# Regular Expressions

# What are Regular Expressions?

- Regular expressions are symbolic notations used to identify patterns in text.
- In some ways, they resemble the shell's wildcard method of matching file and pathnames but on a much grander scale.
- Regular expressions are supported by many command line tools and by most programming languages to facilitate the solution of text manipulation problems

# grep

- The name “grep” is actually derived from the phrase “global regular expression print,” so we can see that grep has something to do with regular expressions.
- In essence, grep searches text files for the occurrence text matching a specified regular expression and outputs any line containing a match to standard output.
- So far, we have used grep with fixed strings, like so:

```
[me@linuxbox ~]$ ls /usr/bin | grep zip
```

- This will list all the files in the /usr/bin directory whose names contain the substring zip.
- The grep program accepts options and arguments this way, where regex is a regular expression:

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

The table describes the commonly used grep options.

| Option | Long Option | Description |
|---|---|---|
| `-i` | `--ignore-case` | Ignore case. Do not distinguish between uppercase and lowercase characters. |
| `-v` | `--invert-match` | Invert match. Normally, `grep` prints lines that contain a match. This option causes `grep` to print every line that does not contain a match. |
| `-c` | `--count` | Print the number of matches (or non-matches if the `-v` option is also specified) instead of the lines themselves. |
| `-l` | `--files-with-matches` | Print the name of each file that contains a match instead of the lines themselves. |
| `-L` | `--files-without-match` | Like the `-l` option, but print only the names of files that do not contain matches. |
| `-n` | `--line-number` | Prefix each matching line with the number of the line within the file. |
| `-h` | `--no-filename` | For multi-file searches, suppress the output of filenames. |
| `-q` | `--quiet, --silent` | Suppress all output. This is useful in shell scripting when we want to test if a match was found. We'll cover testing a command's *exit status* in Chapter 27. |

- To more fully explore grep, let’s create some text files to search.

```
[me@linuxbox ~]$ ls /bin > dirlist-bin.txt
[me@linuxbox ~]$ ls /usr/bin > dirlist-usr-bin.txt
[me@linuxbox ~]$ ls /sbin > dirlist-sbin.txt
[me@linuxbox ~]$ ls /usr/sbin > dirlist-usr-sbin.txt
[me@linuxbox ~]$ ls dirlist*.txt
dirlist-bin.txt   dirlist-sbin.txt     dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

- We can perform a simple search of our list of files like this:

```
[me@linuxbox ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

- In this example, grep searches all the listed files for the string bzip and finds two matches, both in the file dirlist-bin.txt. If we were interested only in the list of files that contained matches rather than the matches themselves, we could specify the –l option.

```
[me@linuxbox ~]$ grep -l bzip dirlist*.txt
dirlist-bin.txt
```

- Conversely, if we wanted to see only a list of the files that did not contain a match, we could do this:

```
[me@linuxbox ~]$ grep -L bzip dirlist*.txt
dirlist-sbin.txt
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

# Metacharacters and Literals

- The regular expression `bzip` is taken to mean that a match will occur only if the line in the file contains at least four characters and that somewhere in the line the characters b, z, i, and p are found in that order, with no other characters in between.
- The characters in the string `bzip` are all literal characters, in that they match themselves. In addition to literals, regular expressions may also include metacharacters that are used to specify more complex matches. Regular expression metacharacters consist of the following:

  `^ $ . [ ] { } - ? * + ( ) | \`
- All other characters are considered literals, though the backslash character is used in a few cases to create meta sequences, as well as allowing the metacharacters to be escaped and treated as literals instead of being interpreted as metacharacters.

# The Any Character

- The first metacharacter we will look at is the dot or period character, which is used to match any character. If we include it in a regular expression, it will match any character in that character position. Here's an example:

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

- We searched for any line in our files that matches the regular expression ".zip"
- Notice that the zip program was not found. This is because the inclusion of the dot metacharacter in our regular expression increased the length of the required match to four characters, one of which must precede the "z". Also, if any files in our lists had contained the file extension .zip, they would have been matched as well, because the period character in the file extension would be matched by the "any character," too.

# Anchors

- The caret (^) and dollar sign ($) characters are treated as anchors in regular expressions.
- This means they cause the match to occur only if the regular expression is found at the beginning of the line (^) or at the end of the line ($).
- Here we searched the list of files for the string zip located at the beginning of the line, the end of the line, and on a line where it is at both the beginning and the end of the line (i.e., by itself on the line). Note that the regular expression ^$ (a beginning and an end with nothing in between) will match blank lines.

