

Learn Data with Bash Shell

Project based Learning



Learn Data with Bash Shell

Explore real-world data at the Linux command line

Scientific Programmer

This book is for sale at <http://leanpub.com/hellobigdata>

This version was published on 2019-01-19



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.

© 2017 - 2019 Scientific Programmer

Tweet This Book!

Please help Scientific Programmer by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#hellobigdata](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#hellobigdata](#)

Also By Scientific Programmer
Python REGEX

Contents

About the author	i
Introduction	i
What is Bash ?	i
When Bash is useful?	i
Bash in data mining	ii
Who is this book for?	ii
How to read this book?	ii
 Part 1: Projects	 iii
Project 1: The ‘US News’ Uni Ranks	1
Dataset Preview	2
Data Analysis	3
Find the colleges	3
Finding the percent of colleges in the ranklist	4
Listing the Institutes from a given state	5
Finding the number of Institutes from each state	5
Finding a correlation between ranks and tuition fees?	7
Chapter Summary	8
 Project 2: Facebook Data Mining	 1
Dataset Preview	2
How many columns and rows?	2
How the data looks like?	3
Data Analysis	3
How many status, in each status type?	3
Find the most popular status entry	4
Chapter Summary	8
 Project 3: Best Australian Cities - Least Crimes	 1
Data Preview	2
Finding the number of rows and columns	4
The hard way	4
The easy way	5
Data Analysis	5
Finding the top most crime in the whole country	5
Finding the top most crime per city	7
Finding the best city in Australia!	9
Chapter Summary	10

CONTENTS

Project 4: Mining Shakespear-era Plays and Poems	1
Data Preview	2
Analysis	3
How many plays/poems?	3
How many plays/poems by each author?	4
What are the most frequent words?	6
Chapter Summary	9
 Part 2: Tutorials	 10
Hello Bash!	11
which bash?	11
Hello world! bash	12
Bash variables	12
Bash functions	14
Bash meta characters	15
Bash quotation basics	16
Read and store user input	17
Bash redirections	18
Bash if-else (conditional statements)	18
Bash case statement	20
Bash loop statements	21
Bash arithmetic	25
Bash arrays	28
 Hello ! Regular Expressions	 29
REGEX Types	29
Basic Regular Expressions	30
Metachar .	30
Metachar []	30
Metachar [^]	30
Metachar ^	30
Metachar \$	30
Metachar ()	30
Metachar *	31
Metachar {m,n}	31
Extended Regular Expressions	31
Metachar ?	31
Metachar +	31
Metachar	31
REGEX Character Classes	31
REGEX Look Arouds	32
REGEX Atomic Groups (?>)	32
How to Use REGEX in Bash?	33
 Hello! AWK	 35
AWK Built-in Variables	35
AWK statements	36
AWK built-in functions	36

CONTENTS

AWK Examples	37
Example 1. AWK <code>print</code> function	37
Example 2. AWK <code>print</code> specific field	38
Example 3. AWK's <code>BEGIN</code> and <code>END</code> Actions	38
Example 4. AWK fields variable (<code>\$1</code> , <code>\$2</code> and so on)	38
Example 5. AWK built-in variables	39
Example 6. AWK fields comparison >	39
Self-contained AWK scripts	39
Hello! SED, GREP and Find	41
SED - Stream Editor	41
SED substitution	42
Some important SED options	42
SED substitute and regular expressions	43
SED delete	44
SED print	44
SED grouping	44
GREP	44
GREP and regular expressions	45
Find command <code>find</code>	45
 Part 3: Hello Big Data!	 46
Big Data Terminologies	47
HDFS	47
Map Reduce	47
YARN	47
Flume	48
SOOOP	48
Hive	48
Pig	49
Spark	49
HBase	49
Big Data file formats	49
 Conclusion	 51
 References	 52
Bash	52
REGEX	52
AWK	52
SED	53
GREP	53
Big data	53
A companion book	53

About the author

Ahmed Arefin, PhD is an enthusiastic computer programmer with more than a decade of well-rounded computational experience. He likes to code and loves to write. Following his PhD and Postdoc research in the area of data-parallelism he moved forward to become a Scientific Computing professional, keeping his research interests on, in the area of parallel, distributed and accelerated computing. In his day job, he pets a few of the world's fastest T500 supercomputers at a large Australian agency for scientific research.

Introduction

Bash may not be the best way to handle all kinds of Data! But, there often comes a time when you are provided with a pure Bash environment, such as what you get in the common Linux based super computers and you just want some early results or view of the data before you drive into the real programming, using Python, R and SQL, SPSS, and so on. Expertise in these data-intensive languages also comes at the cost of spending a lot of time on them.

In contrast, bash scripting is simple, easy to learn and perfect for mining textual data! Particularly if you deal with genomics, microarrays, social networks, life sciences, and so on. It can help you to quickly sort, search, match, replace, clean and optimise various aspects of your data, and you wouldn't need to go through any tough learning curves. We strongly believe, learning and using Bash shell scripting should be the first step if you want to say, **Hello! to Big Data.**

What is Bash ?

Bash, the **Bourne-Again Shell**, refers to both a specific UNIX shell and an associated scripting language. It was initially written for the GNU project and first released in 1989 and is the default shell of Linux and Apple's macOS as well as now is distributed in the Windows 10 Anniversary Update! Just as a note, the name of the shell 'Bourne-again shell' is punned on the name of the Bourne shell that it replaced and also on the term 'born again', pointing to the spiritual re-birthing in the American Christianity.

Due to the fact that it is a commanding language and processor, it runs inside text windows where the commands are needed to be typed. Along with that, Bash also reads comments from any file and supports globbing, command substitution and control structures, and so on. It is mainly a POSIX (Portable Operating System Interface) shell with a number of extensions and is also Unix's sh-compatible, which incorporates the important features from C shell (csh) or Korn shell (ksh).

When Bash is useful?

Bash can be useful in various ways to a user:

- Instead of typing one Linux command at a time, a script in the bash shell can run an entire set of commands at once. This way we can perform specific tasks by sparing ourselves from the efforts to typing repetitive commands over and over again.
- Typical programming languages are well-suited for a set of detailed operations that are not readily accomplished by Bash. However, a Bash script is preferred when it is desired to make use of the existing programs by letting them to do the heavy lifting which are chained together in a particular manner to achieve the desired result. Bash works as glue logic to make the general purpose tools work together. An example scenario would be to

call a multiple number of MATLAB, R and SPSS scripts from Bash where the outputs are feed from one to another while, performing some pre-prepossing at each stage.

Bash in data mining

There are numerous examples of practical data mining works that will have a flow of importing specific data resources into flat text-type files. Bash can run different programs on those files, clean, optimise and extract preliminary views of the data. There is one part of data mining, which involves unstructured data and then transforming it into a structured one. A scripting language like Bash can be very useful for doing the transformation. For example. Bash is preferable to Hadoop (a popular framework for distributed storage and processing of very large data sets using the MapReduce programming model, further discussed later in this book) because scripts in Bash translate well into Hadoop.

Who is this book for?

Students who want to learn Bash and the command line to improve their career prospects, researchers who want to add Bash and other command line tools to their bag of tricks, scientists who want to learn to explore and analyze the data that their lab generates, journalists who want to polish their reporting by analyzing publicly-available datasets. We believe almost everyone can benefit from learning to use Bash in data mining.

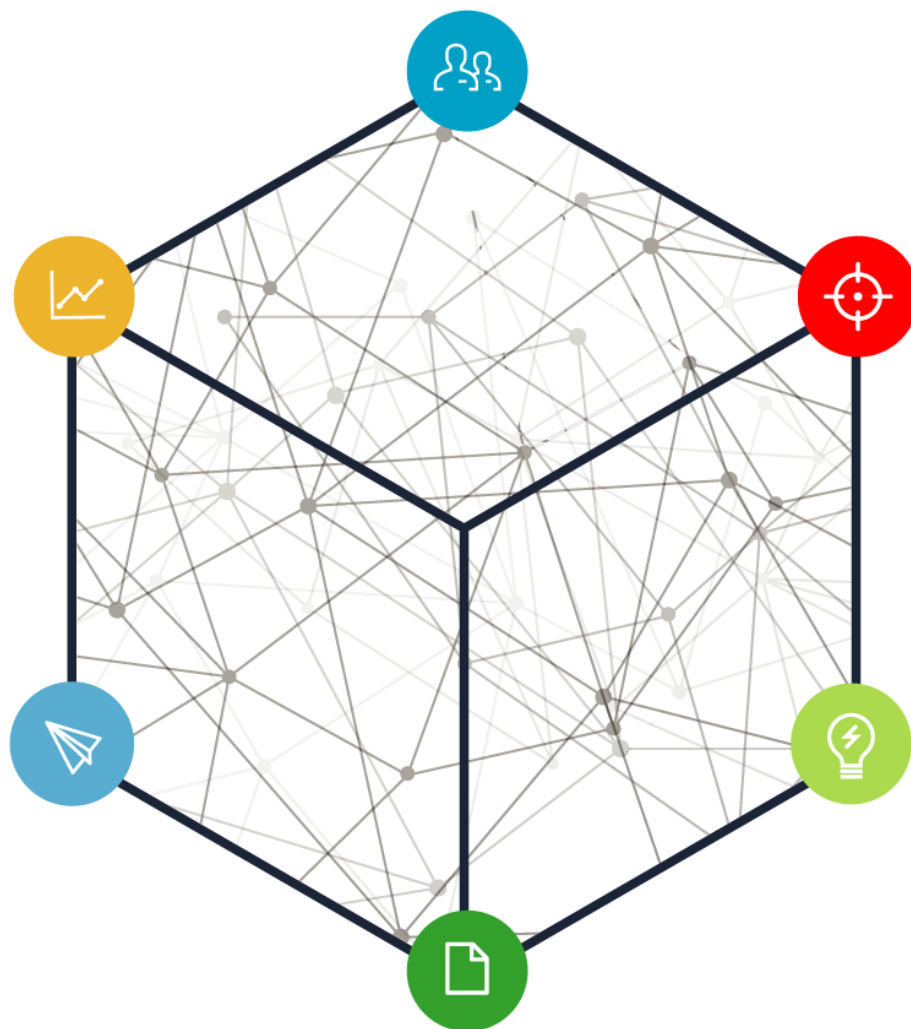
How to read this book?

We recommend reading one section at a time. Then implement that section. Then read the next section, and implement your new learning. If you haven't used Bash before, feel free to skip the projects to the tutorials. Read the tutorials and then come back to the projects, again. You will now find them easy.

Let's start already!

Part 1: Projects

In this part of the book, we will explore a few flat file data handling projects with an increasing order of difficulty.



If you haven't used Bash before, feel free to skip the projects and get to the tutorials part. Read the tutorials and then come back to this part.

Project 1: The ‘US News’ Uni Ranks

The ranking of universities has become a common task performed by many institutions, each of them proposes a different ranking based in several weighted categories. Examples of those rankings are: [Webometrics Ranking of World Universities](#), [THES - QS World Universities Rankings](#) and [Academic Ranking of World Universities](#). The first ranking measures the visibility of the universities and their global performance in the web. The last two attempt to measure the performance of the universities based in categories like prizes received by members, citations, and publications. Employers, especially from the multinational organisations use rankings to find universities to source graduates, so attending a high-ranking university can help in a competitive job market.

In this lesson we will use a simple (publicly available) dataset obtained from the [data.world](#) called: US News Universities Rankings 2017 edition. From this data, using Bash we will explore different features and finally find an interesting fact about the correlation of tuition fees and uni rank. This simple dataset contains the following fields.

- **Name** - institution name
- **Location** - City, State where located
- **Rank** - read methodology [here](#).
- **Description** - a snippet of text overview from U.S. News.
- **Tuition and fees** - combined tuition and fees.
- **Undergrad Enrollment** - number of enrolled undergraduate students

In each project described in this book, we will attempt to learn a few Bash commands and tricks.



Learning objectives

By completing this, you will learn to use the following Bash commands:

- `cd` – change Directory
- `mkdir` – make a directory
- `head` – output the first part of files
- `tail` – opposite to head
- `cat` – concatenate and print files
- `sort` – sort file contents
- `uniq` – remove duplicate entries

Before we go any further, let's setup our working environment by creating a folder on the Desktop. To do so, assuming we have a Linux based OS (e.g., Ubuntu) on our computer and let's first fire up a command line and navigate to our analysis folder:

Setup the working directory

```
1 cd ~/Desktop
2 mkdir unirankingdata
3 cd unirankingdata
```

This will create a folder `unirankdata` on your Desktop. Next, we download the data.



Data download

You should download the data from the book webpage, as we have slightly simplified the data (see below) and Let's save the data as: `unirank.csv`

Dataset Preview

This dataset is small (toy) and we could in principle open it in a text editor or in Excel. However, real-world datasets are often larger and cumbersome to open in their entirety. Let's assume as if it were a Big Data (and unstructured) and we want to get a sneak peak of the data. This is often the first thing to do when you get your hands on new data- previewing; it is important to get a sense for what it contains, how it is organized, and whether the data makes sense in the first place.

To help us get a preview of the data, we can use the command `head`, which as the name suggests, shows the first few lines of a file.

```
1 head unirank.csv
```

However, you will find the outputs are not very interesting on the first place, therefore we install a tool called `csvkit`, which is a suite of command-line tools for converting to and working with CSV (install: `sudo pip install csvkit`).

This will greatly help our future analyses. After we have installed the `csvkit`, we re-run the `head` command, but outputs piped (`|`, soon we'll learn about it) through the `csvlook` command from the `csvkit` suit:

```
1 head unirank.csv | csvlook
```

You should see the first 10 lines of the file output onto the screen, to see more than the first 10 lines, e.g. the first 25, use the `-n` option:

```
1 head -n 25 unirank.csv | csvlook
```

```

unirankingdata : bash
hellobigdata@bash:unirankingdata$ head -n 25 unirank.csv | csvlook

```

Name	City	State	Tuition and fees	Undergrad Enrollment	Rank
Princeton University	Princeton	NJ	45,320	5,402	1
Harvard University	Cambridge	MA	47,074	6,699	2
University of Chicago	Chicago	IL	52,491	5,844	3
Yale University	New Haven	CT	49,480	5,532	3
Columbia University	New York	NY	55,056	6,102	5
Stanford University	Stanford	CA	47,940	6,999	5
Massachusetts Institute of Technology	Cambridge	MA	48,452	4,527	7
Duke University	Durham	NC	51,265	6,639	8
University of Pennsylvania	Philadelphia	PA	51,464	9,726	8
Johns Hopkins University	Baltimore	MD	50,410	6,524	10
Dartmouth College	Hanover	NH	51,438	4,307	11
California Institute of Technology	Pasadena	CA	47,577	1,001	12
Northwestern University	Evanston	IL	50,855	8,314	12
Brown University	Providence	RI	51,367	6,652	14
Cornell University	Ithaca	NY	50,953	14,315	15
Rice University	Houston	TX	43,918	3,910	15
University of Notre Dame	Notre Dame	IN	49,685	8,462	15
Vanderbilt University	Nashville	TN	45,610	6,883	15
Washington University in St. Louis	St. Louis	MO	49,770	7,504	19
Emory University	Atlanta	GA	47,954	6,867	20
Georgetown University	Washington	DC	50,547	7,562	20
University of California--Berkeley	Berkeley	CA	40,191	27,496	20
University of Southern California	Los Angeles	CA	52,217	18,810	23
Carnegie Mellon University	Pittsburgh	PA	52,040	6,454	24

```

hellobigdata@bash:unirankingdata$
unirankingdata : bash

```

Output: unirank.csv data preview

Here, the dataset name `unirankingdata.csv` is a **command-line** argument that is given to the command `head` and the `-n` is an option which allows us to overwrite the 10-line default. Such command-line options are typically specified with a dash followed by a string, a space, and the value of the option (e.g. `-n 25`).

However, often the options don't require a value but instead are made for toggling a feature on or off, for example `top -h` shows the help page for the command `top` that shows off all the running process and apps.

From the first 25 lines of the file, we can infer that the data is formatted as a file with separated values. From the first line (often called a header line) and the first few lines of data, we can infer the column contents: Name, City, State, Tuition and fees, Undergrad Enrollment and Rank

Data Analysis

Let's now proceed to our first analysis:

Find the colleges

To list all the lines in the data file that contain the phrase "college", we need to introduce you with the command `grep` (global regular expression print). In a nutshell, `grep` allows you to look through all the lines in a file but only output those that match a pattern. In our case, we want to find all the lines in the dataset that contain "college". Here's how we do it:

```
1 grep -i "college" unirank.csv
```

```

unirankingdata: bash
hellobigdata@bash:unirankingdata$ grep -i "college" unirank.csv | csvlook
| Dartmouth College | Hanover | NH | 51438 | 4307 | 11 |
|-----|-----|-----|-----|-----|-----|
| Boston College | Chestnut Hill | MA | 51,296 | 9,192 | 31 |
| College of William & Mary | Williamsburg | VA | 41,718 | 6,301 | 32 |
| University of Maryland-College Park | College Park | MD | 32,045 | 27,443 | 60 |
| Texas A&M University-College Station | College Station | TX | 28,768 | 48,960 | 74 |
| SUNY College of Environmental Science and Forestry | Syracuse | NY | 17,620 | 1,839 | 99 |
| St. John Fisher College | Rochester | NY | 31,880 | 2,805 | 146 |
| Edgewood College | Madison | WI | 27,530 | 1,813 | 171 |
hellobigdata@bash:unirankingdata$

```

Institutes containing “colleges” in the unirank.csv dataset

Here, the `grep` command takes two command-line arguments: the first is the pattern, and the second is the file in which we want to search for this pattern. If you run this command you should see some lines that contain the string “college”. Note that we have put `-i` option to make the matching case insensitive. Also, find that the logic by mistake identified two universities as college! due to the fact that their names contained the string (“college”). So, you need to be careful, while using `grep` in data analytics and particularly before reaching a decision!

