Regular Expressions

specifying strings

recognizing strings

“Matching” strings

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 a reg exp or an 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, an RE 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

- Often see Regular Expressions with forward slashes around them: **`/flower/`**.
- The slashes are a form of quoting, are not part of the Regular Expression, and may be omitted.
- 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`** beginning with a single a.
- And watch out for Extended Regular Expressions, or subsets for special purposes, 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`**

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.

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

Using Regular Expressions (cont)

Pattern segments:

[A-Z][A-Z] ← Two Letters (Uppercase)

[0-9][0-9][0-9] ← Three Numbers

[A-Z] ← A 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.
- It's always possible to write them incorrectly in even more ways!

Matching Regular Expressions

- The program used for Regular Expression searches is usually some form of grep: grep itself, egrep, fgrep, rgrep, etc.
  The general form is:

```
grep [options] <reg exp> [filename list]
```

General Linux Commands

- This is 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, and ends with an optional (often) list of filenames (or a filespec).

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
  - `-i` ignore case distinctions
  - `-n` include line numbers
  - `-q` quiet; no output to stdout or Stderr
  - `-v` match only lines not matching the pattern

Regular Expression Examples

- Count the number of "Todd"s in the password file:

```
grep -ic Todd /etc/passwd
```

- List all "Todd"s in **/etc/passwd**, showing the line numbers of the matches:

```
grep -n Todd /etc/passwd
```

Regular Expression 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*
`\{`*n*`\}`	Exactly *n* of preceding item
`\{`*n*`,\}`	*n* or more of preceding item
`\{`*n*`,`*m*`\}`	Between *n* and *m* of preceding item
`\`	The following single character is normal unchanged, , or escaped. Note its use in `[a\-z]`, changing it from `a` to `z` into `a`, `-` or `z`.

Extended metacharacters for egrep

+	**One or more of the preceding item**
?	**None or one (0 or 1) 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}	**Exactly *n* of preceding item**
{n,}	***n* or more of preceding item**
{n,m}	**Between *n* and *m* of preceding item**

Many of these also exist in Regular Expression-intensive languages like Perl. But be sure to check your environment and tools before using any unusual extensions.

Tags

- Tags in grep aren't 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+ → \(...\) \(...\)**
- **RCL → \1 \2**
- **You can have up to 9 "memories"**

Bracketed Classes

`[:alnum:]`	a-z, A-Z, and 0-9
`[:alpha:]`	a-z and A-Z
`[:cntrl:]`	Control characters (ASCII 0x00-0x1F)
`[:digit:]`	0-9
`[:graph:]`	Nonblank characters (ASCII 0x21-0x7E)
`[:lower:]`	a-z
`[:print:]`	`[:graph:]` plus space (0x20-0x7E)
`[:punct:]`	Punctuation characters
`[:space:]`	White space (newline, space, tab)
`[:upper:]`	A-Z
`[:xdigit:]`	Hex digits: 0-9, a-f, and A-F

To use these POSIX classes, they are often enclosed in [ ] again. Thus, to check for punctuation at the end of a line:

/[[:punct:]]$/

- The previous are also 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`	**Word boundary: `\<` or `\>`**

Examples

Basic pattern for a phone number:

(xxx) xxx-xxxx

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

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

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?`