

The kaobook class

Use this document as a template

Example and documentation of the kaobook class

Customise this page according to your needs

Federico Marotta *

August 13, 2019

An Awesome Publisher

* A L^AT_EX lover

The kaobook class

Disclaimer

You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

No copyright

© This book is released into the public domain using the CC0 code. To the extent possible under law, I waive all copyright and related or neighbouring rights to this work.

To view a copy of the CC0 code, visit:

<http://creativecommons.org/publicdomain/zero/1.0/>

Colophon

This document was typeset with the help of KOMA-Script and L^AT_EX using the kaobook class.

The source code of this book is available at:

<https://github.com/fmarotta/kaobook>

(You are welcome to contribute!)

Publisher

First printed in May 2019 by An Awesome Publisher

The harmony of the world is made manifest in Form and
Number, and the heart and soul and all the poetry of
Natural Philosophy are embodied in the concept of
mathematical beauty.

– D'Arcy Wentworth Thompson

Detailed Contents

Detailed Contents	v
1 Introduction	1
1.1 What is Unix?	1
1.2 Why Linux? Windows worked fine so far...	1
2 First Lecture	3
2.1 Hello terminal!	3
2.2 Navigating the command line	4
2.3 Creation and inspection	6
2.4 Migration and Destruction	9
2.5 Self Help	11
2.6 Get Wild	12
2.7 Regular Expressions	13
2.8 Metacharacters	14
2.9 Characters Set	15
 APPENDIX	 17
Notation	19
Alphabetical Index	21

Figures

Tables

1 Introduction

1.1 What is Unix?

Unix is an operating system and a set of tools. The tool we will be using the most in this book is a shell, which is a computer program that provides a command line interface. You have probably seen a command line interface in the movies: an elite computer hacker sits in front of a black screen with green glowing text, furiously typing in commands and shouting something like "Spike them!" Using the command line interface lets you enter lines of code into a shell (also called a console) and that code instructs your computer to perform a specific task. Throughout this book I may use the terms command line, shell, and console interchangeably. You will learn about using the command line in the Command Line Basics chapter.

The shell is a very direct and powerful way to manipulate a computer. You can produce wonderful creations that help thousands of people, or you can wreak havoc on yourself and on others. Like said: "With great power comes great responsibility."

There are several popular shell programs but in this book we'll be using a shell called Bash because it is the default shell program on Mac and Ubuntu.

1.2 Why Linux? Windows worked fine so far...

Some motivations here...

2 First Lecture

2.1 Hello terminal!

Once you have opened up Terminal then you should see a window that looks something like this:

Put here image of the terminal

What you're looking at is the bash shell! Your shell will surely look different than mine, but all bash shells have the same essential parts. As you can see in my shell it says seans-air: sean\$.

This string of characters is called the prompt. You type command line commands after the prompt. The prompt is just there to let you know that the shell is ready for you to type in a command. Press Enter on your keyboard a few times to see what happens with the prompt. Your shell should now look like this:

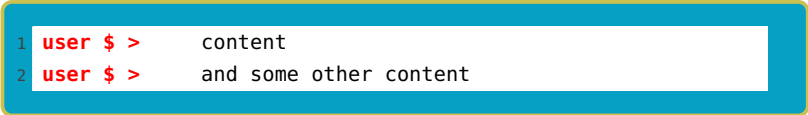
put here another image

If you don't type anything after the prompt and you press enter then nothing happens and you get a new prompt under the old prompt. The white rectangle after the prompt is just a cursor that allows you to edit what you've typed into the shell. Your cursor might look like a rectangle, a line, or an underscore, but all cursors behave the same way. After typing something into the command line you can move the cursor back and forth with the left and right arrow keys.

Don't you think your shell looks messy with all of those old prompts? Don't worry, you're about to learn your first shell command which will clear up your shell! Type clear at the prompt and then hit enter. Voila! Your shell is back to how you started.

Every command line command is actually a little computer program, even commands as simple as clear. These commands all tend to have the following structure:

```
1 | print('print me')
```



```
1 user $ > content
2 user $ > and some other content
```

Some simple commands like clear don't require any options or arguments. Options are usually preceded by a hyphen (-) and they tweak the behavior of the command. Arguments can be names of files, raw data, or other options that the command requires. A simple command that has an argument is echo. The echo command prints a phrase to the console. Enter echo "Hello World!" into the command line to see what happens:

We'll be using the above syntax for the rest of the book, where on one line there will be a command that I've entered into the command line, and then below that command the console output of the command will appear (if there is any console output). You can use `echo` to print any phrase surrounded by double quotes (") to the console.