Finding the percent of colleges in the ranklist

For this step, we need to introduce the concept of pipes, which is one of the most powerful feature in Bash shell. Pipes are represented by the vertical bar symbol (`|`). Essentially, a pipe allows you to forward the output of one command to the input of another command, without having to save the intermediate output to a file. We would like to give the output of `grep` to another command which can tell us how many lines that output has. As it turns out, there is a command that will let us do just that: `wc` which stands for word count and is used to count how many words there are in a given input, but is most often used with the `-l` option that will instead count the number of lines in our input. So let’s start by counting how many lines total there are in the `unirank.csv` file:

```

1 $ wc -l unirank.csv
2 232

```

Now we count the number of colleges, let’s now use pipes and bring everything we learned together:

```
1 $ grep -i "college" unirank.csv | wc -l
2 7
```

The output of `grep`—the lines in the file that contain “college”—is passed on to `wc -l`, which will count the number of lines in the file. Notice that we didn’t specify a filename in the `wc -l` command; we didn’t have to because we piped our data into `wc -l`, so it didn’t need to read the data from a file. Running this command should give you the output of 7. So in this dataset, $((7-2)/232) \times 100\% = 2.15\%$ institutes comes from the US colleges. We deducted the 2 wrongly identified colleges.

```

unirankingdata: bash
hellobigdata@bash:unirankingdata$ grep "CA" unirank.csv | csvlook
| Stanford University | Stanford | CA | 47940 | 6999 | 5 |
|-----|-----|---|-----|-----|---|
| California Institute of Technology | Pasadena | CA | 47,577 | 1,001 | 12 |
| University of California--Berkeley | Berkeley | CA | 40,191 | 27,496 | 20 |
| University of Southern California | Los Angeles | CA | 52,217 | 18,810 | 23 |
| University of California--Los Angeles | Los Angeles | CA | 39,518 | 29,585 | 24 |
| University of California--Santa Barbara | Santa Barbara | CA | 40,704 | 20,607 | 37 |
| University of California--Irvine | Irvine | CA | 39,458 | 25,256 | 39 |
| University of California--Davis | Davis | CA | 40,728 | 28,384 | 44 |
| University of California--San Diego | La Jolla | CA | 41,387 | 26,590 | 44 |
| Pepperdine University | Malibu | CA | 50,022 | 3,533 | 50 |
| University of California--Santa Cruz | Santa Cruz | CA | 40,241 | 16,231 | 79 |
| University of San Diego | San Diego | CA | 46,140 | 5,647 | 86 |
| University of San Francisco | San Francisco | CA | 44,494 | 6,782 | 107 |
| University of the Pacific | Stockton | CA | 44,588 | 3,735 | 111 |
| University of California--Riverside | Riverside | CA | 40,263 | 18,608 | 118 |
| San Diego State University | San Diego | CA | 18,244 | 29,234 | 146 |
| University of California--Merced | Merced | CA | 39,944 | 6,237 | 152 |
| University of La Verne | La Verne | CA | 39,900 | 2,864 | 152 |
| Biola University | La Mirada | CA | 36,696 | 4,225 | 164 |
| Azusa Pacific University | Azusa | CA | 36,120 | 5,883 | 183 |
| California State University--Fullerton | Fullerton | CA | 17,596 | 33,144 | 202 |
| California State University--Fresno | Fresno | CA | 17,209 | 21,482 | 220 |
hellobigdata@bash:unirankingdata$

```

Institutes came from the CA (California) state

Listing the Institutes from a given state

For this step, all we need to do now is to run the same command but by replacing the string “college” with a state name, e.g., “CA” (do not use the `-i` option):

```
1 grep "CA" unirank.csv | wc -l
2 22
```

Note that finding states with `grep` on all the lines is not safe! as it could also find an Institute containing the string “CA”. To avoid this, soon we will learn how to run `grep` on specific columns using the `cut` command.

Finding the number of Institutes from each state

At this point we want to calculate how many Institutes have been ranked from each of the US states in the dataset. Let’s start by extracting only the part of each line that is relevant to us. In our case, notice that we are interested in column #1 and 3 (university and state names, respectively). To extract these columns, we can make use of a command called `cut` as follows:


```
1 $ cat unirank.csv | cut -f1,3 -d,
```

Here, the command-line option `-f` specifies which field (column) to extract or cut out from the file and the option `(d,)` tells that we want delimit the cuts by comma `(,)`. When you run that command, you should see that the output consist only of lines such as university names and states. Note that, despite its name, the `cut` command does not modify the original file it acts on. Now onto the last part. We would like to count how many unis came from each state. However, this is a complex procedure and there isn't one command that can do all that; we will have to use two commands. Here we need the command `uniq -c` to count (hence the `-c`) how many unique appearances of each state. However, `uniq -c` requires the input to be sorted, so the first step is to sort the list of universities and states. We can do this very easily with a command that is conveniently called `sort` :

```
1 $ cat unirank.csv | cut -f1,3 -d, | sort -k 2 -t",,"
```

The `sort` options: `k 2` tells sort function to select the column 2 as a key and `-t",,"` option tells that the delimiter is a comma `(,)`.

```

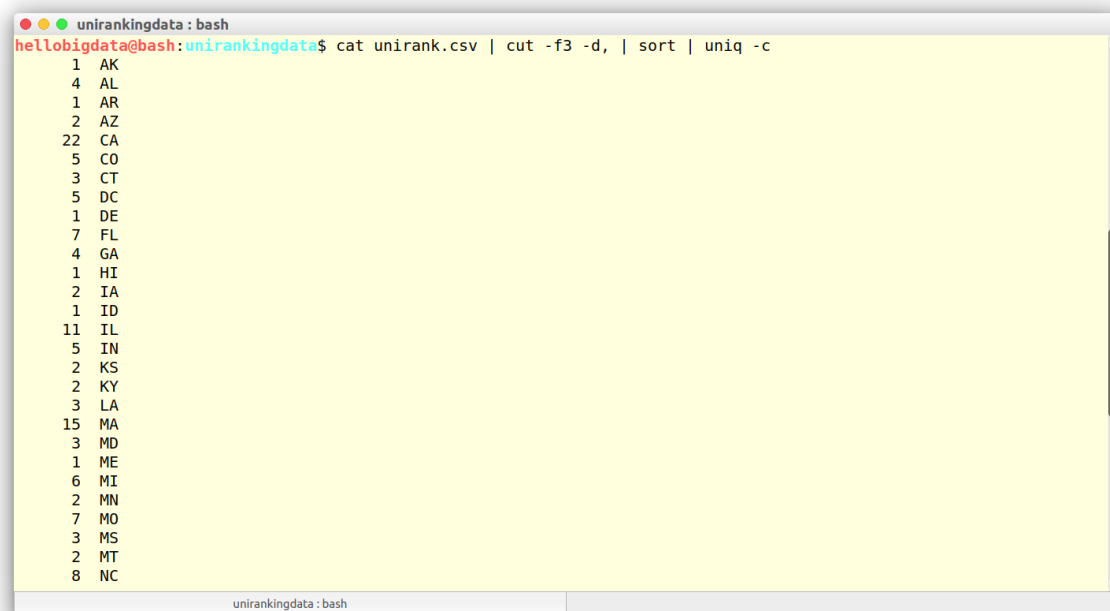
unirankingdata: bash
hellobigdata@bash:unirankingdata$ cat unirank.csv | cut -f1,3 -d, | sort -k 2 -t',,' | csvlook
|-----|-----|
| University of Alaska--Fairbanks | AK |
|-----|-----|
| Auburn University | AL |
| University of Alabama | AL |
| University of Alabama--Birmingham | AL |
| University of Alabama--Huntsville | AL |
| University of Arkansas | AR |
| Arizona State University--Tempe | AZ |
| University of Arizona | AZ |
| Azusa Pacific University | CA |
| Biola University | CA |
| California Institute of Technology | CA |
| California State University--Fresno | CA |
| California State University--Fullerton | CA |
| Pepperdine University | CA |
| San Diego State University | CA |
| Stanford University | CA |
| University of California--Berkeley | CA |
| University of California--Davis | CA |
| University of California--Irvine | CA |
| University of California--Los Angeles | CA |
| University of California--Merced | CA |
| University of California--Riverside | CA |
| University of California--San Diego | CA |
| University of California--Santa Barbara | CA |
| University of California--Santa Cruz | CA |
| University of La Verne | CA |
| University of San Diego | CA |

```

Output: Institutes sorted by states

Notice that, as a result of our list being sorted, all the lines with same state are right next to each other. Now, as mentioned in our plan above, we'll use `uniq -c` to "condense" neighboring lines that are the same and in the process, count how many of each are seen:

```
1 $ cat unirank.csv | cut -f3 -d, | sort | uniq -c
```



```

unirankingdata: bash
hellobigdata@bash:unirankingdata$ cat unirank.csv | cut -f3 -d, | sort | uniq -c
 1 AK
 4 AL
 1 AR
 2 AZ
22 CA
 5 CO
 3 CT
 5 DC
 1 DE
 7 FL
 4 GA
 1 HI
 2 IA
 1 ID
11 IL
 5 IN
 2 KS
 2 KY
 3 LA
15 MA
 3 MD
 1 ME
 6 MI
 2 MN
 7 MO
 3 MS
 2 MT
 8 NC

```

Output: Institutes in the CA (California) state.

We now have a listing of how many unis came from each state, and it's clear that the vast majority of ranked Institutes came from the state of CA - California!

Finding a correlation between ranks and tuition fees?

We already know the ranks and the tuition fees per university (given). An interesting question to investigate would be to find what's the correlation of uni ranks with tuition and fees?

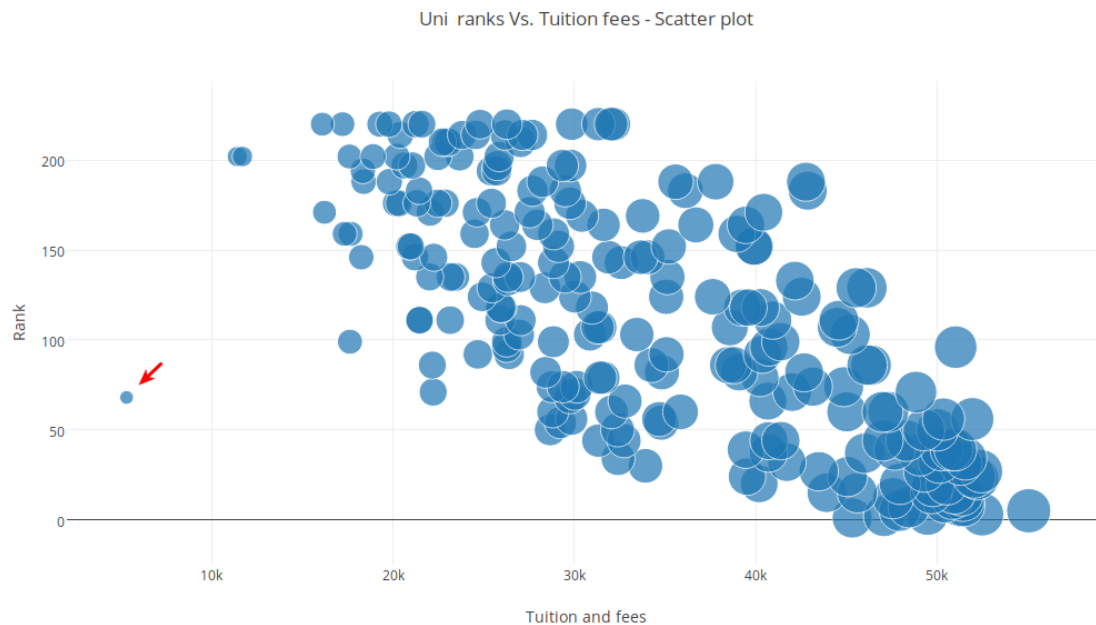
To carry out this, we first cat and cut out the Tuition and fees(col 4) and and Ranks (col 6) from the data into the new dataset called `udata.csv`:

```
1 cat unirank.csv | cut -f4,6 -d, > udata.csv
```

Note that the redirection symbol (`>`), helped us to save the output. Now this data can simply be plotted using a scatterplot tool called `scatter` (install `sudo pip install bashplotlib`).

```
1 cat udata.csv| tail -n +2 | scatter
```

Note that `tail -n +2` excludes the first row i.e., column titles prior to passing the output all the way to end to `scatter`. However, this tool's output doesn't make much sense, as it doesn't show any x,y- axes legends. Therefore, we uploaded the data (`udata.csv`) to an online tool called plot.ly, which produced the following beautiful scatter plot:



Output: Ranks vs. tuition: a scatterplot via *plot.ly*

It's a no brainer to understand from the plot above that highly ranked universities have higher tuition fees! However, the scatterplot also depicts one university (Brigham Young University-Provo) that had a higher rank (rank=68) with an extremely low tuition fees (\$5300 USD p/a). Is this an anomaly (outlier) in the dataset? We leave the question for you to further investigate!

Chapter Summary

In this project we have learned to use some important bash commands like `head`, `tail`, `sort`, `uniq`, `cut`, etc. in the context of mining a csv formatted toy dataset consisting of rankings of the US academic Institutes. We will re-use these commands in a more complicated format in the upcoming chapters.

Project 2: Facebook Data Mining

Now that the Facebook has more than a billion of active users, it really has become a personal, product and corporate branding hub. Companies would like to understand what people think about topics related to their business, so they can make their products and marketing more relevant to their customers. One way to achieve such goal is to analyse company's FB peages which can make marketing content more relevant for marketers.

In this lesson, we're going to mine a dataset generated by using a [Facebook scraper](#) on a particular Facebook page (undisclosed). The goal of this experiment is to find the most vibrant status message on that page, with just one Bash command.



Data download

You should download the data from the book webpage, as we have slightly simplified the data (see below) and Let's save the data as: `facebookdata.csv`

The dataset contains the following attributes: `status_id`, `status_message`, `link_name`, `status_type`, `status_link`, `status_published`, `num_reactions`, `num_comments`, `num_shares`, `num_likes`, `num_loves`, `num_wows`, `num_hahas`, `num_sads`, `num_angrys`. From this data, using Bash we will explore different features and finally find which message was the most vibrant in terms of total number of activities.



Learning objectives

By completing this, you will learn to use the following Bash commands:

- `head` – output the first part of files
- `tail` – opposite to head
- `cat` – concatenate and print files
- `sort` – sort file contents
- `grep` – search the input files for lines containing a match to a given pattern list
- `uniq` – remove duplicate entries
- `awk` – programming language
- Bash functions

Before we go any further, let's setup our working environment by creating a folder on the Desktop. To do so, let's first fire up a command line and navigate to our analysis folder:

Setup the working directory

```
1 cd ~/Desktop
2 mkdir facebookdata
3 cd facebookdata
```

This will create a folder `facebookdata` on your Desktop. Next, we download the data.

Dataset Preview

Same as before, this dataset is also small (toy) and we could in principle open it in a text editor or in Excel. However, as mentioned in the first chapter, real-world datasets are often larger and cumbersome to open in their entirety. Instead, let's get a sneak peak of the data.

How many columns and rows?

First, let us, find some stat about the data using `csvstat` tool from the `csvkit`:

Finding the stat of the columns:

```
1 $ csvstat -n facebookdata.csv
```

Finding the stat of the rows:

```
1 $ csvstat --count facebookdata.csv
```

Final output:

A terminal window titled 'facebookdata: bash' showing the execution of two 'csvstat' commands. The first command, 'csvstat -n facebookdata.csv', lists 15 columns: status_id, status_message, link_name, status_type, status_link, status_published, num_reactions, num_comments, num_shares, num_likes, num_loves, num_wows, num_hahas, num_sads, and num_angrys. The second command, 'csvstat --count facebookdata.csv', shows 'Row count: 3222'.

```
facebookdata: bash
hellobigdata@bash:facebookdata$ csvstat -n facebookdata.csv
1: status_id
2: status_message
3: link_name
4: status_type
5: status_link
6: status_published
7: num_reactions
8: num_comments
9: num_shares
10: num_likes
11: num_loves
12: num_wows
13: num_hahas
14: num_sads
15: num_angrys
hellobigdata@bash:facebookdata$ csvstat --count facebookdata.csv
Row count: 3222
hellobigdata@bash:facebookdata$
```

Output: facebookdata.csv preview

It looks like that the dataset has a total of 11 columns and 3222 rows.

How the data looks like?

This is often the first thing to do when you get your hands on new data; previewing it is important to get a sense for what it contains, how it is organized, and whether the data makes sense in the first place. To help us get a preview of the data, we can use the command `head`, `csvlook` and `csvcut`:

```
1 csvcut -c 1,4,7-11 facebookdata.csv | csvlook | head -n 50
```

status_id	status_type	num_reactions	num_comments	num_shares	num_likes	num_loves
7331091005_10154123560186006	video	5,565	178	461	5,488	43
7331091005_10154123362896006	video	11,997	1,932	3,158	10,385	96
7331091005_10154123319126006	link	2,063	270	400	1,971	28
7331091005_10154123234521006	photo	116,543	11,811	43,923	107,561	1,202
7331091005_10154123219076006	link	10,475	999	2,978	8,833	28
7331091005_10154123189346006	photo	32,111	897	2,525	29,660	398
7331091005_10154123132896006	video	1,261	628	90	1,173	43
7331091005_10154123136121006	video	9,257	283	819	8,776	348
7331091005_10154123102181006	photo	13,314	147	917	12,771	433
7331091005_10154123021136006	video	43,325	4,447	35,534	41,914	621
7331091005_10154122959911006	link	449	19	33	441	3
7331091005_10154122914261006	video	3,717	38	355	3,580	97
7331091005_10154122756301006	video	131	11	10	126	4
7331091005_10154122866096006	video	3,985	149	366	3,786	82
7331091005_10154122823271006	link	4,468	1,026	1,632	4,190	20
7331091005_10154122765336006	photo	25,707	739	2,413	24,667	774
7331091005_10154122722061006	photo	48,708	1,326	3,840	43,658	503
7331091005_10154122686201006	link	2,655	89	100	2,596	14
7331091005_10154122655576006	video	8,894	395	811	7,685	99
7331091005_10154122459461006	link	3,702	222	578	3,610	23
7331091005_10154122428786006	link	4,955	139	184	4,345	36
7331091005_10154121679506006	link	14,426	1,474	1,204	13,797	162
7331091005_10154121610466006	photo	3,719	156	202	3,323	31
7331091005_10154121541501006	photo	6,940	65	285	6,739	141
7331091005_10154121541286006	link	12,719	338	897	12,035	258
7331091005_10154121438506006	link	4,520	346	517	4,138	42

A partial preview of the facebookdata.csv

The `csvcut` command can help us to cut a given set of columns (e.g., 1, 4, 7-11). Note that we have not previewed the column numbers 2 and 3 (`status_message`, `link_name`), which are wider columns and wouldn't fit properly into our preview-screen above!

Data Analysis

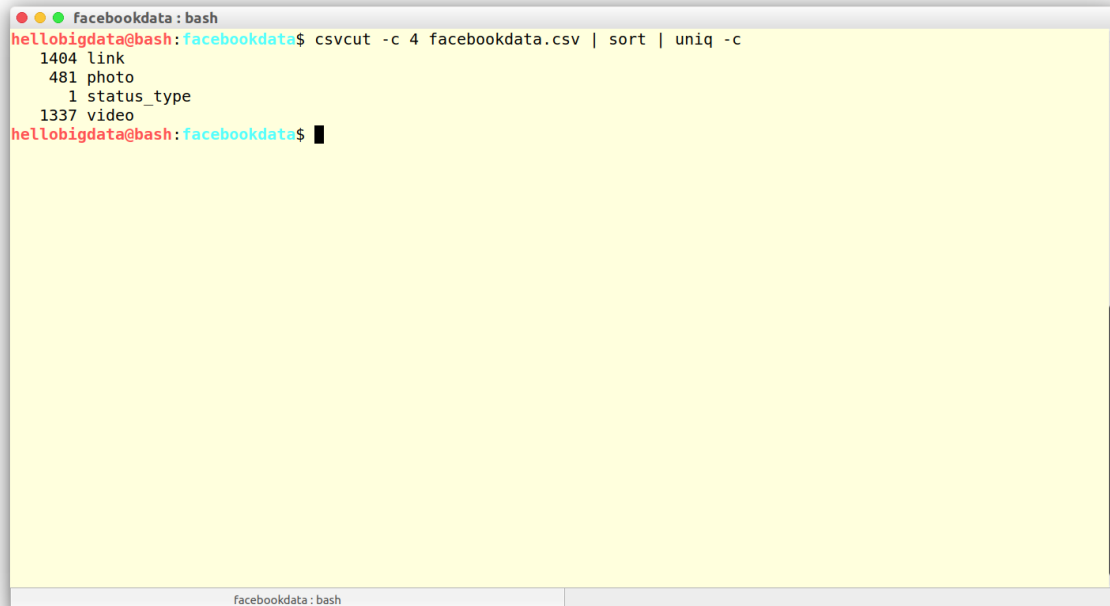
Let's now proceed to our first analysis:

How many status, in each status type?

