A QUICK&DIRTY GUIDE TO REGULAR EXPRESSIONS in GNU/LINUX

HRISHIKESH BARMAN

geekodour.blogspot.com

Quick and Dirty Guide to Regular Expressions in GNU/Linux

[POSIX Standard]

with

grep

Revision.0

Hrishikesh Barman

(Geek Odour)

www.facebook.com/hrishikesh.barman

@Hrishikesh_Bman

www.geekodour.blogspot.com

Who should Read this book?

When I first stared off with Regular Expressions, I was quite frustrated with the books on regular expression. The aim of the the book is, to introduce you to regular expressions and to develop the core basic understanding needed to use regexes and to make you ready to read more advance stuff on this topic. It's a complete beginner's guide, and the best part is; learning this once will also help you being more efficient in programming languages like python and Javascript. This Book will not make you a master, but surely you'll become far more productive than before.

Is the book Linux Centric?

Yes it is, as the name suggests, But you can use the same knowledge with the same efficiency in other platforms too once you build the understanding. We'll be using **grep** for demonstration purpose. If you have lesser knowledge on Linux commands you might want to read a little about them here. Also, we'll be discussing grep extensively in this book. So, all in all I guess this little introduction to the book is enough.

Vote of Thanks

I would like to thank *miso ji* and *IDX root* for providing there valuable suggestions and support. I was supposed to add sed usage but finally decided not to add because it got complex. Explaining sed usage and explaining regular expressions together seems complicated for me, I will surely use mainly sed on the next version of this book, along with a proper introduction to sed.

I am not a native English Speaker, so if there are mistakes in grammar , please forgive me for that :/

Something I would Like To Say

This is my second ebook and this book might not look much as a book compared to other ebooks that you have read, but I have tried my best to give out as much as I could. I'll try my best to make better reads in the future .

~ Hrishikesh

Table of Contents

CHAPTER 1

Introduction

1.1What is Regular Expression/regex/regexp?

Simply put, It is a series of symbolic notations used to identify patterns in text or to identify certain amount of text. Its roots are at computer theory and mathematics but In this book we'll discuss only the high level view with some real life examples towards the end of the book, which is enough to get you started. Regular expressions are supported by many command-line tools and by most of the programming languages out there to facilitate the **solution of text manipulation problems**. So, now we know that regexes are used for text manipulation. That's great! Simple infact! What's more to it that makes it so complex? Well, **Not** all Regular Expressions are the same. They vary from language to language and tool to tool.

For eg. The '%' in SQL is the '*' in POSIX standards.

1.2What makes it vary from tool to tool?

Regular Expression Engines . These are softwares that process regexes. Talking of the real world, Perl, PCRE, PHP, .NET, Java, JavaScript, XRegExp, VBScript, Python, Ruby, Delphi, R, Tcl, POSIX, and many more are there, these happen to support regex . The syntax and behavior of a particular engine is called a **regular expression flavor**, and as the expression flavor differ from engine to engine, the regexes are not identical. For our discussion, we will limit ourselves to regular expressions as described in the POSIX standard, as I stated earlier, this is a Linux Centric book.

1.3Getting Introduced to our good old pal *grep*

There are various utilities to try,practice and use regex, I have listed some in the tools section at the end of this book. We'll be fine with grep though :)

The grep command prints the lines from a file or input stream that matches a **regular expression**. It's the short name for **global regular expression print** . This part is all about using grep, so if you already know how to use it, you might want to skim over. So, Let's lab. Pop up terminal.

```
$ grep root /etc/passwd
```

This command will print the lines in the */etc/passwd* file that contain the text sequence "*root*". The same could be achived using `$ cat /etc/passwd | grep root` (Read On, in case you didn't get what I did there)

```
$ grep root /etc/*
```

This command will check every file in */etc* that contains the text sequence "*root*".

Regular expressions are more powerful than wildcard-style patterns, and they have a different syntax. I'll show you the differences in wildcard-style and regex in the next chapter.

Continuing with our exploration with grep,

Two of the most popular **grep options** are -i (for case-insensitive matches) and -v (which inverts the search, that is, prints all lines that don't match). Following is a list of grep options which otherwise you can get by using the man command for grep. (Check the grep man page for more options)

Some grep Options: -

Commands	Description
-i	`Ignore case.`
-v	`Invert match.`
-c	`Print the number of matches`
-l	`Print the name of each file that contains a match instead of the lines themselves.`
-L	`Like the -l option, but print only the names of files that do not contain matches.`
-n	`Prefix each matching line with the number of the line within the file.`
-h	`For multifile searches, suppress the output of filenames.`

These are some grep specific options, these are not related with regexes yet. Keep that in mind.

The syntax for grep is as follows: -

```
$ grep [options] regex [file...]
```

Now, we shall look at some more grep examples. In real life grep is almost always used after a pipe.

```
$ ls /usr/bin | grep zip
```

This will list all the files in the */usr/bin* directory whose **file-names** contain the substring “zip”. The pipeline just makes the output of the **ls** command the input for the **grep** command.

```
$ grep -l zip mylist*.txt
```

This will list the name of the file which contains the sequence “zip” in it's content where the files present may look like, **mylist_*cool*.txt , mylist*oo*.txt,mylist*anything*.txt.** The astrikes means anything after. Later we'll also get introduced to egrep. We'll be discussing that stuff in detail in a later chapter.

Summary

So In this chapter we had a general introduction to regex and grep which we'll be using throughout the whole book.