If you want to see the last command press the Up arrow key. You can press Up and Down in order to scroll through the history of commands that you've entered. If you want to re-execute a past command, you can scroll to that command then press Enter. Try getting back to the `echo "Hello World!"` command and execute it again.

Summary

You type command line commands after the prompt. `clear` will clean up your terminal. `echo` prints text to your terminal. You can scroll through your command history with the Up and Down arrow keys.

Exercises

Print your name to the terminal. Clear your terminal after completing #1.

2.2 Navigating the command line

As you can see in the image below, my Debussy directory is contained in my Music directory. This is the simplest case of how directories are structured.

PICTURES!

The directory structure on most computers is much more complicated, but the structure on your computer probably looks something like this:

pictures!

There are a few special directories that you should be aware of on your computer. The directory at the top of this tree is called the root directory. The root directory contains all other directories, and is represented by a slash (/).

The home directory is another special directory that is represented by a tilde (~). Your home directory contains your personal files, like your photos, documents, and the contents of your desktop. When you first open up your shell you usually start off in your home directory. Imagine tracing all of the directories from your root directory to the directory you're currently in. This sequence of directories is called a path. The diagram below illustrates the path from a hypothetical root directory to the home directory.

picture!

This path can be written as `/Users/sean`.

Open the command line if you closed it. Your shell starts in your home directory. Whatever directory your shell is in is called the working

directory. Enter the `pwd` command into your shell to print the working directory.

You can change your working directory using the `cd` command. If you use the `cd` command without any arguments then your working directory is changed to your home directory.

Enter `cd` into the command line and then enter `pwd`.

You were in your working directory to start, and by entering `cd` into the command line you did technically change directory, you just changed it to your home directory (the directory you were in to begin with). To use `cd` to change your working directory to a directory other than your home directory, you need to provide `cd` with the path to another directory as an argument. You can specify a path as either a path that is relative to your current directory, or you can specify the absolute path to a directory starting from the root of your computer. Let's say we simply want to change the working directory to one of the folders that is inside our home directory. First we need to be able to see which folders are in our working directory. You can list the files and folders in a directory using the `ls` command. Let's use the `ls` command in our home directory to list the files and folders contained within it.

It looks like I have four folders and one text file in my home directory. Now let's switch into the Music directory:

As you can see the path to the current working directory has changed:

I specified a relative path when I entered `cd Music`. The path to the Music directory is just `Music/` relative to my previous working directory. I can go back to `/Users/sean/` with the command `cd ..` which changes the working directory to the folder above the current working directory:

Notice that `..` is also a relative path, since it specifies the directory above your current working directory. Similarly `.` is the path to your current working directory. Therefore since my current working directory is `/Users/sean` then `cd Music` is the same as `cd ./Music`.

I can `cd` to any folder as long as I know the absolute path to that folder. For example I can `cd` to `/Users/sean/Music` by entering the following into the shell:

It doesn't matter what directory I'm in since I'm using an absolute path, I can jump straight to that directory (Remember that `~` is a shortcut for the path to your home folder). Of course you shouldn't expect yourself to have every absolute path on your computer memorized! You can use a terminal feature called tab completion in order to speed up typing paths and other commands. Enter the following into your shell, and then try pressing the Tab key (on some machines you need to press it twice):

(press Tab)

Pressing tab shows you a list of all files and folders inside of the `/` directory. Now I'm going to type `/D` into my terminal and you can see what happens when I press tab again:

(press Tab)

Since I added a “D” to the path, only folders with names that start with a “D” are listed. If I type `cd /De` into the console and then press Tab then the command will autocomplete to `cd /Desktop/`. If I press tab again, the console will list all of the files and folders on my desktop.

Make sure to pause and try this yourself in your own terminal! You won’t have the same files or folders that I do, but you should try using `cd` and tab completion with directories and files that start with the same letters.

Summary

You can identify a specific file or folder by its path. The root directory (`/`) contains all of the folders and files on your computer. Your home directory (`~`) is the directory where your terminal always starts. Use the `cd` command to change your working directory. The `pwd` command will print the working directory. The `ls` command will list files and folders in a directory.

