

# 09 – Expansions and Regular Expressions

CS 2043: Unix Tools and Scripting, Spring 2019 [2]

---

Matthew Milano

February 11, 2019

Cornell University

# Table of Contents

1. Shell Expansion
2. **grep** and Regular Expressions

As always: Everybody! ssh to wash.cs.cornell.edu

- Quiz time! Everybody! run **quiz-02-11-19**
- You can just explain a concept from last class, doesn't have to be a command this time.

# Shell Expansion

---

# Expansion Special Characters

- There are various special characters you have access too in your shell to expand phrases to match patterns, such as:

```
* ? ^ { } [ ]
```

- These special characters let you match many types of patterns:
  - Any string.
  - A single character.
  - A phrase.
  - A restricted set of characters.
  - Many more, as we will see!

# The \* Wildcard

- The \* matches any *string*, including the null *string*.
- It is a “greedy” operator: it expands as far as it can.
- Is *related* to the **Kleene Star**, matching *0 or more* occurrences.
- For shell, \* is a *glob*. See [3] for more.

```
# Does not match: AlecBaldwin
```

```
$ echo Lec*
```

```
Lec.log Lecture1.tex Lecture1.txt Lecture2.txt Lectures
```

```
# Does not match: sure.txt
```

```
$ echo L*ure*
```

```
Lecture1.tex Lecture1.txt Lecture2.txt Lectures
```

- This is the greedy part:  $L^* \Rightarrow \text{Lect}$

```
# Does not match: tex/ directory
```

```
$ echo *.tex
```

```
Lecture1.tex Presentation.tex
```

- Matches **existing files/dirs**, does *not* define sequence

# The ? Wildcard

- The ? matches a *single* character.

```
# Does not match: Lec11.txt
```

```
$ echo Lec?.txt
```

```
Lec1.txt Lec2.txt Lec3.txt
```

- Lec11 not matched because it would have to *consume* two characters, the ? is *exactly one* character
  - Which character, though, doesn't matter.

```
# Does not match: ca cake
```

```
$ echo ca?
```

```
can cap cat
```

- Again matches existing files/dirs!

# Creating Sets

- **[brackets]** are used to define *sets*.
  - Use a dash to indicate a range of characters.
  - Can put commas between characters / ranges (**[a-z,A-Z]**).
    - Means *either* one lower case or one upper case letter.
  - **[a-z]** only matches **one** character.
    - **[a-z][0-9]**: “find exactly **one** character in **a..z**, *immediately* followed by **one** character in **0..9**”

Input	Matched	Not Matched
<b>[SL]</b> ec*	Lecture Section	Vector.tex
Day <b>[1-3]</b>	Day1 Day2 Day3	Day5
<b>[a-z][0-9].mp3</b>	a9.mp3 z4.mp3	az2.mp3 9a.mp3



# Inverting Sets

- The ^ character is represents *not*.
  - **[abc]** means *either a, b, or c*
  - So **[^abc]** means *any character that is not a, b, or c*.

Input	Matched	Not Matched
<b>[^A-P]</b> ec*	Section.pdf	Lecture.pdf
<b>[^A-Za-z]</b> *	9Days.avi	vacation.jpg

- sets, inverted or not, again match existing files/dirs

# Brace Expansion

- **Brace Expansion:** `{... , ...}` matches any pattern inside the comma-separated braces.
- Supports ranges such as `11..22` or `t..z` as well!
- Brace expansion needs at least two options to choose from.

Input	Output
<code>{Hello,Goodbye}\ World</code>	Hello World Goodbye World
<code>{Hi,Bye,Cruel}\ World</code>	Hi World By World Cruel World
<code>{a..t}</code>	Expands to the range a ... t
<code>{1..99}</code>	Expands to the range 1 ... 99

- **Note:** NO SPACES before / after the commas!
- Mapped onto following expression where applicable:
  - Following expression must be *continuous* (whitespace escaped)
  - See next slide.
- Braces **define a sequence**, unlike previous!

# Brace Expansion in Action

```
# Extremely convenient for loops:
# prints 1 2 3 ... 99
$ for x in {1..99}; do echo $x; done
# bash 4+: prints 01 02 03 .. 99
$ for x in {01..99}; do echo $x; done

# Expansion changes depending on what is after closing brace:
# Automatic: puts the space between each
$ echo {Hello,Goodbye}
Hello Goodbye
# Still the space, then *one* 'World'
$ echo {Hello,Goodbye} World
Hello Goodbye World
# Continuous expression: escaped the spaces
$ echo {Hello,Goodbye}\ Milky\ Way
Hello Milky Way Goodbye Milky Way
# Yes, we can do it on both sides. \\n: lose a \ in expansion
$ echo -e {Hello,Goodbye}\ Milky\ Way\ {Galaxy,Chocolate\ Bar\\n}
Hello Milky Way Galaxy Hello Milky Way Chocolate Bar
    Goodbye Milky Way Galaxy Goodbye Milky Way Chocolate Bar
```