Let us first calculate how many entries for each of the status types in the dataset. Status types are in the column 4. To extract column #4 from our file, we can make use of the `csvcut` again as follows:

```
1 $ csvcut -c 4 facebookdata.csv | sort | uniq -c
```

Here, the command-line option `-c` specifies which column to extract (or cut out). Note that, despite its name, the `cut` command does not modify the original file it acts on. Now, we would like to count how many types of entry came from each status type. Here we need the command `uniq -c` to count (hence the `-c`) how many unique appearances of each status type. However, `uniq -c` requires the input to be 'sorted', so the first step was to sort the list of status types.



```
facebookdata : bash
hellobigdata@bash:facebookdata$ csvcut -c 4 facebookdata.csv | sort | uniq -c
1404 link
481 photo
1 status_type
1337 video
hellobigdata@bash:facebookdata$
```

Output: # of FB status by each status type

Find the most popular status entry

An introduction to awk

To do this analysis efficiently, we'll use the command line language called `awk`, a tool that allows you to filter, extract and transform data files. `awk` is a very useful tool to put in your bag of tricks. To start, let's look at a very simple `awk` program to output every line of our `facebook.csv` file, where we specify the delimiter of the file (comma) using the `-F` option:

```
1 awk -F "," '{ print; }' facebookdata.csv
```

You should see the entire file being output to the screen. To only output the status ids (column 1), use the dollar sign (`$`) to denote columns as follows:

```
1 awk -F "," '{ print $1; }' facebookdata.csv | head
```

However, since the dataset has quoted ("text") cells we will use `csvcut` to extract the columns, e.g., we want to extract the column 1,8-15 into a file called `fbreactions.csv`. The idea is to sum-up all the reactions (columns 8 + ... + 15) on each FB status and then find the status which had the maximum number of reactions.

```
1 csvcut -c 2,8-15 facebookdata.csv > fbreactons.csv
```

```
facebookdata : bash
hellobigdata@bash:facebookdata$ head fbreactons.csv | csvlook
|-----|-----|-----|-----|-----|-----|-----|-----|
| status_id | num_comments | num_shares | num_likes | num_loves | num_wows | num_hahas | num_sads | num_angrys | |
|---|---|---|---|---|---|---|---|---|---|
| 7331091005 | 10154123560186006 | 178 | 461 | 5,488 | 43 | 13 | 19 | 0 | 2 |
| 7331091005 | 10154123362896006 | 1,932 | 3,158 | 10,385 | 96 | 15 | 1,499 | 0 | 2 |
| 7331091005 | 10154123319126006 | 270 | 400 | 1,971 | 28 | 47 | 16 | 0 | 7 |
| 7331091005 | 10154123234521006 | 11,811 | 43,923 | 107,561 | 1,202 | 4,334 | 2,424 | 628 | 394 |
| 7331091005 | 10154123219076006 | 999 | 2,978 | 8,833 | 28 | 932 | 188 | 473 | 21 |
| 7331091005 | 10154123189346006 | 897 | 2,525 | 29,660 | 398 | 26 | 2,009 | 3 | 15 |
| 7331091005 | 10154123132896006 | 628 | 90 | 1,173 | 43 | 11 | 8 | 7 | 19 |
| 7331091005 | 10154123136121006 | 283 | 819 | 8,776 | 348 | 8 | 120 | 1 | 4 |
| 7331091005 | 10154123102181006 | 147 | 917 | 12,771 | 433 | 29 | 23 | 2 | 57 |
hellobigdata@bash:facebookdata$
```

Extract all the reactions into a file fbreactons.csv

To calculate the total number of reactions on each entry (status), all we need to do is horizontally add up all the numbers from the columns #8-15 and we do this easily with awk, as follows:

```
1 $ awk -F "," '{ total = total + $2 + $3 + $4 + $5 + $6 + $7 + $8 + $9; print $1, " total; total=0 }' fbreactons.csv | head
```

Let's pay attention to the awk statement, which not only sums up the columns side by side, but also on each line prints two output (status id and total number of reaction on that row). Finally, at the end of each iteration, it nulls the total=0.

To get the status with max reactions, next, we sort the status ids, based on the number of reactions (column 2) using the sort -n -r -t"," -k 2 function, which tells the system to sort out the piped (|) output numerically (-n), on the column 2 (-k 2) which is delimited by a comma (,):

```
1 $ awk -F "," '{ total = total + $2 + $3 + $4 + $5 + $6 + $7 + $8 + $9; print $1, " total; total=0 }' fbreactons.csv | sort -n -r -t"," -k 2 | head -n 1
```

Final Output:


```
facebookdata: bash
hellobigdata@bash:facebookdata$ awk -F "," '{ total = total + $2 + $3 + $4 + $5 + $6 + $7 + $8 + $9; print $1," total; total=0 }' fbreactons .csv | head
status_id,0
7331091005_10154123560186006,6204
7331091005_10154123362896006,17087
7331091005_10154123319126006,2733
7331091005_10154123234521006,172277
7331091005_10154123219076006,14452
7331091005_10154123189346006,35533
7331091005_10154123132896006,1979
7331091005_10154123136121006,10359
7331091005_10154123102181006,14379
hellobigdata@bash:facebookdata$ awk -F "," '{ total = total + $2 + $3 + $4 + $5 + $6 + $7 + $8 + $9; print $1," total; total=0 }' fbreactons .csv | sort -n -r -t"," -k 2 | head -n 1
7331091005_10154089857531006,668121
hellobigdata@bash:facebookdata$
```

Total reactions on each status entry and the status id with the max number of reactions

The final output, tells us that the status id: 7331091005_10154089857531006 had the maximum number of reaction of total 668121.

If we now use grep, we can easily find the message which had the largest number of reactions.

```
1 cat facebookdata.csv | grep 7331091005_10154089857531006
```

```
facebookdata: bash
hellobigdata@bash:facebookdata$ awk -F "," '{ total = total + $2 + $3 + $4 + $5 + $6 + $7 + $8 + $9; print $1," total; total=0 }' fbreactons .csv | sort -n -r -t"," -k 2 | head -n 1
7331091005_10154089857531006,668121
hellobigdata@bash:facebookdata$ cat facebookdata.csv | csvcut -c 1,2 | grep 7331091005_10154089857531006 | csvlook
| 7331091005_10154089857531006 | LeBron and the Cavs are tired of being bullied |
| ..... | ..... |
hellobigdata@bash:facebookdata$
```

Output: FB Awk total find.

However, we want to make it more interesting! let's efficiently pipe all the steps shown above into a single command and find the message as follows:


```
1 $ cat facebookdata.csv | csvcut -c 2,8-15 | awk -F "," '{ total = total + $2 + $3 + $4 + $5 + $6 + $7 + \
2 $8 + $9; print $1,"total; total=0 }' | sort -n -r -t"," -k 2 | head -n 1
3 LeBron and the Cavs are tired of being bullied
```

It's still a large command, let's rather make a function called `fbfind()` which can just take a file as an argument and spell out the most popular message for us!

```
1 function fbfind() { cat $1 | csvcut -c 2,8-15 | awk -F "," '{ total = total + $2 + $3 + $4 + $5 + $6 + \
2 $7 + $8 + $9; print $1,"total; total=0 }' | sort -n -r -t"," -k 2 | head -n 1 ; }
```

Like all other “real” programming languages Bash has functions, but unfortunately in a somewhat limited implementation. The input argument is stored in the built-in variable `$1`, which does not get mixed with the built-in column variables in `awk` e.g., `$1-$15` our case!

Final output:



```
facebookdata: bash
hellobigdata@bash:facebookdata$ function fbfind() { cat $1 | csvcut -c 2,8-15 | awk -F "," '{ total = total + $2 + $3 + $4 + $5 + $6 + $7 +
$8 + $9; print $1,"total; total=0 }' | sort -n -r -t"," -k 2 | head -n 1 ; }
hellobigdata@bash:facebookdata$ fbfind facebookdata.csv
LeBron and the Cavs are tired of being bullied,668121
hellobigdata@bash:facebookdata$
```

Output: Institutes in the CA (California) state.

We will be able to run our Bash function as follows, instantly get the output:

```
1 $ fbfind facebookdata.csv
2 LeBron and the Cavs are tired of being bullied,668121
```

As you can see, one custom function written in just one line, found our desired out, the most popular message in our dataset (LeBron and the Cavs are tired of being bullied) which had a total of (668121) reactions!

Chapter Summary

In this project we have learned two more important aspects of the Bash: `functions` and the use of `awk` programming language in the context mining a csv formatted toy dataset consisting of Facebook messages and their statistics. We will re-use these commands in a more complicated format in the coming chapters.

Project 3: Best Australian Cities - Least Crimes

There's no doubt that Australia is one of the most beautiful places in the world! The land 'Down Under' has no shortage of opulent beaches, sweeping deserts, dense forests, and endless valleys, but for an overseas visitor, these areas are just waiting to be uncovered. Australia comprises the mainland of the Australian continent, the island of Tasmania and numerous smaller islands. It is the world's sixth-largest country by total area.

However, as soon as you decide to visit a country, the first question that comes into your mind is what would be the safest place/city to visit? In each country, each city has its own characteristics, crime rates, etc. and the same is valid for Australian cities also.

Fortunately, Australia has a number of federal agencies that have an enforcement role, these can be broken into law enforcement agencies and regulatory agencies. In particular, there are two distinct, but similar levels of police force, the various state police forces and the Australian Federal Police (AFP). The state police forces are responsible for enforcing state law within their own states while the AFP are responsible for the investigation of crimes against Commonwealth law which occurs throughout the nation.

In this project, mining a historical dataset (2013, time range unknown) provided by the AFP we will address the following questions:

- What is the top most crime type in Australia by the total number of crimes in all major cities?
- Same question as above, but this time, per city?
- What is the average number of crimes, per city?
- Which city had the least crimes? or, in other words, what's the best city in Australian terms of crimes?

Even though the dataset is not big, mining this (toy) we will be able to develop our initial skills in Bash and .csv data processing.



Learning objectives

By completing this, you will learn to use the following Bash commands:

- Advanced use of the Bash commands: `cat`, `head`, `tail`, `sort`, `wc`;
- Bash function creation;
- Bash `if-else`;
- Bash `for-loop`; and `a`
- `sed`, `awk` and the use of regular expressions.

The chapter ends with a full bash script that we use to find the best city in Australia in terms of number of crimes that happened pre-2013.

Before we go any further, let's setup our working environment by creating a subfolder on our Desktop, where we will store our analysis files. To do so, let's first fire up a command line and navigate to our analysis folder:

Setup the working directory

```
1 cd ~/Desktop
2 mkdir afpdata
3 cd afpdata
```

This will create a folder `afpdata` on your Desktop. Next, we'll download our dataset from the `data.gov.au` web portal, which provides an easy way to find, access and reuse public datasets from the Australian Government. To learn more, go to <https://data.gov.au/about>. Generally, material presented on this website is provided under a Creative Commons Attribution 3.0 Australia licence. In this project, we collect and use a dataset originally contributed by the AFP - Australian Federal Police in the `data.gov.au`. Once you visit the portal, click on search with the string `"afp crime"`, you will get a link to download the data.



Data download

You should download the data from the book web page, as we have slightly simplified the data.

Data Preview

Assuming you have saved your downloaded file as `tmpB2ZP9eAFP---Crime-incidents-data.xls`, the first thing you want to perform is to convert the data into the `.csv` format. Type the following command in the terminal:

```
1 libreoffice --headless --convert-to csv *.xls
```

This command, using command line interface of the LibreOffice (which comes free with most of the Linux systems) converts all Excel (`*.xls`) files in the current folder to `.csv`. Let's also simplify the name of the file with `mv` command:

```
1 mv tmpB2ZP9eAFP---Crime-incidents-data.xls afpdata1.csv crimedata-au.csv
```

Awesome! next what we'll do is get a sneak peak of our data. As a quick reminder, the `cat` command shows us all the lines in a file.

```
1 cat crimedata-au.csv
```

You should see the following output:

```

afpdata: bash
hellobigdata@bash:afpdata$ cat crimedata-au.csv
,,,,,,
,,,,,,
,,,,,,
,,,,,,
,,,,,,
Case Incident Type,National,Adelaide,Brisbane,Cairns,Darwin,Hobart,Melbourne,Perth,Sydney,Total
Drugs ? Imported,8,9,48,4,10,0,121,43,191,434
Drugs ? Exported,0,0,0,1,0,0,0,0,0,1
Drugs ? Trafficked,1,0,1,0,0,0,1,0,2,5
People Smuggling,30,1,0,1,2,0,1,10,1,46
Transnational ? Sexual Servitude,0,0,2,0,0,0,2,1,0,5
Transnational Organised Crime,0,0,1,0,0,0,0,0,0,1
Weapons ? Trafficked,6,0,1,0,0,0,2,0,0,9
Slavery/Human Trafficking,1,1,1,0,1,0,3,4,6,17
Corruption,12,2,2,1,1,1,5,3,5,32
Information And Communications Technology,14,0,5,0,3,1,8,2,5,38
Corporate Or Bankruptcy,0,2,1,0,0,0,1,1,1,6
Counterfeit Currency,2,0,2,0,1,0,0,20,1,26
Environmental Crime,2,0,0,2,0,0,2,1,0,7
Fraud,42,20,48,6,5,10,60,27,63,281
War Crimes,0,1,0,0,0,0,1,0,2,4
Electoral Crime,0,0,0,0,0,0,2,0,0,2
Intellectual Property,1,1,1,0,0,0,12,1,7,23
Identity Crime,1,2,4,0,0,0,5,7,13,32
Migration Crime,1,1,4,1,0,0,5,8,10,30
Illegal Fishing,0,0,0,2,0,0,0,0,0,2
Emerging Crime,28,2,4,1,0,0,7,0,6,48
Offences Commonwealth Property/Premises,0,0,0,1,0,0,3,1,0,5

```

Output: cat crimedata-au.csv

As you can see, there are five rows with no data, there occupied by a series of commas (,). We want to wipe-out these rows using SED - Stream Editor. Let's run the following command:

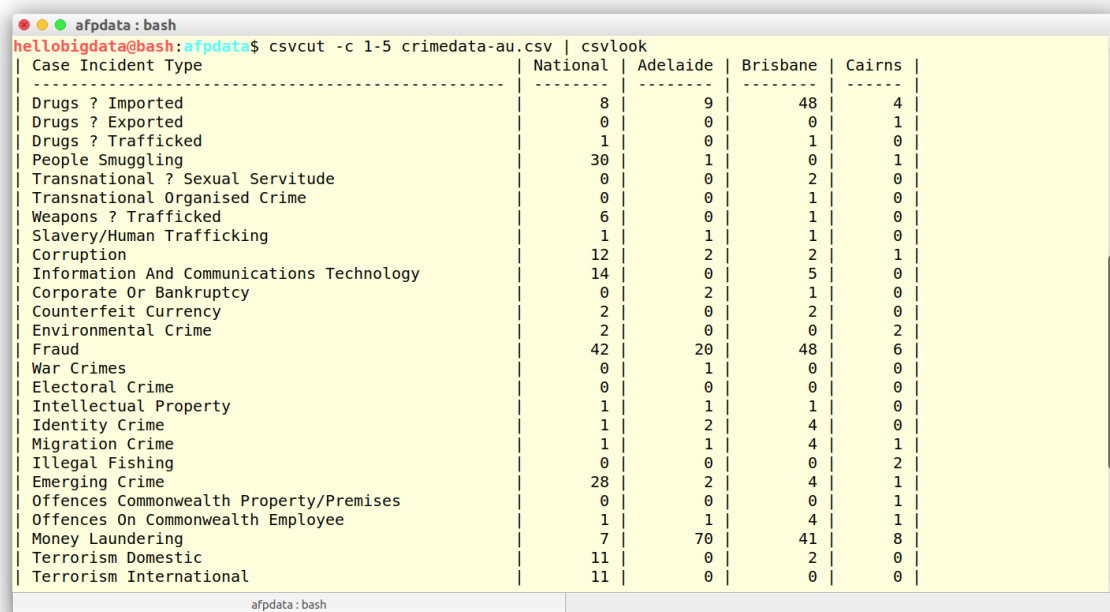
```
1 sed -i '1,5d' crimedata-au.csv
```

This will in-place (due to the option -i) delete the first five lines from the file.

However, before we show the data again, we want to install a tool called `csvkit`. Once installed correctly, we should be able to run the following command:

```
1 csvcut -c 1-5 crimedata-au.csv | csvlook
```

The first command (`csvcut`) cuts out the first five columns and the pipes the output to the `csvlook` command:



```
afpdata: bash
hellobigdata@bash:afpdata$ csvcut -c 1-5 crimedata-au.csv | csvlook
```

Case Incident Type	National	Adelaide	Brisbane	Cairns
Drugs ? Imported	8	9	48	4
Drugs ? Exported	0	0	0	1
Drugs ? Trafficked	1	0	1	0
People Smuggling	30	1	0	1
Transnational ? Sexual Servitude	0	0	2	0
Transnational Organised Crime	0	0	1	0
Weapons ? Trafficked	6	0	1	0
Slavery/Human Trafficking	1	1	1	0
Corruption	12	2	2	1
Information And Communications Technology	14	0	5	0
Corporate Or Bankruptcy	0	2	1	0
Counterfeit Currency	2	0	2	0
Environmental Crime	2	0	0	2
Fraud	42	20	48	6
War Crimes	0	1	0	0
Electoral Crime	0	0	0	0
Intellectual Property	1	1	1	0
Identity Crime	1	2	4	0
Migration Crime	1	1	4	1
Illegal Fishing	0	0	0	2
Emerging Crime	28	2	4	1
Offences Commonwealth Property/Premises	0	0	0	1
Offences On Commonwealth Employee	1	1	4	1
Money Laundering	7	70	41	8
Terrorism Domestic	11	0	2	0
Terrorism International	11	0	0	0

Formatted first five lines of the crimedata-au.csv

Finding the number of rows and columns

Now we want to get some some statistics. Let's first find the total number of columns in the file. There are two ways to do this, let's first check the hard way:

The hard way

Count columns:

Step 1. First get only the first row using head command:

```
1 $ head -1 crimedata-au.csv
2 Case Incident Type,National,Adelaide,Brisbane,Cairns,Darwin,Hobart {...}
```

Step 2. Next use sed to remove everything except commas. To do this, we use the regex (regular expression) pattern `[^,]`, See the appendix for more details on the regex.

```
1 $ head -1 crimedata-au.csv | sed 's/[^,]//g'
```

Step 3. All what has left is to simply use wc command to count number of characters (commas).

```
1 $ head -1 crimedata-au.csv | sed 's/[^,]//g' | wc -c
2 $ 11
```

Count rows:

Now, let's find the total number of rows in the file well:

```
1 $ cat crimedata-au.csv | wc -l
2 41
```

The easy way

As the previous chapter, we could also use the `csvkit`'s command `csvstat` to get the same stats, but with much simpler way:

A terminal window titled 'afpdata: bash' showing the execution of the 'csvstat' command. The first command is 'csvstat -n crimedata-au.csv', which outputs a list of 11 columns: '1: Case Incident Type', '2: National', '3: Adelaide', '4: Brisbane', '5: Cairns', '6: Darwin', '7: Hobart', '8: Melbourne', '9: Perth', '10: Sydney', and '11: Total'. The second command is 'csvstat --count crimedata-au.csv', which outputs 'Row count: 40'. The prompt 'hellobigdata@bash:afpdata\$' is visible before each command.

```
afpdata: bash
hellobigdata@bash:afpdata$ csvstat -n crimedata-au.csv
1: Case Incident Type
2: National
3: Adelaide
4: Brisbane
5: Cairns
6: Darwin
7: Hobart
8: Melbourne
9: Perth
10: Sydney
11: Total
hellobigdata@bash:afpdata$ csvstat --count crimedata-au.csv
Row count: 40
hellobigdata@bash:afpdata$
```

Output: Use `csvstat` to find #rows and #cols.