Exercises

Set your working directory to the root directory. Set your working directory to your home directory using three different commands. Find a folder on your computer using your file and folder browser, and then set your working directory to that folder using the terminal. List all of the files and folders in the directory you navigated to in #3.

2.3 Creation and inspection

Now that you can fluidly use your terminal to bound between directories all over your computer I’ll show you some actions you can perform on folders and files. One of the first actions you’ll probably want to take when opening up a fresh terminal is to create a new folder or file. You can make a directory with the `mkdir` command, followed by the path to the new directory. First let’s look at the contents of my home directory:

I want to create a new directory to store some code files I’m going to write later, so I’ll use `mkdir` to create a new directory called Code:

It worked! Notice that the argument Code to `mkdir` is a relative path, however I could have specified an absolute path. In general you should expect Unix tools that take paths as arguments to accept both relative and absolute paths.

There are a few different ways to create a new file on the command line. The most simple way to create a blank file is to use the `touch` command, followed by the path to the file you want to create. In this example I’m going to create a new journal entry using `touch`:

A new file has been created! I’ve been using `ls` to list the files and folders in the current directory, but using `ls` alone doesn’t differentiate

between which of the listed items are folders and which are files. Thankfully you can use the `-l` option with `ls` in order to get a long listing of files in a directory.

There is a row in the resulting table for each file or folder. If the entry in the first column is a `d`, then the row in the table corresponds to a directory, otherwise the information in the row corresponds to a file. As you can see in my home directory there are five directories and two files. The string of characters following the `d` in the case of a directory or following the first `-` in the case of a file represent the permissions for that file or directory. We'll cover permissions in a later section. The columns of this table also show who created the file, the group that the creator of the file belongs to (we'll cover groups later when we cover permissions), the size of the file, the time and date when the file was last modified, and then finally the name of the file.

Now that we've created a file there are a few different ways that we can inspect and edit this file. First let's use the `wc` command to view the word count and other information about the file:

The `wc` command displays the number of lines in a file followed by the number of words and then the number of characters. Since this file looks pretty small (only three lines) let's try printing it to the console using the `cat` command.

The `cat` command is often used to print text files to the terminal, despite the fact that it's really meant to concatenate files. You can see this concatenation in action in the following example:

The `cat` command will combine every text file that is provided as an argument.

Let's take a look at how we could view a larger file. There's a file inside the Documents directory:

Let's examine this file to see if it's reasonable to read it with `cat`:

Wow, over 1000 words! If we use `cat` on this file it's liable to take up our entire terminal. Instead of using `cat` for this large file we should use `less`, which is a program designed for viewing multi-page files. Let's try using `less`:

You can scroll up and down the file line-by-line using the up and down arrow keys, and if you want to scroll faster you can use the spacebar to go to the next page and the `b` key to go to the previous page. In order to quit `less` and go back to the prompt press the `q` key.

As you can see the `less` program is a kind of Unix tool with behavior that we haven't seen before because it "takes over" your terminal. There are a few programs like this that we'll discuss throughout this book.

There are also two easy to remember programs for glimpsing the beginning or end of a text file: `head` and `tail`. Let's quickly use `head` and `tail` on `a-tale-of-two-cities.txt`:

As you can see `head` prints the first ten lines of the file to the terminal. You can specify the number of lines printed with the `-n` option followed by the number of lines you'd like to see:

The `tail` program works exactly the same way:

We've now gone over a few tools for inspecting files, folders, and their contents including `ls`, `wc`, `cat`, `less`, `head`, and `tail`. Before the end of this section we should discuss a few more techniques for creating and also editing files. One easy way to create a file is using output redirection. Output redirection stores text that would be normally printed to the command line in a text file. You can use output redirection by typing the greater-than sign (`>`) at the end of a command followed by the name of the new file that will contain the output from the proceeding command. Let's try an example using `echo`:

Only the first command printed output to the terminal. Let's see if the second command worked:

Looks like it worked! You can also append text to the end of a file using two greater-than signs (`>>`). Let's try this feature out:

Now for a word of warning. Imagine that I want to append another line to the end of `echo-out.txt`, so typed `echo "A third line." > echo-out.txt` into the terminal when really I meant to type `echo "A third line." >> echo-out.txt` (notice I used `>` when I meant to use `>>`). Let's see what happens:

Unfortunately I have unintentionally overwritten what was already contained in `echo-out.txt`. There's no undo button in Unix so I'll have to live with this mistake. This is the first of several lessons demonstrating the damage that you should try to avoid inflicting with Unix. Make sure to take extra care when executing commands that can modify or delete a file, a typo in the command can be potentially devastating. Thankfully there are a few strategies for protecting yourself from mistakes, including managing permissions for files, and tracking versions of your files with Git, which we will discuss thoroughly in a later chapter.