# Combining Them

- Of course, you can combine all of these!
- `cd /course/cs2043/demos/09-demos/combined`

```
# Doesn't match: hello.txt
```

```
$ ls *h[0-9]*
```

```
h3 h3llo.txt
```

```
# Doesn't match: foo.tex bar.tex
```

```
$ ls [bf][ao][row].t*t
```

```
bar.text bar.txt foo.text foo.txt
```

```
# Careful with just putting a * on the end...
```

```
$ ls [bf][ao][row].t*
```

```
bar.tex bar.text bar.txt foo.tex foo.text foo.txt
```

```
# Doesn't match: foo.text bar.text
```

```
$ ls {foo,bar}.t{xt,ex}
```

```
bar.tex bar.txt foo.tex foo.txt
```

# Special Characters Revisited

- The special characters are

```
# Expansion related special characters
* ? ^ { } [ ]
# Additional special characters
$ < > & ! #
```

- The shell interprets them in a special way unless we escape them (`\$`), or place them in **single** quotes (`'$'`).
- When executing a command in your shell, the expansions happen **before** the command is executed. Consider `ls *.txt`:
  1. Starts parsing: `ls` is a command that is known, continue.
  2. Sees `*.txt`: expand **now** e.g. `*.txt`  $\Rightarrow$  `a.txt b.txt c.txt`
  3. `ls a.txt b.txt c.txt` is *then* executed.
- Shell expansions are your friend, and we'll see them again...

# Shell Expansion Special Characters Summarized

Symbols	Meaning
*	Multiple character wildcard: 0 or <b>more</b> of <i>any</i> character.
?	Single character wildcard: exactly one, don't care which.
[ ]	Create a set, e.g. <b>[abc]</b> for <i>either a, or b, or c</i> .
^	Invert sets: <b>[^abc]</b> for anything <i>except a, b, or c</i> .
{ }	Used to create enumerations: <b>{hello,world}</b> or <b>{1..11}</b>
\$	Read value: <b>echo \$PWD</b> reads <b>PWD</b> variable, then <b>echo</b>
<	Redirection: create stream out of file <b>tr -dc '0-9' &lt; file.txt</b>
>	Redirection: direct output to a file. <b>echo "hiya" &gt; hiya.txt</b>
&	Job control.
!	Contextual. In Shell history, otherwise usually negate.
#	Comment: anything after until end of line not executed.

- Non-exhaustive list: see [4] for the full listing.

# Single vs Double Quotes

- Special characters inside *double* quotes “prefer” not to expand
  - some still need escaping
- Special characters in *single* quotes are **never** expanded.

```
# prints the letters as expected
$ for letter in {a..e}; do echo "$letter"; done
# escaping the money sign means give literal $ character
$ for letter in {a..e}; do echo "\$letter"; done
# $ is literal now, so doesn't read variable
$ for letter in {a..e}; do echo '$letter'; done
```

- Pay attention to your text editor when writing scripts.
  - Like the slides, there is syntax highlighting.
  - It *usually* changes if you alter the meaning of special characters.
- If you remember anything about shell expansions, remember the difference between single and double quotes.

# tr Revisited with Sets

## Useful POSIX Sets

Set Name	Set Value
<code>[:lower:]</code>	lowercase letters
<code>[:upper:]</code>	uppercase letters
<code>[:alpha:]</code>	alphabetic characters (upper and lower)
<code>[:digit:]</code>	digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9
<code>[:alnum:]</code>	alphanumeric characters
<code>[:punct:]</code>	punctuation characters
<code>[:space:]</code>	whitespace characters

```
# Get excited. Note single quotes because of !
$ echo 'I am excited!' | tr [[:lower:]] [[:upper:]]
I AM EXCITED!

# Component-wise: e->3, t->7, a->4, o->0, s->5
$ echo 'leet haxors' | tr [etaos] [37405]
l337 h4x0r5
```



# **grep** and Regular Expressions

---

# Time for the Magic

## Globally Search a Regular Expression and Print

`grep <pattern> [input]`

- Searches **input** for all lines containing **pattern**.
- As easy as searching for a **string** in a **file**.
- Or it can be much more, using regular expressions.
- Common use:

`<command> | grep <thing you need to find>`

- You have some **command** or sequence of commands producing a large amount of output.
  - The output is longer than you want, so filter through **grep**.
  - Reduces the output to only what you really care about!
- Understanding how to use **grep** is **really** going to save you a lot of time in the future!

## Some Useful Grep Options

- **-i**: ignores case.
- **-A 20 -B 10**: print 10 lines **B**efore, 20 lines **A**fter each match.
- **-v**: inverts the match.
- **-o**: shows only the matched substring.
- **-w**: “word-regexp” – exclusive matching, **read the man page**.
- **-n**: displays the line number.
- **-H**: print the filename.
- **--exclude <glob>**: ignore **glob** e.g. **--exclude \*.o**
- **-r**: recursive, search subdirectories too.
  - **Note**: your Unix version may differentiate between **-r** and **-R**, check the **man** page.
  - **grep -r [other flags] <pattern> <directory>**
    - That is, you specify the **pattern** first, and where to search after (just like how the **file** in non-recursive **grep** is specified last).

