooded with beads that you are trying to eat? I didn't lie, it's true. The third one, however, is to show the content of text files.

Back in the ages when I was a teenager (the 90s, just to be clear) we used to exchange data on floppy disks. Yes I know, it sounds like a time where dinosaurs still walked on earth and men went hunting with flint spears (spoiler alert, these two things never happened during the same age), but it's true. Text files were very important, song lyrics, programming tutorials, jokes, all these things came in text files. And you know what? Now, when I open a web page with some band lyrics and my screen is flooded by advertisements, pop-ups, cookie alerts, and unwanted videos I definitely miss those amazing plain dull simple text files.

OK, enough riding the train of nostalgia. My point is that these days text files are… still one of the most important things in the world of computer science! Let's consider this: every Web page and every computer program are, at their source, text files. That's it, easily 80% of what is travelling on data networks these days is still a text file. So, let's see the content of a text file, for once without firing up a text editor.

If you followed the instructions in the setup section, you should have the example files in your current directory. One of these files is `examples.txt`, and we will use it throughout the next lessons to test some commands.

```
$ cat examples.txt
```

You should have been glorified by the content of the `examples.txt` file, and you are probably wondering why things so unrelated with each other are listed together in a text file. Trust me, I'm not completely crazy yet, these things will come in handy.

There is not much more to say, so far. I warned you, text files are not as exciting as all those pop-ups and advertisements you get on your browser.

As this lesson was a bit short and there are not many smart exercises that you can run through, I will entertain you with a fun fact about `cat`. While it is a very appealing name for feline lovers like I am, `cat` actually stands for conCATenate. Indeed, you can name multiple file names on its command line, and what you will get as an output is the concatenation of their content. A side effect is that if you name only one file you are just asking the command to print its content.

Exercises

Printing the content of a file is pretty straightforward, but there are a couple of interesting tricks that may come handy later, so these exercises will help you to discover them.

Exercise 3.01

Print the content of the file `examples.txt` numbering lines.

Go to solution

Exercise 3.02

Check if the file `examples.txt` contains spaces at the end of any line.

Go to solution

Cool, have a walk, enjoy some music, relax. From next lesson we will start to go a bit deeper into Unix command line tricks, and I want you ready for the fight! Feel free to bring the cat along.

Suggested film for the evening: That Darn Cat! (1965) - A beginner Unix programmer has issues with one of the simplest commands and involves the FBI in their struggle.

Day 4 - Beginnings and ends

> – *Come, let's begin.*
> – *We ended in F major.*
> Amadeus (1984)

Many times people don't read long messages. You should be careful when you send emails, for example. If they are too long people tend to read the beginning and then skip the central part and read the end. That is, if the core of the message was the final result of your 20 years long research on time travel which shows how to build a time machine out of a dishwasher, it is lost. Gone. Sorry, you should have been more concise.

While this might be a bad habit people have, reading the initial and the final part of some stream of data comes in handy very often. For example, we might want to list files in order of size and then get the top 5 ones, or we might want to print out the last 10 log messages of your system, just to keep an eye on what is happening.

Well, Unix provides two nice commands to perform those actions, namely `head` and `tail`. You should be able to tell which is which from the name, which is a very nice thing their authors did for us. When you run

```
$ head <file>
```

the system will show you by default the first 10 lines of the file. You can try this and the following examples on the `slices.txt` file provided in the examples repository. That file is just a sequence of numbered lines, from 1 to 20, so that you can easily check if what you did is correct. So `head slices.txt` should shows you lines 1-10. Conversely, if you run `tail slices.txt` you will get lines 11-20, that is the last 10 ones of the file.

Ten lines are usually a good amount of content if you want to have a quick look into a file, maybe to just have a glimpse of how the content is structured, but you might want to print less or more lines. This can be done in both commands with the `-n` switch followed by the amount of lines that you want. That is, `head -n3 slices.txt` will print the first 3 lines of the file.

Please note an important thing about the Unix command line: switches and their values can be optionally separated by spaces, so executing `head -n3 slices.txt` or `head -n 3 slices.txt` gives you the very same result. You can also add more than one space between the switch and the value, if this makes you happy, but when we will start dividing the text in columns you will not be that happy anymore about multiple spaces, so maybe just don't do it. Friendly advice.

Exercises

There are some exercises that I can leave to you about `head` and `tail`. All exercises can be performed on the `slices.txt` example file, to get an immediate feedback from the numbered lines, but if you feel confident you can use other files as well.

Exercise 4.01

Show the first 3 entries of `slices.txt`

Go to solution

Exercise 4.02

Show the last 3 entries of `slices.txt`

Go to solution

Exercise 4.03

Show the content of `slices.txt` skipping the last 3 lines

Go to solution

Exercise 4.04

Show the content of `slices.txt` starting from line 3 (that is, skipping the first 2 lines)

Go to solution

That's all for today. Was it complex? Was it boring? Well, I'm sorry but I can't hear your answer, I wrote this book some time ago. At any rate, if you think we are going to slowly you can go on freely, the daily frequency is just a suggestion. For today, I also suggest a good cup of tea and maybe some biscuits. Next stop, some plumbing: one of the most important things you can learn about Unix.

Suggested film for the evening: Highlander (1986) - There are many beginnings, but in the end there will be only one.

Day 5 - Semicolon, logical AND, pipes

> *– All right. Let's just think this thing through logically.*
> Back to the Future Part III (1990)

So far it looks like bash and Unix are not using those nasty strange symbols many other programming languages or systems use. Well, that's good, isn't it? Yes, but it's also tragically untrue. In this lesson we will meet some of the fancy hieroglyphs that Unix uses to express complex behaviour, so if you are still ruminating on that class of ancient Babylonian language that you missed at the university, be happy. You will soon catch up.

First of all let's explore the semicolon. Try to add a semicolon between two commands, like for example

```
$ echo "First string"; echo "Second string"
```

As you can see bash runs both in sequence, so `;` clearly stands for a sort of "and" between two commands. It is interesting to see what happens if one of the commands fails, though. Let's try to get the content of a file that doesn't exists as first command.

```
$ cat nofile.txt; echo "Second string"
```

Well, the output of this double command is an error message from `cat` (which mercilessly sends us to the naughty step), followed by the output of the `echo` command. So, the semicolon is executing both commands whatever the result of them.

It is however often useful to chain commands in a different way, where a failing command interrupts the execution. For example you might want to find all files with a certain name and create a log file with some statistics, but you don't want to run this last step if there are no files. For the time being let's see what the operator `&&` does to the previous two commands

```
$ cat nofile.txt && echo "Second string"
```

It looks like this is exactly what we were looking for. We get the error message from `cat` (and the naughty step again, sorry), but no output from `echo`. Bash interrupted the execution because one command failed. Put this is your belt, it will come in handy sooner or later.

There is a third ancient glyph that made it through the fall of the ancient Egyptian kingdom and landed in the Unix system, and it is one of the most important operators that you have to learn. So, grab some strong coffee, you need to be awake.

So far, when we run commands the output was printed (magically?) on the terminal, and that's fine, as most of the time we want to see what a command does. The input of the commands, on the other hand, came almost exclusively from files that are already on your hard disk. What happens you you run

```
$ head -n 10 slices.txt | tail -n 5
```

If your system doesn't have the hiccup you should see the lines from 6 to 10 of the `slices.txt` file. But, but… we asked `head` to show us 10 lines, this is definitely fishy. The `|` operator is called *pipe* and its effect is that of piping the output of the first command, preventing it to reach the terminal, and instead going directly into the `tail -n 5` command. So we got the last 5 lines of the first 10 lines of `slices.txt`, that is lines from 6 to 10.

I told you this lesson was about plumbing.

Many Unix commands that work on files accept input from a pipe instead, and you might see now why this is so important. Through pipes you can connect commands, creating your own complex operators from simpler ones.

And now, welcome to the Operating Systems Philosophy class of professor Unix! I'm only half joking here, the so-called *Unix philosophy* dictates that you should provide small programs that do a very specific things, and that do it at their best. You will then compose them with pipes and other operators to create more complex programs.

Caveat You will notice that later in the book I always use the pattern `cat <file> | <command>`, even though the man page of many commands shows that you might as well run `<command> <file>` omitting the `cat` and the pipe. While I use both versions in my daily tasks, I preferred the first one in this book because it makes clearer what the `<command>` does, and it helps familiarising with pipes.

Enough for today! You will probably have nightmares about leaky pipes, philosophers and Pharaohs. I can only recommend a good book and a cup of tea. Just don't read anything related to ancient cultures!

Suggested film for the evening: Stargate (1997) - Some Unix programmers work together on a manufact that apparently shows that ancient Egyptians were familiar with the command line.

Day 6 - Printing specific lines

> – *You've been targeted for termination.*
> Terminator (1984)

Today I want to introduce you to a friend of mine, its name is `sed`. It's full name is actually Stream EDitor, but few remember it and it doesn't care that much. Actually it answers only when you call it with the nickname. Anyway, it is an incredibly powerful command, but for the moment I can't show you exactly what it can do, as you might not get how useful it will be for your work. We will thus explore just one of the many things it can do, and probably the less important one, which is to print lines.

The problem we often have is that of printing a specific line of a text. With `head` and `tail` we learned how to print a certain number of lines starting from the beginning or the end, but what if we want a line somewhere in the middle of the file? One possible solution comes from pipes. If you run

```
$ head -n 5 slices.txt | tail -n 1
```

you are asking `head` to show you the first 5 lines of the file, and then filtering this output with `tail`, getting the last line. And this ultimately gives you exactly the line number 5.