Finally we should discuss how to edit text files. There are several file editors that are available for your terminal including `vim` and `emacs`. Entire books have been written about how to use both of these text editors, and if you're interested in one of them you should look for resources online about how to use them. The one text editor we will discuss using is called `nano`. Just like `less`, `nano` uses your entire terminal window. Let's edit `todo.txt` using `nano`:

Once you've started `nano` you can start editing the text file. The top line of the `nano` editor shows the file you're currently working on, and the bottom two lines show a few commands that you can use in `nano`. The carrot character (`^`) represents the Control key on your keyboard, so you can for example type Control + O in order to save the changes you've made to the text file, or Control + X in order to exit `nano` and go back to the prompt.

nano is a good editor for beginners because it works similarly to word processors you've used before. You can use the arrow keys in order to move your cursor around the file, and the rest of the keys on your keyboard work as expected. Let's add an item to my to-do list and then I'll save and exit nano by typing Control + O followed by Control + X.

Now let's quickly check if those changes were saved correctly:

You can also create new text files with nano. Instead of using an existing path to a file as the argument to nano, use a path to a file that does not yet exist and then save your changes to that file.

Summary

Use `mkdir` to create new directories. The `touch` command creates empty files. You can use `>` to redirect the output of a command into a file. `»` will append command output to the end of a file. Print a text file to the command line using `cat`. Inspect properties of a text file with `wc`. Peek at the beginning and end of a text file with `head` and `tail`. Scroll through a large text file with `less`. nano is simple text editor.

Exercises

Create a new directory called `workbench` in your home directory. Without changing directories create a file called `readme.txt` inside of `workbench`. Append the numbers 1, 2, and 3 to `readme.txt` so that each number appears on its own line. Print `readme.txt` to the command line. Use output redirection to create a new file in the `workbench` directory called `list.txt` which lists the files and folders in your home directory. Find out how many characters are in `list.txt` without opening the file or printing it to the command line.

2.4 Migration and Destruction

In this section we'll discuss moving, renaming, copying, and deleting files and folders. First let's revisit the contents of our current working directory:

It's gotten a little sloppy, so let's clean this directory up. First I want to make a new directory to store all of my journal entries in called `Journal`. We already know how to do that:

Now I want to move my journal entry `journal-2017-01-24.txt` into the `Journal` directory. We can move it using the `mv` command. `mv` takes two arguments: first the path to the file or folder that you wish to move followed by the destination folder. Let's try using `mv` now:

Looks like it worked! I just realized however that I want to move the `Journal` directory into the `Documents` folder. Thankfully we can do this with `mv` in the same way:

Let's just make sure it ended up in the right place:

Looks good! Another hidden use of the `mv` command is that you can use it to rename files and folders. The first argument is the path to the

folder or file that you want to rename, and the second argument is a path with the new name for the file or folder. Let's rename `todo.txt` so it includes today's date:

Looks like it worked nicely. Similar to the `mv` command, the `cp` command copies a file or folder from one location to another. As you can see `cp` is used exactly like `mv` when copying files, the file or folder you wish to copy is the first argument, followed by the path to the folder where you want the copy to be made:

Be aware that there is one difference between copying files and folders, when copying folders you need to specify the `-r` option, which is short for recursive. This ensures that the underlying directory structure of the directory you wish to copy remains intact. Let's try copying my Documents directory into the Desktop directory:

Finally, let's discuss how to delete files and folders with the command line. A word of extreme caution: in general I don't recommend deleting files or folders on the command line because as we've discussed before there is no undo button on the command line. If you delete a file that is critical to your computer functioning you may cause irreparable damage. I highly recommend moving files or folders to a designated trash folder and then deleting them the way you would normally delete files and folders outside of the command line (The path to the Trash Bin is `/.Trash` on Mac and `/.local/share/Trash` on Ubuntu). If you decide to delete a file or folder on your computer make absolutely sure that the command you've typed is correct before you press Enter. If you do delete a file or folder by accident stop using your computer immediately and consult with a computer professional or your IT department so they can try to recover the file.