```
[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
```

# Bracket Expressions and Character Classes

- In addition to matching any character at a given position in our regular expression, we can also match a single character from a specified set of characters by using bracket expressions. With bracket expressions, we can specify a set of characters (including characters that would otherwise be interpreted as metacharacters) to be matched. In this example, using a two-character set, we match any line that contains the string bzip or gzip:

```
[me@linuxbox ~]$ grep -h '[bg]zip' dirlist*.txt
bzip2
bzip2recover
gzip
```

- 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

- If the first character in a bracket expression is a caret (^), the remaining characters are taken to be a set of characters that must not be present at the given character position. We do this by modifying our previous example, as follows:

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

- With negation activated, we get a list of files that contain the string zip preceded by any character except b or g. Notice that the file zip was not found. A negated character set still requires a character at the given position, but the character must not be a member of the negated set.
- The caret character only invokes negation if it is the first character within a bracket expression; otherwise, it loses its special meaning and becomes an ordinary character in the set.

# Traditional Character Ranges

- If we wanted to construct a regular expression that would find every file in our lists beginning with an uppercase letter, we could do this:

```
[me@linuxbox ~]$ grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXZY]' dirlist*.txt
```

- It's just a matter of putting all 26 uppercase letters in a bracket expression. But the idea of all that typing is deeply troubling, so there is another way.

```
[me@linuxbox ~]$ grep -h '^[A-Z]' dirlist*.txt
MAKEDEV
ControlPanel
GET
HEAD
POST
X
X11
Xorg
MAKEFLOPPIES
NetworkManager
NetworkManagerDispatcher
```

- By using a three-character range, we can abbreviate the 26 letters. Any range of characters can be expressed this way including multiple ranges, such as this expression that matches all filenames starting with letters and numbers:

```
[me@linuxbox ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

- In character ranges, we see that the dash character is treated specially, so how do we actually include a dash character in a bracket expression? By making it the first character in the expression. Consider these two examples:

```
[me@linuxbox ~]$ grep -h '[A-Z]' dirlist*.txt
```

- This will match every filename containing an uppercase letter. The following will match every filename containing a dash, or an uppercase A or an uppercase Z:

```
[me@linuxbox ~]$ grep -h '[-AZ]' dirlist*.txt
```

# POSIX Character Classes

- The traditional character ranges are an easily understood and effective way to handle the problem of quickly specifying sets of characters. Unfortunately, they don't always work.
- While we have not encountered any problems with our use of grep so far, we might run into problems using other programs.
- Earlier, we looked at how wildcards are used to perform pathname expansion. In that discussion, we said that character ranges could be used in a manner almost identical to the way they are used in regular expressions, but here's the problem:

```
[me@linuxbox ~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

- Depending on the Linux distribution, we will get a different list of files, possibly an
- empty list. This example is from Ubuntu.
- This command produces the expected result — a list of only the files whose names begin with an uppercase letter

- But with the following command we get an entirely different result (only a partial listing of the results is shown):

```
[me@linuxbox ~]$ ls /usr/sbin/[A-Z]*
/usr/sbin/biosdecode
/usr/sbin/chat
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/chroot
/usr/sbin/cleanup-info
/usr/sbin/complain
/usr/sbin/console-kit-daemon
```

- As the popularity of Unix spread beyond the United States, there grew a need to support characters not found in U.S. English.
- The ASCII table was expanded to use a full eight bits, adding characters 128-255, which accommodated many more languages.
- To support this ability, the POSIX standards introduced a concept called a locale, which could be adjusted to select the character set needed for a particular location. We can see the language setting of our system using this command:

```
[me@linuxbox ~]$ echo $LANG
en_US.UTF-8
```

- With this setting, POSIX-compliant applications will use a dictionary collation order rather than ASCII order. This explains the behavior of the previous commands. A character range of [A-Z] when interpreted in dictionary order includes all of the alphabetic characters except the lowercase a, hence our results.
- To partially work around this problem, the POSIX standard includes a number of character classes that provide useful ranges of characters as described in the table on the next slide

# POSIX Character Classes

| Character Class | Description |
|---|---|
| `[:alnum:]` | The alphanumeric characters. In ASCII, equivalent to: `[A-Za-z0-9]` |
| `[:word:]` | The same as [:alnum:], with the addition of the underscore (_) character. |
| `[: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. |
| `[:graph:]` | The visible characters. In ASCII, it 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]` |
| `[:upper:]` | The uppercase characters. |
| `[:xdigit:]` | Characters used to express hexadecimal numbers. In ASCII, equivalent to: `[0-9A-Fa-f]` |

- Even with the character classes, there is still no convenient way to express partial ranges, such as [A-M].
- Using character classes, we can repeat our directory listing and see an improved result.

```
[me@linuxbox ~]$ ls /usr/sbin/[[:upper:]]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

- Remember, however, that this is not an example of a regular expression; rather, it is the shell performing pathname expansion. We show it here because POSIX character classes can be used for both.

# POSIX Basic vs. Extended Regular Expressions

- Just when we thought this couldn't get any more confusing, we discover that POSIX also splits regular expression implementations into two kinds: basic regular expressions (BRE) and extended regular expressions (ERE). The features we have covered so far are supported by any application that is POSIX compliant and implements BRE. Our grep program is one such program.
- What's the difference between BRE and ERE? It's a matter of metacharacters. With BRE, the following metacharacters are recognized:

  ^ $ . [ ] *
- All other characters are considered literals. With ERE, the following metacharacters (and their associated functions) are added:

  ( ) { } ? + |
- However (and this is the fun part), the (, ), {, and } characters are treated as metacharacters in BRE if they are escaped with a backslash, whereas with ERE, preceding any metacharacter with a backslash causes it to be treated as a literal.

# Alternation

- The first of the extended regular expression features we will discuss is called alternation, which is the facility that allows a match to occur from among a set of expressions. Just as a bracket expression allows a single character to match from a set of specified characters, alternation allows matches from a set of strings or other regular expressions.
- To demonstrate, we'll use grep in conjunction with echo. First, let's try a plain old string match.

```
[me@linuxbox ~]$ echo "AAA" | grep AAA
AAA
[me@linuxbox ~]$ echo "BBB" | grep AAA
[me@linuxbox ~]$
```

- This is a straightforward example, in which we pipe the output of echo into grep and see the results. When a match occurs, we see it printed out; when no match occurs, we see no results.

- Now we'll add alternation, signified by the vertical-bar metacharacter.

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB'
AAA
[me@linuxbox ~]$ echo "BBB" | grep -E 'AAA|BBB'
BBB
[me@linuxbox ~]$ echo "CCC" | grep -E 'AAA|BBB'
[me@linuxbox ~]$
```

- Here we see the regular expression 'AAA|BBB', which means "match either the string AAA or the string BBB." Notice that since this is an extended feature, we added the –E option to grep (though we could have just used the egrep program instead), and we enclosed the regular expression in quotes to prevent the shell from interpreting the vertical-bar metacharacter as a pipe operator. Alternation is not limited to two choices.

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB|CCC'
AAA
```

- To combine alternation with other regular expression elements, we can use () to separate the alternation.

```
[me@linuxbox ~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

- This expression will match the filenames in our lists that start with either bz, gz, or zip.
- Had we left off the parentheses, the meaning of this regular expression changes to match any filename that begins with bz or contains gz or contains zip:

```
[me@linuxbox ~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

# Quantifiers

Extended regular expressions support several ways to specify the number of times an element is matched, as described in the sections that follow.

