CSE 374 Programming Concepts and Tools

Lecture 05 - Glob, Regular Expressions

Review

Review: Scripting

Goal: create small, reusable pieces of code that you can execute on the command line

- Automate tasks, do something useful, etc.
  - Computers are very good at simple, tedious tasks, and they're much faster than we are
  - Instead of doing something tedious multiple times, you can use your knowledge of bash to spend a little bit of time to write a script, and then do it instantly forever

Need to know how to:

- Use the script (usage messages!)
- Do some basic error handling/checking (using tests and conditionals)
- Tell the computer what we want to do, and how (using variables, conditionals, loops)
- Make it easily runnable (the shebang (#!) line and chmod)

Review: Usage comment and messages

Useful for future you and *anyone else* who wants to use the script

- Document what the script should do
  - Put a usage comment at the top of the file, after the shebang, like
    `# Prints the names of all image files in the current directory`
  - Doesn't have to be long – short and sweet is best
  - If it ends up being long, your idea for the script may be too complicated for bash – maybe should use Python, Perl, Ruby, etc. instead
- Document how to use it
  - If the inputs (i.e., args) are incorrect, output the usage message

Portability

- The ability to use the same script on multiple different systems, and its behavior stays the same

How to interpret usage

How to use a command is often given in one or more lines that look something like this:

```
mycommand [OPTION…] STR_ARG [STR_ARG…]
```

- parts that are uppercase are to be determined by the user: `OPTION` and `STR_ARG`
- parts NOT in brackets are **required**
  - so `mycommand -myoption "hi"` works, but `mycommand -myoption` doesn't -- it's missing the necessary STR_ARG
- parts in brackets mean that they are optional, i.e., you can run the command without them
  - `mycommand "hi"` should be valid too
- anything with … after means that you can have multiple

Review: Scripting Variables, Return Codes

- `$#` stores number of arguments (strings) entered
- `$0` stores the first string entered – the name of the command
- `$N` returns the Nth argument passed to script
- `$?` returns state of last exit
- `$@` returns a list of the arguments – a space separated string

$1 $2 ...

When a process exits you can use a number to indicate either a success (0) or failure (1-255)

- Success: `exit` or `exit 0`
- Failure: `exit 1` or.. {1-255}

Review: Tests/[…] and Conditionals

Boolean expressions can be eval'd by the `test` or `[` command

- file tests
  - file existence `-e`
  - is a directory `-d`
  - etc.
- number tests: `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`, etc.
- string tests: `=`, `!=`

Bash also has if statements

- Use the square brackets test syntax for the condition

```
if [ <test> ]; then
    commands
elif [ <test> ]
    commands
else
    commands
fi
```

The blocks in red are optional, and you can have multiple elif blocks.

Review: Loops

Iterate until the <test> is false

```
while [ <test> ]; do
  <commands>
done
```

Iterate over a list

```
for <var> in <list>; do
   <commands>
done
```

Iterate over lines of a file, until we run out of lines

```
while read <var>; do
  <commands, can use the $<var>
   variable>
done < <file>
```

Iterate over script arguments

```
for <var> in "$@"; do
  <commands>
done
```

Review: Loops

Iterate until the <test> is false: Print numbers from 1 to 10

```
counter=1
while [ $counter -le 10 ]; do
    echo $counter
    ((counter++))
done
```

Iterate over lines of a file: looking for lines that say “bananas”

```
while read line; do
  if [ $line = “bananas” ]; then
    echo “Bananas!”
  fi
done < file_containing_bananas.txt
```

Iterate over a list: Print numbers from 1-10


```
for i in {1..10}; do
    echo “$i”
done
```

Iterate over script arguments

```
for arg in "$@"; do
  echo "$arg"
done
```

Discussion

Turn and talk:

What's something small (and/or annoying) you do on your computer every day that could probably be replaced with a short Bash script?