# Regular Expressions

- **grep**, like many programs, takes in a **regular expression** as its **input**. Pattern matching with regular expressions is more sophisticated than shell expansions, and also uses different syntax.
- More precisely, a regular expression *defines* a set of strings – if any part of a line of text is *in the set*, **grep** returns a **match**.
- When we use regular expressions, it is (usually) best to enclose them in quotes to stop the shell from expanding it before passing it to **grep** / other tools.

## WARNING

When using a tool like **grep**, the shell expansions we have learned *can* and do still occur! I **strongly** advise using *double quotes* to circumvent this. Or if you want the literal character (e.g. the *\**), use *single quotes* to disable all expansions entirely.

# Regular Expression Similarities

- Some **regex** patterns are similar / the same.

## Single Characters are Different

Shell Expansion:	<code>?</code>
Regular Expressions:	<code>.</code>

- `?` means something different in regex (Differences slide).
- Example: **grep** `"t.a"`  $\Rightarrow$  lines with **tea**, **taa**, and **steap**

## Sets are almost the Same

Shell Expansion:	<code>[a-z]</code>
Regular Expressions:	<code>[a-z]</code>

- Matches one of the indicated characters.
- Don't separate multiple characters with commas in the **regex** form (e.g. `[a,b,q-v]` becomes `[abq-v]`).

## A Note on Ranges in Sets

- Like shell wildcards, regex is case-sensitive.
- How would you match any letter, regardless of case?
  - If you take a look at the ASCII codes ([1]), you will see that the lower case letters come **after** the upper case letters.
  - You should be careful about trying to do something like **[a-Z]**.
  - Instead, just do **[a-zA-Z]**.
  - Or use the POSIX set **[[:alpha:]]**.
  - **Note:** some programs *may* accept the range **[a-Z]**.
    - But it may not actually be the range you think. It depends.

# Regular Expression Differences

- Some of the shell expansion tools are **completely** different.

## Modifiers Apply to the Expression *Before* Them

? is 0 or 1 occurrences:	$a? \Rightarrow 0 \text{ or } 1 \text{ } a$
* is 0 or more occurrences:	$a^* \Rightarrow 0, 1, \dots n \text{ } a\text{'s}$
+ is 1 or more occurrences:	$a^+ \Rightarrow 1, 2, \dots n \text{ } a\text{'s}$

- **Note:** + and ? are *extended* regular expression characters.
- Must escape (\+ and \?) or use -E or **egrep**.

# Nothing happens, they weren't escaped

```
$ grep "f?o+" combined/*.*
```

# f\? can be 0, so h{e,3}llo are found

```
$ grep "f\?o\+" combined/*.*
```

```
combined/foo.tex:1:foo
```

```
combined/foo.text:1:foo
```

```
combined/foo.txt:1:foo
```

```
combined/h3llo.txt:1:h3llo
```

```
combined/hello.txt:1:hello
```

# Curly Braces in Pattern Creation

- Recall that curly braces are an expansion:

```
$ echo h{e,3}llo
hello h3llo
$ echo "h{e,3}llo"
h{e,3}llo
```

- However, you cannot use them with **grep** like this:

```
# Second expansion: treated as file input to grep
# You can only supply *ONE* pattern!
$ grep h{e,3}llo combined/*.*
grep: h3llo: No such file or directory
combined/hello.txt:1:hello
# Double quotes won't save you: that's the literal
# string 'h{e,3}llo' at this point (so no match).
$ grep "h{e,3}llo" combined/*.*
```

- AKA you cannot *easily* do these expansions when using **grep**.
- {}**.bash are *fundamentally different* from the other expansions
  - defines a sequence, does not match existing targets.



# Final Thoughts and Additional Resources

- The regular expressions we use in our shell are the “Perl Regular Expressions.”
  - There are other regular expression syntaxes.
  - **Most** tools / languages use **perl** RE syntax.
- “Regular” regular expressions
- *Extended* regular expressions
- Python **re** (Regular Expression) module
  - Many **excellent** examples, and thorough explanations.
  - Topics of interest:
    - Greedy vs non-greedy,
    - Positive lookahead vs negative lookahead
    - Capturing vs non-capturing
- Probably the **best step-by-step tutorial there is**

# References

- [1] ASCII Table. *ASCII Character Codes and html, octal, hex, and decimal chart conversion*. 2010. URL: <http://www.asciitable.com/>.
- [2] Stephen McDowell, Bruno Abrahao, Hussam Abu-Libdeh, Nicolas Savva, David Slater, and others over the years. “Previous Cornell CS 2043 Course Slides”.
- [3] The Linux Documentation Project. *Globbering*. 2017. URL: <http://www.tldp.org/LDP/abs/html/globberingref.html>.
- [4] The Linux Documentation Project. *Special Characters*. 2017. URL: <http://www.tldp.org/LDP/abs/html/special-chars.html>.