**? - Match an Element Zero or One Time**

- This quantifier means, in effect, “Make the preceding element optional.” Let’s say we wanted to check a phone number for validity and we considered a phone number to be valid if it matched either of these two forms, where n is a numeral:

```
(nnn) nnn-nnnn
nnn nnn-nnnn
```

- We could construct a regular expression like this:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

- In this expression, we follow the parentheses characters with question marks to indicate that they are to be matched zero or one time. Again, since the parentheses are normally metacharacters (in ERE), we precede them with backslashes to cause them to be treated as literals instead.

Let’s try it.

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9]
\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)
? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
555 123-4567
[me@linuxbox ~]$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)
? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$'
[me@linuxbox ~]$
```

- Here we see that the expression matches both forms of the phone number but does not
- match one containing non-numeric characters. This expression is not perfect as it still al-
- lows mismatched parentheses around the area code, but it will perform the first stage of a
- verification.

# * - Match an Element Zero or More Times

- Like the ? metacharacter, the * is used to denote an optional item; however, unlike the ?, the item may occur any number of times, not just once.
- Let's say we wanted to see whether a string was a sentence; that is, it starts with an uppercase letter, then contains any number of uppercase and lowercase letters and spaces, and ends with a period. To match this (crude) definition of a sentence, we could use a regular expression like this:

```
^[[:upper:]][[:upper:][:lower:] ]*\.
```

- The expression consists of three items: a bracket expression containing the [:upper:] character class, a bracket expression containing both the [:upper:] and [:lower:] character classes and a space, and a period escaped with a backslash.
- The second element is trailed with an * metacharacter so that after the leading uppercase letter in our sentence, any number of uppercase and lowercase letters and spaces may follow it and still match.

- The expression matches the first two tests, but not the third, since it lacks the required leading uppercase character and trailing period.

```
[me@linuxbox ~]$ echo "This works." | grep -E '^[[:upper:]][[:upper:]
[:lower:] ]*\.'
This works.
[me@linuxbox ~]$ echo "This Works." | grep -E '^[[:upper:]][[:upper:]
[:lower:] ]*\.'
This Works.
[me@linuxbox ~]$ echo "this does not" | grep -E '^[[:upper:]][[:upp
er:][:lower:] ]*\.'
[me@linuxbox ~]$
```

# + - Match an Element One or More Times

- The + metacharacter works much like the *, except it requires at least one instance of the preceding element to cause a match. Here is a regular expression that will match only the lines consisting of groups of one or more alphabetic characters separated by single spaces:
- ^([[:alpha:]]+ ?)+$

```
[me@linuxbox ~]$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'
This that
[me@linuxbox ~]$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'
a b c
[me@linuxbox ~]$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$ echo "abc  d" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$
```

- We see that this expression does not match the line a phabetic character; nor does it match abc separates the characters c and d.

# { } - Match an Element a Specific Number of Times

- The { and } metacharacters are used to express minimum and maximum numbers of required matches. They may be specified in four possible ways as outlined in the table.

| Specifier | Meaning |
|---|---|
| {*n*} | Match the preceding element if it occurs exactly *n* times. |
| {*n*,*m*} | Match the preceding element if it occurs at least *n* times but no more than *m* times. |
| {*n*,} | Match the preceding element if it occurs *n* or more times. |
| {,*m*} | Match the preceding element if it occurs no more than *m* times. |

- Going back to our earlier example with the phone numbers, we can use this method of specifying repetitions to simplify our original regular expression from the following:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

- to the following:

```
^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$
```

- Let’s try it.

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-
9]{3}-[0-9]{4}$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]
{3}-[0-9]{4}$'
555 123-4567
[me@linuxbox ~]$ echo "5555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9
]{3}-[0-9]{4}$'
[me@linuxbox ~]$
```

- As we can see, our revised expression can successfully validate numbers both with and without the parentheses, while rejecting those numbers that are not properly formatted.