So far so good, but this is a pretty convoluted way to just print line 5, isn't it? Indeed, `sed` thinks the same, and so it provides a simpler way to do it.

```
$ sed -n 5p slices.txt
```

I know, I know. You think that `sed -n 5p` is not the first thing that comes to your mind when simplicity is the subject of the conversation. At any rate, I hope you at least agree on the fact that having only one command is better than having to pipe two of them. Your new friend `sed` accepts input from pipes as well, so whenever you want to see a specific line you can pipe the current command into it. We will see examples of this later in the book.

I said `sed` is extremely powerful, so is it possible it can only print a single line? Not at all, it can definitely print a sequence of lines, you just need to provide the first and the last line separating them with a comma.

```
$ sed -n 3,8p slices.txt
```

This prints lines 3-8 of the given file. Are you not satisfied yet? OK, `sed` can also print a certain number of lines starting from a given one. This often saves me from the shame of showing the world that I am terrible at math.

```
$ sed -n 3,+5p slices.txt
```

This gives you the same output of the previous command, as it start at line 3, printing that and the following 5 lines. Pay attention that `sed` is not showing you 5 lines starting from line 3, but 6 lines. I think you probably still need you math skills.

Did you notice I haven't given you any exercise last time? Yes you did, I'm sure. Well, no exercises for this lesson either, just practice a bit printing lines and try to get used to the weird syntax.

Yesterday I gave you a headache with strange symbols, today I hope I just regained your trust. Let's wrap it up here, there is a new film that awaits you, and some crisps. Remember to put some beers in the fridge, one for you, and one for your new friend `sed`. And maybe give some food to the `cat` as well, poor little thing.

Suggested film for the evening: The Conversation (1974) - Sometimes getting a single line is not the best choice, as this Unix programmer discovers while looking at conversationg logs.

Day 7 - Sequences and counting

> *– A coding sequence cannot be revised once it's been established.*
> Blade Runner (1982)

Today we will relax a bit, lately we have been doing a lot, so it's time to learn something easy and straightforward. So, we will discuss distributed consensus in a network system. No, I'm just joking, that is actually far from being simple. OK, let's pick another topic: what do you think about sequences? I find them extremely useful, for example when you need to rename files and you want to give them a proper order.

Whatever the use, Unix provides a very nice command to create sequences of integers, aptly named `seq`.

Note: I'm always unsure about the proper pronunciation of `seq`. As an Italian, I tend to pronounce it as "sec", but if you want to go for something else I understand you. You can spell it, with the benefit that if you use the right tone you can sound like a field officer during a battle.

The basic usage is very simple, as you just need to specify a number grater than 1 and `seq` will output all integers between 1 and that number.

```
$ seq 10
```

Now, this is useful, but sometimes you need to count between two specific numbers, for example because you want to include 0, or because you simply need to start from 42, which is always a good initial value. In this case you just give `seq` the first and the last number

```
$ seq 4 11
```

We are not restricted to positive numbers, however. If you give `seq` a negative starting point it will happily digest it

```
$ seq -4 6
```

Unfortunately, the opposite doesn't work out of the box. If you try to execute `seq 4 -5` you won't get any output. To perform a reverse count you need to specify the interval between the steps, which is 1 by default

```
$ seq 4 -1 -5
```

This will give you the sequence of integers `4, 3, 2, ... -4, -5`. As you probably guessed, you can change the step arbitrarily, for example using 2 to skip every other number, or any other requirement you might have, for example

```
$ seq 1 2 10
```

As you can read in the man page, the `seq` command has some command-line options. It is particularly useful to learn the `-w` switch, that prepends enough 0s to keep all numbers at the same length. For example

```
$ seq -w 1 10
```

prepends one 0 to the numbers between 1 and 9 to get the same length of the last number, 10. This is called left zero-padding, in the elite circles of programmers, but I bet you can also call it zero left padding and everybody would understand. Now shout "Launch a zero left-padded sequence!" and tell me if you don't feel like one the mecha pilots of some Japanese anime. I honestly thing it's cool.

Another useful and simple thing we can learn today is counting elements. When it comes to lines, words, or characters, often you need to know how many of them are in a file or in the output of a command, and in those cases `wc` is your friend. Arguably, the name of the tool is not the best, but it stands for "word count" and after a while you will get used to it, and it will no more conjure up any idea of private spaces.

Despite of the name, `wc` can count several different things, lines and characters being the most useful ones. Let's test the line count with `seq`, using the `-l` options that makes `wc` output only the number of lines

```
$ seq 1 10 | wc -l
10
```

The tool is very useful when we want to count the number of characters in a string

```
$ echo "This is a test" | wc -c
15
```

Wait a minute... 15? I count 14 characters there, including spaces. What's going on? Well, you know that `echo` adds a newline at the end of the string, right? That newline is a specific non-printable character, which makes `wc` add one to the length. We discussed the `-n` option of `echo` that prevents the newline, and using that you will get the correct length

```
$ echo -n "This is a test" | wc -c
14
```

Exercises

I have one exercise for you this time! Yay! Your excitement is so great that it is travelling in time and reaching me in the past, while I'm writing the book. You can skip it if you want, but I think you should try to solve it to see if we are on the same page.

Exercise 7.01

Print the numbers from 5 to 15 padded with enough zeros to fill 5 digits, i.e. `00005`, `00006`, `00007`, and so on.

Go to solution

Exercise 7.02

Count the number of lines of the file simple.log

Go to solution

That's it for today, I promised you it would have been a relaxing lesson: simple concepts, no homework, what more can you desire? A nice walk, this is what you need! Go, enjoy the nature if you can, or get your mecha and fight the alien invaders one zero-padded sequence at a time!

Suggested film for the evening: WarGames (1983) - A young Unix programmer teaches a supercomputer to run a sequence of battle simulations and to count the casualties.

Day 8 - Variables

> – *You're not part of this equation.*
> Die Hard (1988)

The Unix terminal provides variables, like many programming languages. Unfortunately, unlike other programming languages, bash does not provide data types, and this is a very important thing to remember. In bash, all variables are untyped, or, if you prefer, are just strings.

The assignment operator is `=`, so if you run

```
$ x=pdf
```

the variable `x` assumes the value `pdf` as a string. As happened with `echo` you *can* use quotes, which are usually a good way to signal that we are dealing with a string, therefore you *should*. Remember they are not strictly necessary, though, as you can find many scripts that do not use them.

```
$ x="pdf"
```

You cannot, instead, use spaces, as the line `x = "pdf"` would be interpreted as the execution of the command `x` with two parameters `=` and `"pdf"`. Weird, isn't it? It is, but this is due to the double nature of the terminal. It's a command-line interface to the operating system, but it's also a programming language, so the two aspects sometimes clash.

As I said, variables are always strings, so

```
$ x=5
```

is not assigning the integer value `5` to `x`, but the string `"5"`. This has some implications, but the most important for the time being is that mathematical operators are slightly convoluted in bash. We will discuss this later, however, so you don't need to worry now. Just remember that if you type `x=5+1` you are assigning x the value of the string `"5+1"`. Facepalm moment for you, the first time you will write this in one of your scripts.

To refer to a variable, that is, to use its value, we use the syntax `${variable}`. For example

```
$ x=5; echo ${x}
```

prints `5` on the terminal. You will often see people using a shorthand version of this syntax, which avoids the the curly braces. That means that `$x` is most of the time the same as `${x}`. My advice is to get used to the full syntax now and never use the short one, as it can be confusing if underscores are involved. I will dive into this is a later lesson, so for now do yourself and your fellow programmers a favour and forget about the short notation. It's just a pair of curly braces, after all, and using them makes the code clearer.

You can use variables anywhere in bash, as they are replaced before commands get executed. For example

```
$ x=5; sed -n ${x}p slices.txt
```

Arguably not the easiest syntax to read, but it is you who signed up for the course "Ancient spells and mystical rituals", after all. Wait a minute, what do you mean you just wanted to be a programmer? What is the difference?

After variables, command substitution is one of the features you will use the most if you get into bash scripting. Command substitution simply means that you assign to a variable the full output of a command, and this is used often because it allows you to manipulate that output, to loop over its parts, and in general to use it creatively.

Command substitution is expressed by the syntax `$(command)`, where `command` is any line that you could execute on the terminal. For example we might assign the output of a sequence to a variable

```
$ x=$(seq 1 5); echo ${x}
```

Please note that in this case the variable `x` gets the value of the string `"1 2 3 4 5"`. It is not a list, or any other complex data type, just a simple string made of digits and spaces. Bash automatically converted the newline characters used by `seq` into spaces, and this will come in very handy when we will learn how to write loops.

Cool, now enjoy the rest of the day, we will soon learn how to transform a pumpkin into a coach, and mice into servants, just remember that at midnight everything goes back to text files.

Suggested film for the evening: A Beautiful Mind (2001) - A film about the importance of variables, and the dangers of getting too involved in Unix scripting without having a good walk sometimes.

Day 9 - Fields

> – *In the plus column, though she makes a hell of a cup of coffee.*
> Batman Returns (1992)

Data is often structured, and many times a good way to split data is to use fields separated by a specific character. This is true for Comma-Separated Values files (CSV), but it is also true for other types of data. For example, the words in a sentence are separated by spaces, the numbers in an IP address are separated by dots (`192.168.0.1`), and hours, minutes, and seconds in a time string can be separated by colons (`08:41:12`).