Note that the `csvstat` is smarter than `wc -l` as it omitted the titles line and counted only the data rows!

Data Analysis

Finding the top most crime in the whole country

To find the top most crime that happened the highest number of times in Australia, we want to sort our `crimedata-au.csv` file by the `Total` column (column 11).

As mentioned earlier, Bash has a command just for doing that, and as you may have guessed, it's called `sort`. However, running `sort` on the `.csv` will output all lines of the file on the screen! which is not essentially very useful.

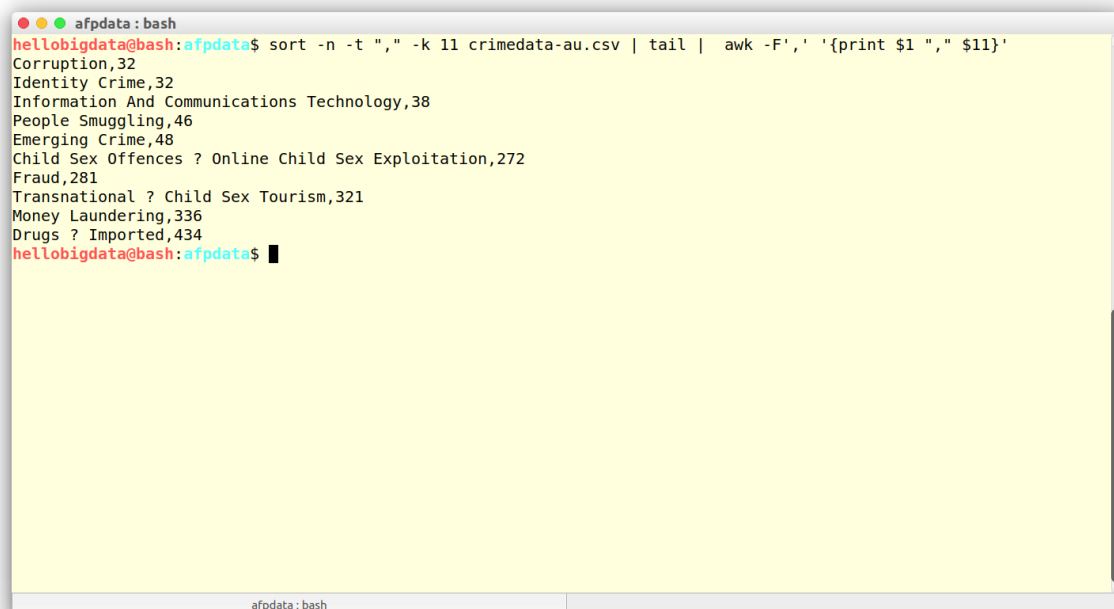
Instead, we would like to sort the file, and then only show the first few lines of the sorted result. To do this, we'll need to use pipes (`|`) as we did before. As a quick reminder, pipes are convenient because they allow you to move the output of a command into the input of another command, without saving the intermediate result to a file.

To sort our comma-separated file by the Total column, which is the 11th column (see above), enter the following command:

```
1 sort --numeric-sort -t "," --key 11 crimedata-au.csv | tail
```

This looks scary but don't worry, it's simply a `sort` command, but with a few additional command-line options, so let's break it down!

The `--numeric-sort` informs Bash to sort numerically (otherwise, it will sort alphabetically!). Next, the `-t ","` specifies that the columns of our file are defined by a comma, and `--key 11` specifies that we want to sort only on column 11 (note: `--key 10,11` would combine the data in column 10 and 11 for the sorting).



```
afpdata: bash
hellobigdata@bash:afpdata$ sort -n -t "," -k 11 crimedata-au.csv | tail | awk -F',' '{print $1 "," $11}'
Corruption,32
Identity Crime,32
Information And Communications Technology,38
People Smuggling,46
Emerging Crime,48
Child Sex Offences ? Online Child Sex Exploitation,272
Fraud,281
Transnational ? Child Sex Tourism,321
Money Laundering,336
Drugs ? Imported,434
hellobigdata@bash:afpdata$
```

Output: Finding the total number of crimes, per crime type.

In this case, the column of our interest is the last column so using `--key 11` is equivalent to `--key 11,11`, but had our column of interest been column 10, using `--key 10` instead of `--key 10,10` would sort the data starting at column 10 till the end of the line, and therefore yield incorrect results!

Now, using `awk` (see tutorials for details) you can only print the first column, which has the names of the crimes and the last or 11th column (total number crimes for that crime type):

```
1 $ sort -n -t "," -k 11 crimedata-au.csv | tail | awk -F',' '{print $1 "," $11}'
```

In the `awk` call `-F','` tell that the separator is a comma. Also note that we have shorthanded the `--numeric-sort` with `-n` and `--key` with `-k`. Now, if you want just the first column's last row, which means the top most crime!

```
1 $ sort -n -t "," -k 11 crimedata-au.csv | tail -n 1 | awk -F',' '{print $1 " ", " $11}'
```

This will capture or `tail` the last line (`tail -n 1`) and print the first column:

```
1 $ Drugs ? Imported, 434
```

Finding the top most crime per city

```
afpdata: bash
hellobigdata@bash:afpdata$ cat crimedata-au.csv | csvcut -c "Case Incident Type","Sydney" | sort -n -r -t "," -k 2
Drugs ? Imported,191
Money Laundering,83
Fraud,63
Child Sex Offences ? Online Child Sex Exploitation,54
Identity Crime,13
Migration Crime,10
Intellectual Property,7
Terrorism Domestic,6
Slavery/Human Trafficking,6
Emerging Crime,6
Information And Communications Technology,5
Corruption,5
Offences On Commonwealth Employee,4
Child Sex Offences ? Not Child Sex Tourism,4
War Crimes,2
Threats,2
Terrorism Financing,2
Drugs ? Trafficked,2
Weapons/Prohibited Items In The Aviation Environ.,1
Transnational ? Child Sex Tourism,1
Theft ? General,1
Terrorism International,1
People Smuggling,1
Counterfeit Currency,1
Corporate Or Bankruptcy,1
Weapons ? Trafficked,0
Transnational ? Sexual Servitude,0
Transnational Organised Crime,0
```

Output: Finding the total number of crimes, for Sydney.

Our next compute quest is to find the total number of crimes, given a city name.

This essentially will require us to compute the summation of a column which contains the city names. Ideally, we would like to have a function like `top_crime ()` that would take an argument (e.g., Sydney) and tell us the top crime for that city. Let's start with a basic version first:

```
1 $ cat crimedata-au.csv | csvcut -c "Case Incident Type","Sydney" | sort -n -t "," -k 2
```

Note that the `csvcut` function can take column titles with the `-c` option. Now, if we just want the top crime we add `tail -n 1` at the end, as follows:

```
1 $ cat crimedata-au.csv | csvcut -c "Case Incident Type","Sydney" | sort -n -t "," -k 2 | tail -n 1
```

Output:

```
1 Drugs ? Imported,191
```

Now, let's build our desired function. For knowing more about bash function, please refer to the tutorials section.

```

1 $ function topcrime() { cat $1 | csvcut -c "Case Incident Type",$2 | sort -n -t "," -k 2 | tail -n 1 | \
2 awk -F',' '{print $1 "=" $2}' ; }

```

Example run:

```

1 $ topcrime crimedata-au.csv Sydney
2 Drugs ? Imported= 191

```

Note that the first (crimedata-au.csv) and second (city name: Sydney) arguments were automatically placed into the \$1 and \$2 variables inside the function, however inside awk's call \$1 and \$2 represented the first (crime type) and second (number of crimes) columns of the piped output.

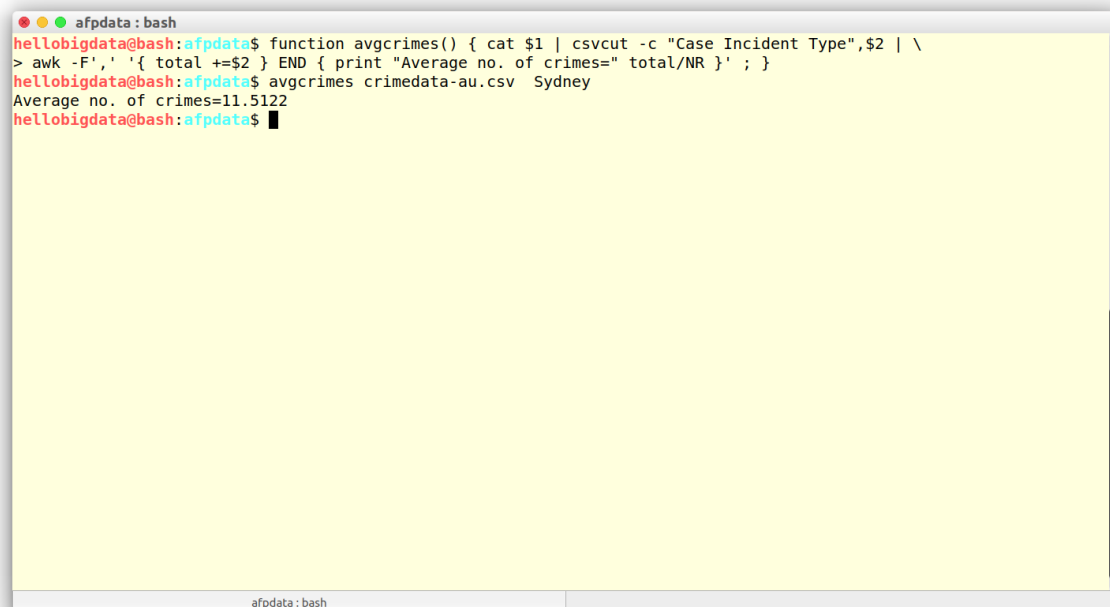
Now, we can slightly modify the above function so that it can also compute, given any city name, the average number of crimes.

```

1 $ function avgcrimes() { cat $1 | csvcut -c "Case Incident Type",$2 | awk -F',' '{ total += $2 } END { \
2 print "Average no. of crimes=" total/NR }' ; }

```

Here, everything remains as above, except there's no sort function and the awk function computes the sum of crimes (col 2) and then divide the sum by the total number of rows (NR).



```

afpdata: bash
hellobigdata@bash:afpdata$ function avgcrimes() { cat $1 | csvcut -c "Case Incident Type",$2 | \
> awk -F',' '{ total += $2 } END { print "Average no. of crimes=" total/NR }' ; }
hellobigdata@bash:afpdata$ avgcrimes crimedata-au.csv Sydney
Average no. of crimes=11.5122
hellobigdata@bash:afpdata$

```

Output: Finding the average number of crimes, for Sydney

```

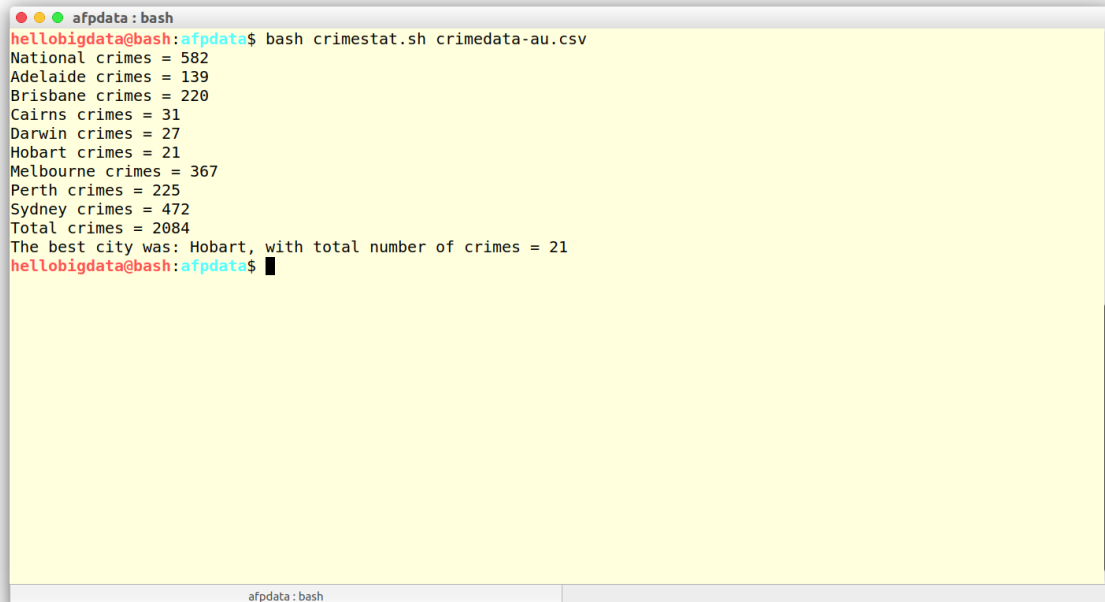
1 $ avgcrimes crimedata-au.csv Sydney
2 Average no. of crimes=11.5122

```

Finding the best city in Australia!

Our final quest in this project is to find the best city in Australia that had the least number of crimes over that pre-2013 unknown period. Essentially, we would need to find summation of the city columns and then compare the sums to find the city with the lowest total sum!

To develop the code, we will reuse the average function (sum part) developed above and call it inside a loop. Some important points:



```
afpdata: bash
hellobigdata@bash:afpdata$ bash crimestat.sh crimedata-au.csv
National crimes = 582
Adelaide crimes = 139
Brisbane crimes = 220
Cairns crimes = 31
Darwin crimes = 27
Hobart crimes = 21
Melbourne crimes = 367
Perth crimes = 225
Sydney crimes = 472
Total crimes = 2084
The best city was: Hobart, with total number of crimes = 21
hellobigdata@bash:afpdata$
```

Output: Which one's the best city down under in terms of crime? Hobart.

- The first line in the script (`crimestat.sh`) indicates the system (`bash`) which program to use to run the file. If you get something like `./crimestat.sh: Command not found`. Probably the first line `'#!/bin/bash'` is wrong, issue `whereis bash` to see how should you write this line. Also, do not forget to issue `chmod +x crimestat.sh` to enable execution right on the script for you!
- The function `totalcrimes()` is no different from the `avgcrimes()` explained above, just that in the `awk` part we do not divide `total` crimes by the number of rows.
- We read the first line of the input file (stored in the variable `$1`) as an array and store in the `fields` variable.
- In the `for`-loop part, we iterate over the field titles (`field: city names`), except the first first field, which is not a city!
- Inside the `for`-loop, we find the city that had the lowest number total crimes, comparing with the previously stored maximum number of crimes, using a simple logic of find the lowest number from a set of given numbers.

Now, see the full code below:

```

1  #!/bin/bash
2
3  # totalcrimes() function to find summation of cols, given the city (column) names
4  function totalcrimes() { cat $1 | csvcut -c "Case Incident Type",$2 | awk -F',' '{ total += $2 } END { \
5  print total }' ; }
6
7  # Read the first line
8  IFS=' ' read -r fields < $1
9
10 # Set some variables
11 FIRST=1
12 CRIME_MAX=99999
13
14 # Loop over the comma separated cities, stored in the array called $fields
15 IFS=","
16 for i in $fields
17 do
18     if [ $FIRST -eq 1 ]
19     then
20         FIRST=0
21     else
22         totalcrime=$(totalcrimes $1 $i)
23         echo "$i crimes = $totalcrime"
24
25         #Check if the total number of crimes is less the MAX value
26         if [ $totalcrime -lt $CRIME_MAX ]; then
27             bestcity=$i
28             CRIME_MAX=$totalcrime
29         fi
30     fi
31 done
32
33 echo "The best city was: $bestcity, with total number of crimes = $CRIME_MAX"

```

If you have already set execute permission on the file, run the script as follows:

```

1  $ ./crimestat.sh crimedata-au.csv | tail -n 1
2  The best city was: Hobart, with total number of crimes = 21

```

Chapter Summary

In this project we have learned two more important aspects of the Bash: functions, the use of `awk` programming language and more importantly a full-length bash shell script along with loops and conditional statement, in the context mining a csv formatted toy dataset consisting of Australian cities and their crime statistics. We will re-use these commands in a more complicated format in the coming chapters.

Project 4: Mining Shakespear-era Plays and Poems

In this project, we utilise a text corpus containing plays and poems from the Shakespeare-era (16th and 17th centuries) and find which are the words most frequently used by some of the known authors (e.g., Shakespeare) of that time!

You may wonder to know that, so far there is no comprehensive collection of electronic texts of these works in the public domain, rather a portion of the plays and poems are held in an machine readable archive at the Centre for Literary and Linguistic Computing at The University of Newcastle.

This has been assembled over many years by editing versions available from commercial online collections or from other sources such as keyboarding from early printed versions. Later, by developing and using a software tool called Intelligent Archive (IA) by [Craig and Whipp](#) identified in total a set of approximately 66,907 unique words in the 256 texts. IA calculated the frequency of each of the aforementioned 66,907 words in each work and stored the final outcome in the form of a 66,907×256 matrix, which is freely downloadable from a relevant [publication](#)'s supporting material.



Learning objectives

By completing this, you will learn to use the following Bash commands:

- Advanced use of the Bash commands: cat, head, tail, sort, wc, paste;
- Bash function creation;
- sed, awk and the use of regular expressions (complex examples).



Data download

You should download the data from the book webpage, as we have slightly simplified the data.

Let's first setup a command line and navigate to our analysis folder:

Setup the working directory

```
1 cd ~/Desktop
2 mkdir playsandpoemsdata
3 cd playsandpoemsdata
```

This will create a folder playsandpoems on your Desktop. Next, we'll download our dataset:

Data Preview

As in Chapter 1, the first thing we will do is get a sneak peak of our data. As a quick reminder, the `cat` command shows us all the lines in a file, the `cut` command given the `-f` option cuts out the desired columns (e.g., 1-3 in this case), the `head` command will show us the first few lines specified in the `-n` option and the `csvlook` command, as mentioned in the previous chapter, makes the output readable! Assuming you saved your file as `plays_and_poems_stat.csv`, type the following command in the terminal:

```
1 $ cat plays_and_poems_stat.csv | cut -f 1-3 | head -n 30 | csvlook
```

tokens	Knight_of_the_Burning_Pestle__play__Beaumont	Maids_Tragedy__play__Beaumont_and_Fletcher
the	2.630...	2.256...
and	3.743...	2.709...
i	3.478...	4.093...
of	1.791...	1.244...
a	1.853...	1.625...
you	2.018...	2.016...
is	1.294...	1.602...
my	1.493...	1.738...
to_infinite__	0.862...	1.456...
in_preposition__	1.000...	0.886...
it	1.265...	1.729...
to_preposition__	0.801...	0.909...
not	0.986...	1.423...
me	1.023...	1.668...
with	0.810...	0.740...
will_verb__	1.161...	1.225...
be	0.867...	0.937...
your	0.886...	0.692...
but	0.720...	0.876...
his	0.715...	0.424...
he	0.862...	0.692...
this	0.787...	1.079...
have	0.881...	0.919...
for_preposition__	0.777...	0.537...
as	0.474...	0.678...
all	0.649...	0.721...
that_relative__	0.493...	0.716...
what	0.431...	0.523...

