CST8177

Regular Expressions

What is a "Regular Expression"?

- The term “Regular Expression” is used to describe a pattern-matching technique that is used into many different environments.
- A regular expression (commonly called regex, reg exp, or RE, often pronounced rej-exp or rej-ex) can use a simple set of characters with special meanings (called metacharacters) to test for matches quickly and easily.

Regular Expressions (RE)

- At its most basic, a regex pattern is a sequence of characters that matches the item being compared:

  Pattern: **flower**

  Match: **flower**

  And nothing else!

- A key thing to remember is that a Regular Expression will try to match the *first* and the *longest* string of characters that match the pattern. This will sometimes give you a surprising result, one you didn't expect!

Once Again!

A Regular Expression will try to match the *first* and the *longest* string of characters that match the pattern.

Sometimes this will surprise you.

- You may see Regular Expressions with forward slashes around them: **`/flower/`**
- These slashes are a form of quoting, are **not** part of the Regular Expression, and may be omitted in most cases (some commands may require them, though, or something similar).
- Do not confuse Regular Expressions with filespec globbing.
  - Even though some of the forms appear similar, they are not the same thing at all:

    `/a*/` matches any number of **`a`**'s (even 0)

    `ls a*` matches all files in the **`PWD`** that begin with a single **`a`** (at least 1)
- And watch out for Extended Regular Expressions, or subsets for special purposes, PCRE, different languages, and other confusions.

Using Regular Expressions

Q: So what if we want to match either
‘**`flower`**’ or ‘**`Flower`**’?

A: One way is to provide a simple choice:

Pattern: **`[Ff]lower`**

Match: **`flower`** or **`Flower`**

Unfortunately for the confusion factor, this closely resembles **`[ ]`** in filespecs. Remember that it's different, however.

Using Regular Expressions

Q: So the [square brackets] indicate that either character may be found in that position?

A: Even better, any *single* character listed in **[ ]** will match, or any sequence in a valid ASCII range like **`0-9`** or **`A-Z`**.

Just like file globs, unfortunately.

Using Regular Expressions

Q: Does that mean I can match (say) product codes?

A: You bet. Suppose a valid product code is 2 letters, 3 numbers, and another letter, all upper-case, something like this (**`C`** for a character, **`9`** for a number):

`CC999C`

Using Regular Expressions

Pattern segments:

`[A-Z][A-Z]`	Two Letters (Uppercase)
`[0-9][0-9][0-9]`	Three Numbers
`[A-Z]`	One Letter (Uppercase)

Giving a Pattern:

`[A-Z][A-Z][0-9][0-9][0-9][A-Z]`

Good match:	**`BX120R`**
Bad match:	**`BX1204`**

Using Regular Expressions

Q: Good grief! Is there an easier way?

A: There are usually several ways to write a Regular Expression, all of them correct.

- Of course, some ways will be more correct than others (see also *Animal Farm* by George Orwell)
- It's always possible to write them incorrectly in even more ways!

Matching Regular Expressions

- The program used for Regular Expression searches is often some form of **grep**: **grep** itself, **egrep**, even **fgrep** (fixed strings) and **rgrep** (recursive), which are also **grep** options, etc.
- The general form is:

```
grep [options] regex [filename list]
```

- You will also see regexes in **sed**, **awk**, **vi**, and **less** among other places

General Linux Commands

- This is indeed the general form for all Linux commands, although there are (as always) some exceptions:

  `command [flags & keywords] [filename list]`

- That is, the command name (which might include a path) is followed by an optional (usually) set of command modifiers (usually) in any order or combination, and ends with an optional (often) list of filenames (or filespecs)
- In a pipe chain (**`cmd1 files | cmd2 | cmd3`**) of filters, the filename list is commonly found only on the first command.

Matching Regular Expressions

- **grep** is a filter, and can easily be used with **stdin**:

  echo <a string> | grep [options] <reg exp>

- Some useful options (there are more) include:
  - **-c** count occurrences only
  - **-E** extended patterns (**egrep**)
  - **-i** ignore case distinctions
  - **-n** include line numbers
  - **-q** quiet; no output to **stdout** or **stderr**
  - **-r** recursive; search all subfolders
  - **-v** select only lines not matching
  - **-w** match "words" only (be careful: what constitutes a word?)

Regular Expression Examples

- Count the number of "**robert**"s or "**Robert**"s (or any case combination) in the password file:

  grep –ic robert /etc/passwd

- List all lines with "**Robert**" in all files with "**name**" as part of the file name, showing the line numbers of the matches in front of each matching line:

  grep -n "Robert" *name*

Metacharacters

.	Any single character except newline
[…]	Any character in the list
[^…]	Any character not in the list
*****	Zero or more of the preceding item
^	Start of the string or line
$	End of the string or line
\<	Start of word boundary
\>	End of word boundary
\(…\)	Form a group of items for tags
\n	Tag number **n**

Metacharacters

\{n,m\}	**n** to **m** of preceding item (plus others)
****	The following character is unchanged, or escaped. Note its use in **[a\-z]**, changing it from **a** to **z** into **a**, **-**, or **z**.