It is thus definitely useful to learn how to pick a specific field or set of fields from a string, and the best way to do it in Unix is to use the command `cut`. This command has two important options: `-d` specifies the character used to split the fields, and `-f` specifies the fields themselves, counting from 1. Let's consider an example

```
$ echo "12:34:56" | cut -d ":" -f 2
```

This splits the string `12:34:56` into three fields, namely `12`, `34`, and `56`, as the separator given to the command is the colon character, and then picks field number 2, which corresponds to minutes. Using command substitution we might then assign this to a variable

```
$ minutes=$(echo "12:34:56" | cut -d ":" -f 2)
```

The `-d` option accepts a single character as a delimiter, and the default value is `TAB` as in the special character for tabulation, being that a common way to space fields. As you can see I wrapped the colon in quotes, which are not strictly necessary, but they become so when we want to use a space

```
$ echo "This is a sentence with many words" | cut -d " " -f 4
```

My advice is to just always use the quotes, it is useless to waste time thinking if you need them or not. As we said, the field option accepts the number of a field, but it can also parse sequences like `4-7`, lists like `4,7`, and combinations of the two. Read the manual page to discover all the supported formats.

Exercises

Time for some exercises! At this point we have a pretty good amount of tools in our belt, so we can start performing useful tasks. Let's consider the simple.log file, which represents the part of the logs of a web server (a simplified version of an actual log of the Apache server). Remember that you can print the content of the file beforehand with either `cat` or `head`

Exercise 9.01

Print fields 1,6,7, and 8 of the file `simple.log`

Go to solution

Exercise 9.02

Extract the time of each request as HH:MM:SS

Go to solution

Speaking of cutting, we made it in one piece apparently. Well, at least I am still OK, on the bus on my way to the office. Where are you? If you are commuting I hope you'll get soon to your office or home, and in this second case that you can have a good time relaxing and not worrying too much about fields.

Suggested film for the evening: Edward Scissorhands (1991) - Cutting has never been more fun!

Day 10 - Sorting and reducing

> *– But in the Latin alphabet, Jehovah begins with an "I".*
> Indiana Jones and the Last Crusade (1989)

Sorting data is paramount in so many cases. We sort data to ease the visual browsing, to pick the top entries according to some quantity, or to spot duplicates. Needless to say, Unix provides a specific tool for sorting, which is unsurprisingly called `sort`.

So, we are hackers, we pose as the keepers of deep and powerful knowledge about computers, and we go around sorting things with a command called `sort`? We definitely have to come up with some command with an obscure and meaningless name! Oh, wait, there are `grep` and `sed`, but we will discuss about them in a later lesson.

For the time being, let's spend some time with the self-explanatory `sort`. Since sorting is so simple and straightforward, `sort` is a command with… only 30 possible options. Well, it turns out simple things are many times not so simple! I will show you only a couple of these options, the ones I happen to use the most, but remember that you can read the man page and discover new useful ones.

For starters, let's use the bare command

```
$ cat examples.txt | sort
```

or, if you prefer

```
$ sort examples.txt
```

Nice and simple. You might notice some oddities, though. Why the empty line at the beginning? And why is "* TM Sony Pictures" listed after "Spider-Man [*]"? By default, `sort` uses the so-called dictionary order, that, according to the man page, "considers only blanks and alphanumeric characters". This means that the " " part of the string is not considered, and the position is given by the "T" letter of "TM", while the first line is the empty line that was in the second-to-last position in the original file. being empty, that is considered as a pure space that comes before letters in the ASCII code. According to the ASCII code, numbers come before letters as well, which is why "007" is positioned before "aardvark" in the sorted output.

When sorting numbers, however, the dictionary order gives results that are probably not what we want

```
$ seq 1 20 | sort
1
10
11
12
13
14
15
16
17
18
19
2
20
3
4
5
6
7
8
9
```

We probably want `sort` to understand that `9` comes before `10`. This can be done with the `-n` option, which implements the numeric sort.

```
$ seq 1 20 | sort -n
1
2
3
4
[...]
```

The default sorting order used by the command is ascending, that is following the alphabet (or the value of the numbers). To show the sorted list in reverse order we can use the `-r` option

```
$ cat examples.txt | sort -r
```

which can be combined with `-n` to reverse a numerical sort

```
$ seq 1 20 | sort -nr | head -n 5
20
19
18
17
16
```

The final command that I want to show you in this chapter appears often after `sort` and is called `uniq`. Its job is that of removing duplicated lines, leaving only one occurrence. This command, however, works comparing a line with the following one only, which is the reason why we run it after a `sort`.

The file `examples.txt` contains the word `cat` several times (because I'm not a cat lover, I'm a feline worshipper. Guess what's my favourite bash command). You can notice that the pure `sort` command lists that word three times in a row. If you run

```
$ cat examples.txt | sort | uniq
```

though, you will see it listed only once. Not everybody hate duplicates, though (ask Gaius Baltar), so `uniq` has several options that perform different tasks like for example printing only the duplicate lines. At any rate, in my experience the default behaviour is the most useful one.

Is there anything that people like more than sorting? Oh yeah, and it is pizza! Oh, sorry, I must have messed up my notes. What was I saying? Oh yes, what do we love more than sorting? Counting, naturally!

So, `uniq` can compress a sorted text, removing duplicated lines, but counting them, giving as output a nice report of the number of times a certain line appeared.

```
$ cat examples.txt | sort | uniq -c
      1
      1 007
      1 aardvark
      1 basilisk
      1 beholder
      1 Big Bad Wolf
      1 bull
      1 C-3PO
      3 cat
      1 corn dog
      1 Cyborg 009
      1 direwolf
      2 dog
      1 dryad
      1 Dug the Dog
```

```
      1 elephant
      1 gorilla
[...]
```

As you can see results are ordered and reduced, but `uniq` also counts them before removing duplicates, so we know that there are `3` lines containing `cat` and `2` lines containing `dog`, while `basilisk` appears only once in the file. The result of `uniq -c` can be sorted again to get the results that appear the most or the least, you will have to find a way in the first exercise of this chapter.

Exercise 10.01

Print the `5` more frequent IP addresses in the file simple.log

Go to solution

Exercise 10.02

Print the HTTP methods used by the requests in the file simple.log and count how many occurrences are there for each one.

Go to solution

Now, this is what I think you should know about sorting and removing duplicates. Time to relax! What about some pizza? After all I was wrong before, we probably love pizza much more than sorting! Cowabunga!

Suggested film for the evening: Honey, I Shrunk the Kids (1989) - A scientist gets reducing a bit wrong and his kids have to sort it out.

Day 11 - Find and replace text

> *– I've got to find a replacement for the chief.*
> Das Boot (1981)

Searching is one of the more frequent activities of computers, algorithms, and also programmers. You search a given variable in a text, you want to find all the occurrences of a name in a book, you want to extract all the timestamps in a log file with a specific day and hour. These are just some examples of how searching can be used in your daily life (as a programmer or not).

Replacing text is less frequent than searching only because you fist need to search for the text in order to replace it. You want to replace text because you need to format some data in a different way, because you want to reuse part of it, and in general because you want to transform data.

In this chapter I will introduce two tools that I use every single day, and definitely in every script that I write, `grep` and `sed`. We already briefly covered `sed` in chapter 6, but in this chapter we will start discovering its true potential. Both tools, `sed` in particular, are very powerful and provide many options, thus I will not cover them completely in this book. You will find a lot of information in the respective man pages and in plenty of online resources covering these tools.

We will unlock the full power of both `grep` and `sed` only once we will learn regular expressions in Part 2 of the book, but in this chapter we can start familiarising with the syntax and with their role. In the whole chapter I will make use of the examplex.txt file that you should have among the files provided for the book.

Let's start with `grep`. This tool was born inside the grandfather of all editors, `ed`, as the "Globally search a Regular Expression and Print" command, and was eventually converted into the stand-alone utility that we are still using nowadays. The tool's man page says that `grep` prints lines that match patterns, thus it's the tool that we will use every time we need to find specific parts of some text.

Are you already asleep? I am dozing. Enough chatting! Code first and boring explanations later!

If you execute this code

```
$ grep "elephant" examples.txt
elephant
```

you notice that `grep` found the line that contained the proovided string `elephant` and printed it. The tool find partially matching lines by default, that is lines that contain the searched string and other characters.

```
$ grep "red" examples.txt
undead red dragon
```

If you want to find only the matches that cover the whole line use the `-x` option

```
$ grep "dog" examples.txt
dog
dog
corn dog
$ grep -x "dog" examples.txt
dog
dog
```

By default `grep` performs a case-sensitive search, so lowercase and uppercase are meaningful. If you want to perform a case-insensitive search use the `-i` option

```
$ grep "H" examples.txt
H2O
HTTP/1.1
$ grep -i "h" examples.txt
elephant
ostrich
Dug the Dog
beholder
H2O
phase spider
Johnny 5
hogwash
wild hog
HTTP/1.1
hog
```

Once we find what we need, or when we have a certain text in a variable, we often want to change it, replacing parts of it with something different. The `sed` utility can do this (and actually many other things). Try to execute

```
$ grep "elephant" examples.txt | sed s,"ele","oli",
oliphant
```

which replaces the string “ele” with the string “oli”. The `s` command of `sed` needs to receive the search and the replacement patters surrounded by a character that is not part of them. In this case I used the comma, but you will find many times the `/`, which is another typical character used for this tool

```
$ grep "elephant" examples.txt | sed s/"ele"/"oli"/
oliphant
```

I tend to avoid the `/` in my `sed` command because many times I run it on file paths, which contain the `/`, thus making the command fail. Obviously, the comma doens't work well if your main activity is to process comma-separated values files (CSV), so your mileage may vary. Just to see that `sed` really doesn't mind the separation character, try to execute

```
$ grep "elephant" examples.txt | sed svelevoliv
oliphant
```

To be clear, not only I do not endorse such a cryptic syntax, but if you dare to use it and tell people that I told you to do so I will `grep ${you} universe.txt | sed s,"${you}","void",`. Friendly advice. =) You can also notice from this last example that quotes are optional. As usual, I believe they make the whole command easier to read, so my suggestion is to use them always.