Output: cat plays_and_poems_stat.csv

As you can see from the data snapshot above each column title, except the first column has a pattern and contains three pieces of information, separated by three underscores (`__`).

Example:

```
1 Knight_of_the_Burning_Pestle__play__Beaumont
```

- **Text name** : Knight of the Burning Pestle
- **Type** : Play; and
- **Author**: Beaumont

We can effectively use this knowledge to extract various stats from the data. See some examples below:

Analysis

How many plays/poems?

If we want to find there are how many plays or poems in the dataset, the first thing we need to do is - take out the first line of the file using `head` and then replace all the commas (,) with newline (\n) characters. We do the replacement using a regular expression (\ '\$'\n) inside `sed` tool.

Let's run the following:

```
1 $ head -n 1 plays_and_poems_stat.csv | sed -e 's/,/\ '$'\n/g'
```

This command will convert the first line (all column titles) into a single column

```
1 Knight_of_the_Burning_Pestle___play___Beaumont
```

```
playsandpoemsdata : bash
hellobigdata@bash:playsandpoemsdata$ head -n 1 plays_and_poems_stat.csv | sed -e 's/,/\ '$'\n/g'
tokens
Knight of the Burning Pestle__play__Beaumont
Maids Tragedy__play__Beaumont and Fletcher
Virtuous Octavia__play__Brandon Samuel
Jovial Crew__play__Brome
Two Books of Airs Book 1__poem__Campion
Poems 1640 first 25__poem__Carew
Mariam__play__Carey
All Fools__play__Chapman
Andromeda Liberata__poem__Chapman
Blind Beggar of Alexandria__play__Chapman
Bussy dAmbois__play__Chapman
Byrons Conspiracy__play__Chapman
Byrons Tragedy__play__Chapman
Caesar and Pompey__play__Chapman
Chapman Iliads 1 3__poem__Chapman
Epicede__poem__Chapman
Eugenia__poem__Chapman
Euthymiae__poem__Chapman
Gentleman Usher__play__Chapman
Hero and Leander Ch__poem__Chapman
Humorous Days Mirth__play__Chapman
Hymnus in Cynthia__poem__Chapman
Hymnus in noctem__poem__Chapman
May Day__play__Chapman
Monsieur dOlive__play__Chapman
Revenge of Bussy__play__Chapman
Sir Giles Goosecap__play__Chapman
Widows Tears__play__Chapman
Hoffman__play__Chettle
```

Convert the title row to a column

Now, we should be able to `grep` the text types (play/poems) from the output. For example, if we want to grab the “poems”, we execute the following:


```
1 $ head -n 1 plays_and_poems_stat.csv | sed -e 's/,/\''$'\n/g' | grep "poem"
```

```

playsandpoemsdata: bash
hellobigdata@bash:playsandpoemsdata$ head -n 1 plays_and_poems_stat.csv | sed -e 's/,/\''$'\n/g' | grep "poem"
Two Books of Airs Book 1 poem__Campion
Poems 1640 first 25 poem__Carew
Andromeda Liberata poem__Chapman
Chapman Iliads 1 3 poem__Chapman
Epicede poem__Chapman
Eugenia poem__Chapman
Euthymiae poem__Chapman
Hero and Leander Ch poem__Chapman
Hymnus in Cynthiam poem__Chapman
Hymnus in noctem poem__Chapman
Delia poem__Daniel
Holy Rood poem__Davies_John
Humours Heaven poem__Davies_John
Mirum in Modum poem__Davies_John
Select Second Husband poem__Davies_John
Summa Totalis poem__Davies_John
Tears of the Muses Da poem__Davies_John
Wits Pilgrimage poem__Davies_John
First Anniversary poem__Donne
Poems section poem__Donne
Second Anniversary poem__Donne
Endimion and Phoebe poem__Drayton
Christs Bloody Sweat poem__Ford
Fames Memorial poem__Ford
Funeral Elegy by WS poem__Ford
Treatise on Religion poem__Greville_Fulke
Church Militant poem__Herbert_George
Church Porch poem__Herbert_George
The Temple 20 poems poem__Herbert_George
Sonnets 44 to 63 poem__Herbert_Mary

```

Output: grep poems from the plays_and_poems_stat.csv data

Finally, we count the number of lines, which will give us the total number of “poems” in the datasets.

```
1 $ head -n 1 plays_and_poems_stat.csv | sed -e 's/,/\''$'\n/g' | grep "poem" | wc -l
2 54
```

Using the same command, we can also extract the works by “Shakespeare” or any other authors in the dataset, easy!

```
1 $ head -n 1 plays_and_poems_stat.csv | sed -e 's/,/\''$'\n/g' | grep "Shakespeare" | wc -l
2 40
```

How many plays/poems by each author?

Well, so far we do not now how many authors are there in the dataset and also their name’s spellings. Therefore, it is not feasible to insert each authors names in the command shown above. We need to do a smarter, but slightly complicated approach!

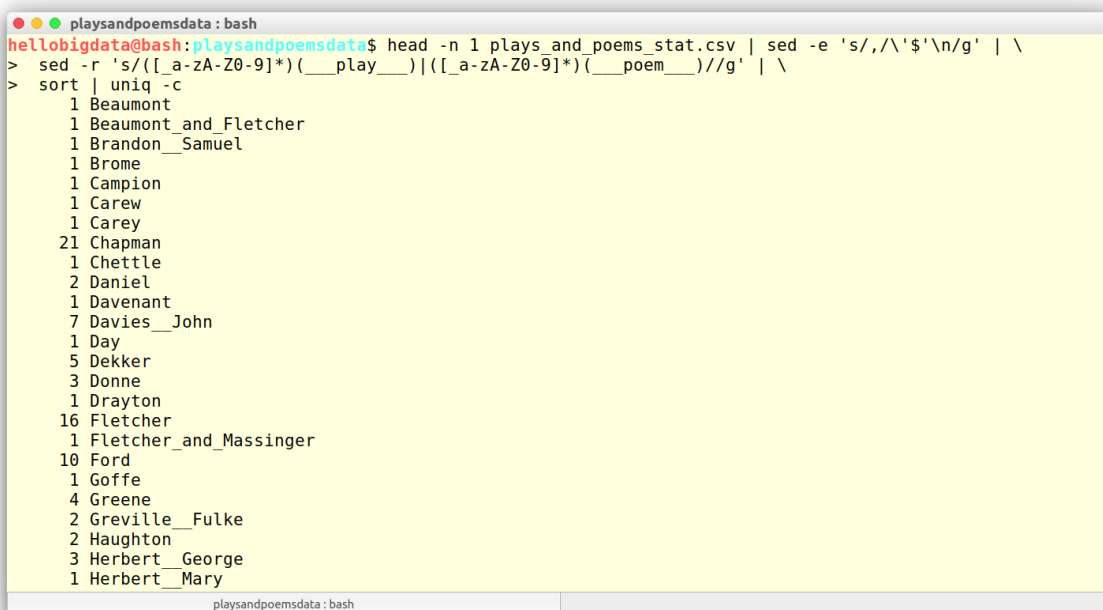
- **Step 1:** Convert the first line into a column, `$head -n 1 plays_and_poems_stat.csv | sed -e 's/,/\''$'\n/g'`
- **Step 2:** For each line, remove all the bits that appeared before the (___play___) and (___poem___), including them. This way we get the raw ‘author names’: `sed -r 's/([_a-zA-Z0-9]*)(___play___)|([_a-zA-Z0-9]*)(___poem___)/g'`

- **Step 3:** Find the unique appearances of each author `sort | uniq -c`. Note that it is a requirement that you need to run `sort`, before you call the `uniq`. We use `uniq -c`, because it will then not only “condense” the neighboring lines that are the same, but also count how many of each are seen!

Now let’s combine all the steps into a piped (|) command as the steps need to appear in a tandem:

```
1 $ head -n 1 plays_and_poems_stat.csv | \
2 sed -e 's/,/\ '$'\n/g' | \
3 sed -r 's/([_a-zA-Z0-9]*)(___play___)|([_a-zA-Z0-9]*)(___poem___)//g' | \
4 sort | uniq -c
```

Notice that the OR (|) in the `sed` statement does not get affected by the pipes (|).



```
playsandpoemsdata: bash
hellobigdata@bash:playsandpoemsdata$ head -n 1 plays_and_poems_stat.csv | sed -e 's/,/\ '$'\n/g' | \
> sed -r 's/([_a-zA-Z0-9]*)(___play___)|([_a-zA-Z0-9]*)(___poem___)//g' | \
> sort | uniq -c
 1 Beaumont
 1 Beaumont_and_Fletcher
 1 Brandon__Samuel
 1 Brome
 1 Campion
 1 Carew
 1 Carey
21 Chapman
 1 Chettle
 2 Daniel
 1 Davenant
 7 Davies__John
 1 Day
 5 Dekker
 3 Donne
 1 Drayton
16 Fletcher
 1 Fletcher_and_Massinger
10 Ford
 1 Goffe
 4 Greene
 2 Greville__Fulke
 2 Haughton
 3 Herbert__George
 1 Herbert__Mary
```

Number of contributions by each author

We can also add a reverse (-r) numeric sort on the number of works (i.e., the first column), but at the end of the above command. This will definitely make the output more interesting!

```
1 $ head -n 1 plays_and_poems_stat.csv | \
2 sed -e 's/,/\ '$'\n/g' | \
3 sed -r 's/([_a-zA-Z0-9]*)(___play___)|([_a-zA-Z0-9]*)(___poem___)//g' | \
4 sort | uniq -c | sort -n -r -k1
```

Note that the final output is showing that Shakespeare only contributed 31 works, whereas in the previous example we computed 40! The numbers are different because in the previous example we have used `grep` and counted all the lines that have the string “Shakespeare” which also included the works by “Shakespeare and others” in the context.

```

playsandpoemsdata: bash
hellobigdata@bash:playsandpoemsdata$ head -n 1 plays_and_poems_stat.csv | sed -e 's/,/\ '$'\n/g' | \
> sed -r 's/([_a-zA-Z0-9]*)(__play__)([_a-zA-Z0-9]*)(__poem__)/g' | \
> sort | uniq -c | sort -nr -k1
31 Shakespeare
21 Chapman
19 Jonson
18 Middleton
16 Fletcher
14 Uncertain
10 Ford
8 Lyly
7 Spenser
7 Marlowe
7 Davies__John
6 Heywood
5 Peele
5 Dekker
4 Webster
4 Marston
4 Greene
3 Wilson
3 Unknown
3 Shirley
3 Shakespeare_and_others
3 Shakespeare_and_Middleton
3 Herbert__George
3 Donne
2 Shakespeare_and_Fletcher

```

Contribution by each author, sorted

What are the most frequent words?

Given a text, what are the most frequent words?

Finding the most frequent words for a given text (e.g., `Knight_of_the_Burning_Pestle`) is easy, we can build a function `toptokens()`, which is nothing but the `topcrimes()` function developed in our previous project:

```

1 function toptokens() { cat $1 | \
2 csvcut -c "tokens",$2 | \
3 sort -nr -t "," -k 2 | \
4 head -n 20 | \
5 awk -F',' '{print $1 " ", $2}' ; }

```

For example, if we want to grab the most frequent words in the `Romeo_and_Juliet` play, we can execute the following:

```

1 $ toptokens plays_and_poems_stat.csv Romeo_and_Juliet__play__Shakespeare | csvlook

```

```

playsandpoemsdata: bash
hellobigdata@bash:playsandpoemsdata$ function toptokens() { cat $1 | csvcut -c "tokens",$2 | sort -nr -t "
," -k 2 | head -n 20 | awk -F',' '{print $1 "," $2}' ; }
hellobigdata@bash:playsandpoemsdata$ toptokens plays_and_poems_stat.csv Romeo_and_Juliet__play__Shakespe
are | csvlook
| and | 2.7536173 |
|-----|-----|
| the | 2.704... |
| i | 2.651... |
| is | 1.917... |
| a | 1.830... |
| of | 1.587... |
| my | 1.467... |
| in_preposition__ | 1.253... |
| you | 1.208... |
| it | 1.162... |
| thou | 1.150... |
| me | 1.092... |
| to_infinite__ | 1.088... |
| not | 1.068... |
| to_preposition__ | 1.026... |
| with | 0.985... |
| will_verb__ | 0.932... |
| this | 0.923... |
| be | 0.866... |
| but | 0.742... |
hellobigdata@bash:playsandpoemsdata$

```

The top 20 frequent words in the work “Romeo and Juliet”

Given an author, what are the most frequent words?

This is slightly complicated! because we again need to perform several steps:

- For the given author, trim out the plays/ poems names, including text types (i.e., plays | poems)
- Combine all the columns, i.e., sum horizontally the frequencies of words for all the texts of that author
- Sort the words, based on the accumulated frequencies on all works by that author.

Don't be scared! we will take you there.

Step 1. Trim out the plays/ poems names, for a given author:

Let's consider that the author in question is Shakespeare. The following `awk` based regular expression will trim out all the bit before the name of the author. If you look closely, you will see that inside the `sed` regex, it's actually finding the pattern of plays OR (|) poems names that end with the string “Shakespeare” and then replacing inplace (due to the `-i -r`) the whole matched pattern e.g., `Romeo_and_Juliet__play__Shakespeare` with the string `Shakespeare`:

```

1 $ sed -i -r 's/([_a-zA-Z0-9]*)(__play__)(Shakespeare)| \
2 ([_a-zA-Z0-9]*)(__poem__)(Shakespeare)/Shakespeare/g' \
3 plays_and_poems_stat.csv

```

At this stage, we have a file, where all the Shakespeare works have renamed to “Shakespeare”.

Step 2 Separate all the works of “Shakespeare”

In this step, we build a function (`colcut()`), which, given the column title (e.g., “Shakespeare”) spit out all the columns with that title including the first column (tokens), which we will write

onto a file (Shakespeare.csv). Also note the use of the new command `paste`, which merges lines of files and writes to standard output lines consisting of sequentially corresponding lines of each given file.

```
1 function colcut() { cut -f 1, $(head -1 $1 | sed 's/,./\'$'\n/g' | \
2 grep -n "$2" | \
3 cut -f1 -d: | \
4 paste -sd",") \
5 -d, $1 ; }
```

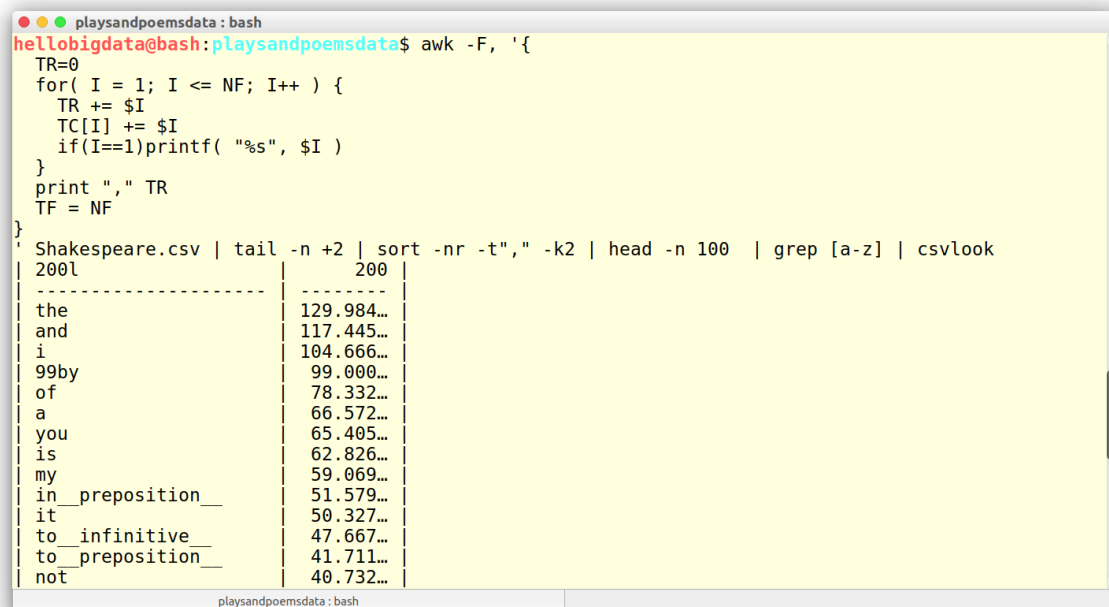
We use this function as follows:

```
1 colcut plays_and_poems_stat.csv Shakespeare > Shakespeare.csv
```

Note that we can not use `csvcut` because it can not handle multiple columns with 'same' title, which is our case (Shakespeare).

Step 3. Combine/sum horizontally all the columns with same titles (e.g., Shakespeare).

Finally, our final bit of code looks like below. We apply the following `awk` code to the `Shakespeare.csv` file which will do the trick for us!



```
playsandpoemsdata : bash
hellobigdata@bash:playsandpoemsdata$ awk -F, '{
  TR=0
  for( I = 1; I <= NF; I++ ) {
    TR += $I
    TC[I] += $I
    if(I==1)printf( "%s", $I )
  }
  print ", " TR
  TF = NF
}' Shakespeare.csv | tail -n +2 | sort -nr -t"," -k2 | head -n 100 | grep [a-z] | csvlook
```

Word	Frequency
2001	200
the	129.984...
and	117.445...
i	104.666...
99by	99.000...
of	78.332...
a	66.572...
you	65.405...
is	62.826...
my	59.069...
in_preposition__	51.579...
it	50.327...
to_infiniteive__	47.667...
to_preposition__	41.711...
not	40.732...

The most frequent words in all Shakespearean works

```

1  awk -F, '{
2      TR=0
3      for( I = 1; I <= NF; I++ ) {
4          TR += $I
5          TC[I] += $I
6          if(I==1)printf( "%6s", $I )
7      }
8      print "," TR
9      TF = NF
10 }
11 ' Shakespeare.csv | tail -n +2 | sort -nr -t"," -k2

```

This small `awk` code will combine and sum horizontally all the columns (for any number of columns). Note that at the end we again sort the output based on the second column (i.e., combined and summed frequencies).