CHAPTER 2

Globbing and Regular Expressions

2.1The Common Newcomer Mistake

When I was starting off I thought Standard Wildcards and regex were same. So wrong I was!

Shell file name globbing(Standard Wildcard) and **regular expressions** use some of the same characters, and they have similar purposes, but they aren't compatible.

Let's get deeper,

A **wildcard** is a generic term referring to something that can be substituted for all possibilities. For a GNU/Linux system there are two different major ways that wildcards are used,

- Globbing patterns/Standard wildcards (Commonly Used)
- Regular Expressions (More Powerful)

2.2 More on the differences.

- File name expansion predates regular expressions *(That's why Quoting is necessary as we'll get to know later)*
- Regexes are slower than standard wildcards
- While `*.txt` is easily understandable by casual users, the equivalent `.*\.txt$` is something more targeted to experienced users. [Explained in Chapter 6]

  Here `.txt` is used in globbing while `.*\.txt` is used in regex.
- In regular expressions, the period **('.')** character refers to any single character. It is similar to the question mark **("?")** character in wildcard pattern matching.

  As a result:

  example.com not only matches **example.com** but also matches **exampleacom**, **examplebcom**, **exampleccom** and so on.
- There is a rough similarity between file globbing syntax and regexes, but if you need regex matching of file names, you need to do it another way. Take the all time popular **find** command for example, `find -regex` is one option. *(Notice that there is also* `find -name`*, by the way, which uses glob syntax.)*

  Learn more about *standard wildcards* here.(You Must)

CHAPTER 3

Regular Expressions: A Deeper Look

Uptill now we have been discussing the abstracts and facts about Regular Expressions. In this chapter, you and I get introduced to the real workhorses in regular expressions.

Also, when you'll feel like this can't get anymore complex! I will have you know that POSIX splits regular expression implementations into two kinds: **BRE** and **ERE**. About Which We'll be discussing extensively in a later chapter.

3.1 Metacharacters and Literal characters

Metacharacters are also sometimes refered to as Special Characters.

If you have ever used grep you already have been using regular expressions all along, albeit very simple ones. Let's first look into literal characters.

3.1.1 Literal Characters

Let's have a string, ***"hrishi is a fan of lapti"***

In this string if grep is asked to search for "*a*" It will find you the single "a", the "a" from "f**a**n" and also the "a" from "l**a**pti". So, it does not really matter to the regex engine if it's inside a word or it's independent. If it matters to you, you will need to tell that to the regex engine by using *word boundaries*. More on that later.

```
$ echo hrishi is a fan of lapti | grep a
Output:
hrishi is a fan of lapti
```

Self Explanatory, Let's have a look at another one.

```
$ echo hrishi is a fan of lapti | grep is
hrishi is a fan of lapti
$ echo hrishi is a fan of lapti | grep hrishi
hrishi is a fan of lapti
```

When we give in **“is”** we are essentially asking the regex engine to search for an “i” immediately followed by a “s”.

When we give in **“hrishi”** we are essentially asking the regex engine to search for a “h” immediately followed by a “r” immediately followed by an “i”immediately followed by a “s” immediately followed by a “h” immediately followed by a “i”.

In the above examples, what we used were literals. I'll be formally defining it in a while. Also, regex engines are by default case sensitive. That's a great opportunity to leverage the -i option in grep.

3.1.2 Metacharacters or Special Characters

In short, these are metacharacters,

```
^ $ . [ ] { } - ? * + ( ) | \ [^]
```

This is not perfectly correct. That's because the metacharacters differ from engine to engine. The next chapter is all about Metacharacters, we'll be discussing each in details.

Formal Definition for Literals (I guess it's not a formal one!)

> All other characters excluding Metacharacters are considered literals.

At this point, I would highly recommend you to read about shell expansions and quotings in Bash(If possible), if you don't know about them.

Characters like the single quote and double quote are not metacharacters with a good reason. Single quotes and double quotes are language specific. Their use differ from language to language.

> As we can see, many of the regex metacharacters are also characters that have meaning to the shell when expansion is performed. When we pass regular expressions containing metacharacters on the command line, it is vital that they be enclosed in quotes to prevent the shell from attempting to expand them.

Use of backslash in shell expansion is also a very important topic here, you might want to brush up on it.

A Tip not to get confused

Well, In Regular expressions, we use /(backslash) as an escape character. And as we'll read in Chapter 5 that POSIX splits into two forms ERE and BRE. In BRE the backslash is used to activate the metacharacter. While in ERE it's the opposite.

Yes you need to use / even if you are quoting. Quoting is just for letting the terminal know that "*No terminal No, You don't have to expand me, I am a regular expression*" whereas \ (backslash) is to activate/deactivate the metacharacter in the expression itself.

Summary

This was overview on the real workhorses that we are going to ride on an on till the end of this book.

CHAPTER 4

Metacharacters

In this Chapter, We'll be going through metacharacters as described in POSIX standards.

In regular expressions, you will come across many core concepts. Before we even start on metacharacters, I would like to give you a quick headsup to all popular regex lingos.