By default, `sed` will replace only the first occurrence of the search string, and if you want to replace all of them you need to use the `g` option

```
$ echo "abracadabra" | sed s,"a","_",
_bracadabra
$ echo "abracadabra" | sed s,"a","_",g
_br_c_d_br_
```

One interesting feature of `sed` is that the replacement pattern can contain the matched pattern, which avoids repetitions if we need to reuse it

```
$ echo "abracadabra" | sed s,"a","&-",g
a-bra-ca-da-bra-
```

This will be superseded by groups and backreferences as soon as we will learn regular expressions, but for the time being it's a nice trick to learn.

Exercises

There are not many interesting exercises to do with `grep` and `sed` in this simple form. Be sure to practice their different options, though, as they will come handy.

Exercise 11.01

Using `grep` and `wc`, count how many times the file `examples.txt` contains the word `dog`

Go to solution

Exercise 11.02

Print all the lines of `examples.txt` that do not contain either a lowercase or an uppercase `h`

Go to solution

Exercise 11.03

Replace all letters `e` in the file `examples.txt` with a question mark `?`, then find among the resulting lines all the ones that have a space in them

Go to solution

We will meet `grep` and `sed` again (and again and again) in the book and basically every time we will write some script. When we will learn regular expressions they will be back in play, and we will unleash their true power! Have a nice evening, or, if it's still morning `s,"evening","day",`!

Suggested film for the evening: Fantasia (1940) - What else do you think about when you read "elephants" and "abracadabra"?

Part 2 - Regular expressions

Day 12 - Regular expressions - Single characters

> – *You know, you got to keep regular if you want to be happy.*
> The Shining (1980)

Some time ago, I needed to review around 1000 files to check if IP numbers were mentioned in them. I wasn't looking for a specific number, the requirement was to find IP numbers in general. Trying all of them was absolutely out of the question, as IPv4 has a pool of 4 billion possible addresses. At 1 address per second it would take a little bit more than 136 years, and while I am still in my forties I somehow doubt that I have that amount of time. Especially to spend finding IP addresses! Fortunately I know regular expressions, and the whole task took me approximately 1 minute.

Regular expressions (regex) are the most powerful tool you can learn when it comes to searching (and replacing) text. They are strings, with a specific syntax, that are used to search other strings. They unfortunately have a reputation of being incredibly complicated, and I hope I will succeed in debunking this myth. You can definitely come up with complex regular expressions, and sometimes it can take you a while to figure out how to express a specific pattern, but you will see that they are definitely accessible.

Did I manage to instill hope you? Yes? Good, because this chapter will be incredibly difficult. Just kidding!

To practice regular expressions we are going to use the file examples.txt and our good friend `grep`. Let's see `grep` in action with the option `-E` that enables regular expressions

```
$ grep -E "dog" examples.txt
dog
dog
corn dog
```

So, apparently the option `-E` doesn't change the tool's behaviour, so disappointing. This is because the string `dog` is already a valid regular expression, namely the one that searches for the character `d`, followed by the character `o`, followed by the character `g`. You are probably thinking that I'm just beating around the bush, right?

Well, time to write the first "proper" regular expression, or at least to use some of the special syntax that regular expressions provide. After all, we are not here to just see cats and dogs. Let's run the following command (remember there is no `-o` option this time)

```
$ grep -E "d." examples.txt
dog
beholder
dryad
dog
aardvark
corn dog
direwolf
phase spider
undead red dragon
Spider-Man [*]
wild hog
Big Bad Wolf
```

As you can see `grep` highlights groups of two letters, all of them starting with `d`. The "do" in **do**g, the "de" in spi**de**r, the "dr" in **dr**agon, all these lines have one thing in common: they contain a `d` followed by another symbol (a letter or a space, as happens for example in "wil**d** hog").

This is what the symbol `.` does in a regular expression. It doesn't mean a full stop, like in the standard punctuation usage, but "any character". Whenever a regular expression contains a `.` there can be any single character. Mind the fact that a single `.` matches a single character.

As you can see, then, regular expressions are simple strings, but they can contain either normal characters (mostly letters of the alphabet, both lowercase and uppercase, and numbers) and special ones. So far we learned about only one of the special characters, that is `.`, commonly called "dot" in this context.

How do we match a proper dot in the string? Since regular expressions assign a special meaning to some characters, when you want to use those characters for their original value you have to escape them with a `\` (backslash). So, while

```
$ grep -E "1.1" examples.txt
Police 101
HTTP/1.1
```

matches two characters "1" separated by any single character, the regular expression

```
$ grep -E "1\.1" examples.txt
HTTP/1.1
```

matches only those separated by a literal dot. Pay attention that the dot can be a punctuation mark, a decimal point, or have any other meaning. Regular expressions don't know anything about the text that you are parsing, they just consider pure characters.

The other important tool that can use regular expressions is `sed`, that we already met twice in the previous chapters. To activate them in `sed` you need to use the `-r` option, and this makes the search pattern in an `s/` command a regular expression.

```
$ head examples.txt | sed -r s,".g","--",g
d--
cat
elephant
ostrich
D-- the D--
beholder
dryad
d--
Police 101
aardvark
```

The previous command searches for any character followed by a lowercase g (`.g`) and replaces it with a double dash. It obviously doesn't make sense to have regular expressions in the replacement part of the command, as regexes are used to search.

Regular expressions in `sed` are often used to delete unwanted parts of a line. Let's say that I want to print the first `5` lines of the file simple.log removing everything after the `GET` part of the line

```
$ head -n 5 simple.log | sed -r s,"GET.*","GET",
83.149.9.216 [17/May/2015:10:05:03] GET
83.149.9.216 [17/May/2015:10:05:43] GET
83.149.9.216 [17/May/2015:10:05:47] GET
83.149.9.216 [17/May/2015:10:05:12] GET
83.149.9.216 [17/May/2015:10:05:07] GET
```

I used the `.*` patter that matches all to remove the part od the line that I didn't want in the output. As you can see I need to repeat `GET` in order to preserve it. We will learn later in the book a technique to avoid this repetition.

Exercises

Exercise 12.01

Match "dog", "Dog", and "hog" into examples.txt

Go to solution

Exercise 12.02

Log entries in the file `simple.log` contain the string `HTTP/<version> <code>`, where `<version> is the version of the HTTP protocol in use (either 1.0 or 1.1) and <code>` is the three-digits HTTP request status code. Extract all lines with a status "4xx" (that is a status between 400 and 499). Count how often each status occurs.

Go to solution

Let's wrap up this short (and hopefully simple enough) introduction to regular expressions. You can already appreciate that a simple syntax like "d." has a powerful meaning and can match many different strings. In the exercises we were already able to create a nice and useful filter for HTTP codes without knowing in advance which ones were contained in the file. In the next lesson we will keep discussing regular expressions introducing new parts of their syntax.

If you head is spinning, sit, relax, have a drink (maybe a soft one, otherwise the spinning will just get worse). It's time for something not related to regular expressions, editors, and command lines. What about a documentary on the history of operating systems? Just kidding! Have a walk, watch a TV series, enjoy your day!

Suggested film for the evening: The AristoCats (1970) - There are too many dogs in this chapter, it's time to fix it properly.

Day 14 - Regular expressions - Classes

> *– I sat next to you in Mrs. Walsh's English class!*
> Groundhog Day (1993)

You were so amazed by the power of regular expressions that you decided to come back and proceed with your education! What did you say? I see, your boss forced you to learn regular expressions but your real dream is to be an action films star. You should consider taking some acting classes. Speaking of which, the topic of this lesson is classes in regular expressions, which are not evening courses, but collections of characters.

In the previous chapter we learned how to match a single specific character in a regular expression and how to match any single character. These two choices are very handy but often we need to match a set of character, for example the numbers 1, 2, or 3, or the letters between "a" and "f". Neither of the two syntaxes we discussed last time can provide this sort of match, so we need a new one.

In a regular expression, the syntax `[<characters>]` means "any single character in the list", and it is exactly what we need in this case. For example

```
$ cat examples.txt | grep -E "[abc]"
```

matches a single "a", a single "b", or a single "c". Remember that `grep` highlights all matching elements in each line, and prints the whole line, use the `-o` option if you want to print the matching part only. The line "dog", for example, is excluded from the output as it doesn't contain any of the three letters in the class.

Classes are especially useful because they allow you to use ranges. For example

```
$ cat examples.txt | grep -E "[a-z]"
```

matches all lowercase letters of the English alphabet. This will highlight whole words like "gorilla" and "aardvark", as they are composed of lowercase letters only. "Johnny 5", instead, is not completely highlighted, as the capital "J" and the number 5 are not matched by the regular expression.

If you use regular expressions in an editor to search for strings (I let you discover how your favourite editor allows you to do this) the syntax `[a-z]` will match the first lowercase letter in the text. Repeating the search will find the second one, and so on.

Typical ranges are `a-z` for lowercase letters, `A-Z` for uppercase ones, and `0-9` for digits. You can use more than one range in a class, for example

```
$ cat examples.txt | grep -E "[a-zA-Z]"
```

matches any letter, but not digits. Pay attention that a range like `A-z` (uppercase A and lowercase z) works but will match also other characters that are encoded between uppercase and lowercase letters, namely `[`, `\`, `]`, `^`, `_`, and “`. If your intention is that of including those characters you might want to explicitly add them to the class, as the `A-z` syntax might be easily overlooked. Be kind to whoever will have to debug your code, the Unix terminal is already more cryptic than ancient tomes of magic, don’t make it worse.

Remember that a class, that is the whole block between square brackets, brackets included, is just an element of the regular expression, and can be followed or preceded by other elements. Exercises 2 and 3 will help you to see this in practice.