The final output will look like below:

Note that due to some garbage characters (e.g., page numbers) in the data set, we excluded tokens that are numbers. We only have shown word tokens, using a `grep [a-z]` at the end of the command. There we go, the most frequent five words in all Shakespearean works:

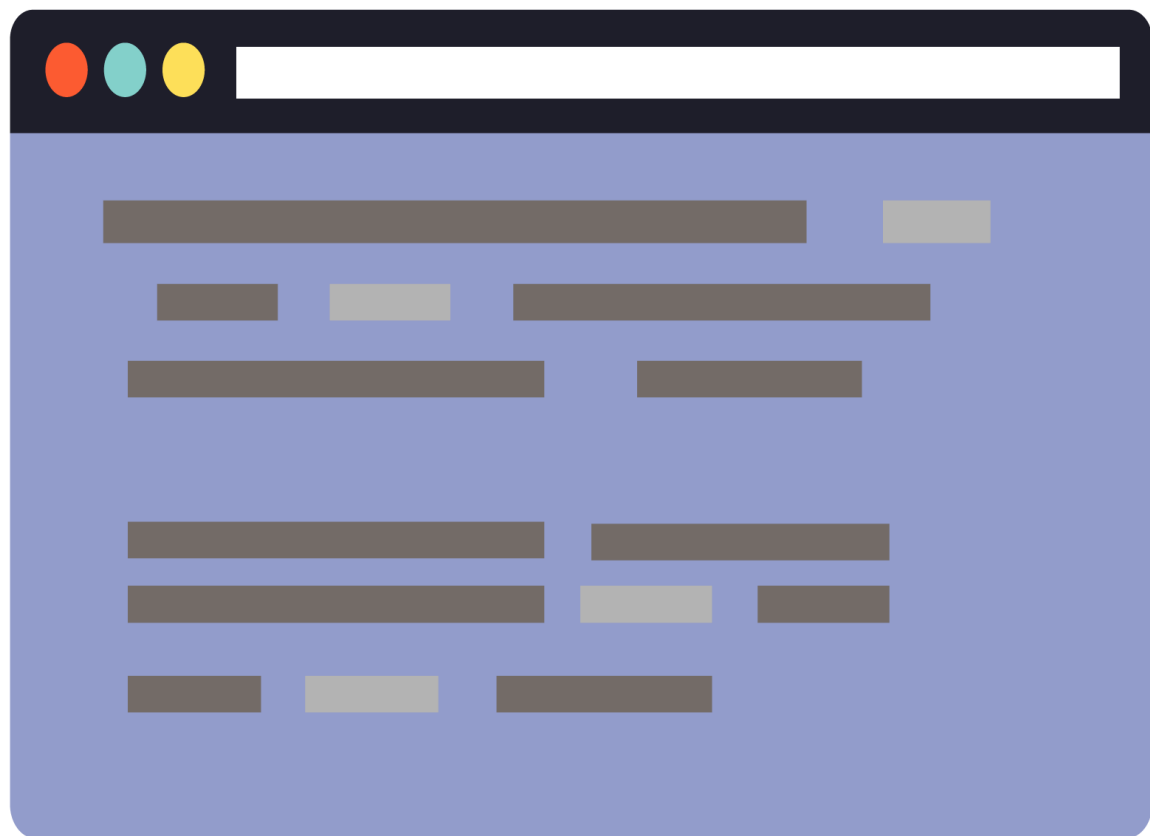
- the,
- and,
- I,
- of, and
- a.

Chapter Summary

In this project we have learned two more important aspects of the Bash: extended Bash functions (such `colcut` which extends `cut` and much more robust than the `csvcut`), a more complicated use of `awk` in the context mining a csv formatted data set consisting of Shakespeare-era 256 plays and poems statistics.

Part 2: Tutorials

This part of the book will introduce you with the Bash shell scripting, regular expressions, AWK language, SED and a few other valuable commmandline utilities.



This tutorial will give you just enough knowledge to read and understand this book, to be a master on these topics, you need to explore relevant literature referenced at end of this book.

Hello Bash!

A Bash shell is a nothing but a program that provides users with an interface to interact with other programs in a Linux system. You can infact find a large variety of Bash shell like programs, each with their own language. Some popular ones are the C shell (csh), Z shell (zsh), Korn shell (ksh), Debian's Almquist shell (dash), etc. However, Bash is currently the most popular and ubiquitously available shell. All of the shells tend to use similar syntax, however it is important to be fully aware of what shell you're actually writing code for. This tutorial will teach you how to write bash shell codes.



Important Information

This tutorial will give you just enough knowledge to read and understand this book, to be a master on Bash shell programming, you need to explore relevant literature referenced at end of this book.

which bash?

Let's find out where is the bash interpreter located. We enter the following into your command line:

which bash

```
1 $ which bash
2 /bin/bash
3
4 $ bash --version
5 GNU bash, version 4.3.46(1)-release (x86_64-pc-linux-gnu)
6 Copyright (C) 2013 Free Software Foundation, Inc.
7 License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
8
9 This is free software; you are free to change and redistribute it.
10 There is NO WARRANTY, to the extent permitted by law.
```

This means that our bash resides in a folder called `bin` inside the root (`/`) partition and our bash version is 4.3.46.

With bash installed, we can run use it to start any binary program or script. Before we do so, it's important to take note of the two distinct modes of operation in Bash:

Interactive mode: In interactive mode, the bash shell waits for your commands before performing them. Each command you pass (e.g., `bash --version`) it is executed and while it is being executed, you cannot interact with the bash shell. You need to wait until it gets finished.

Non-interactive mode: In non-interactive more, bash shell generally execute scripts. A script (see below: `helloworld.sh`) is a pre-written series of commands which bash executes without needing to ask you what to do next. It helps to automate tasks.

In this chapter, we are mostly interested about to leanring about he non-ineractive mode.

Hello world! bash

Let's open up a text editor (e.g., gedit, kate, atom, you name it) and create a file called `helloworld.sh` and insert the following lines:

Hello world! bash

```
1 $ cat helloworld.sh
2
3 #!/bin/bash
4
5 # Declare string variable
6 s="Hello World!"
7
8 # Print the variable on a screen
9 echo $s
10
11 $ bash helloworld.sh
12 Hello Word!
```

Note that every bash shell script in this tutorial starts with shebang: `#!/` which is not read as a comment. The first line is also a place where we put our interpreter with its location, which is in this case: `/bin/bash`. Let's now navigate to a directory where your `hello_world.sh` is located and make the file executable:

```
1 $ chmod +x helloworld.sh
```

The `chmod` (interactive) command changes the permissions of a given file according to mode specified in the option (read `r`, write `w` and execute `x`), where mode describes the permissions to modify. Here we have changed the file's mode to execute, therefore we will be able to run it using the command:

```
1 $ ./helloworld.sh
```

Or, alternatively we run by calling `bash` (`bash helloworld.sh`). Both will display an output to the screen. Awesome! you have successfully said a hello to Bash!

Bash variables

A variable is a temporary store for a piece of information. There are two actions we may perform for variables: first - setting a value for a variable, second - reading the value for a variable. To read the variable we place its name preceded by a `$` sign. To learn more, let's create a shell script that will backup the home directory into a zipped (`tar.gz`) file.

A Bash backup script

```

1  #!/bin/bash
2
3  filename=homedirbackup_$(date +%Y%m%d).tar.gz
4  tar -czf $filename /$HOME

```

Here the variable `filename` first accepts a file name that's being constructed off three strings `homedirbackup_`, current date, and `tar.gz` and then we use the variable to zip up our home dir. Note that when the variable gets called, it has a `$` in front of it. The `date` function with the given options will return today's date and built-in OS variable `$HOME` will give us the path to user's home directory.

Bash variable types:

In normal situations, we don't need to declare a variable type to use it. However, bash lets us declare integer, read only, array, associative array, and export type variables. For example, we can declare an integer (number) using `declare -i` we can read only a variable using `declare -r`, we use this when we want to assign a value to a variable that should not be allowed to change. Further more, Bash allows us to declare arrays with `-a` option or associative arrays with `-A` option (details given later in this chapter).

Bash variables can be local or global:

Bash global and local variables

```

1  $ cat var.sh
2
3  #!/bin/bash
4
5  x="global value"
6
7  function bashfunction {
8      local x="local value"
9      echo $x
10  }
11
12  echo $x
13
14  # call the bashfunction
15  bashfunction
16
17  echo $x
18
19  $ bash var.sh
20
21  global value
22  local value
23  global value

```

The bash global variable's value do not change by the out of function activities, also note that "local" is bash reserved word.

Bash functions

We have already used a function above (called `bashfunction()`), as in other programming language, you can use bash functions to group pieces of code in a logical way or practice the divine art of recursion. It can also take arguments, however, Bash functions don't allow us to return a value, rather they allow us to set a return status.

Bash functions

```

1  $ cat bashprintfunction.sh
2
3  #!/bin/bash
4
5  printfunction () {
6      echo Hello $1
7      return 0
8  }
9
10 printfunction Hello
11
12 printfunction Big data!
13
14 echo "The previous function has a return value of $?"
15
16 $ bash bashprintfunction.sh
17
18 Hello
19 Big data!
20 The previous function has a return value 0

```

Note that the built-in variable `$?` contains the return status of the previously run command or function. Typically a return status of 0 indicates that everything went successfully. A non zero value indicates an error occurred.

Executing commands and passing arguments:

Inside bash scripts, you can run any command or call another script, see an example of calling `uname` command which displays some OS details.

Bash: passing arguments and external command execution

```

1  $ cat bashargs.sh
2
3  #!/bin/bash
4
5  osname=$1
6  echo $osname
7  echo `uname -a`
8
9  $ bash bashargs.sh Ubuntu
10 Ubuntu
11 Linux bash 4.4.0-36-generic #55-Ubuntu SMP Thu Aug 11 18:01:55 UTC 2016 x86_64 GNU/Linux

```

Note that we use backticks to execute shell command and the input argument is initially stored in the built-in variable `$1`.

Bash meta characters

Symbol	Example	How it works?
<	<code>sort < filename.txt</code>	Get input for the command to the left from the file listed to the right of this symbol.
>	<code>echo "BASH" > filename.txt</code>	Send the output of the command on the left into the file named on the right of this symbol. If the file does not exist, create it. If it does exist, overwrite it (erase everything in it and put this output as the new content).
>>	<code>date >> filename.txt</code>	Send the output of the command on the left to the end of the file named on the right of this symbol. If the file does not exist, it will be created. If it does exist, it will be appended.
?	<code>ls e?.txt</code>	This symbol matches any single character.
*	<code>ls e*.txt</code>	This symbol matches any number of any characters listing.
[]	<code>ls -l e[abc].txt</code>	One of the characters within the square brackets must be matched.
\$	<code>my_variable="this string" and echo \$my_variable</code>	Denotes that a string is a variable name. Not used when assigning a variable.
\	<code>echo "I have \\$10.00"</code>	Escape. Ignore the shell's special meaning for the character after this symbol. Treat it as though it is just an ordinary character.
()	<code>(cal; date) > filename.txt</code>	If you want to run two commands and send their output to the same place, put them in a group together.
{ }	<code>numbers=(5 2 7) and echo \${numbers[1]}</code>	Used in special cases for variables with the \$. One use is in parameter substitution (see the \$ definition, above), the other is in arrays. Example:
"	<code>my_variable="This is a test."</code>	Used to group strings that contain spaces and other special characters.
'	<code>my_variable='This is a backslash: \'</code>	Used to prevent the shell from interpreting special characters within the quoted string.
	<code>my_variable="This is the date: date "</code>	Unquoting. Used within a quoted string to force the shell to interpret and run the command between the backticks.

Symbol	Example	How it works?
&&	mkdir stuff && echo "Made the directory"	Run the command to the right of the double-ampersand only if the command on the left succeeded in running.
&	cat /etc/passwd &	Run the process in the background, allowing you to continue your work on the command line.
;	date;cat passwd; date	Allows you to list multiple commands on a single line, separated by this character.
=	my_variable="Hello World!"	Assignment. Set the variable named on the left to the value presented on the right.

Bash quotation basics

Single quotes in bash will suppress special meaning of every meta characters. Therefore meta characters will be read literally. It is not possible to use another single quote within two single quotes not even if the single quote is escaped by backslash.

Single quotes in bash will suppress special meaning of every meta characters. Therefore meta characters will be read literally. It is not possible to use another single quote within two single quotes not even if the single quote is escaped by backslash.

Bash: single quotes

```

1  #!/bin/bash
2
3  VAR="Hello Big Data!"
4
5  echo $VAR
6
7  echo '$VAR "$VAR"'

```

Double Quotes in bash will suppress special meaning of every meta characters except "\$", "\", and backtick (). Any other meta characters will be read literally. It is also possible to use single quote within double quotes. If we need to use double quotes within double quotes bash can read them literally when escaping them with ". Example:

Bash: double quotes

```

1  #!/bin/bash
2
3  VAR="Hello Big Data!"
4
5  # echo variable VAR
6  echo $VAR
7
8  echo "It's $VAR and \"${VAR}\" using backticks: `date`"

```

Bash quoting with ANSI-C style There is also another type of quoting and that is ANSI-C. In this type of quoting characters escaped with " will gain special meaning according to the ANSI-C standard.

Char	Meaning
\a	alert/ bell
\e	an escape character
\n	newline
\t	horizontal tab
\\	backslash
\b	backspace
\f	form feed
\r	carriage return
\v	vertical tab
'	single quote
\xnn	hexadecimal value of characters

Example:

ANSI style chars

```

1  #!/bin/bash
2
3  $ cat bash-ansi.sh
4
5  # \n Newline
6  # \x40 is hex value for @
7  echo $Hello!\nBig Data\x40Bash
8
9  $ bash bash-ansi.sh
10 Hello!
11 Big Data@Bash

```

Read and store user input

Bash's built-in `read` function helps you to read user input and the variable next to `read` will store the input value:

User input

```

1  $ bash userinput.sh
2
3  #!/bin/bash
4
5  echo -e "Hi, please your name: \c "
6  read name
7  echo "Hello $name !"
8
9  $ bash userinput.sh
10
11 Hi, please your name:
12 data
13 Hello data !

```

Bash redirections

Redirection makes it possible to control where the output of a command goes to, and where the input of a command comes from. Under normal circumstances, there are 3 files open, accessible by the file descriptors (FD) 0, 1 and 2, all connected to your terminal:

Name	FD	Description
stdin	0	standard input stream (e.g. keyboard)
stdout	1	standard output stream (e.g. monitor)
stderr	2	standard error output stream (usually also on monitor)

Examaples:

Bash redirection examples

```
1 #!/bin/bash
2
3 # stdout from bash script to stderr
4 $ echo "Hello! Big Data @ Bash" 1>&2
5
6 # stderr from bash script to /dev/null
7 # Note that there's no command called 'dates'
8
9 $ dates 2>&1 // This will display errors
10 $ dates 2>/dev/null // This will not display errors
11
12 # stderr and stdout to file
13
14 $ dates &> outfile.txt
```

Bash if-else (conditional statements)

To compare integers we use can the following operators:

Operator	Description
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than OR equal to
-lt	less than
-le	less than OR equal to

The following example shows how to use the number comparison operators in an if statement.

Bash if-else

```

1  $ bash bash-ifelse.sh
2
3  #!/bin/bash
4
5  $ cat operators-integer.sh
6  total=${1}
7  if [ $total -eq 1 ]; then
8      echo "the number is equal to 1"
9  else
10     echo "the number is NOT equal to 1"
11 fi
12
13
14 $ bash bash-ifelse.sh 0
15 the number is NOT equal to 1

```

Bash `elif` is short for `else if` which can allow us to select one of many blocks of code to execute by testing two or more conditional expressions. The `If-elif-else` syntax is given below:

Bash if-elif-else

```

1  if [ conditional expression1 ]
2  then
3      statement 1
4      statement 2
5      ..
6  elif [ conditional expression2 ]
7  then
8      statement 3
9      statement 4
10     ..
11 else
12     statement 5
13 fi

```

Bash file test operators:

File Test	Operator Description
-e	File exists (this could be regular file, directory, block device, character device, etc.)
-f	It's a regular file (for example: <code>/etc/shadow</code>)
-d	It's a directory (for example: <code>/var</code>)
-b	It's a block device (for example: <code>/dev/sdb</code>)
-c	It's a character device (for example: <code>/dev/tty2</code>)
-s	File is not empty
-r	File read permission
-w	File write permission
-x	File execute permission
-u	suid set on the file
-g	sgid set on the file
-k	Sticky bit set on the file

File Test	Operator Description
-p	It's a pipe
-S	It's a socket
-h	It's a symbolic link
-t	Checks whether the given FD is opened in a terminal.
-O	You own the file
-G	File group id and my group id are the same.
-N	Did the file got modified since last read?
file-a -nt file-b	file-a is newer than file-b
file-a -ot file-b	File1 is older than file2
file-a -ef file-b	Both file1 and file2 are hard linked to the same file

Bash file/directory checking

```

1 $ mkdir scriptsdir
2 $ cat bashdircheck.sh
3
4
5 #!/bin/bash
6
7 directory=$1
8
9 # bash check if directory exists
10 if [ -d $directory ]; then
11     echo "Directory exists!"
12 else
13     echo "Directory does not exists!"
14 fi
15
16 $ bash bashdirchack.sh scriptsdir
17 Directory exists!

```

Bash case statement

In the following Bash case statements, if value of the “var” matches “pattern1”, it will execute command1, command2, and any other commands in the “pattern1” block.

Bash case statement

```

1 case var in
2     pattern1 )
3         command1
4         command2
5         ...
6     ;;
7     pattern2 )
8         command3
9         command4
10        ...
11    ;;
12 esac

```

Example:

Bash case statement

```

1  $ cat bash-case-statement.sh
2
3  #!/bin/bash
4  echo "What is your scripting language? (0 = EXIT)"
5  echo "1) bash"
6  echo "2) perl"
7  echo "3) python"
8  echo "4) None of the above !"
9  read case;
10
11  case $case in
12      1) echo "You've selected bash!";;
13      2) echo "You've selected perl!";;
14      3) echo "You've selected python!";;
15      0) exit
16  esac
17
18
19  $ bash bash-case-statement.sh
20
21  What is your scripting language? (0 = EXIT )
22  1) bash
23  2) perl
24  3) python
25  4) None of the above !
26
27  1
28  You've selected bash!

```

Bash loop statements

for loop

For loops allow repeated execution of a command sequence based on an iteration variable. Bash supports two kinds of for loop, a “list of values” and a “traditional” c-like method.

Bash for-loop Syntax

```

1  for varname in list
2  do
3      commands
4  done

```

Note that

- Bash for, in, do and done are keywords
- list contains a list of items, which can be in the statement or fetched from a variable that contains several words separated by spaces.
- If list is missing from the for statement, then bash uses positional parameters that were passed into the shell.

Example 1:

Bash for-loop

```
1 $ cat bash-day-for.sh
2
3 i=1
4 for day in Mon Tue Wed Thu Fri
5 do
6 echo "$(( i++ )) : $day"
7 done
8
9 $ bash bash-day-for.sh
10
11 1 : Mon
12 2 : Tue
13 3 : Wed
14 4 : Thu
15 5 : Fri
```

Note the counter increment command `$((i++))` which increases `i` by 1, each time called.

Bash c-style for-loop:

Bash c-style for-loop

```
1 $ cat bash-c-style-for.sh
2
3 for (( i=1; i <= 3; i++ ))
4 do
5     echo "Your random number $i: $RANDOM"
6 done
7
8 $ bash bash-c-style-for.sh
9
10 Your random number 1: 12348
11 Your random number 2: 166342
12 Your random number 3: 42343
```

Note that the built-in variable `$RANDOM` generates a random number each time called.

Bash while loop

Bash `while` allows for repetitive execution of a list of commands, as long as the command controlling the while loop executes successfully (exit status of zero). The syntax is:

Bash while-loop Syntax

```
1 while expression
2 do
3     commands
4 done
```

Note that

- `while`, `do`, `done` are keywords
- Expression is any expression which returns a scalar value

- Commands between do and done are executed while the provided conditional expression is true.