⚠️ Stuff to watch out for !!! ⚠️

White space: Spacing of words and symbols matters

- Use **double quotes** around variable substitutions so that spaces don't mess you up
- (You may want to avoid creating directories, files, etc. that have spaces in their names. This removes a significant number of these issues)

Assign WITHOUT spaces around the equal, brackets are WITH SPACES

- `[ 5 -lt 4 ]` vs. `name=chloe`

Math: Non-numbers (including the empty string) converted to numbers can produce '0'

- `x=$((5 + A)) => x=$((5 + 0)) => x=5`

Placeholders and Metasyntactic Variables

Placeholders: parts of a code example that are meant to be replaced with your own value(s) before running

Often surrounded with <> (sometimes [])

- The brackets should be replaced when you put your own value in
- e.g., if the example is **`echo <string>`**, and you want to print **`"hi"`**, you should use **`echo "hi"`**

Metasyntactic variable: A fancy name for words used as placeholder names, typically used in computer science.

"A word that is a variable for other words"
(Wiki)

- `Examples: foo, bar, baz, quux, etc.`

Although quite common, they can be a bit of a barrier to entry

- If you're not in the know, it can be confusing
- Another con: Not as descriptive!

Demo: `lyric_count` bash script

Glob Patterns

Bash: “Glob patterns”

- Pattern syntax
  - Bash will replace a “pattern” with list of file names matching that pattern
- Enables you to pass multiple file names as arguments without typing them out individually
- Pattern matches are **based on your location** within the **file tree**
  - i.e., you will get different things in directory ~/dir1 vs. directory ~/dir2

Glob Wildcard Expansion

- "Wildcard", often written as "`*`", means "anything goes here"
- Wildcard expands into all the terms that match the pattern
- **`echo src/*`**
  - Prints every file/folder in the `src/` directory
  - Becomes: `echo src/file1.txt src/file2.txt src/file3.txt`
  - Outputs: `src/file1.txt src/file2.txt src/file3.txt`
- **`echo *`**
  - Prints every file/folder in the current directory

File Prefix/Suffix Search

- How can we find all the files beginning with a certain string?
  - e.g. Starting with `python`
  - **`echo /bin/python*`** (can also use **`ls`** instead here, instead of **`echo`**)
- How can we find all the files in the current directory ending with a certain string? (e.g. **`.txt`**)
  - **`ls *.txt`** (you can also use **`echo`** instead of **`ls`**!)
- How do we recursively find all files, in all subdirectories of our current directory, ending in **`.txt`**?
  - **`find -name "*.txt"`**

Demo: globbing

Polling Time!

Please answer in the LP5 assignment on Gradescope!

How can we copy all the files from one directory to another?
e.g. Copy all files in `src/` to `dest/`

- `cp src/ dest/`
- `cp src/* dest/`
- `cp src/ dest/*`
- `cp src/* dest/*`

`cp -t dest/ src/*`

cp(1) - Linux man page

Name

cp - copy files and directories

Synopsis

cp [*OPTION*]... [*-T*] *SOURCE DEST*
cp [*OPTION*]... *SOURCE*... *DIRECTORY*
cp [*OPTION*]... *-t DIRECTORY SOURCE*...

Description

Copy SOURCE to DEST, or multiple **SOURCE**(s) to DIRECTORY.

Regexes

Regexes

```
/[a-zA-Z_\-]+@([a-zA-Z_\-])+\.)+[a-zA-Z]{2,4}/
```

regular expression ("regex"): a pattern that describes strings that it matches

- can test whether a string matches the regex's pattern
- can use a regex to search/replace characters in a string
- very powerful, but tough to read, and tough to write

Regexes occur in many places:

- text editors (Sublime, Vim, etc.): allow regexes in search/replace
- languages: JavaScript; Java Scanner, String split
- Unix/Linux/Mac shell commands (`grep`, `sed`, `find`, etc.)
- The site regexr is useful for testing a regex

Basic Regex

```
/abc/
```

Regexes are typically written with surrounding `/`'s

The simplest regexes simply match a particular substring

The above regex matches any string containing "abc"

- Match: "abc", "abcdef", "defabc", ".=.abc.=.", ...
- Don't Match: "fedcba", "ab c", "PHP", …

Regex Wildcards, Case (in)sensitivity

`.` (a dot) matches *any character* except a line break (represented as \n for newline)

- `/.ax../` matches "Faxes", "Jaxes", "Taxes", "maxie", "faxing", etc.
- use `\.` to literally match a dot . character

`^` matches the beginning of a line, `$` the end

- `/^if$/` matches lines that consist entirely of `if`

A trailing i at the end of a regex (after the closing `/`) signifies a case-insensitive match

- `/cal/i` matches "Pascal", "California", "GCal", etc.
- This is equivalent to the **grep -i** flag (more on this shortly)

Polling Time!