Exercises

Exercise 14.07

Match any line of examples.txt containing a digit

Go to solution

Exercise 14.08

Match any line of examples.txt containing a lowercase “a” followed by any letter (that is “aa”, “ab”, “ac”, and so on)

Go to solution

Exercise 14.09

Match any line of examples.txt containing an upper case letter followed by a digit

Go to solution

Exercise 14.10

Match any line of examples.txt containing a dash

Go to solution

Exercise 14.11

Match any line of examples.txt containing a left square bracket "["

Go to solution

That's all about classes, as you can see the regular expressions syntax allows you to express rich patters in a short way, which is good! Now, use the time you saved thanks to the smart regular expressions! Have a chat with a friend, read a comic, lay in the sun if it is summer, build a snowman if it is winter. Class(es) dismissed!

Suggested film for the evening: Dead Poets Society (1989) - A Unix programming class with a teacher that knows which man pages to keep and which to rip out.

Day 15 - Regular expressions - Anchors

> – *Okay, here. Stop. Throw anchor.*
> The Karate Kid (1984)

This is day 3 of your journey into regular expressions, are you ready for some more magic? Today we will learn how to use anchors. No, we are not setting boats on fire, what did you understand? We are going to discover how to tell a regular expression where the text is and how to manage repetitions.

Let's start with anchors. You might like to know that the HTML tag `<a>` commonly used to include hypertext links, comes from anchor, as it was a way to anchor an element to a specific place in the markup. Old times, now things are pretty different. Anyway, this has nothing to do with regular expressions but for the fact that we generally have the need of specifying where some text has to be found.

Before we dive into positioning, however, I want to discuss text that spans multiple lines. Generally speaking, the separation of a text into lines is a pure convention. We agree that the characters `\n`represent a new line, and this convention comes from the C language that wanted to ensure maximum portability of the code, as the ASCII hexadecimal value `0x0a` might not be recognised by some systems.

Text on a new line is usually considered separated from the previous one, or at least identifying a different entry in a list. For example, when we make a list of people in a group, we generally put two different people of two different lines, which avoid any possible misunderstanding between names, middle names, and surnames.

So, Unix tools generally work line by line, as we already noticed with `grep`. This is not different when we use regular expressions, as they are applied by tools that read and process a text line by line. There are ways to apply a regular expressions to a multiline text, but I leave this for a future lesson.

Let's go back to anchors. When we want to find a string in a longer text, the two more frequent cases we will encounter are when the string is at the beginning of the line and when the string is at the end of it. The syntax of regular expressions provides two special characters to signal this, aptly named *anchors*, which are `^` (*caret*, or *hat*) and `$` (*dollar*). Please not that the `$` used by some shells as prompt (and used in the examples of this book) is not related the the `$` anchor at all.

I think you remember the first regular expression that we learned

```
$ grep -E "dog" examples.txt
dog
dog
corn dog
```

which returns all lines with the string `dog` somewhere in them. If you run

```
$ cat examples.txt | grep -E "^dog"
dog
dog
```

you will notice that the last line is not there, because the `^` anchors the string `dog` to the beginning of the line. Conversely, if you run

```
$ cat examples.txt | grep -E "dog$"
dog
dog
corn dog
```

you will get the same result of the first run, as all those `dog` strings are at the end of the line.

You can also combine the two in a single regular expression if you need to anchor something at the beginning and something else at the end

```
$ cat examples.txt | grep -E "^co.*og$"
corn dog
```

From the last example, it follows that if you want to match the exact content of a line you just need to surround the search pattern with anchors

```
$ cat examples.txt | grep -E "^hog$"
hog
```

If you try the last example without either the `^` or the `$` you will see that the result changes as the pattern matches only part of the line.

Exercises

Exercise 15.01

Match every line of examples.txt that ends with an upper case letter and a number (in this order)

Go to solution

Exercise 15.02

Count how many empty lines are contained in the file examples.txt

Go to solution

Anchoring text is very important as many times the pattern we are looking for is repeated in other parts of the line, but those are not interesting. The beginning and the end of the line are good reference points to start analysing text, so it is good that regular expressions provide this two special positioning characters.

In the next lesson we will discuss multiple matches, but for the time being be happy, your knowledge of regular expressions is increasing day by day!

Suggested film for the evening: Jason and the Argonauts (1963) - When is comes to use anchors mister Jason is the system administrator you need; he also knows a lot about escaping.

Day 16 - Regular expressions - Multiple matches

> – *A date gives you a corsage, not a multiple fracture.*
> Little Shop of Horrors (1986)

Well, not bad at all! We are still alive after 3 lessons about something that is considered an advanced topic. Congrats! I hope you are not only surviving, but actually enjoying the journey. I think you start to appreciate that regular expressions are not actually difficult, they are however complicated, full of special symbols and rules. So far we learned how to use `.` for any character, square brackets `[` and `]` for classes and ranges with `-` inside them, and finally the two anchors `^` and `$`.

Today we'll have a look at multiple matches. Generally speaking a multiple match is a repeated match of a previous regular expressions, and typical use case is when you need to match a specific number of digits or letters, but multiple matches can also be less specific, for example matching an indefinite number of lowercase letters.

Let's start with exact matches, which are performed with the syntax `{N}`, where `N` is the number of matches . As I said, all multiple matches operations refer to a previous regular expression, so if you write

```
$ grep -E "a{2}" examples.txt
aardvark
```

you are asking `grep` to match all groups of `2` adjacent characters `a`, as in `aardvark`. The number between brackets can be any positive number, even though using `1` makes no sense, as a single character is already a regular expression matching one repetition of it. So, while you can execute

```
$ grep -E "a{1}" examples.txt
```

and get the correct result, this is equivalent to

```
$ grep -E "a" examples.txt
```

and I personally don't see a point in making the regular expression more complex to read introducing the braces. If you like complicating your life try to create a social network in PHP. Wait a minute, what you mean they did it?

The braces repeat the previous regular expression component, so the syntax `a{2}` is equivalent to a literal `aa`. The syntax can be used to repeat more than letters, though, as they apply to any previous component of the regular expression. This command, for example

```
$ grep -E "[a-z]{3}" examples.txt
dog
cat
elephant
ostrich
Dug the Dog
beholder
[...]
```

matches exactly three adjacent lowercase letters. Is is worth noting that the patterns do not overlap: for example, in `elephant` the regular expression matches `ele` and `pha` only, and not `ele`, `lep`, `eph`, and so on. Once a pattern has been matched it is skipped to continue the line analysis. This is pretty clear if you use the `-o` option of `grep` that we learned previously, which outputs only the matching part of the string

```
$ grep -Eo "[a-z]{3}" examples.txt
dog
cat
ele
pha
ost
ric
the
beh
old
[...]
```

The braces allow you to specify ranges, so `{n,m}` means from `n` to `m` repetitions, `{n,}` means `n` or more, and `{,m}` form zero to `m`. For example

```
$ grep -E "[a-z]{8,12}" examples.txt
elephant
beholder
aardvark
direwolf
manticore
basilisk
```

matches all lines containing at least one word made of 8 to 12 lowercase letters. Two specific ranges a re used very often, namely `{1,}` and `{0,}`. The first matches one or more repetitions of a component, and it is useful when you know that at least one occurrence will be there, but you are unsure of the upper limit. The second one, instead, is used to match possible occurrences of a component. As these two ranges are very important, there is a special syntax for them. `{1,}` can be written `+`, while `{0,}` can be written `*`. So, the code

```
$ grep -Eo "D[a-z]+" examples.txt
Dug
Dog
```

matches the letter `D` followed by at least one lowercase letter, so the output contains both `Dug` and `Dog` (pay attention to the `-o` option used here to clearly show the matches). The code

```
$ grep -Eo "D[a-z]*" examples.txt
Dug
Dog
D
```

instead, also matches the single `D` that comes from the line `R2-D2`, as that `D` is actually followed by zero or more lowercase letters. Pay attention to the last two examples: as you can see the `+` and the `*` are applied to the last *component* of the regular expression, as happened for the braces, and not to the last *character*. The syntax `a+` works because a single letter or digit is also a regular expression component on its own.

Exercises

Exercise 16.01

Match all the occurrences of exactly three digits in the file examples.txt

Go to solution

Exercise 16.02

The log file simple.log contains the status of HTTP responses, which is a three digit number preceded and followed by a space. HTTP statuses are categorised according to the initial digit and 4xx responses (that is 400, 401, 402, and so on) are considered resource errors. Count how many HTTP 4xx responses are in the file for each type of error.

Go to solution

Exercise 16.03

The log file simple.log contains the IP address of the client for each request. IP addresses are made of four numbers separated by dots (i.e. `A.B.C.D`), where each number goes from 0 to 255 (thus having from 1 to 3 digits). Find the 5 IP addresses that occur the highest number of times in the file, counting them

Go to solution

Exercise 16.04

The file simple.log contains the HTTP method used in the request (for example `GET` or `POST`) followed by a space and the rest of the log line. For each request that uses a `GET` print the HTTP method and everything that follows.

Go to solution

Multiple matches are one of the most used components of the regular expressions' syntax, so make sure you feel comfortable with them. After all, they save a lot of typing, and allow to cover a great number of cases with just a couple of symbols, so they are a good tool for your belt.

Now it's maybe time to relax, if it's evening and you are home, or to focus on something else if you still have to begin your day. Have a great day/a lovely evening!

Suggested film for the evening: The Social Network (2010)

Day 17 - Regular expressions - Groups

> – _They come in groups of threes.__
> Escape from New York (1981)

Oh, groups. How much do I love groups? Let's see, definitely less than pizza, but after all that's an easy win. Well, culinary comparisons aside, I consider groups one of the best features of regular expressions, both for their expressive power and for the fact that they introduce a certain tool that is usually provided by programming languages only: variables.

Let's start considering why groups are important. Generally speaking groups allow you to isolate specific parts of the regular expression and reuse them later, but the way you can reuse them depends on the tool that you are using. Groups are particularly useful with `sed`, as they greatly improve the search and replace syntax.