Example:

Bash while-loop

```
1 $ cat bash-whileloop.sh
2
3 #!/bin/bash
4
5 # This script prints 4 sequential numbers 0 1 2 3.
6
7 x="0"
8
9 while [ $x -lt 4 ]
10 do
11     x=$((x+1))
12     echo $x
13 done
14
15 $ bash bash-whileloop.sh
16 0
17 1
18 2
19 3
```

Infinite while loop:

Bash infinite while-loop

```
1 #!/bin/bash
2
3 while :
4 do
5     echo "Do something; hit [CTRL+C] to stop!"
6 done
```

A more (on-purpose) complicated example:

Bash nested if-else and while-loop

```
1 $ cat bash-nested-if-and-loop.sh
2
3 #!/bin/bash
4
5
6 select=0
7
8 echo "1. Apple"
9 echo "2. Orange"
10 echo "3. Lime"
11
12 echo -n "Please select [1,2 or 3] : "
13
```

```

14 # Loop while the variable 'select' is equal 0
15 # bash while loop
16
17 while [ $select -eq 0 ]; do
18
19     # read user input
20     read select
21
22     # bash nested if/else
23     if [ $select -eq 1 ] ; then
24
25         echo "You have selected: Apple"
26
27     else
28
29         if [ $select -eq 2 ] ; then
30             echo "You have selected: Orange"
31         else
32
33             if [ $select -eq 3 ] ; then
34                 echo "You have selected: Lime"
35             else
36                 echo "Please select between 1-3 !"
37                 echo "1. Apple"
38                 echo "2. Orange"
39                 echo "3. Lime"
40                 echo -n "Please select [1,2 or 3] : "
41                 choice=0
42             fi
43         fi
44     fi
45 done
46
47 $ bash bash-nested-if-and-loop.sh
48
49 1. Apple
50 2. Orange
51 3. Lime
52
53 Please select [1,2 or 3]: 1
54 You have selected: Apple

```

until loop

Bash until loop is very similar to the while loop, except that the loop executes until the condition executes successfully.

Bash until-loop example

```
1 $ cat bash-until-loop.sh
2
3 #!/bin/bash
4
5 i=0
6 # bash until loop
7 until [ $i -gt 3 ]; do
8     echo "i : $i"
9     i=$((i+1))
10 done
11
12 $ bash bash-until-loop.sh
13 i : 1
14 i : 2
15 i : 3
```

Bash arithmetic

Bash arithmetic expansion provides a powerful tool for performing arithmetic operations in scripts. Translating a string into a numerical expression is relatively straightforward using backticks (`), double parentheses (()), or `let`.

Example 1:

Bash arithmetic example

```
1 i=`expr $i + 1`
```

Where, `expr` is an all-purpose expression evaluator.

Example 2:

Bash arithmetic example

```
1 $(( i1+1 ))
```

Example 3:

Bash arithmetic example

```
1 $ let i=3+5
2 $ echo "3 + 5 =" $i
```

Order of Precedence operators are evaluated in order of precedence. The levels are listed in order of decreasing precedence:

Operator's order of Precedence in Bash

```

id++ id--
    variable post-increment and post-decrement
++id --id
    variable pre-increment and pre-decrement
- +    unary minus and plus
! ~    logical and bitwise negation
**     exponentiation
* / %  multiplication, division, remainder
+ -    addition, subtraction
<< >> left and right bitwise shifts
<= >= < >
        comparison
== !=  equality and inequality
&      bitwise AND
^      bitwise exclusive OR
|      bitwise OR
&&     logical AND
||     logical OR
expr?expr:expr
        conditional operator
= *= /= %= += -= <<= >>= &= ^= |=
        assignment
expr1 , expr2
        comma

```

Apart from the precedence, **operators that work with integers** are given below with some examples:

Operator	Description	Example	Output
+	Addition	echo \$((10 + 1))	11
-	Subtraction	echo \$((11 - 1))	10
/	Division	echo \$((10 / 2))	5
*	Multiplication	echo \$((10 * 5))	50
%	Modulus	echo \$((10 % 3))	1
++	post-increment (add variable value by 1)	x=5;echo \$((x++));echo \$((x++))	5 6
--	post-decrement (subtract variable value by 1)	x=5; echo \$((x--))	4
**	Exponentiation	x=3;y=3;echo \$((x ** y))	9

Code example:

Bash arithmetic examples

```

1  $ cat bash-arithmetic.sh
2
3  echo -e "Enter two numbers : "
4
5  read x y
6  declare -i n
7  n=$((x+y))
8  echo "Result is:$n "
9
10 # bash convert binary number input x
11 n=2 # $x
12 echo $n
13
14 # bash convert octal number input x
15 n=8 # $x
16 echo $n
17
18 # bash convert hex number input x
19 result=$((16 * $x))
20 echo $n

```

Bash floating point calculations

You can perform floating point operation in Bash using the `bc` arbitrary precision calculator language. Note the need to escape the multiply operator `*` with a backslash (`\`) or enclose the arithmetic expression in single quotes (`' '`).

Examples:

Bash floating point arithmetic examples

```

1  $ x = 1.1
2  $ y = 2.2
3  $ echo x + y | bc -l
4  3.3
5
6  $ echo x - y | bc -l
7  -1.1
8
9  $ echo x \* y | bc -l
10 2.42
11
12 $ echo 'x * y' | bc -l
13 2.42
14
15 $ echo 'x / y' | bc -l
16 .5000
17
18 $ z=`echo '$x / $y' | bc -l`
19 $ echo $z
20 .5000
21
22 # Wrong use
23

```



```
24 $ echo x * y | bc -l
25 1.1
```

Note that there should be no space between the variable name and the equal sign (=) in the assignment, otherwise an error occurs.

Bash arrays

Bash only provides one-dimensional indexed variables (arrays). The `declare` builtin command will explicitly declare an array. There is no maximum limit on the size of an array, nor any requirement that members be indexed or assigned contiguously!

Bash arrays example

```
1 $ cat basharrays.sh
2
3 #!/bin/bash
4
5 # Let's declare array with 3 elements
6 a=( 'Hello Big Data' book ! )
7
8 # get number of elements in the array
9 elements=${#a[@]}
10
11 # echo each element in array
12 # for loop
13 for (( i=0;i<${elements};i++)); do
14     echo ${elements[$i]}
15 done
16
17
18 $ bash basharrays.sh
19 Hello Big Data
20 book
21 !
```

Notice that the first element has three words as the element was confined by the single quotes (' '). The number of elements were identified by putting # in front of the array `#a[@]` which then stored into the `elements` variable as `${#a[@]}`.

Hello ! Regular Expressions

A regular expression (regex) is a way to describe a particular pattern of characters that a *regular expression engine* would attempt to match in a given input text.

Mathematician Stephen Cole Kleene invented (1956) regular languages using his mathematical notation called ‘regular sets’ which then entered into popular use from 1968 in two use cases: *pattern matching* in a text editor and lexical analysis in a compiler. Among the first appearances of regular expressions in a programming form was seen when *Ken Thompson* built Kleene’s notation into the editor *QED* as a means to match patterns in text files. In the 1980s the more complicated regexes started to arise in *Perl* and then in 1997, *Philip Hazel* developed *Perl Compatible Regular Expressions*, which attempts to closely mimic Perl’s regex functionality and is currently used by many modern tools including PHP and Apache HTTP Server. Today regexes are widely supported in programming languages, text processing programs (particularly lexers), advanced text editors, and some other programs.



Important Information

This tutorial will give you just enough knowledge to read and understand this book, to be a master on RegEx, you need to explore relevant literature referenced at end of this book.

With regular expressions we can:

- **Search for particular items**, within a large body of text.
- **Replace particular items**
- **Validate input**, for example, a password meets certain criteria such as, a mix of uppercase and lowercase, digits and punctuation, etc. and
- **Coordinate actions**, for example process certain files in a directory, but only if they meet particular conditions.

In short Regexes are super useful in text processing tasks, and also in string processing, where the data need not be textual. Common applications include data validation, data scraping (especially web scraping), data wrangling, simple parsing, the production of syntax highlighting systems.

REGEX Types

The IEEE POSIX standard has three sets of compliance: **Basic Regular Expressions (BRE)**, **Extended Regular Expressions (ERE)**, and **Simple Regular Expressions (SRE)**. The SRE is deprecated.

Basic Regular Expressions

Metachar .

Matches any single character.

Example:

`x.z` matches "xyz", etc., but within bracket expressions, the dot character matches a literal dot, e.g., `[x.y]` matches only "x", ".", or "y"

Metachar []

Matches a single character that is contained within the brackets.

Example:

`[xyz]` matches "x", "y", or "z", where `[a-z]` specifies a range which matches any lowercase letter from "a" to "z". Note that the - character is treated as a literal character if it is the last or the first e.g., `[-xyz]`

Metachar [^]

Matches a single character that is not contained within the brackets.

Example:

`[^xyz]` matches any character other than "x", "y", or "z".

Metachar ^

Matches the starting position within the string.

Example:

`^[xterm]` matches any string that starts with `xterm`.

Metachar \$

Matches the ending position of the string or the position just before a string-ending newline.

Example:

`[mb]at$` matches "mat" and "bat", but only at the end of the string or line.

Metachar ()

The string matched within the parentheses can be recalled later, also called a `block` or `capturing group`.

Example:

`([0-9]+)([a-z]+)` the first group matches atleast one digit and the second group atleast one alphabet.

Metachar *

Matches the preceding element zero or more times.

Example:

`xy*z` matches `"xy"`, `"xyz"`, `"xyyz"`, etc.

Metachar {m,n}

Matches the preceding element at least `m` and not more than `n` times.

Example:

`Y{2,3}` matches only `"YY"`, `"YYY"`.

Extended Regular Expressions**Metachar ?**

Matches the preceding element zero or one time.

Example:

`xy?z` matches only `"xy"` or `"xyz"`.

Metachar +

Matches the preceding element one or more times.

Example:

`xy+z` matches `"xyz"`, `"xyyz"`, and so on, but not `"xz"`.

Metachar |

Alternation operator, matches either the expression before or the expression after the operator.

Example 1:

`(abc)|(xyz)` matches `"abc"` or `"xyz"`

Example 2:

`(data)|(information)` matches `"data"` or `"information"`.

See also the grouping meta char `()` described above.

REGEX Character Classes

Regex character class can make our life easy, as it makes one small sequence of characters match a larger set of characters. However, character classes can only be used within bracket expressions. For example, `[[:upper:]]xy` matches all the uppercase letters and lowercase `"x"` and `"y"`.

Char Class	Description
[:alnum:]	alphanumeric characters [A-Za-z0-9]
[:lower:]	lowercase letters [a-z]
[:upper:]	uppercase letters [A-Z]
[:digit:]	digits [0-9]
[:blank:]	Space and tab [\t]
[:space:]	Space []
[:punct:]	punctuation characters e.g., [] [! " # \$ % & ' () * + , . / : ; < = > ? @ \ ^ _ { } ~ -]

REGEX Look Arouds

Let's have a look into the following examples first:

Example	Type	Description
data(?:data)	Look ahead positive	finds the 1st data (data which has data after it)
data(?:!data)	Look ahead negative	finds the 2nd data (data which does not have data after it)
(?<=big)data	Look behind positive	finds the 1st bar (data which has big before it)
(?<!big)data	Look behind negative	finds the 2nd bar (data which does not have big before it)

Collectively, lookbehinds and lookaheads are known as **lookarounds**. The examples above give us a very basic syntax, but further down the track we encourage you to read the dedicated regex reference materials.

REGEX Atomic Groups (?>)

By definition, a REGEX **atomic** group is a non-capturing group that exits the group and throws away all alternatives after the first match of the pattern inside the group, so backtracking is **disallowed**".

While a **non-atomic** group will **allow backtracking**, it will still find the first match, then if the matching ahead fails it will backtrack and find the next match, until a match for the entire expression is found or all possibilities are exhausted.

Example 1 :

Let's consider the input string: `xoots`.

- A non-atomic group in the expression `(xoo|xoot)s` applied to `xoots` will:
 - match its 1st alternative `xoo`, then fail as `s` does not immediately follow in `xoot`, and backtrack to its 2nd alternative;
 - match its 2nd alternative `xoot`, then succeed as `s` immediately follows in `xoots`, and stop.
- An atomic group in the expression `(?>xoo|xoot)s` applied to `xoots` will match its 1st alternative `xoo`, then fail as `s` does not immediately follow, and stop as backtracking is disallowed.

Example 2:

Let's consider the input string: bbabbbabbbbc.

- Without an atomic grouping for the regex pattern `/(. *|b*)[ac]/`, the string will have a single match which is the whole string, due to backtracking at the end to match `[ac]`.
- In contrast, for the atomic pattern: `/((?>.*|b*)[ac])/`, there will be only three matches to this regex, which are bba, bbba, bbbbc.

How to Use REGEX in Bash?

You can always use regex `grep` or `sed` or some other external command/programs, but since the version 3 of bash (released in 2004) bash's provided a built-in regular expression comparison operator `"=~"`.

Bash regular expressions support sub-patterns surrounded by parenthesis for capturing parts of the match. The matches are assigned to an array variable `BASH_REMATCH`. The entire match is assigned to `BASH_REMATCH[0]`, while the first sub-pattern is assigned to `BASH_REMATCH[1]`, `BASH_REMATCH[2]`, etc.

Regex in Bash

```

1  $ cat bash-regex.sh
2
3  #!/bin/bash
4
5  input=$1
6
7  if [[ "$input" =~ 'data(.*)' ]]
8  then
9      echo $BASH_REMATCH : ${BASH_REMATCH[1]}
10 fi
11
12
13 $ bash bash-regex.sh databig
14 big

```

You can also do the following trick (see the code below) to perform a global matching as in a more advanced language like Perl.

Regex in Bash

```

1  #!/bin/bash
2
3  regex="aa(b{2,3}[xyz])cc"
4  mystring=aabbxcc aabbcc
5
6  global_rematch() {
7      local string=$1 regex=$2
8      while [[ $string =~ $regex ]]; do
9          echo "${BASH_REMATCH[1]}"
10         string=${string#*${BASH_REMATCH[1]}}
11     done

```

```
12 }  
13  
14 global_rematch "$mystring" "$regex"
```

Regular expressions are a language of their own, please refer to the further reading section to know about the REGEX.

Hello! AWK

Awk is a powerful tools in the commandline used for processing the rows and columns of a flat texy file. Awk has built in string functions and associative arrays. Awk supports most of the operators, conditional blocks, and loops available in C language. You may want to know, what Awk stands for? It comes from the surnames of its authors “**Aho, Weinberger, and Kernighan**”. AWK was created at Bell Labs in the 1970s. It is pronounced the same as the name of a bird called auk. The GNU implementation of awk is called gawk.



Important Information

This tutorial will give you just enough knowledge to read and understand this book, to be a master on AWK, you need to explore relevant literature referenced at end of this book.

The AWK language is a fully data-driven scripting language consisting of a set of actions to be taken against streams of textual data - either run directly on files or used as part of a pipeline for purposes of extracting or transforming text, such as producing formatted reports.

The very basic syntax of AWK:

AWK syntax

```
awk 'BEGIN {start-action} {action} END {stop-action}' filename
```

Note that the actions in the *begin block* are performed before processing the file and the actions in the *end block* are performed after processing the file. The rest of the actions are performed while processing the file!

It can be also written as:

AWK syntax

```
awk '/search pattern1/ {Actions}  
    /search pattern2/ {Actions}' file
```

In the above AWK syntax:

- search pattern is a regular expression;
- Actions are the statement(s) to be performed;
- several patterns and actions are possible in AWK;
- a file is an input file; and
- single quotes around program is to avoid shell not to interpret any of its special characters.

AWK Built-in Variables

The following tables lists some important AWK builtin variables:

Variable	Description
ARGC, ARGV	command-line arguments
FILENAME	name of the file that awk is currently reading.
FNR	current record number in the current file, incremented on new records read
NF	number of fields in the current input record
NR	number of input records processed since the beginning of the program's execution
RLENGTH	length of the substring matched by the AWK's <code>match</code> function
RS	input record separator (default: newline)
OFS	output field separator (default: blank)
ORS	output record separator (default: newline)
OFMT	output format for numbers (default: <code>%.6g</code>)
ENVIRON	array of environment variables; subscripts are names.

AWK statements

An AWK action is a sequence of statements. A statement can be one of the following:

- `If(expression) statement [else statement]`
- `while(expression) statement`
- `for(expression ; expression ; expression) statement`
- `for(var in array) statement`
- `do statement while(expression)`
- `break`
- `continue`
- `{ [statement ...] }`
- `expression`
- `print [expression-list] [> expression]`
- `printf format [, expression-list] [> expression]`
- `return [expression]`
- `next` (skips remaining patterns on this input line)
- `nextfile` (skips rest of this file, open next, start at top)
- `delete array[expression]` (deletes an array element)
- `delete array` (deletes all elements of array)
- `exit [expression]` (exits immediately; exit status is the evaluation of expression)

AWK built-in functions

AWK has the mathematical functions like `exp`, `log`, `sqrt`, `sin`, `cos`, `atan2`, etc. built-in, other built-in functions are:

- `length` the length of its argument taken as a string, or of `$0` if no argument.
- `rand` random number between 0 and 1
- `srand` sets seed for `rand` and returns the previous seed.
- `int` truncates to an integer value
- `substr(s, m, n)` the `n`-character substring of `s` that begins at position `m` counted from 1.

- `index(s, t)` the position in `s` where the string `t` occurs, or `0` if it does not.
- `match(s, r)` the position in `s` where the regular expression `r` occurs, or `0` if it does not. The variables `RSTART` and `RLENGTH` are set to the position and length of the matched string.
- `split(s, a, fs)` splits the string `s` into array elements `a[1]`, `a[2]`, ..., `a[n]`, and returns `n`. The separation is done with the regular expression `fs` or with the field separator `FS` if `fs` is not given. An empty string as field separator splits the string into one array element per character.
- `sub(r, t, s)` substitutes `t` for the first occurrence of the regular expression `r` in the string `s`. If `s` is not given, `$0` is used.
- `gsub` same as `sub` except that all occurrences of the regular expression are replaced; `sub` and `gsub` return the number of replacements.
- `sprintf(fmt, expr, ...)` the string resulting from formatting `expr ...` according to the `printf` format `fmt`.
- `system(cmd)` executes `cmd` and returns its exit status.
- `tolower(str)` returns a copy of `str` with all upper-case characters translated to their corresponding lower-case equivalents.
- `toupper(str)` returns a copy of `str` with all lower-case characters translated to their corresponding upper-case equivalents.

AWK Examples

The best way to learn AWK is probably looking at some examples in a given context, let us create file `utility.data` file which has the following content

input file: utility.data

```

1  $cat data.txt
2  Months,Water,Gas,Elec,Phone
3  Jan,647,2851,3310,1200
4  Feb,287,3544,25500,1000
5  Mar,238,3212,21900,1710
6  Apr,468,6986,35000,2000
7  May,124,1097,96300,13000

```

Example 1. AWK print function

By default Awk prints every line from the file.

AWK print funtion

```

1  $ awk -F, '{print;} ' data.txt
2  Months,Water,Gas,Elec,Phone
3  Jan,647,2851,3310,1200
4  Feb,287,3544,25500,1000
5  Mar,238,3212,21900,1710
6  Apr,468,6986,35000,2000
7  May,124,1097,96300,13000

```

Action `print` with out any argument prints the whole line by default. So it prints all the lines of the file with out fail. Note that the actions need to be enclosed with in the braces.

Example 2. AWK print specific field

AWK fields printing

```

1 awk -F, '{print $1,$2}' utility.data
2 Months Water
3 Jan 647
4 Feb 287
5 Mar 238
6 Apr 468
7 May 124

```

Example 3. AWK's BEGIN and END Actions

AWK print formatting

```

1 awk -F, 'BEGIN {print "Period\tBill1\tBill2";}
2 {print $1,"\t",$2,"\t",$3,"\t",$NF;}
3 END{print "END REPORT\n-----"; }' utility.data
4 Period      Bill1      Bill2
5 Months      Water      Gas      Phone
6 Jan         647        2851     1200
7 Feb         287        3544     1000
8 Mar         238        3212     1710
9 Apr         468        6986     2000
10 May        124        1097     13000
11 END REPORT
12 -----

```

Here, the actions specified in the BEGIN section executed before AWK starts reading the lines from the input and END actions will be performed after completing the reading and processing the lines from the input.