Note that repeating items can also use the forms

\{n\} for exactly **n** items; and

\{n,\} for at least **n** items.

Ranges must be in ascending collating sequence. That is **[a-z]** and **[0-9]** are valid but **[9-0]** is not. Note that you may have to set the correct locale. We will use **LOCALE=C** for our collating sequence.

Extended metacharacters

(used, for example, with **egrep**)

+	One or more of the preceding item
?	None or one of the preceding item
|	Separates a list of choices (logical OR)
(…)	Form a group of items for lists or tags
\n	Tag number **n**
{n,m}	Between **n** and **m** of preceding item

Many of the extended metacharacters also exist in regex-intensive languages like Perl (see PCRE). Be sure to check your environment and tools before using any unusual extended expressions.

Tags

- **sed** is the stream editor, handy for mass modifications of a file.
- Tags are often used in **sed** to keep some part of the regex being searched for in the result.
- Imagine you have a file of the part numbers above and you need to replace an extra digit with the letter '**x**'. The regex for this kind of bad entry is

  [A-Z]\{2\}[0-9]\{3\}[0-9]

  so the full command is

  ```
  sed 's+\([A-Z]\{2\}[0-9]\{3\}\)[0-9]+\1X+' \
                      data > data.1
  ```

- The '**s**' operator for **sed** means substitute, but there are more available. See **man sed** for details.

Tags

- Tags in **grep** may not be as obvious to use. However, here is an example.

  echo abc123abc | grep "\(abc\)123\1“

- Think of tags as the **STR** or **M+** and **RCL** keys on your calculator
- **STR/M+** as a regex **\(...\) \(...\)**
- **RCL** as a regex **\1 \2**
- You can have up to 9 "memories" or tags in any one regex.

Tags

- Tags can be used with grep and its variants, but they are often used with tools like **sed**:

```
sed 's/\([0-9][0-9]*\)/\1\.0/g'  \
raw.grades > float.grades
```

  will insert **.0** after every string of digits in the **raw.grades** file.

- There are, as usual, other ways to do this in sed, including:

```
sed 's/[0-9][0-9]*/&\.0/g'   \
raw.grades > float.grades
```

- I recommend you use the first style for now.

Note on sed

```
sed 's/\([0-9][0-9]*\)/\1\.0/g' raw.grades
```

- Note that **sed**'s delimiter is the character that immediately follows the command option **s**; it could be any character that doesn't appear in the rest of the operand, such as

```
sed 'sX\([0-9][0-9]*\)X&\.0Xg' raw.grades
```

or as first used in the example

```
sed 's+\([0-9][0-9]*\)+&\.0xg' raw.grades
```

- The **g** after the last delimiter is for **g**lobal, to examine all matches in each line; otherwise, only the first match is used.

Bracketed Classes

[:alnum:]	**a – z**, **A - Z**, and **0 - 9**
[:alpha:]	**a - z** and **A - Z**
[:cntrl:]	control characters (**0x00** - **0x1F**)
[:digit:]	**0 - 9**
[:graph:]	Non-blanks (**0x21** - **0x7E**)
[:lower:]	**a - z**
[:print:]	**[:graph:]** plus **[:space:]**
[:punct:]	Punctuation characters
[:space:]	White space (newline, space, tab)
[:upper:]	**A - Z**
[:xdigit:]	Hex digits: **0 - 9**, **a - f**, and **A - F**

These POSIX classes are often enclosed in **[ ]** again.
Check for a char at the end of a line: **/[[:print:]]$/**

- The previous items are part of the extended set, not the basic set. A few more in the extended set that can be useful:

\w	**[0-9a-zA-Z]**
\W	**[^0-9a-zA-Z]**
\b	Any word boundary: **\< or \>**

Examples

Basic pattern for a phone number:

(xxx) xxx-xxxx

^([0-9]\{3\})_*[0-9]\{3\}-[0-9]\{4\}$

(The underscore _ is used to represent a blank)

Area code:	**([0-9]\{3\})**
Spaces:	*****
Exchange:	**[0-9]\{3\}**
Dash:	**-**
Number:	**[0-9]\{4\}**

Another Example

Extended pattern for an email address:

xxx@xxx.xxx

^\w+@\w{2,}\.[a-zA-Z]{2,4}$

Personal ID:	**\w+**
At:	**@**
Host:	**\w{2,}**
Dot:	**\.**
TLD:	**[a-zA-Z]{2,4}**

One Last Example

Extended pattern for a web page URL (regex folded)

http://xxxx.xxx/xxxx/xxxx.xxx

```
http://
   (\w{2,}\.)+[a-zA-Z]{2,4}(/\w+)*/\w+\.html?$
```

Prefix:	**http://**
Host:	**(\w{2,}\.)+**
TLD:	**[a-zA-Z]{2,4}**
Path:	**(/\w+)***
Filename:	**/\w+**
Dot:	**\.**
Extension:	**html?**