The following code uses `sed` to parse a regular expression (enabled by the option `-r`), isolate a group with the parenthesis and use the matched pattern in the replacement string.

```
$ echo "10:30" | sed -r s,"([0-9]{2}).*","H:\1",
H:10
```

The regular expression tries to match 2 adjacent digits (`[0-9]{2}`), but putting them in a group with the parenthesis makes `sed` store the value in a variable. In this case the value stored is `10`, and the rest is ignored by the `.*` that matches anything. As you can see the way `sed` uses to access the value of groups stored during the search is a backslash followed by a number. The first (and only, in this case) group is named `\1`, the second `\2`, and so on. So, the replacement pattern is a string beginning with `H:` and followed by the value matched and stored during the search, that is `10`.

Let's try something a bit more complex, with two groups.

```
$ echo "10:30" | sed -r s,"([0-9]{2}):([0-9]{2})","H:\1 - M:\2",
H:10 - M:30
```

This time the regular expression looks for two digits in the first group, then for a colon, and then for another group of two digits. The two groups are named `\1` and `\2`, so they can be used in the replacement string. Please note that the colon is not included in any group as we don't need to store it, it is just an anchor to separate the two parts of the time.

Groups can include other groups, but it is important to note that the outer group still matches the whole expression. An example can clarify the matter. If you run

```
$ echo "alpha10" | sed -r s,"([a-z]+([0-9]+))","Full code:\1 - Number:\2",
Full code:alpha10 - Number:10
```

you will realise that the inner group `([0-9]+)` matches the digits `10` but the outer group matched both the letters and the digits, basically acting as it was `([a-z]+[0-9]+)`, without the internal group.

When you use groups in the replacement string you are not forced to keep them in order, so for example

```
$ echo "First,Second" | sed -r s/"(.*),(.*)"/"\2,\1"/
Second,First
```

is a very simple way to swap two fields in a comma-separated string. Well, maybe you don't think it is that simple, let's review it together. First of all I used a `/` to separate the search and replacement strings because the search string will contain a comma. The first group matches anything but up to the first comma, after which the second group captures the rest. In the replacement string I print the content of the second group, then a comma, and the content of the first group.

We can use groups in `grep` as well, even though the reuse of the matching values is obviously limited by the fact that `grep` doesn't replace text. The best use of groups in `grep` is with the so-called lookaround expressions, that are provided by the Perl-compatible regular expression syntax (PCRE), activated by the `-P` switch, as opposed to the `-E` that we used so far, that activates the Extended syntax (ERE). The differences between these syntaxes are outside the scope of this book, so feel free to investigate the matter online.

Lookaround expressions can provide information about the surroundings of a matching pattern without including the surroundings themselves. Let's look at an example: the simplest form of lookaround is the positive lookahead, where a group specifies what should follow the matching part of the string

```
$ cat examples.txt | grep -P "[A-Za-z ]+(?=[0-9]+)"
Police 101
H2O
R2-D2
Johnny 5
Cyborg 009
```

Here, the matching regular expression is `[A-Za-z ]+`, so strings of lowercase letters, uppercase letters, and spaces. We are however interested only in those strings that are also followed by one or more digits `(?=[0-9]+)`. The effect is clear if you have a coloured output on your shell, or if you use the `-o` option

```
$ cat examples.txt | grep -oP "[A-Za-z ]+(?=[0-9]+)"
Police
H
R
D
Johnny
Cyborg
```

The immediate evolution of the positive lookahead is the negative lookahead, which is expressed by `?!`. You can start to see a pattern here (apt, speaking of regular expressions). Lookaround groups are introduced by a `?` and followed by a criteria, which can be equality (`=`) or inequality (`!`).

You can also use lookbehind expressions, which start with a `?<` instead of starting with a simple `?`. These expressions match patterns that follow the lookbehind group, for example

```
$ cat examples.txt | grep -oP "(?<=[A-Z])[a-z]+"
ug
og
olice
ohnny
pider
an
yborg
ig
ad
olf
ony
ictures
```

That matches all the lowercase letters that follow an uppercase one, without including the latter. The lookbehind expression is `(?<=[A-Z])`.

A warning: lookaround expressions are often difficult to manage, and their behaviour can be surprising because it strongly depends on the implementation of the engine. You won't hit such complex cases now that you just learned how to use groups, but it might happen in the future. For the time being, please keep in mind that there are important things to learn about regular expression engines, such as if they are greedy or not. This book wants to be a primer, so I will simply pretend those issues do not exist, but remember that there is a lot to learn out there!

Back-references are actually supported by `grep`, but their behaviour can be surprising. The code

```
$ cat examples.txt | grep -E "[A-Z]([0-9])-[A-Z]\1"
R2-D2
```

matches an uppercase letter followed by a digit (grouped), a dash, another uppercase letter, and the same digit that matched in the group. A string like `R2-D3` would not match this regular expression, because the digit 3 doesn't match the previous digit. Or maybe it won't match because the processing engine has a Star Wars lore checker inside!

A final interesting feature of groups is that they allow to use the `OR` logical operator at a local level. Let's first have a look at the operator in a standard regular expression without groups. Having read so far, you clearly understand the following regular expressions

```
$ cat examples.txt | grep -E "^o"
ostrich
ogre
$ cat examples.txt | grep -E "a$"
gorilla
```

The first one matches the lines beginning with the letter `o`, while the second one matches those ending with `a`. You can match both at the same time with the logical OR represented by a pipe `|`

```
$ cat examples.txt | grep -E "^o|a$"
ostrich
gorilla
ogre
```

Don't be confused by the use of the pipe symbol. In a regular expression this character doesn't have the meaning it has on the command line, that is to connect commands, it just represents a logical OR. It is a powerful tool, as it allows you to run multiple unrelated regular expressions at the same time, without forcing you to split them into several executions of `grep` or any other tool.

So far, though, the operator can only separate two whole expressions. Groups allow you to use the logical OR at a local level, as you can see in this example

```
$ cat examples.txt | grep -E "[A-Z]([a-z]|[0-9]-)"
Dug the Dog
Police 101
R2-D2
Johnny 5
Spider-Man [*]
Cyborg 009
Big Bad Wolf
* TM Sony Pictures
```

The regular expression matches an uppercase letter, followed by either a lowercase letter (`Du`, `Cy`) or a digit and a dash (`R2-`). As I want the dash to follow only the digit, if present, this condition would

be impossible to express in a single run without the OR operator, and without the support of a group I would have to repeat part of the expression twice. Always remember that duplicated code is evil, even in regular expressions.

Exercises

Exercises! Groups can be complex sometimes, so testing your knowledge can't but be beneficial. Remember that you can always write regular expressions in an incremental way, adding pieces and testing what they do (maybe adding the occasional `head` to avoid getting too many output lines)

Exercise 17.01

Extract all the lines of `simple.log` that contain an HTTP method `GET` or `POST`, rewrite each line in the form `<time> <HTTP status> <HTTP method>`. The result for the first 10 lines should be

```
10:05:03 200 GET
10:05:43 200 GET
10:05:47 200 GET
10:05:12 200 GET
10:05:07 200 GET
10:05:34 200 GET
10:05:57 200 GET
10:05:50 200 GET
10:05:24 200 GET
10:05:50 200 GET
```

Go to solution

Exercise 17.02

The file `simple.log` contains lines with requests concerning files like

```
83.149.9.216 [17/May/2015:10:05:03 GET /presentations/logstash-monitorama-2013/image\
s/kibana-search.png HTTP/1.1 200 203023 http://semicomplete.com/presentations/logsta\
sh-monitorama-2013/
```

Extract a list of all file extensions and count them. Assume that extensions are made of lowercase letters only.

Go to solution

Exercise 17.03

There are three lines in the file `simple.log` where a request received an HTTP 500 status code, for example

```
66.249.73.135 [18/May/2015:15:05:42 GET /misc/Title.php.txt HTTP/1.1 500 - -
```

Find them and print the IP addresses of each client in a single comma-separated line (i.e. `<IP number 1>,<IP number 2>,<IP number 3>`)

Go to solution

I think you should be proud of yourself, you made it so far and you are still alive. Are you? Hello? Just joking, I'm pretty sure you followed along and that the exercises made you appreciate groups even more. This chapter concludes our journey in the land of regular expressions, it's time to get into bash scripting. I warmly recommend to stop here for now, grab a drink, enjoy a walk, a game, something relaxing. See you in the next part of the book!

Suggested film for the evening: The Avengers (2012) - Groups are powerful, and when your Unix system is attacked by aliens from the outer space you need the most powerful one.

Part 3 - The Unix file system

TODO

This part of the book is still in the works.

Part 4 - Scripting

TODO

This part of the book is still in the works.

Appendix 1 - Solutions to exercises

Day 1 - Getting help

Solutions to exercises

Exercise 1.01

Enter the man page for the `echo` command and locate the `AUTHOR` (actually the authors).

Solution

Run the command

```
$ man echo
```

Inside the man page press `/` (slash) and type `AUTHOR`, then hit Enter. You should have found that `echo` has been written by Brian Fox and Chet Ramey. Now silently say thanks to then for the time they spent creating these utilities. They did it for free, so you should be grateful.

Go back to the exercise

Exercise 1.02

Enter the man page for the `sort` command (we haven't used it yet, but the man page is there). Enter the online help and locate how to `Undo (toggle) search highlighting` with the search pattern command. Now exit the help, search for `ignore` and practise the search highlighting toggle command.

Solution

Run the command

```
$ man sort
```

Inside the man page press `h` for the online help, then press `/` to search for a pattern, type `Undo` and press Enter. The first (and actually only) occurrence is

```
ESC-u                Undo (toggle) search highlighting.
```

Now exit the help with `q`, hit `/`, type `ignore` and Enter. This highlights the occurrences. Now press ESC and then `u`. You don't need to keep ESC pressed, just hit ESC and then hit `u`, and the highlighting should disappear. Press `ESC-u` again and the highlighting should reappear. Now exit the help with `q`.