Example 4. AWK fields variable (\$1, \$2 and so on)

Let's consider we want to find a total of all bills in all months in the data. We then create the following script:

BASH and AWK script to find a grand total

```

1 $ cat awk-total.sh
2
3 #!/bin/bash
4
5 awk -F "," '{
6     if(FNR == 1){
7         next;
8     }
9
10    Water=$2;
11    Gas=$3;
12    Electricity=$4;

```

```

13         Phones=$5;
14
15         fields_sum=Water + Gas + Electtricty + Phones;
16
17         total +=fields_sum;
18
19 } END { print total; }' utility.data

```

Note that it's a bash script that calls `awk` from inside and we have used `FNR` to detect the first row which we want to avoid in the sum calculation.

Example 5. AWK built-in variables

As mentioned earlier, the built-in variable `$NF` represents number of field, in this case last field (5):

AWK built-in variables

```

1  awk -F, '{print $1,$NF;}' utitlity.data
2  Months Phone
3  Jan 1200
4  Feb 1000
5  Mar 1710
6  Apr 2000
7  May 13000

```

Example 6. AWK fields comparison >

Let's find the months with water bills > 500:

AWK syntax

```

1  $ awk -F, '$2 > 500' utitlity.data
2  Months,Water,Gas,Elec,Phone
3  Jan,647,2851,3310,1200

```

Self-contained AWK scripts

In Linux systems self-contained AWK scripts can be constructed using. For example, a script that prints the content of a given file may be built by creating a file named `printfile.awk` with the following content:

AWK self-contained script

```
1 $ cat printfile.awk
2
3 #!/usr/bin/awk -f
4 { print $0 }
5
6 $ ./printfile.awk utility.data
```

The `-f` flag tells AWK that the argument that follows is the file to read the AWK program from.

Hello! SED, GREP and Find

SED - Stream Editor

Stream Editor (SED) is an important text-processing utilities on GNU/Linux. It uses a simple programming language and is capable of solving complex text processing tasks with few lines of code. This easy, yet powerful utility makes GNU/Linux more interesting.

SED can be used in many different ways, such as:

- Text substitution,
- Selective printing of text files,
- In-a-place editing of text files,
- Non-interactive editing of text files, and many more.



Important Information

This tutorial will give you just enough knowledge to read and understand this book, to be a master on the SED, GREP and Find command, you need to explore relevant literature referenced at end of this book.

Sed works as follows: it reads from the standard input, one line at a time. for each line, it executes a series of editing commands, then the line is written to STDOUT.

An example which shows how it works : we use the `s` sommand. `s` means “substitute” or search and replace. The format is

SED basic syntax

```
1 s/regular-expression/replacement text/{flags}
```

In the example below, we have used `g` as a flag, which means “replace all matches” (global replacement):

SED basic syntax

```
1 $ cat datafile.txt
2   I have a big data!
3
4 $ sed -e 's/big/small/g' -e 's/data/list/g' datafile
5   I have a small list!
```

Let’s try to learn what happened.

Step 1, sed read in the line of the file and executed

```
1      s/big/data/g
```

which produced the following text:

```
1      I have a small data!
```

Step 2, then the second replacement command ('s/data/list/g') was performed on the edited line and the result was:

```
1      I have a small list!
```

SED substitution

The format for the substitute command is as follows:

```
1      [address1[ ,address2]]s/pattern/replacement/[flags]
```

The flags can be any of the following:

- **n** replace *n*th instance of pattern with replacement
- **g** replace all instances of pattern with replacement
- **p** write pattern space to STDOUT if a successful substitution takes place
- **w file** Write the pattern space to file if a successful substitution takes place
- **i** match REGEXP in a case-insensitive manner.

We can use different delimiters (one of @ % ; :) instead of /. If no flags are specified the first match on the line is replaced. note that we will almost always use the **s** command with either the **g** flag or no flag at all.

If one address is given, then the substitution is applied to lines containing that address. An address can be either a regular expression enclosed by forward slashes `/regex/`, or a line number. The **\$** symbol can be used in place of a line number to denote the last line. If two addresses are given separated by a comma, then the substitution is applied to all lines between the two lines that match the pattern.

Example 1: substitute only third occurrence of a word `sed s//3`

```
1  $ sed 's/Data/Big-Data/3' datafile
```

Example 2: print and write to a file `sed s//gpw`

```
1  $ sed -n 's/Data/Big-Data/gpw output' datafile
```

Some important SED options

Option: `-n`, `--quiet` OR `--silent`

The `"-n"` option will not print anything unless an explicit request to print is found:

```
1 sed -n 's/PATTERN/&/p' file
```

Option: -e

Combines multiple commands:

```
1 sed -e 's/a/A/' -e 's/b/B/' file
```

Option: -f

If you have a large number of sed commands, you can put them into a file and use

```
1 sed -f sedscrip file
```

Option: -r

Extended regular expressions (ERE) have more power, but SED them normal characters. Therefore you must explicitly enable this extension with a command line option.

```
1 % echo "123 abc" | sed -r 's/[0-9]+/& &/'
2 123 123 abc
```

Option: -i

Substitutions are performed in-place, on the file which was fed to SED:

```
1 sed -i 's/^\t/' *.txt
```

SED substitute and regular expressions

Example 1: match patterns and REGEX (\!.*)

```
1 $ sed '/!/s/\!.*//g' datafile
```

In this example, if the line matches with the pattern “!”, then it replaces all the characters from “!” with an empty char.

Example 2: use REGEX to delete the last x characters

```
1 $ sed 's/...$//' datafile
```

This sed example deletes last 3 characters from each line.

Example 3: use REGEX to eliminate comments (#)

```
1 $ sed -e 's/#.*//' datafile
```

This sed example deletes last 3 characters from each line.

Example 4: use REGEX and eliminate Comments (#) as well as the empty lines


```
1 $ sed -e 's/#.*//;/^$/d' datafile
```

This sed example deletes last 3 characters from each line.

SED delete

A useful command that deletes line that matches the restriction: “d.” For example, if you want to chop off the header of a mail message, which is everything up to the first blank line, use:

```
1 $ sed '1,/^$/d' file
```

SED print

If sed was not started with an “-n” option, the “p” command will duplicate the input.

```
1 sed '/^$/ p'
```

Adding the “-n” option turns off printing unless you request it.

SED grouping

The curly braces, “{” and “}” are used to group the commands.

Previously, we have showed you how to remove comments starting with a “#.” If you wanted to restrict the removal to lines between special “begin” and “end” key words, we use:

```
1 sed -n '
2     /begin/,/end/ {
3         s/#.*//
4         s/[ ^I]*$//
5         /^$/ d
6         p
7     }
8 '
```

GREP

The command `grep` is a small utility for searching plain-text data sets for lines matching a regular expression. Its name comes from the globally search a regular expression and print.

A simple example of a common usage of `grep` is the following, which searches the file `colors.txt` for lines containing the text string `blue`:

```
1 $ grep "red" colors.txt
```

The `v` option reverses the sense of the match and prints all lines that do not contain `blue`, as in this example.

```
1 $ grep -v "blue" colors.txt
```

The `i` option in `grep` helps to match words that are case insensitive, as shown in below example.

```
1 $ grep -i "bLuE" colors.txt
```

The `n` option identifies the lines where matches occurred:

```
1 $ grep -n "orange" colors.txt
2 4: Orange color
3 6: Ornage company
```

GREP and regular expressions

While `grep` supports a handful of regular expression commands, it does not support certain useful sequences such as the `+` and `?` operators. If you would like to use these, you will have to use extended `grep` (`egrep`).

The Following command illustrates the `?`, which matches 1 or 0 occurrences of the previous character `w`:

```
1 $ grep "yellow?" colors.txt
```

`egrep` example:

```
1 $ egrep "red|yellow" colors.txt
```

Note that `grep` does not do the pipe (`|`), which functions as an "OR" in the expression.

Find command `find`

The command `find` is a command-line utility that searches one or more directory trees of a file system, locates files based on some user-specified criteria.

The following example will print out the files that `find` returns that contain the text "lime":

```
1 $ find | grep "lime"
```

Now, by using the `exec` switch with the `find` command, we can find all the files that contain the search string ("lime").

```
1 $ find . -exec grep "lime" {} \;
```

Part 3: Hello Big Data!



We have learned about Bash, SED, AWK, REGEX, and finally, the time has come to go beyond Bash and learn some systems relevant to Big data, and perhaps apply the learned technique over there! The following Big data related terms should be fairly easy to understand for you.

Big Data Terminologies

HDFS

One of the major components of the Hadoop ecosystem is the Hadoop Distributed File System, generally known as HDFS (storage) and the other essential part is the processing or compute. The major factors that set it apart from other file systems are the scalability and distribution. Hadoop stores various duplicates of large files over various data nodes, which appears like it is in its default mode. This is an architecture that can work perfectly with two indispensable functions: a) Data is available on various nodes, making it possible for the compute engine to work in parallel across respective nodes and then the results are summarized later, by that means, enabling very large parallelism. b) Another vital advantage is the fault tolerance. Hadoop is known as a system geared towards the processing of very large amount of jobs over various machines, thus, it is crucial that a single lost node does not affect the job to result into failure or data loss. c). This is one of the major ideas to comprehend in a distributed environment. The loss of just one node ought not to negatively affect the operation of the system or result into making a long-running task become a failure. HDFS likewise provides some commands, just like Unix, for the manipulation of file system.

Example:

```
1 hdfs dfs -ls /home/hellobigdatauser/datafile1
2 hdfs dfs -mkdir hellobigdata
```

For every Hadoop cluster, there is a single master as well as a group of data nodes clustered together. A master node has the regular components for the cluster just like job tracker, metastore, software, etc. and there will be some data nodes also known as the worker nodes – they are the real data which does the required computations. There is the availability of other distributed file systems which provide the same function, such as MapR-FS – a MapR distributed file system and Amazon S3.

Map Reduce

Map Reduce is another major component of the Hadoop Framework. This functions as the compute part (while the other center component is Storage as explained earlier). Map reduce offers a general purpose framework for the distribution of your work through various nodes and then the combination of the results for the purpose of getting the end results you desire. Map Reduce is also referred to as a programming model. The procedure of the `map()` helps in dividing your tasks over various queues, in a manner that each task can be autonomously executed. The procedure of the `reduce()` execute data summary from the individual nodes' output, for the purpose of obtaining the end result of your task. Just as expected, a specific task may incorporate various stages of map reduce.

YARN

A typical need in the majority of distributed processing engines is the necessity to oversee resources. Hadoop YARN deals with such functions that are critical. The Hadoop world began

with Map Reduce as the first data processing framework, but afterwards improved to include various project tasks. The majority of these projects have the same flow with regards to resource management. Basically, YARN abstracts a great deal of these projects and deals with the complex work for you. There are applications in the current Hadoop ecosystem that are composed in written format as YARN applications – as if each application converses with yarn and informs it what it needs to ensure the job is done. On the other hand, Yarn finds out what's available and what to designate to a specific task.

Flume

One of the basic components of any system of big data pipeline is the export layer and data ingestion. Commercial tools or common enterprise utilized for this reason include tools such as Data Stage, Informatica, and more. In the world of Hadoop, the most famous choice is the Flume. Based on documented information, Flume is known as a distributed, trusted, and accessible service for efficiently gathering, aggregating, and moving a lot of log data. Like a ton of different tools in the Hadoop environment, Flume is intended in its design to be fault tolerant. It has an easy architecture on the basis of streaming data flows, despite the fact that not all data requires to be streaming data.

SOOOP

Hadoop is also known as a new paradigm of handling data processing at monstrous scale. In any case, so many data processing pipelines and applications still make use of relational databases. A typical need, in cases like this, is to consider the importation of data from the relational database just like SQL Server, Oracle, etc. directly to Hadoop for the purpose of processing and then export the data being processed back from Hadoop straight to the RDBMS. The final data being processed in the RDBMS is utilized as one of an application or together with the data in the RDBMS for any analytic needs. In the Hadoop environment, Sqoop is referred to as a tool of choice for this purpose. Apache Sqoop is another tool designed to efficiently transfer enormous data between Hadoop and the stores of structured data like the relational databases (for instance, SQL Server, Oracle). Sqoop benefits greatly from both worlds. It depends on the database for the definition of schema and utilizes Map Reduce for the importation and exportation of the data. With the use of Map Reduce, you have the ability of scaling to any number of machines and you will be able to run codes in parallel. Also, it offers fault tolerance, which is a vital piece present in the machines with hundreds or even thousands of machines.

Hive

Map Reduce is difficult and for a few developers, a totally unique method for handling data processing than they are conversant with. The most widely recognized language to manage data is SQL, also, Apache Hive is referred to as a project that allows you develop SQL alike warehouse of your data on Hadoop top. The Apache Hive is an exceptionally productive project in the Hadoop environment as it joins the following: a) The simplicity and definitive nature of SQL, all you need is to simply state the data you require and the way in which it should be displayed, SQL will deal with developing the right plan to access and get the data. b) SQL queries run in the use of Hive Query Language (HQL) are then converted into jobs of Map Reduce, so you'll enjoy all the benefits Hadoop has to offer (parallel processing, fault tolerance,

and more) c) There are so many data processing codes and expertise are presently in SQL, Hive gives you the chance to reuse a vast majority of that, by offering a SQL layer on the Map Reduce top.

Pig

Apache Pig offers a high-level scripting language, popularly used for the writing of data pipelines in the world of Hadoop. The scripting language is known as Pig Latin. It appears as a blend of SQL and many other scripting languages, it is reasonably simple to read and write, so you tend to move up and running with the use of Pig Latin pretty rapidly. It is explanatory and concise just like SQL. You also enjoy more explicit control on various datasets at each phase in the pipeline.

Spark

Apache Spark is one of the most recent and the quickest developing projects in the big data space. Its design is intended to offer solution to similar problems as Hadoop. It additionally provides similar advantages, including Distributed computing, higher level interfaces, and Fault Tolerance. Spark separates itself in different significant ways: a) Spark offers various performance improvements and makes extraordinary use of the memory. The documentation of Spark claims programs keep running up to 100X quicker than Map Reduce with respect to memory and 10X faster with respect to disk. b) Spark is designed with a rich set of interfaces and APIs making it simple to write applications of distributed data processing. It is possible for you to write spark jobs in Scala, R, Java, and Python. You can additionally utilize SQL to the created query tables in Hive, while making use of Spark as the initial execution engine. c) The initial concept of Spark's Resilient distributed datasets is utilized by all of the other tools present in the Spark environment, so any further developments made to the core is translated into advancements for the remaining tools. d) The APIs of Spark support functional programming concepts, making the code concise and expressive much more in this way, when it is compared to Map Reduce code.

HBase

A Hadoop database is the Apache HBase, in which operations keep running in real time on its database instead of Map Reduce jobs, just like the Pig or Hive. Apache HBase offers random read or write access to big data. Also, it can accommodate expansive tables, for instance, billions of rows 'by' millions of columns. It is similar to other tools in the ecosystem of Hadoop, due to the fact that it is fault-tolerant and also designed to operate on commodity hardware. One significant contrast between HBase and different other RDBMSs is that HBase relates to a NoSQL Database. Rather than SQL as the essential access language, HBase offers an API in order to read and write data.

Big Data file formats

- Text/CSV Files: These are the usual delimited files that you normally see for most raw.
- Avro: Apache Avro is a data serialization system that provides a compact, fast binary format. It relies on schemas to make sense of the data in the file.

- Parquet: Apache Parquet is a columnar storage format that can be used by different projects in the Hadoop ecosystem. It is built to support very efficient compression and encoding schemes.
- ORC (optimized Row Columnar): In this format data is stored in a hybrid fashion, it stores collections of rows and within a collection different columns. It also introduces indexing and statistics like min and max.

Conclusion

Thank you for joining us! Through each chapter, we covered a some commands, analyzed some data, but most importantly, asked a few interesting questions and used command line tools to answer them. We've also introduced with a concise beginner friendly guide to the big data landscape including an overview of the critical Big Data tools such as HDFS, MapReduce, YARN, Flume, Hive and more. With all the topics we discussed, you are now be well-equipped to do some data analysis of your own!

References

Bash

- [Linuxconfig.org Bash Scripting Tutorial](#)*
- [Bash guide on Greg's wiki](#)
- [Steve Parker's shell scripting guide](#)
- [Advanced Bash Scripting Guide \(ABS\)](#)
- [IBM developerWorks "Bash by example"](#)
- [Bash Programming Introduction HowTo \(TLDP\)](#)*
- [LinuxCommand.org: Writing shell scripts.](#)
- [Beginner Linux Tutorial](#)
- [Beginner Bash Scripting Tutorial](#)
- [Wikipedia: BASH](#)

(*some examples are used in this book)

REGEX

- [Regular Expressions Cookbook](#) by Jan Goyvaerts and Steven Levithan
- [Teach Yourself Regular Expressions in 10 Minutes](#) by Ben Forta
- [Mastering Regular Expressions](#) by Jeffrey Friedl
- [Java Regular Expressions](#) by Mehran Habibi
- [Oracle Regular Expressions Pocket Reference](#) by Jonathan Gennick & Peter Linsley
- [Regular Expression Pocket Reference](#) by Tony Stubblebine
- [Regular Expression Recipes](#) by Nathan Good
- [Regular Expression Recipes for Windows Developers](#) by Nathan Good
- [Wikipedia: REGEX](#)

AWK

- [The GNU Awk User's Guide](#) at [gnu.org](#)
- [awk\(1\) OS X Manual Page](#) at [developer.apple.com](#)
- [Unix awk\(1\) manual page](#) at [man.cat-v.org](#)
- [awk POSIX specification](#) at [opengroup.org](#)
- [Getting started with awk](#) at [cs.hmc.edu](#)
- [Wikipedia:AWK](#)

SED

- GNU sed user's manual as one page at gnu.org
- sed(1) OS X Manual Page at developer.apple.com
- Unix sed(1) manual page at man.cat-v.org
- Sed FAQ at sed.sourceforge.net
- Sed by example, Part 1 at ibm.com
- Wikipedia:sed

GREP

- grep Pocket Reference by John Bambenek
- GNU grep user's manual as one page at gnu.org
- grep(1) OS X Manual Page at developer.apple.com
- Unix grep(1) manual page at man.cat-v.org
- Wikipedia:grep

Big data

- Hadoop For Dummies Dirk Deroos
- Hadoop Operations by Eric Sammer
- Hadoop: The Definitive Guide, 4th Edition
- Agile Data Science: Building Data Analytics Applications with Hadoop by Russell Journey
- Learning Spark
- Programming Hive
- Professional Hadoop Solutions by Boris Lublinsky, Kevin T Smith, Alexey Yakubovich
- MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop Donald Miner
- Learning Spark: Lightning -Fast Big Data Analysis by by Holden Karau, Andy Konwinski, Patrick Wendell and Matei Zaharia
- Advanced Analytics with Spark: Patterns for Learning from Data at Scale by Sandy Ryza, Uri Laserson, Sean Owen and Josh Wills

A companion book

- Data Science at the Command Line by Jeroen Janssens