Now that you've been warned, let's discuss `rm`, the Avada Kedavra of command line programs. When removing files `rm` only requires the path to a file in order to delete it. Let's test its destructive power on `echo-out.txt`:

I felt a great disturbance in the Force, as if millions of voices suddenly cried out in terror, and were suddenly silenced. - Obi-wan Kenobi

The file `echo-out.txt` is gone forever. Remember when we copied the entire Documents directory into Desktop? Let's get rid of that directory now. Just like when we were using `cp` the `rm` command requires you to use the `-r` option when deleting entire directories. Let's test this battle station:

Now that the awesome destructive power of `rm` is on your side, you've learned the basics of the command line! See you in the next chapter for a discussion of more advanced command line topics.

Summary

`mv` can be used for moving or renaming files or folders. `cp` can copy files or folders. You should try to avoid using `rm` which permanently removes files or folders.

Exercises

Create a file called `message.txt` in your home directory and move it into another directory. Copy the `message.txt` you just moved into your home directory. Delete both copies of `message.txt`. Try to do this without using `rm`.

2.5 Self Help

Each of the commands that we've discussed so far are thoroughly documented, and you can view their documentation using the `man` command, where the first argument to `man` is the command you're curious about. Let's take a look at the documentation for `ls`:

The controls for navigating man pages are the same as they are for `less`. I often use man pages for quickly searching for an option that I've forgotten. Let's say that I forgot how to get `ls` to print a long list. After typing `man ls` to open the page, type `/` in order to start a search. Then type the word or phrase that you're searching for, in this case type in long list and then press Enter. The page jumps to this entry:

Press the `n` key in order to search for the next occurrence of the word, and if you want to go to the previous occurrence type `Shift + n`. This method of searching also works with `less`. When you're finished looking at a man page type `q` to get back to the prompt.

The `man` command works wonderfully when you know which command you want to look up, but what if you've forgotten the name of the command you're looking for? You can use `apropos` to search all of the available commands and their descriptions. For example let's pretend that I forgot the name of my favorite command line text editor. You could type `apropos editor` into the command line which will print a list of results:

The second result is `nano` which was just on the tip of my tongue! Both `man` and `apropos` are useful when a search is only a few keystrokes away, but if you're looking for detailed examples and explanations you're better off using a search engine if you have access to a web browser. Summary

Use `man` to look up the documentation for a command. If you can't think of the name of a command use `apropos` to search for a word associated with that command. If you have access to a web browser, using a search engine might be better than `man` or `apropos`.

Exercises:

Use `man` to look up the flag for human-readable output from `ls`. Get help with `man` by typing `man man` into the console. Wouldn't it be nice if there was a calendar command? Use `apropos` to look for such a command, then use `man` to read about how that command works.

2.6 Get Wild

Let's go into my Photos folder in my home directory and take a look around:

I've just been dumping pictures and figures into this folder without organizing them at all! Thankfully (in the words of Dr. Jenny Bryan) I have an unwavering commitment to the ISO 8601 date standard so at least I know when these photos were taken. Instead of using `mv` to move around each individual photo I can select groups of photos using the `*` wildcard. A wildcard is a character that represents other characters, much like how joker in a deck of cards can represent other cards in the deck. Wildcards are a subset of metacharacters, a topic which we will discuss in detail later on in this chapter. The `*` ("star") wildcard represents zero or more of any character, and it can be used to match names of files and folders in the command line. For example if I wanted to list all of the files in my Photos directory which have a name that starts with "2017" I could do the following:

Only the files starting with "2017" are listed! The command `ls 2017*` literally means: list the files that start with "2017" followed by zero or more of any character. As you can imagine using wildcards is a powerful tool for working with groups of files that are similarly named.

Let's walk through a few other examples of using the star wildcard. We could only list the photos starting with "2016":

We could list only the files with names ending in `.jpg`:

In the case above the file name can start with a sequence of zero or more of any character, but the file name must end in `.jpg`. Or we could also list only the first photos from each set of photos:

All of the files above have names that are composed of a sequence of characters, followed by the adjacent characters `01.`, followed by another sequence of characters. Notice that if I had entered `ls *01*` into the console every file would have been listed since `01` is a part of all of the file names in my Photos directory.