*You Might Want To Skip this part and Jump right into Metacharacters, But I highly reccomned that you read his part, also these terms will be discussed extensively when I'll be describing metacharactersand giving examples of regular expressions.. **Terms are listed in NO perticular order***

- **Non-Printable Characters :** Special character sequences to put non-printable characters in your regular expression. eg. *\n for line feed (ASCII 0x0A)*
- **Anchors/Where do we match :** Anchors do not match characters. Instead, they match zero-length features of a piece of text, such as the start and end of the text. *eg. ^ and $*
- **Alternation/Boolean OR :** The pipe subdivides a regular expression into alternative subpatterns.
- **Repetitions:** Repetition relates to the subpattern that immediately precedes the metacharacter in the regular expression. By default, this is just the previous character.
- **Character Classes or Character Sets :** Match only one out of several characters. Simply place the characters you want to match between square brackets. Metacharacters lose their

special meaning when placed within brackets.(With some exception)

- **Word Boundaries:** The metacharacter **\b** is an anchor like the caret and the dollar sign. **\b** allows you to perform a "whole words only" search using a regular expression in the form of **\bword\b**.
- **Grouping and Capturing:** By placing part of a regular expression inside round brackets or parentheses, you can group that part of the regular expression together. This allows you to apply a quantifier to the entire group or to restrict alternation to part of the regex. There are non-capturing groups too(Not covered in this book)
- **Backreferences/Remembering patters :** Another pattern that requires a special mechanism is searching for repeated words. The expression "[a-z][a-z]" will match any two lower case letters. If you wanted to search for lines that had two adjoining identical letters, the above pattern wouldn't help. You need a way of remembering what you found. They are a mess in some substandards of POSIX, we'll know why soon. eg. `(\d)\d\1`

  The \1 refers back to what was captured in the group enclosed by parentheses. "\d" is a shorthand for degit.
- **Quantifiers/How often do we match :** Specifies how often the preceding regex should match. They are of 3 kinds and of 4 types. *Greedy,Reluctant,Possessive* **and** +,*?,*,{}* respectively.

4.1 Metacharacters In POSIX Standard

In the POSIX standard, there are 11 metacharachters to my knowledge with special meanings. If you want to use any of these characters as a literal in a regex, you need to escape them with a *backslash* in BRE. Yes backslash is also a metacharacter itself. Right after this quick refence table, we'll look into each metacharacter along with it's usage.

METACHARACTER	USAGE
\	Escape Character
^	Anchors, Negation
$	Anchors
.	Matches almost any single character.
|	Alternation
?	Repetitions , Optional Items, Quantifier (ERE)??
*	Repetitions , Quantifier
+	Repetitions, Quantifier
BRE \(\) ERE ( )	Grouping and Capturing
[ ]	Character Sets/classes
{ } (Braces) BRE `\{m,n\}` ERE `{m,n}`	General repetition quantifiers.

Now we'll be discussing all the metacharacters one by one, along with which concepts are being used and in *Chapter 6* I have 6 Examples for you which I think are sufficient to dive home the concepts.

Just a quick note on How Metacharacters are different from literals :-
If you want to use any of the metacharacters as a literal in a regex, you need to escape them with a *backslash.*

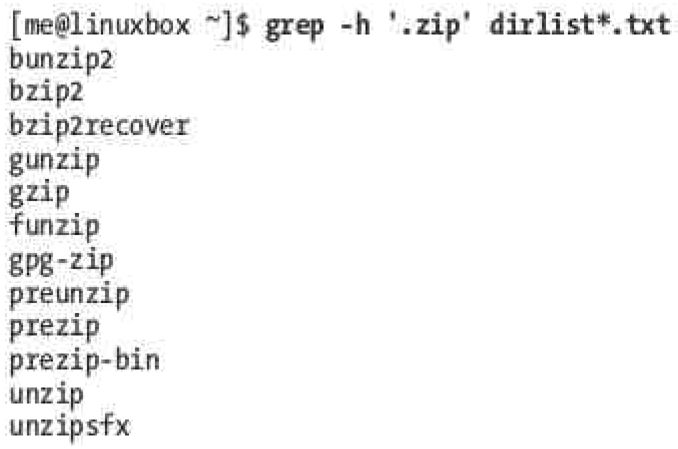
If you want to match 1+1=2, the correct regex is 1\+1=2. Otherwise, the plus sign has a special meaning.

Note that 1+1=2, with the backslash omitted, is a valid regex. So you won't get an error message. But it doesn't match 1+1=2. More on that later.

I'll Start explaining the special characters and concepts now.

4.1.1 The Any / Dot / Period Character

The dot (.) or period character is used to match any character. If we include it in a regular expression, it will match any character in that character position. **With an exception of the newline or line break character.** This exception exits mostly because of historic reasons. .



```
[me@linuxbox ~]$ grep -h '.zip' dirlist*.txt
bunzip2
bzip2
bzip2recover
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

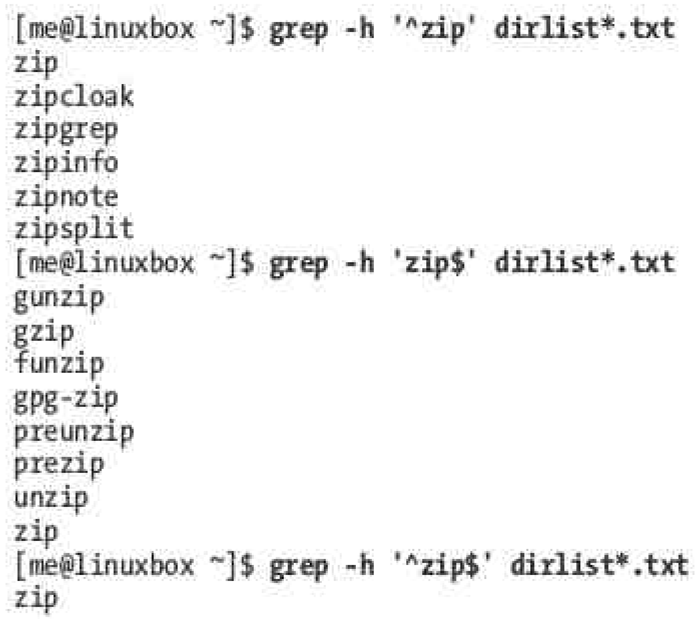
This Example is taken from *The Linux Command Line* By *William E Shot TS, JR*

. ^ $ \ ( ) | [ ] [^] - ? + * { }

4.1.2 The Anchors

The caret (^) and dollar sign ($) characters are treated as anchors in regular expressions. This means that they cause the match to occur only if the regular expression is found at the beginning of the line (^) or at the end of the line ($).

For Example,