Go back to the exercise

Day 2 - Printing

Solutions to exercises

Exercise 2.01

Print the string “Just a test”

Solution

```
$ echo "Just a test"
Just a test
```

As we discussed in the chapter this can also be solved running

```
$ echo Just a test
Just a test
```

but I believe the first solution is better. It is immediately clear which part of the command is the string and later, when we will create scripts in an editor, strings between quotes are highlighted.

Go back to the exercise

Exercise 2.02

Print a string without the trailing newline (check the manual page)

Solution

Open the man page for `echo` and search for the pattern `trailing newline`. You should find that

```
-n     do not output the trailing newline
```

So, if you run

```
$ echo -n "Just a string"
Just a string$
```

You should get the string immediately followed by the prompt, as you can see above. This is useful when you want to give the user some feedback on a running process like a for loop, and you want to print a single character like for example a dot. Inside the loop you want to print those characters without the newline to keep then on the same output line.

Go back to the exercise

Exercise 2.03

Run

```
$ echo "First line\nSecond line"
```

What happens? Can you find a way to convert that `\n` into a newline?

Solution

When you run the given command you get as an output the literal string `First line\nSecond line`. The `\n` sequence is used in Unix systems to indicate a newline, and apparently `echo` doesn't understand it out of the box. This is because, by default `echo` ignores *backslash escapes* like `\n`. In the programming world, an *escape* is a way to avoid the default interpretation of some character and to signal the language that we give it a special meaning. In this case, if echo encounters an `n`, it would just print out the character `n`, while we want to use it in a special way.

Long story short, if you search for `backslash` in the man page of `echo`, you will find

```
The man page says `-e     enable interpretation of backslash escapes`, so
```

which means that if you run

```
$ echo -e "First line\nSecond line"
First line
Second line
```

you will get the desired result.

Go back to the exercise

Day 3 - Showing the content of a file

Solutions to exercises

Exercise 3.01

Print the content of the file `examples.txt` numbering lines.

Solution

According to the man page of `cat` the `-n` option numbers all output lines. So

```
$ cat -n examples.txt
     1  dog
     2  cat
     3  elephant
     4  ostrich
     5  Dug the Dog
[...]
```

gives the desired result.

Go back to the exercise

Exercise 3.02

Check if the file `examples.txt` contains spaces at the end of any line.

Solution

According to the man page of `cat` the `-E` option displays `$` at the end of each line. So

```
$ cat -E examples.txt
dog$
cat$
elephant$
ostrich$
Dug the Dog$
[...]
```

shows you that each line terminates with a `$` immediately after the text, so there are no nasty "invisible" spaces. Later in the book we will learn how to check this without having to manually look at each line.

Go back to the exercise

Day 4 - Beginnings and ends

Solutions to exercises

Exercise 4.01

Show the first 3 entries of `slices.txt`

Solution

```
$ head -n3 slices.txt
01 ########################################
02 ########################################
03 ########################################
```

Go back to the exercise

Exercise 4.02

Show the last 3 entries of `slices.txt`

Solution

```
$ tail -n3 slices.txt
18 ########################################
19 ########################################
20 ########################################
```

Go back to the exercise

Exercise 4.03

Show the content of `slices.txt` skipping the last 3 lines

Solution

```
$ head -n -3 slices.txt
01 ########################################
02 ########################################
03 ########################################
04 ########################################
[...]
14 ########################################
15 ########################################
16 ########################################
17 ########################################
```

Go back to the exercise

Exercise 4.04

Show the content of `slices.txt` starting from line 3 (that is, skipping the first 2 lines)

Solution

```
$ tail -n +3 slices.txt
03 ########################################
04 ########################################
05 ########################################
06 ########################################
[...]
17 ########################################
18 ########################################
19 ########################################
20 ########################################
```

Go back to the exercise

Day 7 - Sequences and counting

Solutions to exercises

Exercise 7.01

Print the numbers from 5 to 15 padded with enough zeros to fill 5 digits, i.e. `00005`, `00006`, `00007`, and so on.

Solution

```
$ seq -w 5 10000 | head -n 11
00005
00006
00007
00008
00009
00010
00011
00012
00013
00014
00015
```

Go back to the exercise

Exercise 7.02

Count the number of lines of the file simple.log

Solution

```
$ cat simple.log | wc -l
10000
```

Go back to the exercise

Day 9 - Fields

Solutions to exercises

Exercise 9.01

Print fields 1,6,7, and 8 of the file `simple.log`

Solution

You can either use `cut` directly

```
$ cut simple.log -d " " -f 1,6-8
83.149.9.216 200 203023 http://semicomplete.com/presentations/logstash-monitorama-20\
13/
83.149.9.216 200 171717 http://semicomplete.com/presentations/logstash-monitorama-20\
13/
83.149.9.216 200 26185 http://semicomplete.com/presentations/logstash-monitorama-201\
3/
[...]
```

or use `cat` and a pipe

```
$ cat simple.log | cut -d " " -f 1,6-8
83.149.9.216 200 203023 http://semicomplete.com/presentations/logstash-monitorama-20\
13/
83.149.9.216 200 171717 http://semicomplete.com/presentations/logstash-monitorama-20\
13/
83.149.9.216 200 26185 http://semicomplete.com/presentations/logstash-monitorama-201\
3/
[...]
```

Go back to the exercise

Exercise 9.02

Extract the time of each request as HH:MM:SS

Solution

There are several ways to achieve this result. Two possible ones are

```
$ cat simple.log | cut -d ":" -f 2-4 | cut -d "]" -f 1
10:05:03
10:05:43
10:05:47
[...]
```

and

```
$ cat simple.log | cut -d "]" -f 1 | cut -d ":" -f 2-4
10:05:03
10:05:43
10:05:47
```

In the first one I first cut using the colon and then we remove the part separated by a closing bracket, while in the second one I do the opposite, first selecting the first column separated by a closing bracket and then removing the first part separated by colon.

Go back to the exercise

Day 10 - Sorting and reducing

Solutions to exercises

Exercise 10.01

Print the 5 more frequent IP addresses in the file simple.log

Solution

```
$ cut -d " " -f 1 simple.log | sort | uniq -c | sort -nr | head -n 5
    482 66.249.73.135
    364 46.105.14.53
    357 130.237.218.86
    273 75.97.9.59
    113 50.16.19.13
```

Go back to the exercise

Exercise 10.02

Print the HTTP methods used by the requests in the file simple.log and count how many occurrences are there for each one.

Solution

```
$ cut -d " " -f 3 simple.log | sort | uniq -c
   9952 GET
     42 HEAD
      1 OPTIONS
      5 POST
```

Go back to the exercise

Day 11 - Find and replace text

Solutions to exercises

Exercise 11.01

Using `grep` and `wc`, count how many times the file `examples.txt` contains the word `dog`

Solution

```
$ grep "dog" examples.txt | wc -l
3
```

Go back to the exercise

Exercise 11.02

Print all the lines of `examples.txt` that do not contain either a lowercase or an uppercase `h`

Solution

```
$ grep -vi "h" examples.txt
dog
cat
dryad
dog
Police 101
aardvark
[...]
```

The lesson doesn't tell you how to reverse-match the given string, so you need to read the man page for `grep`, which says

```
       -v, --invert-match
              Invert the sense of matching, to select non-matching lines.
```

while the `-i` or `--ignore-case` option was already shown in the chapter.

Go back to the exercise

Exercise 11.03

Replace all letters `e` in the file `examples.txt` with a question mark `?`, then find among the resulting lines all the ones that have a space in them

Solution

```
$ cat examples.txt | sed s,"e","?",g | grep " "
Dug th? Dog
Polic? 101
corn dog
phas? spid?r
und?ad r?d dragon
[...]
```

There are no specific issues in using question mark and spaces, as long as you use quotes to surround the latter.

Go back to the exercise

Day 12 - Regular expressions - Single characters

Solutions to exercises

Exercise 12.01

Match “dog”, “Dog”, and “hog” into examples.txt

Solution

```
$ grep -E ".og" examples.txt
dog
Dug the Dog
dog
corn dog
hogwash
wild hog
hog
```

Go back to the exercise

Exercise 12.02

Log entries in the file `simple.log` contain the string `HTTP/<version> <code>`, where `<version> is the version of the HTTP protocol in use (either 1.0 or 1.1) and <code>` is the three-digits HTTP request status code. Extract all lines with a status “4xx” (that is a status between 400 and 499). Count how often each status occurs.