Let's organize these photos by year. First let's create one directory for each year of photos:

Now we can move the photos using wildcards:

Notice that I've moved all files that start with "2017-" into the 2017 folder! Now let's do the same thing for files with names starting with "2016-":

Finally my photos are somewhat organized! Let's list the files in each directory just to make sure all was moved as planned:

Looks good! There are a few more wildcards beyond the star wildcard which we'll discuss in the next section where searching file names gets a little more advanced.

Summary

Wildcards can represent many kinds and numbers of characters. The star wildcard (*) represents zero or more of any character. You can use wildcards on the command line in order to work with multiple files and folders.

Exercises

Before I organized the photos by year, what command would have listed all of the photos of type .png? Before I organized the photos by year, what command would have deleted all of my hiking photos? What series of commands would you use in order to put my figures for a data science course and the pictures I took in the lab into their own folders?

2.7 Regular Expressions

The ability to search through files and folders can greatly improve your productivity using Unix. First we'll cover searching through text files. I recently downloaded a list of the names of the states in the US, which you can find [here](#). Let's take a look at this file:

It makes sense that there are 50 lines, but it's interesting that there are 60 total words. Let's take a peak at the beginning of the file:

This file looks basically how you would expect it to look! You may recall from Chapter 3 that the kind of shell that we're using is the bash shell. Bash treats different kinds of data differently, and we'll dive deeper into data types in Chapter 5. For now all you need to know is that text data are called strings. A string could be a word, a sentence, a book, or a file or folder name. One of the most effective ways to search through strings is to use regular expressions. Regular expressions are strings that define patterns in other strings. You can use regular expressions to search for a sub-string contained within a larger string, or to replace one part of a string with another string.

One of the most popular tools for searching through text files is `grep`. The simplest use of `grep` requires two arguments: a regular expression and a text file to search. Let's see a simple example of `grep` in action and then I'll explain how it works:

In the command above, the first argument to `grep` is the regular expression "x". The "x" regular expression represents one instance of the letter "x". Every line of the `states.txt` file that contains at least one instance of the letter "x" is printed to the console. As you can see New Mexico and Texas are the only two state names that contain the letter "x". Let's try searching for the letter "q" in all of the state names using `grep`:

Nothing is printed to the console because the letter "q" isn't in any of the state names. We can search for more than individual characters though. For example the following command will search for the state names that contain the word "New":

In the previous case the regular expression we used was simply "New", which represents an occurrence of the string "New". Regular expressions are not limited to just being individual characters or words, they can also represent parts of words. For example I could search all of the state names that contain the string "nia" with the following command:

All of the state names above happen to end with the string "nia".

2.8 Metacharacters

Regular expressions aren't just limited to searching with characters and strings, the real power of regular expressions come from using metacharacters. Remember that metacharacters are characters that can be used to represent other characters. To take full advantage of all of the metacharacters we should use `grep`'s cousin `egrep`, which just extends `grep`'s capabilities. The first metacharacter we should discuss is the "." (period) metacharacter, which represents any character. If for example I wanted to search `states.txt` for the character "i", followed by any character, followed by the character "g" I could do so with the following command:

The regular expression "i.g" matches the sub-string "irg" in Virginia, and West Virginia, and it matches the sub-string "ing" in Washington and Wyoming. The period metacharacter is a stand-in for the "r" in "irg" and the "n" in "ing" in the example above. The period metacharacter is extremely liberal, for example the command `egrep "." states.txt` would return every line of `states.txt` since the regular expression "." would match one occurrence of any character on every line (there's at least one character on every line).

Besides characters that can represent other characters, there are also metacharacters called quantifiers which allow you to specify the number of times a particular regular expression should appear in a string. One of the most basic quantifiers is "+" (plus) which represents one or more occurrences of the preceding expression. For example the regular expression "s+as" means: one or more "s" followed by "as". Let's see if any of the state names match this expression:

Both Arkansas and Kansas match the regular expression "s+as". Besides the plus metacharacter there's also the "*" (star) metacharacter which represents zero or more occurrences of the preceding expression. Let's see what happens if we change "s+as" to "s*as":

As you can see the star metacharacter is much more liberal with respect to matching since many more state names are matched by "s*as". There are more specific quantifiers you can use beyond "zero or more" or "one or more" occurrences of an expression. You can use curly brackets () to specify an exact number of occurrences of an expression. For example the regular expression "s2" specifies exactly two occurrences of the character "s". Let's try using this regular expression:

Take note that the regular expression "s2" is equivalent to the regular expression "ss". We could also search for state names that have between two and three adjacent occurrences of the letter "s" with the regular expression "s2,3":

Of course the results are the same because there aren't any states that have "s" repeated three times.