```
[me@linuxbox ~]$ grep -h '^zip' dirlist*.txt
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
[me@linuxbox ~]$ grep -h 'zip$' dirlist*.txt
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
unzip
zip
[me@linuxbox ~]$ grep -h '^zip$' dirlist*.txt
zip
```

This Example is taken from *The Linux Command Line* By *William E Shot TS, JR*.

But, the ^(caret) character has other usage too, i.e **Negation.**

We'll talk about it when we'll be discussing *character sets.*

. ^ $ \ ( ) | [ ] [^] - ? + * { }

4.1.3 The Backslash (\ )

Used to remove the special value for a metacharacter. For eg. the regex 'Comin?' will match the words 'Coming', 'Comint', and the question 'Comin?'. But the regular expression 'Comin\?' will only match the question 'Comin?'. *(We'll be discussing **"?"** in a while)* Along with that, It is also used to give special meaning to ordinary literals. For eg. '\b' matches a word boundary. And '\d' matches 0 to 9. grep does not support \d though.

In case of BRE :

- Grouping is done with backslashes followed by parentheses '\(', '\)'.
- The alternation operator is '\|'.
- Many more usage, you'll get to know when you'll be uning it on your own.

A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example '\2' matches the second group expression.

. ^ $ \ ( ) | [ ] [^] - ? + * { }

4.1.4 Subexpression / Grouping and Capturing Group

\`(' and \`)' Parentheses can be used to define a subpattern within a regular expression. This is useful for providing *alternation* just within a subpattern of a regular expression and for applying a repetition metacharacter to more than a single character.

For eg. The regular expression a(bee)?t matches at or abeet but not abet

Basically (bee) becomes one. If it exists, it will exist together otherwise it won't exist.

. ^ $ \ ( ) | [ ] [^] - ? + * { }

4.1.5 Alteration

Separates tokens, one of which must be matched, much like a logical OR statement. Any portion of a regular expression that uses the '|' symbol is often enclosed in parentheses to disambiguate the tokens to which the '|' applies. *Use \| for BRE.*

For eg. The regex "cat|sat" means: the letter \`c', followed by the letter \`a', followed by the letter \`t', or the letter \`s', followed by the letter \`a', followed by the letter \`t'.

Whereas If we want to improve the first example to match *whole words only*, we would need to use \b(cat|dog)\b

\b uses a concept called word boundries which is discussed later.

. ^ $ \ ( ) | [ ] [^] - ? + * { }

4.1.6 Bracket Expressions Character Classes / Sets

A bracket expression matches a single character that is contained within the brackets. For eg. [abc] matches "a", "b", or "c", and [a-z] specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: [abcx-z] matches "a", "b", "c", "x", "y", or "z", as does [a-cx-z].

A set may contain any number of characters, and *metacharacters lose their special meaning when placed within brackets*. However, there are two cases in which metacharacters are used within bracket expressions and have different meanings. The first is the caret (^), which is used to indicate **negation**; the second is the dash (-), which is used to indicate a character range.

`. ^ $ \ ( ) | [ ] [^] - ? + * { }`

Negation and Character Ranges

Negation: Matches a single character that is not contained within the brackets. For example, [^abc] matches any character other than "a", "b", or "c", and [^a-z] matches any single character that is not a lowercase letter from "a" to "z". These forms can be mixed: [^abcx-z] matches any character other than "a", "b", "c", "x", "y", or "z". The caret character invokes negation only if it is the first character

within a bracket expression; otherwise, it loses its special meaning and becomes an ordinary character in the set.

`. ^ $ \ ( ) | [ ] [^] - ? + * { }`

Character Ranges : If we wanted to construct a regex that would find every word in our list.txt file whose name begins with an uppercase letter, we could do this:

```
grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXYZ]' list.txt
```

Well there must be some easy and short way out, There comes (-) dash. The following will do the job for us too.

```
grep -h '^[A-Z]' list.txt
```

- The - character is treated as a literal character if it is the last character or the first characted after "^" : [^abc-], [^-abc].
- The] character is treated as a literal character if it is the first character after "^" : [^]abc].

Some other popular ranges are:

[A-Z], [a-z], [A-Za-z], [0-9], [0-9A-Fa-f], [A-Za-z0-9]

. ^ $ \ ( ) | [ ] [^] - ? + * { }

*So, Now we are left with {}, ? * and +* . Before we dive into them, I would like to give you some examples so that you build up your confidence along the way.

When you try regexes on terminal always put them in quotes.

Otherwise, shell will try to interpret it as one of its spcl. chars.

Some Super Easy Examples

Use grep to test these. Make a file with words like hat, pat, cat, xyzcat, hatuuu etc.

- `[hc]at` matches "*hat*", "*cat*", *"hahahat", "catlulu" etc.*
- `[^b]at` matches all strings matched by `.at` except "*bat*".
- `^[hc]at` matches "*hat*" and "*cat*", but only at the beginning of the string or line.
- `[hc]at$` matches "*hat*" and "*cat*", but only at the end of the string or line.
- `\[.\]` matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "*[a]*" and "*[b]*". Here even if you quote you have to use backslash because, to the regex engine [] are reserved for character sets.

Pop Quiz : ***Regular Expression Matching a Valid Date? Answer:-***