Solution

The `HTTP/1.0` or `HTTP/1.1` strings can be used as anchors to identify the three digits we are interested in, but then we want to get rid of them to aggregate the HTTP status codes only

```
$ grep -Eo "HTTP/.\.. 4.." simple.log  | sed -r s,"HTTP/.\.. ",, | sort | uniq -c
      2 403
    213 404
      2 416
```

If we don't get rid of the protocol version we get a different histogram, which can be interesting anyway

```
$ grep -Eo "HTTP/.\.. 4.." simple.log  | sort | uniq -c
      1 HTTP/1.0 403
     30 HTTP/1.0 404
      1 HTTP/1.1 403
    183 HTTP/1.1 404
      2 HTTP/1.1 416
```

Go back to the exercise

Day 14 - Regular expressions - Classes

Solutions to exercises

Exercise 14.07

Match any line of examples.txt containing a digit

Solution

```
$ cat examples.txt | grep "[0-9]"
Police 101
H2O
007
R2-D2
Johnny 5
Cyborg 009
HTTP/1.1
C-3PO
```

Go back to the exercise

Exercise 14.08

Match any line of examples.txt containing a lowercase “a” followed by any letter (that is “aa”, “ab”, “ac”, and so on)

Solution

```
$ cat examples.txt | grep "a[a-z]"
cat
elephant
dryad
aardvark
phase spider
manticore
undead red dragon
Spider-Man [*]
cat
basilisk
hogwash
cat
Big Bad Wolf
```

Go back to the exercise

Exercise 14.09

Match any line of examples.txt containing an upper case letter followed by a digit

Solution

```
$ cat examples.txt | grep "[A-Z][0-9]"
H2O
R2-D2
```

Go back to the exercise

Exercise 14.10

Match any line of examples.txt containing a dash

Solution

```
$ cat examples.txt | grep -E "[-]"
R2-D2
Spider-Man [*]
C-3PO
```

Go back to the exercise

Exercise 14.11

Match any line of examples.txt containing a left square bracket "["

Solution

```
$ cat examples.txt | grep -E "[[]"
Spider-Man [*]
```

Go back to the exercise

Day 15 - Regular expressions - Anchors

Solutions to exercises

Exercise 15.01

Match every line of examples.txt that ends with an upper case letter and a number (in this order)

Solution

```
$ cat examples.txt | grep "[A-Z][0-9]$"
R2-D2
```

Go back to the exercise

Exercise 15.02

Count how many empty lines are contained in the file examples.txt

Solution

```
$ cat examples.txt | grep "^$" | wc -l
1
```

Go back to the exercise

Day 16 - Regular expressions - Multiple matches

Solutions to exercises

Exercise 16.01

Match all the occurrences of exactly three digits in the file examples.txt

Solution

```
$ grep -E "[0-9]{3}" examples.txt
Police 101
007
Cyborg 009
```

Go back to the exercise

Exercise 16.02

The log file simple.log contains the status of HTTP responses, which is a three digit number preceded and followed by a space. HTTP statuses are categorised according to the initial digit and 4xx responses (that is 400, 401, 402, and so on) are considered resource errors. Count how many HTTP 4xx responses are in the file for each type of error.

Solution

```
$ grep -Eo " 4[0-9]{2} " simple.log | sort | uniq -c
      2  403
    213  404
      2  416
      1  481
```

Go back to the exercise

Exercise 16.03

The log file simple.log contains the IP address of the client for each request. IP addresses are made of four numbers separated by dots (i.e. `A.B.C.D`), where each number goes from 0 to 255 (thus having from 1 to 3 digits). Find the 5 IP addresses that occur the highest number of times in the file, counting them

Solution

```
$ grep -Eo "[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}" simple.log | sort | uniq\
 -c | sort -nr | head -n 5
    482 66.249.73.135
    364 46.105.14.53
    357 130.237.218.86
    273 75.97.9.59
    113 50.16.19.13
```

The regular expressions is made of four repetitions of `[0-9]{1,3}`, which matches 1 to 3 adjacent digits, separated by `\.` which is a literal dot (remember that a dot without the escape backslash matches any character). The following `sort` and `uniq -c` provide the counting, the last `sort -nr` orders the list again using the numerical sort (which orders according to the count, as this is at the beginning of each line), and in reverse order, starting from bigger numbers down to 1. The last `head -n5` at last selects the top 5 from the list.

Go back to the exercise

Exercise 16.04

The file simple.log contains the HTTP method used in the request (for example `GET` or `POST`) followed by a space and the rest of the log line. For each request that uses a `GET` print the HTTP method and everything that follows.

Solution

```
$ grep -Eo "GET .*" simple.log
GET /presentations/logstash-monitorama-2013/images/kibana-search.png HTTP/1.1 200 20\
3023 http://semicomplete.com/presentations/logstash-monitorama-2013/
GET /presentations/logstash-monitorama-2013/images/kibana-dashboard3.png HTTP/1.1 20\
0 171717 http://semicomplete.com/presentations/logstash-monitorama-2013/
GET /presentations/logstash-monitorama-2013/plugin/highlight/highlight.js HTTP/1.1 2\
00 26185 http://semicomplete.com/presentations/logstash-monitorama-2013/
[...]
```

The pattern `.*` is one of the most common ones in regular expressions. As `.` matches any character and `*` matches 0 or more repetitions of the previous component, `.*` means any repetition of any character, or "anything" for short. This is extremely useful whenever there are long parts of a line that you want to manage without writing a (probably very complex and long) regular expressions that matches them.

Go back to the exercise

Day 17 - Regular expressions - Groups

Solutions to exercises

Exercise 17.01

Extract all the lines of `simple.log` that contain an HTTP method `GET` or `POST`, rewrite each line in the form `<time> <HTTP status> <HTTP method>`. The result for the first 10 lines should be

```
10:05:03 200 GET
10:05:43 200 GET
10:05:47 200 GET
10:05:12 200 GET
10:05:07 200 GET
10:05:34 200 GET
10:05:57 200 GET
10:05:50 200 GET
10:05:24 200 GET
10:05:50 200 GET
```

Solution

```
$ head simple.log | grep -E " (GET|POST) " | sed -r s,".*[0-9]{4}:(.*)] (GET|POST).*\
HTTP/1.[01] ([0-9]{3}).*","\1 \3 \2",
10:05:03 200 GET
10:05:43 200 GET
10:05:47 200 GET
10:05:12 200 GET
10:05:07 200 GET
10:05:34 200 GET
10:05:57 200 GET
10:05:50 200 GET
10:05:24 200 GET
10:05:50 200 GET
```

The idea behind this solution it to find all the lines that contain `GET` or `POST` using the logical OR in a group, so that either can be surrounded by spaces, which helps avoiding other mentions of those letters (like for example a line with an URL that contains “BUDGET” or “POSTER”). Then

I decompose the rest of the line using the year (`[0-9]{4}:`) as an anchor. Mentioning the closing square bracket `]` outside the group makes it disappear from the output. The HTTP protocol can be either `HTTP/1.0` or `HTTP/1.1`, and the HTTP status is always a three digits number (`[0-9]{3}`).

Go back to the exercise

Exercise 17.02

The file `simple.log` contains lines with requests concerning files like

```
83.149.9.216 [17/May/2015:10:05:03 GET /presentations/logstash-monitorama-2013/image\
s/kibana-search.png HTTP/1.1 200 203023 http://semicomplete.com/presentations/logsta\
sh-monitorama-2013/
```

Extract a list of all file extensions and count them. Assume that extensions are made of lowercase letters only.

Solution

There are many ways to solve this exercise. One possible solution is to use lookaround expressions with `grep` to isolate the file path and later the file extension.

```
$ cat simple.log | grep -Po "(?<=GET )\/.*(?= HTTP)" | grep -Po "(?<=\.)[a-z]+$" | s\
ort | uniq -c
      4 access
     10 c
     14 conf
      1 cpp
   1458 css
      4 deb
[...]
```

Go back to the exercise

Exercise 17.03

There are three lines in the file `simple.log` where a request received an HTTP 500 status code, for example

```
66.249.73.135 [18/May/2015:15:05:42 GET /misc/Title.php.txt HTTP/1.1 500 - -
```

Find them and print the IP addresses of each client in a single comma-separated line (i.e. `<IP number 1>,<IP number 2>,<IP number 3>`)

Solution

The exercise can be solved using `grep`, sed `and` xargs‘

```
$ grep -E "HTTP/1.1 500" simple.log | sed -r s,"([0-9.]+) .*","\1", | xargs echo | s\
ed s/" "/","/g
66.249.73.135,66.249.73.135,64.131.102.243
```

The initial `grep` uses the `HTTP\1.1 500` mentioned in the text of the exercise to find the relevant lines. Then, using sed we can extract the IP addresses, put them on a single line with `xargs` and replacing the space used as a separator with a comma.

Instead of `sed` we could have used `grep`

```
$ grep -E "HTTP/1.1 500" simple.log | grep -Eo  "^[0-9.]+" | xargs echo | sed s/" "/\
","/g
66.249.73.135,66.249.73.135,64.131.102.243
```

counting on the fact that the space following the IP address is not matched by the class. A common replacement for the `xargs | sed` pattern is the `paste` command (that wasn't introduced in the book)

```
$ grep -E "HTTP/1.1 500" simple.log | grep -Eo  "^[0-9.]+" | paste -s -d,
66.249.73.135,66.249.73.135,64.131.102.243
```

Make sure you read the man page of the `paste` command, it's pretty simple and useful to have in your toolkit.

Go back to the exercise

Part 3 - Appendices

Changelog

> – *The security code has been changed.*
> The Empire Strikes Back (1980)

I will track here changes between releases of the book, following Semantic Versioning[10]. A change in the **major** number means an incompatible change, that is a big rewrite of the book, also known as 2nd edition, 3rd edition, and so on. I don't know if this will ever happen, but the version number comes for free. A change in the **minor** number means that something important was added to the content, like a new section or chapter. A change in the **patch** number signals minor fixes like typos in the text or the code, rewording of sentences, and so on.

The book is not finished yet, but I thought the amount of chapters I wrote was enough to give readers useful information, so I decided to publish it. The plan is to add at least 2 more parts, one on the Unix filesystem and one on bash scripting.

Current version: 1.0.2

Version 1.0.0 (2019-12-25)

- Initial release

Version 1.0.1 (2019-12-25)

- Emil Jaregran (https://github.com/emiljaregran) fixed some typos and a mistake in exercise 9.02

Version 1.0.1 (2020-05-13)

GitHub users Sundeep Agarwal[11], Vatsalya Gupta[12], and Adithya Venkateswaran[13] spotted several typos and fixed them. Thanks!

[10] https://semver.org/
[11] https://github.com/learnbyexample
[12] https://github.com/vatsalya-gupta
[13] https://github.com/MajorCarrot