# Putting Regular Expressions to Work

Let's look at some of the commands we already know and see how they can be used with regular expressions.

**Validating a Phone List With grep**

```
[me@linuxbox ~]$ for i in {1..10}; do echo "(${RANDOM:0:3}) ${RANDO
M:0:3}-${RANDOM:0:4}" >> phonelist.txt; done
```

- This command will produce a file named phonelist.txt containing ten phone numbers. Each time the command is repeated, another ten numbers are added to the list.
- We can also change the value 10 near the beginning of the command to produce more or fewer phone numbers. If we examine the contents of the file, however, we see we have a problem.

```
[me@linuxbox ~]$ cat phonelist.txt
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
```

- Some of the numbers are malformed, which is perfect for our purposes, since we will use grep to validate them.
- One useful method of validation would be to scan the file for invalid numbers and display the resulting list.

```
[me@linuxbox ~]$ grep -Ev '^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$'
phonelist.txt
(292) 108-518
(129) 44-1379
[me@linuxbox ~]$
```

- Here we use the -v option to produce an inverse match so that we will output only the lines in the list that do not match the specified expression. The expression itself includes the anchor metacharacters at each end to ensure that the number has no extra characters at either end. This expression also requires that the parentheses be present in a valid number, unlike our earlier phone number example.

# Finding Ugly Filenames with find

- The find command supports a test based on a regular expression. There is an important consideration to keep in mind when using regular expressions in find versus grep.
- Whereas grep will print a line when the line contains a string that matches an expression, find requires that the pathname exactly match the regular expression. In the following example, we will use find with a regular expression to find every pathname that contains any character that is not a member of the following set:

  `[-_./0-9a-zA-Z]`

- Such a scan would reveal pathnames that contain embedded spaces and other potentially offensive characters

```
[me@linuxbox ~]$ find . -regex '.*[^-_./0-9a-zA-Z].*'
```

- Because of the requirement for an exact match of the entire pathname, we use .* at both ends of the expression to match zero or more instances of any character. In the middle of the expression, we use a negated bracket expression containing our set of acceptable pathname characters.

# Searching for Files with locate

- The locate program supports both basic (the --regexp option) and extended (the --regex option) regular expressions. With it, we can perform many of the same operations that we performed earlier with our dirlist files.
- Using alternation, we perform a search for pathnames that contain either bin/bz, bin/gz, or /bin/zip.

```
[me@linuxbox ~]$ locate --regex 'bin/(bz|gz|zip)'
/bin/bzcat
/bin/bzcmp
/bin/bzdiff
/bin/bzegrep
/bin/bzexe
/bin/bzfgrep
/bin/bzgrep
/bin/bzip2
/bin/bzip2recover
/bin/bzless
/bin/bzmore
/bin/gzexe
/bin/gzip
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

# Searching for Text with less and vim

- less and vim both share the same method of searching for text. Pressing the / key followed by a regular expression will perform a search. If we use less to view our phonelist.txt file, like so:

```
[me@linuxbox ~]$ less phonelist.txt
```

- then search for our validation expression, like this:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
/^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

- less will highlight the strings that match, leaving the invalid ones easy to spot.

```
(232) 298-2265
(624) 381-1078
(540) 126-1980

(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
(END)
```

- vim, on the other hand, supports basic regular expressions, so our search expression would look like this:

```
/([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}
```

- We can see that the expression is mostly the same; however, many of the characters that are considered metacharacters in extended expressions are considered literals in basic expressions. They are treated only as metacharacters when escaped with a backslash. Depending on the particular configuration of vim on our system, the matching will be highlighted. If not, try this command mode command to activate highlighting:

```
:hlsearch
```