```
^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])$
```

Wow! That looks overwhelming! Let me explain it to you. Let our format be **yyyy/mm/dd** . This is not the perfect one like it's not made for leap-year and other special needs, but this is runnable. So here's the explanation. The command on the terminal will look like this :-

```
cat foo | egrep '^(19|20)[0-9][0-9][- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])$'
```

Try to solve the whole command and regular expression. It's based on what we've already discussed. Use grep and find out why I haven't used grep but egrep (HINT: see table) . Click Here For Full Explanation.

4.1.7 Quantifiers

Various metacharacters indicate how many instances of a character, character set or character class should be matched. A quantifier must NOT

- Follow another quantifier
- Follow an opening parenthesis
- Be the expression's first character.

**(star) , +(plus) , ?(Question mark), {}(Curly Braces).* These are the 4 types of quantifiers. Let's talk about each and after that we'll finish up with the kinds of quantifiers and some light on advanced topics which are otherwise out of the scope of this book.

These metacharacters are used to specify *how many times a subpattern can repeat*. The repetition relates to the subpattern that immediately precedes the metacharacter in the regular expression. **By default**, this is just the **previous character**, but if the preceding character is a closing square bracket or a closing parenthesis then the modifier relates to the entire character set or the entire subpattern within the parentheses. You'll be clear on that when we'll be doing some examples.

Star (*) : "*Zero or more quantifier*". Preceeding item must match zero or more times.
eg. ke* matches k or ke or keeeeeeeeee

Plus (+) : "*One or more quantifier*". Preceeding item must match one or more times.
eg. be+ matches be or bee but not b

Question Mark (?) : "*Zero or one quantifier*". Preceeding item must match one or zero times. eg. colou?r matches color or colour but not colouur

Curly brackets ({}) : "*Min/Max quantifier*". This again can be used in 3 ways. Python programmers will find the concept similar to slicing. But Values must not be -ve and greater than 255. Intervals are specified by '\{' and '\}'. Invalid intervals such as 'a\{1z' are not accepted. Most regular expression flavors treat the brace { as a literal character, unless it is part of a repetition operator like a{1,3}. So you generally do not need to escape it with a backslash if it's only one {.

Exact no. of times for the preceeding item to match.	**Min** no. of times for the preceeding item to match.	**Min & Max** no. of times for the preceeding item to match.
{3} means Exactly 3	{3, } means 3 or more	{3,5} means 3,4 or 5
[0-9]{3} matches any three digits in range 0-9	[0-9]{3, } matches any three or more digits	[0-9]{3,5} matches any three, four, or five digits

?	*	+	{}
0 or 1 (Match Itself)	0 or More (Not Match Itself)	1 or More (Match Itself)	Min /Max

We'll do command line examples in chapter 6 , so if right now you are not able to get this down on the terminal, it's perfectly fine. By the way, on he terminal, it would look like:

```
cat filename | grep '^[0-9]\{3\}$'
```

where filename contains numbers of our need + other numbers. Try removing the anchors and see the results.

You need to escape the quantifiers using / . * is an exception sometimes though.

Now we know about the types, let's find about it's kinds.

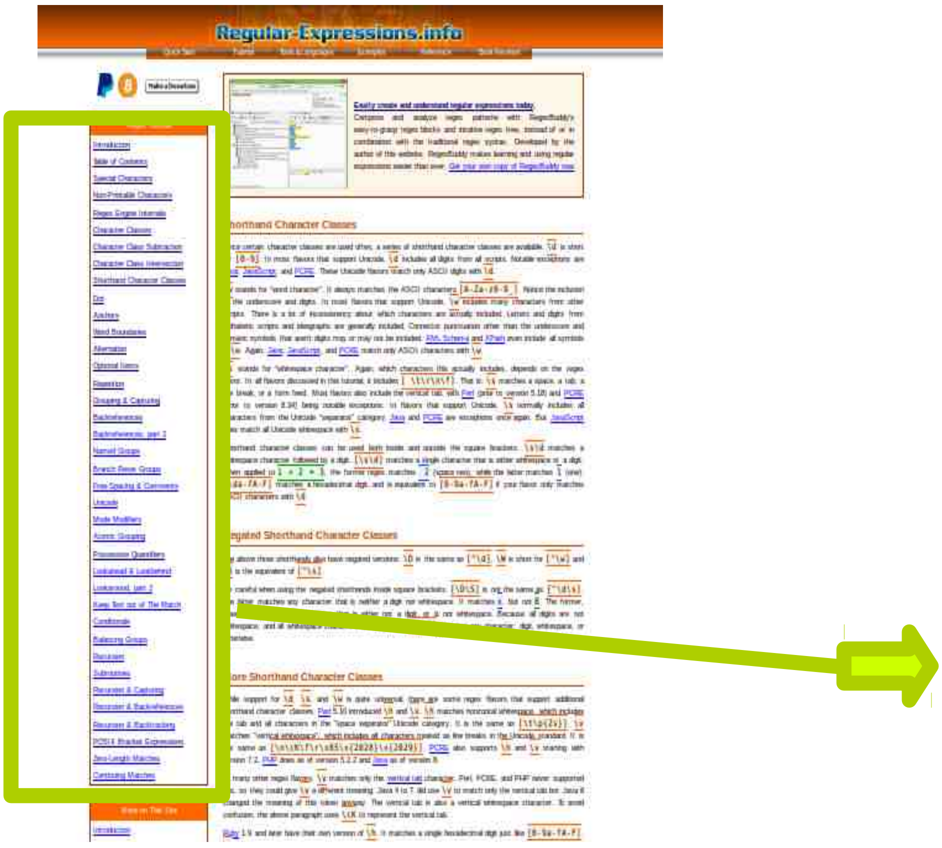
They are of three types, greedy, reluctant and possessive. There's a good read here about them. Stackoverflow and Oracle Docs

. ^ $ \ ( ) | [ ] [^] - ? + * { }

4.2 Other Concepts and Links

This is the first version of this book and I am trying to make this as beginner friendly as possible. Here I am listing links to articles and documents on some advanced topics. I'll try to add those to the book in the next revision. Also if you want me to add something more to the next revision ping me on twitter. *@Hrishikesh_Bman*

Backrefrences , **Wordboundaries** , **Lookaround** , **Lookbehind**, **backtracking** and many other concepts are there which you need to know to truly master regular expressions. There are books for that. But as of now, Here is a screen-shot of regularexpression.info,



Here is the index for topics, You might find this useful. Explaining all of these is quite pointless as this book aims at beginners . If you want to get deeper you want to,
*Google about the topic
*Read about them from the books I recommend in the **What's Next** section.

Chapter 5

BRE , ERE and POSIX classes

Well we are in a stage where we can discuss some facts about POSIX. The IEEE POSIX standard has three sets of compliance: BRE, ERE, and SRE for Basic, Extended, and Simple Regular Expressions respectively. SRE is deprecated. So, we'll keep out discussion to BRE and ERE only.

To be honest by today's standard, the POSIX ERE flavor is very basic. The POSIX standard was defined in 1986, and regular expressions have come a long way since then, as you can see in the regex flavor comparison on this site.

BRE

When we talk the BRE flavor, it's important to mention that it's pretty much the oldest flavor still in use today. The GNU utilities grep, ed and sed use it. What separates this particular flavous from other flavors is the use of *backslash* to activate or make use of the metacharacter. (See the GNU manual for Grep here.)

Whereas, other flavous including it's brother i.e POSIX ERE use a backslash to suppress the meaning of metacharacters.

I mean in BRE, if you want to use alteration you'll have to type in “/|” instead of “|”

ERE (grep –e or egrep)

The meaning of metacharacters escaped with a backslash is reversed for some characters in the POSIX Extended Regular Expression (ERE) syntax. With this syntax, a backslash causes the metacharacter to be treated as a literal character. So, for example, \(\) is now ( ) and \{ \} is now { }. Additionally, support is removed for **\n** backreferences .

The Extended Regular Expressions or ERE flavor is used by the GNU utilities egrep and awk and the emacs editor.

Remember I said backrefrences in POSIX is a mess? That's because ERE does not even support backrefrences while BRE does. The GNU Extension adds them.

Also, The advanced topics that I have listed in he previous chapter, here's an addition to that *(addition stolen from regular-expressions.info)* :- “The new features are word boundaries and anchors. Like modern flavors, GNU supports

- \b to match at a position that is at a word boundary
- \B at a position that is not.
- \< matches at a position at the start of a word,
- \> matches at the end of a word.

- The anchor `` \` `` (backtick) matches at the very start of the subject string
- `\'` (single quote) matches at the very end. These are useful with tools that can match a regex against multiple lines of text at once.
- `^` will match at the start of a line, and `$` at the end."

POSIX Character Sets / Classes

There are some problem with ranges, such as if we want to print file-names which have only capital letters in themselves. We'll do `grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXYZ]' dirlist*.txt`

But hey! We can use range there right? Like : -