Please answer in the LP5 assignment on Gradescope!

Which string is *not* matched by `/^cse...$/i`?

- "cse374"
- "CSE 373"
- "Cse216"
- "CSE12x"

Reminders:

- `^` anchors to the beginning of the line
- `.` is the wildcard – any character can go here
- `$` anchors to the end of the line
- an `i` after the last `/` makes it case insensitive

Grouping

Sequences of characters can be grouped together using parentheses like: `/(abc)/` or `/a(bc)/`

`/^(abc)$/` matches “abc” – the parentheses are ignored!

- They’re special characters
- Important for applying *quantifiers* to a whole sequence of characters

Quantifiers: *, +, ?

`*` means 0 or more occurrences of the preceding pattern

- `/abc*/` matches "ab", "abc", "abcc", "abccc", ...
- `/a(bc)*/` matches "a", "abc", "abcbc", "abcbcbc", ...
- `/a.*a/` matches "aa", "aba", "a8qa", "a!?xyz__9a", ...

`+` means 1 or more occurrences of the preceding pattern

- `/Goo+gle/` matches "Google", "Gooogle", "Goooogle", ...
- `/a(bc)+/` matches "abc", "abcbc", "abcbcbc", … but not just "a"

`?` means 0 or 1 occurrences of the preceding pattern

- `/a(bc)?/` matches substrings with only "a" or "abc"

More Quantifiers: {n}, {m, n}

`{n}` means *exactly* **`n`** occurrences

- `/abc{2}$/` matches "ab**cc**" but not "abc" or “abcd”

`{m,n}` means between **`m`** and **`n`** occurrences

- `/Go{2, 4}gle/` matches "G**oo**gle", "G**ooo**gle", "G**oooo**gle", but not "Gooooogle", ...

Character range: [start-end]

Inside a character set, denoted with `[]`, specify a range of characters with `-`

- `/[a-z]/` matches any lowercase letter
- `/[a-c]/` matches any a, b, or c
- `/[a-zA-Z0-9]/` matches any lowercase or uppercase letter or digit

Can be negated using `^`

- `/[^a-z]/` matches any character that is NOT lowercase
- `/[^a-zA-Z0-9]/` matches anything that isn't a letter or a digit

Inside a character set, - must be escaped to be matched, or put at the beginning

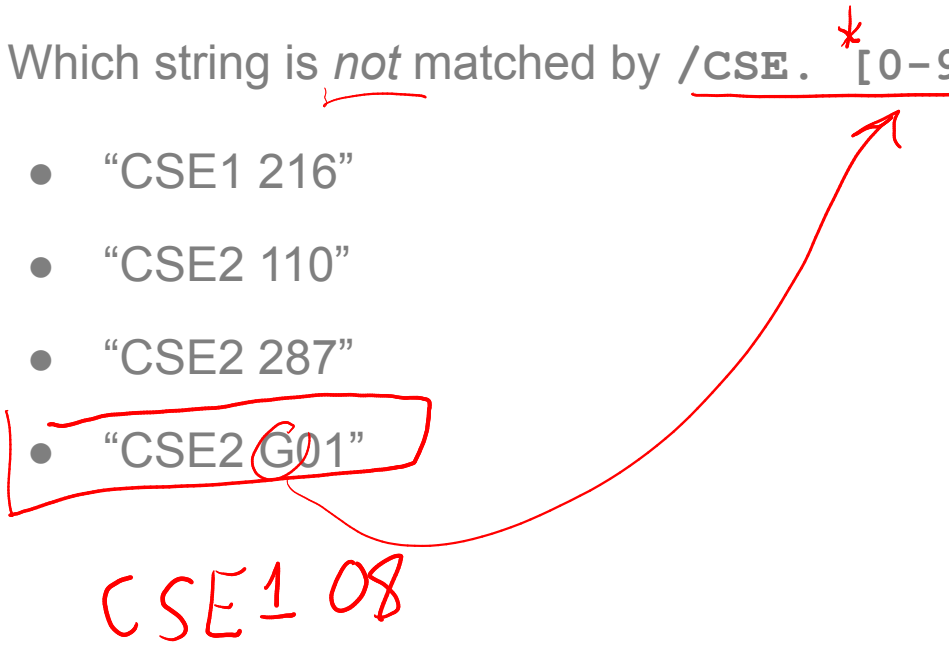
- `/[+\-]?[0-9]+/` matches an optional + or -, followed by at least one digit
- `/[-+]?[0-9]+/` does the same, but is shorter

Polling Time!