You can use a capturing group in order to search for multiple occurrences of a string. You can create capturing groups within regular expressions by using parentheses ("()"). For example if I wanted to search states.txt for the string "iss" occurring twice in a state name I could use a capturing group and a quantifier like so:

We could combine more quantifiers and capturing groups to dream up even more complicated regular expressions. For example, the following regular expression describes three occurrences of an "i" followed by two of any character:

The complex regular expression above still only matches "Mississippi".

2.9 Characters Set

For the next couple of examples we're going to need some text data beyond the names of the states. Let's just create a short text file from the console:

In addition to quantifiers there are also regular expressions for describing sets of characters. The `\w` metacharacter corresponds to all "word" characters, the `\d` metacharacter corresponds to all "number" characters, and the `\s` metacharacter corresponds to all "space" characters. Let's take a look at using each of these metacharacters on small.txt:

As you can see in the example above, the `\w` metacharacter matches all letters, numbers, and even the underscore character (`_`). We can see the compliment of this grep by adding the `-v` flag to the command:

The `-v` flag (which stands for invert match) makes grep return all of the lines not matched by the regular expression. Note that the character sets for regular expressions also have their inverse sets: `\W` for non-words, `\D` for non-digits, and `\S` for non-spaces. Let's take a look at using `\W`:

The returned strings all contain non-word characters. Note the difference between the results of using the invert flag `-v` versus using an inverse set regular expression.

In addition to general character sets we can also create specific character sets using square brackets (`[]`) and then including the characters we wish to match in the square brackets. For example the regular expression for the set of vowels is `[aeiou]`. You can also create a regular expression for the compliment of a set by including a caret (`^`) in the beginning of a set. For example the regular expression `[^aeiou]` matches all characters that are not vowels. Let's test both on small.txt:

Notice that the word “rhythms” does not appear in the result (it’s the longest word without any vowels that I could think of).

Every line in the file is printed, because every line contains at least one non-vowel! If you want to specify a range of characters you can use a hyphen (-) inside of the square brackets. For example the regular expression `[e-q]` matches all of the lowercase letters between “e” and “q” in the alphabet inclusively. Case matters when you’re specifying character sets, so if you wanted to only match uppercase characters you’d need to use `[E-Q]`. To ignore the case of your match you could combine the character sets with the `[e-qE-Q]` regex (short for regular expression), or you could use the `-i` flag with `grep` to ignore the case. Note that the `-i` flag will work for any provided regular expression, not just character sets. Let’s take a look at some examples using the regular expressions that we just described:

APPENDIX

Notation

The next list describes several symbols that will be later used within the body of the document.

c Speed of light in a vacuum inertial frame

h Planck constant

Greek Letters with Pronunciation

Character	Name	Character	Name
α	alpha <i>AL-fuh</i>	ν	nu <i>NEW</i>
β	beta <i>BAY-tuh</i>	ξ, Ξ	xi <i>KSIGH</i>
γ, Γ	gamma <i>GAM-muh</i>	\omicron	omicron <i>OM-uh-CRON</i>
δ, Δ	delta <i>DEL-tuh</i>	π, Π	pi <i>PIE</i>
ϵ	epsilon <i>EP-suh-lon</i>	ρ	rho <i>ROW</i>
ζ	zeta <i>ZAY-tuh</i>	σ, Σ	sigma <i>SIG-muh</i>
η	eta <i>AY-tuh</i>	τ	tau <i>TOW (as in cow)</i>
θ, Θ	theta <i>THAY-tuh</i>	υ, Υ	upsilon <i>OOP-suh-LON</i>
ι	iota <i>eye-OH-tuh</i>	ϕ, Φ	phi <i>FEE, or FI (as in hi)</i>
κ	kappa <i>KAP-uh</i>	χ	chi <i>KI (as in hi)</i>
λ, Λ	lambda <i>LAM-duh</i>	ψ, Ψ	psi <i>SIGH, or PSIGH</i>
μ	mu <i>MEW</i>	ω, Ω	omega <i>oh-MAY-guh</i>

Capitals shown are the ones that differ from Roman capitals.

Alphabetical Index

ls, 5