```
grep -h '^[A-Z]' dirlist*.txt
```

Well actually there will be a problem if we use Ranges. It won't give us the desired output. Try it yourselves too. The reason is historical.

So, POSIX decided to use dictionary Collation instead of ASCII order. In POSIX regular expressions, common character ranges can be specified using special character sequences of the form `[:keyword:]` . The advantage of this approach is that the regular expression will work in different languages. For example, `[a-z]` will not capture all characters in languages that include accented characters, but `[[:alpha:]]` will. These are used in brackets. They are described on the next page.

Along with character classes, *there* comes character class **subtraction**, **intersection** and **shorthand character classes**.

Explaining character class subtraction and intersection is out of the scope of this book, so here are two links to them and you can always google more about it. CC Subtraction , CC Intersection .

Shorthand Character Classes

Certain character classes are used more often than others. So we have some shorthands for that. [0123456789] = [0-9] = \d

but \d is not supported by grep so you'll have to use [0-9] instead. Here's a link to GNU grep online manual for supported shorthands. And here's a link to an article on shorthand classes .

List of POSIX Character Classes

An expression of the form `[[:name:]]` matches the named character class "name", for example `[[:lower:]]` matches any lower case character. Look at these examples and then look at the table.

- `a[[:digit:]]b` matches "a0b", "a1b", ..., "a9b".
- `a[:digit:]b` is an error: character classes must be in brackets
- `[[:digit:]abc]` matches any digit, "a", "b", and "c".
- `[abc[:digit:]]` is the same as above
- `[^ABZ[:lower:]]` matches any character except lowercase letters, A, B, and Z.

Character Class	Description	Shorthand
[:alnum:]	The alphanumeric characters; in ASCII, equivalent to [A-Za-z0-9]	
[:word:]	The same as [:alnum:], with the addition of the underscore character (_)	\w
[:alpha:]	The alphabetic characters; in ASCII, equivalent to [A-Za-z]	
[:blank:]	Includes the space and tab characters	
[:cntrl:]	The ASCII control codes; includes the ASCII characters 0 through 31 and 127	
[:digit:]	The numerals 0 through 9	\d
[:graph:]	The visible characters; in ASCII, includes characters 33 through 126	
[:lower:]	The lowercase letters	
[:punct:]	The punctuation characters; in ASCII, equivalent to [-!"#$%&'()*+,./:;<=>?@[\\\]_`{|}~]	
[:print:]	The printable characters; all the characters in [:graph:] plus the space character	
[:space:]	The whitespace characters including space, tab, carriage return, newline, vertical tab, and form feed; in ASCII, equivalent to [\t\r\n\v\f]	\s
[:upper:]	The uppercase characters	
[:xdigit:]	Characters used to express hexadecimal numbers; in ASCII, equivalent to [0-9A-Fa-f]	

There are Some terms terms that floats with character classes, Collating Sequence(POSIX only, as far as I know) , Collating symbols, Equivalence Classes.

Collating symbols, like character classes, are used in brackets and have the form `[.ch.]`. Here `ch` is a digraph. Collating systems are defined by the locale.

Equivalence classes, like character classes and collating symbols, are used in brackets and have the form `[=a=]`. They stand for any character which is equivalent to the given.

For eg, if 'a', 'à', and 'â' belong to the same equivalence class, then "`[[=a=]b]`", "`[[=à=]b]`", and "`[[=â=]b]`" are each equivalent to "`[aàâb]`". Equivalence classes are defined by the locale.

CHAPTER 6

Examples. Examples and Examples.

This chapter has a total of 6 examples based on what we have understood till now, and some from the advanced topics too.

The Examples are random. So I hope you'll be enjoying this chapter. Examples are taken from various books and online sources.

Example 1 : Matching an American Phone Number

Let's say we have a file with file-name "k" . Pop up Terminal. Inside the file "k" we have a string "223-824-7645" . Our Job is to print out that string using grep.

```
$cat k | grep  '^\(([0-9]\{3\})\|^[0-9]\{3\}[.-]\?\)\?[0-9]\{3\}[.-]\?[0-9]\{4\}$'
223-824-7645
$cat k | egrep  '^(\([0-9]{3}\)|^[0-9]{3}[.-]?)?[0-9]{3}[.-]?[0-9]{4}$'
223-824-7645
```

The first one uses BRE, so we had to use so many escape character to activate the metacharacter in each. In the other hand, with ERE you need not use so many escape characters.

I'll be explaining the ERE version, figure out the BRE version yourselves. Make sure you are quoting the regular expression to run in the terminal smoothly.

^ → Starting Anchor

(→ start of capturing group

\(→ opening of *literal* parenthesis

[0-9] → Digit range, could have used \d but grep does not support that.

{ → start of min/max quantifier

3 → Exactly 3 number of occurences

} → End of min/max quantifier

\) → Closing of literal parenthesis

| → Alteration, this alteration says that the first 3 digits can either be inside parenthesis or without parenthesis, look at the expression and you'll get the idea of it.

[.-] → this 3 digit code followed by a . or a - .

[.-]? → making the [.-] optional. ***? = 0 or 1*** times . Remember?

) → Closing of capturing group.

? → Making whatever we figured out till now optional. Because first 3 digit is the area code, can be optional.

[0-9]{3} → Digit, Exactly 3 occurences

[.-]? → making the [.-] optional.

[0-9]{4} → Digit, Exactly 4 occurences

$ → Anchor. End of line.

Well that was a long explanation, from now on I'll be giving the solution and you'll figure out how it happened by yourselves as we already discussed the concepts.

Example 2 : Back references (\n)

```
$ grep -e '^\(abc\)\1$' filename.txt
```

will match abc and abcabc

Example 3 : Match the pattern “Breaking Bad”

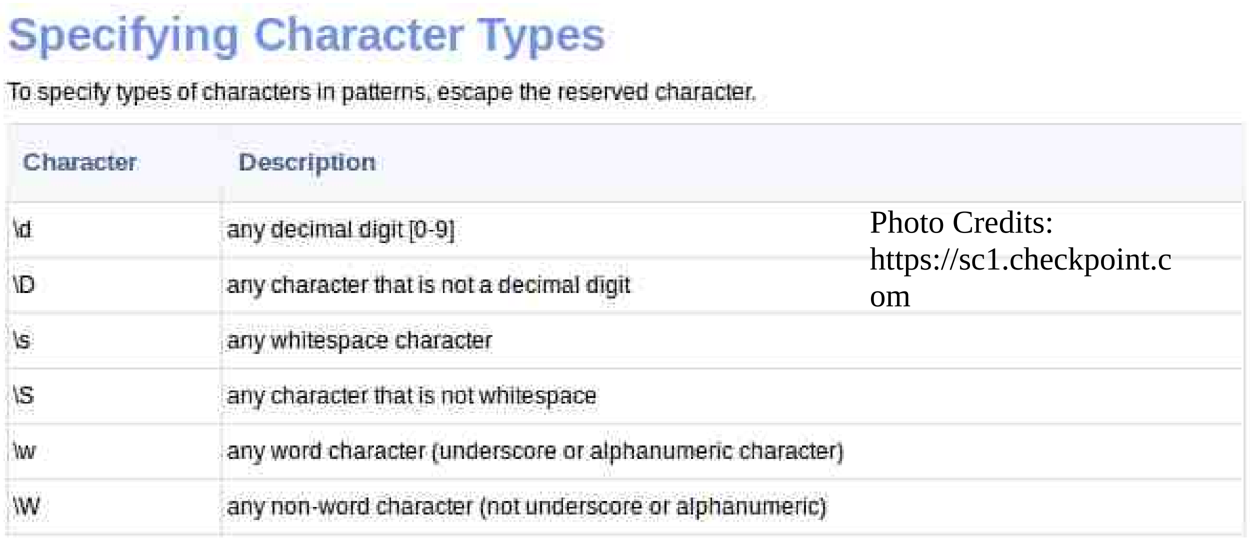
```
$ grep "BB\|\([bB]reaking\( \|\-\)[Bb]ad\)"
```

Here we are using BRE, notice we had to escape the “–“ too. This will match BB, Breaking Bad, breaking Bad, Breaking bad and breaking bad. Can You see how much powerful are regular expressions?

Example 4 : Matching an IP Address

IP addresses have classs, A,B,C etc. ipv4 I am talking about. So to match a valid IP address we'll use the following regular expression.

