Introduction to Regular Expressions

Demitri Muna

4 August 2017

What Are Regular Expressions?

- Regular expressions are used for textual pattern matching.
- It's like half a computer language.
- Built into most modern programming languages and all good text editors
- Functionality provided on the command line though `grep`.
- Can be linked as a library into any code, e.g. C.
- Best learned through examples.

grep

Open a terminal window in the same directory as the sample file `student_data.txt`.

Usage:

```
% grep <pattern> <files to search>
```

Simple searches:

```
% grep Junior student_data.txt
```
select all lines containing 'Junior'

```
% grep '^E' student_data.txt
```
^ = start of line: select all lines starting with 'E'

```
% grep 's$' student_data.txt
```
$ = end of line: select all lines ending with 's'

```
% grep 'oom 3' student_data.txt
```
find all offices on third floor (note space requires quotes)

```
% grep 'oom [23]' student_data.txt
```
find all offices on second and third floors

Square Brackets

Square brackets match (i.e. find) a single character.

```
[bc]at
```

will match 'bat' and 'cat' but not 'mat' or 'hat'. You can place a range of letters or numbers inside the brackets.

`[b-k]at`	matches 'bat', 'cat', 'gat' but not 'mat', 'nat', 'pat'
`[3-7]5`	matches 35, 45, 55, 65, 75 but not 85, 95, etc.
`[a-z]`	matches any single lowercase letter
`[a-zA-Z]`	matches any single upper or lowercase letter
`[a-zA-Z]$`	matches any character that appears at the end of a line
`[^abc]`	matches any character *except* 'a', 'b', 'c' – (a '^' anywhere else in brackets just matches a caret)

On the command line:

`% ls -d .[a-zA-Z]*`	* = any length of characters : list any hidden (starting with a '.') file or directory beginning with a letter

Shortcuts

These are shortcuts that define frequently used patterns.

Shortcut	Matches
\s	any whitespace, e.g. space, tab, end of line (\n, \r), etc.
\S	any character *except* whitespace
\d	any digit : [0-9]
\D	any character *excluding* numbers
\w	effectively, a word : [0-9a-zA-Z]
\W	not a word : [^0-9a-zA-Z] (i.e. inverse of \w)

Grouping

You can identify groups of characters by using parentheses. This is useful for extracting text that matches a pattern or search and replace operations.

Example:
Swap the columns
'`first_name`' and '`last_name`'.

```
first_name|last_name|city|supervisors|club|status
Cara|Rogers|New Britain|Tennant/Room 101|Sophomore|Che
Ori|Mejia|Lakeland||Senior|Debate
Leandra|Stevens|Rockford||Freshman|
Danielle|Moody|Oro Valley|Baker/Room 315, Eccleston/Ro
Josiah|Barber|Rancho Cordova||Sophomore|
Wing|Gordon|Reedsport|Baker/Room 315|Freshman|Rugby, C
```

Search for: `^([A-Za-z_]+)\|([A-Za-z_]+)\|`

- start of line
- any upper or lowercase letter or underscore
- one or more characters of the set

Replace with: `\2|\1|`

- second set of parentheses
- first set of parentheses

Note: what represents the groups differs between Python, Perl, text editors, etc.

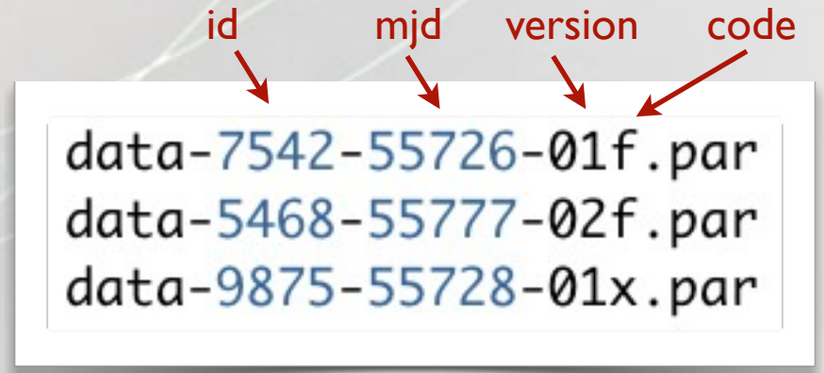
Grouping

Grouping can also let you specify more complex queries:

```
([ea][^r])
```

Matches two characters beginning with either ‘e’ or ‘a’, but *not* when followed by ‘r’.

Imagine data files that look like this, and you want to extract the information from the filename.



data – 7542 – 55726 – 0 1 f . par

```
([a-z]+)-([0-9]+)-([0-9]+)-([0-9][0-9])[a-z]\.par
```

between 1 and 7 lowercase letters — exactly 4 digits — exactly 5 digits — two or three digits — single letter

```
([a-z]{1,7})-([0-9]{4})-([0-9]{5})-([0-9]{2,3})[a-z]\.par
```

‘.’ actually matches any character, so it should be escaped with a backslash to match a period

Special Characters

Character	Meaning
\n, \t, \r	new line, tab, linefeed
.	any single character except a line break (\n, \r)
?	last item optional, e.g. `[a-z][0-9]?` matches 'a9' and 'x'
+	one or more of the last item, e.g. `[a-z]+` one or more lowercase letters
\	escape character, e.g. `\+` is an actual plus
^	start of line
$	end of line

Example: `^.+` matches any full line.

Command Line Tricks

Useful for deleting certain ranges of files:

```
% rm file-2011-0[167]-*.txt
```

Deletes files from January, June, and July.

```
% rm */.[a-zA-Z0-9]*
```

Deletes hidden files in all subdirectories.

RegExp in Python

The `match` method searches from the *beginning* of the string.

```
import re

d = "The date is Jan-04-2011 today."

m = re.match(".+([A-Za-z]{3})\-([0-9]+)\-([0-9]+).+", d)

assert m is not None, "The pattern was not matched."

month = m.group(1)
day = m.group(2)
year = m.group(3)

print m.group(0)
print m.groups()

m = re.match("Jan", d)
m = re.match("The", d)


# typical usage
if (re.match(...)):
    # do stuff
```

Import the regular expression module.

If the string does not match the regular expression, match returns None.

Three groups of parentheses for three matches, starting from index 1.

index 0 returns the original string

returns an array of all the grouped items

m is None (doesn't match start of line)

The `search` method works in exactly the same way, but searches anywhere in the string (not strictly from the start).

Build Patterns Interactively

The following web sites allow you to write regular expressions to test against sample text. You can see the matches as you type the regular expression, a detailed description of what each piece of the expression is doing, and any syntax errors.

- `https://regex101.com`
- `http://regexr.com`
- `https://www.debuggex.com`

Regular Expression Exercises

Exercise 1

Read email list file (`data/random_name_list.txt`), write a new file with the data in this format (spaces between):

first_name last_name <email>

Exercise 2

Given the `spec-*` files in the `data/` directory, write a script that lists the three numbers in each filename to a new file where the values are tab delimited, e.g.

    4055 55359 001

Hint: Look up the Python module "glob".