```
$ egrep  '\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)' filename
```



Specifying Character Types

To specify types of characters in patterns, escape the reserved character.

Character	Description
\d	any decimal digit [0-9]
\D	any character that is not a decimal digit
\s	any whitespace character
\S	any character that is not whitespace
\w	any word character (underscore or alphanumeric character)
\W	any non-word character (not underscore or alphanumeric)

Photo Credits: https://sc1.checkpoint.com

Read basics about ipv4 addressing and you'll be getting the expression. But what's the /b ? The metacharacter \b is an anchor like the caret and the dollar sign. It matches at a position that is called a "word boundary". This match is zero-length.

There are three different positions that qualify as word boundaries:

- Before the first character in the string, if the first character is a word character.
- After the last character in the string, if the last character is a word character.
- Between two characters in the string, where one is a word character and the other is not a word character.

\B is the negated version of \b. \B matches at every position where \b does not. Effectively, \B matches at any position between two word characters as well as at any position between two non-word characters.

Example 5 : Apple Anyone?

```
grep 'ap*le' filename
```

This will match ale, aple, appple. A regular expression is different from a shell wildcard as I have explained earlier. (A `*` in bash is approximately `.*` in regular expressions: Here, 0 or more occurences of 'any character'[the dot is the previous character]) The above expression is equivalent to `ap{0,}le`

. But if you do not want to match `ale` and only match aple, appple, apppple, apppppple, use the + instead of the * which means one or more: ap+le And is equivalent to `ap{1,}le`

Example 6 : To list files with extensions

`ls | grep 'pdf$'` This will print files ending with "pdf" but this may create some problem , for eg. A file has a name "thisismypdf" but it's not a pdf file, so the regex we are using is not perfect but surely it will work fine for home use, but if it's for your company, you really want to set the regex appropriately . ***Here's a tip***, If you want to list the directories only in linux. You would do `ls -al| grep '^d'` because the output of `ls -al` looks like this:-

```
drwx------  2 hrishi hrishi     4096 Feb  9 20:08 mydir
-rw-r--r--  1 hrishi hrishi  1026322 Jul  3 19:52 xyz.pdf
```

The first character is 'd' for directories and '–' for files. So you now know that you can use regex in infinite conditions. Very Powerful. Another example , .*\.txt this will match names containing ' .txt ' with anything before the ' . ' , the first '.' is anything character , the * is a quantifier saying '.' may occur 0 or more times, then \. is a literal '.' then txt is txt. Again this will change on whether you are using ERE or BRE. I think now you are getting the feel of it. If you are, tap yourself on your back. :) You have done something really good. :)

Other Tools and Links

I used LibreOffice for writing an compiling this ebook. Here are some books that I would want you to read, So What's Next?

- Introducing Regular Expressions - Michael Fitzgerald
- Mastering Regular Expressions, 3rd Edition - Jeffrey E. F. Friedl
- Regular Expression Pocket Reference, 2nd Edition - Tony Stubblebine

- Regular Expressions Cookbook, 2nd Edition - Jan Goyvaerts and Steven Levithan

Really Good Books, All Of them. Power is Good, Invest some time on those books, you'll be far more productive than you are now. Let's say you make a whole function in your program to find text, by using regular expression you can achieve that in one single line. :) Hope You Enjoyed Reading the book.

Here are Some Websites and Tools Worth your Time,

- regular-expressioninfo for sure
- checkout regexpal too

Please Give me your Feedback on twitter @Hrishikesh_Bman

BIBLIOGRAPHY

Links:

http://www.boost.org/doc/libs/1_50_0/libs/regex/doc/html/boost_regex/syntax/basic_extended.html#boost_regex.syntax.basic_extended.character_sets_

http://regular-expressions.info

https://en.wikibooks.org/wiki/Regular_Expressions/POSIX-Extended_Regular_Expressions

https://en.wikipedia.org/wiki/POSIX

http://www.thegeekstuff.com/2011/01/advanced-regular-expressions-in-grep-command-with-10-examples-%E2%80%93-part-ii/

http://www.pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html

http://www.google.com/support/enterprise/static/postini/docs/admin/en/admin_ee_cu/cm_regex.html