Please answer in the LP5 assignment on Gradescope!

Which string is *not* matched by `/CSE. [0-9]+/`?

- "CSE1 216"
- "CSE2 110"
- "CSE2 287"
- "CSE2 G01"



Reminder:

- `.` is the wildcard – any character can go here
- `[0-9]` means any digit from 0 to 9
- `+` is the "1 or more" quantifier

Regex special characters

`\`	escapes the following character. Many characters must be escaped: `/ \ $ . [] () ^ * + ?`	`\.\\n` matches lines containing ".\n"
`.`	matches any single character at least once	`c.t` matches `{cat, cut, cota}`
`|`	or, enables multiple patterns to match against	`a|b` matches `{a} or {b}`
`*`	Matches 0 or more of the previous pattern	`a*` matches `{, a, ,aa, aaa, …}`
`?`	Matches 0 or 1 of the previous pattern	`a?` matches `{ , a}`
`+`	Matches one or more of the previous pattern	`a+` matches `{a, aa, aaa, …}`
`{n}`	Matches exactly `n` repetitions of a pattern	`a{3}` matches "`aaa`"

Regex special characters

`()`	group patterns for order of operations	`a(bc)+` matches "abc", "abcbc", "abcbcbc", ...
`[]`	Contains literals to be matched, single or range	`[a-z]` matches all lowercase letters
`^`	Anchors to beginning of line	`^a` matches lines that start with a
`$`	Anchors to end of line	`;$` matches lines that end with ;
`{min, max}`	Between min and max occurrences. Min/max be omitted to specify any number at least/at most	`a(bc){2, 4}` matches lines that contain “abcbc”, “abcbcbc”, or “abcbcbcbc”

grep

grep

A command for doing regex searches on files/stdin

- We've seen this a few times to do basic searches
- Also supports regex syntax*
- Very useful utility, often used to
  - Quickly searching within a project repo (without having to open it up in VS code or other IDEs)
  - Looking for files with a certain name
  - etc.
- *Lots* of options to do a lot of different things
  - We'll just go over a few of them in class

*Some syntax may not be supported on all systems, more on this later.

grep with options

```
grep [options] pattern [files]
```

```
grep -c "<string>" <FILENAME>          # -c count number of matches
grep -i "<string>" <FILENAME>          # -i case insensitive
grep -w "<string>" <FILENAME>          # -w checks for whole words
grep -A <N> "<string>" <FILENAME>      # -A prints N lines after match
grep -B <N> "<string>" <FILENAME>      # -B prints N lines before match
grep -r "<string>" *                   # -r recursive search from current directory
```

grep with Regex

Useful Regex Patterns

- `[a-zA-Z]` - matches any English letter
- `[0-9]*` - match any number of digits
- `(abc)*` - match any number of “abc”s
- `(foo | bar)` – matches either “foo” or “bar”

```
grep “^hello” file1        # Match all lines that start with ‘hello’

grep “done$” file1         # Match all lines that end with ‘done’

grep “[a-e]” file1         # Match all lines that contain any of the letters a-e
```

Extended Regex

`grep -E` uses “extended” regex

- In basic grep the meta-characters `?`, `+`, `{`, `|`, `(`, and `)` lose their special meaning; instead use the backslash versions `\?`, `\+`, `\{`, `\|`, `\(`, and `\)`.
- Traditional egrep did not support the `{` meta-character, so portable scripts should avoid `{` in `grep -E` patterns and should use `[{]` to match a literal **{**.
- Also `grep -e` allows to use several strings for searching: `grep -e 'abc' -e 'def' -e '123'` will look for any of the three of these strings: abc as well as def and 123.

```
grep -E '^[A-Z].*[.,]$' file.txt
```

- match all lines that start with a capital letter and end with either period or comma
- `.*` matches any number of any character

Demo: `grep` regexes

Polling Time!

Please answer in the LP5 assignment on Gradescope!

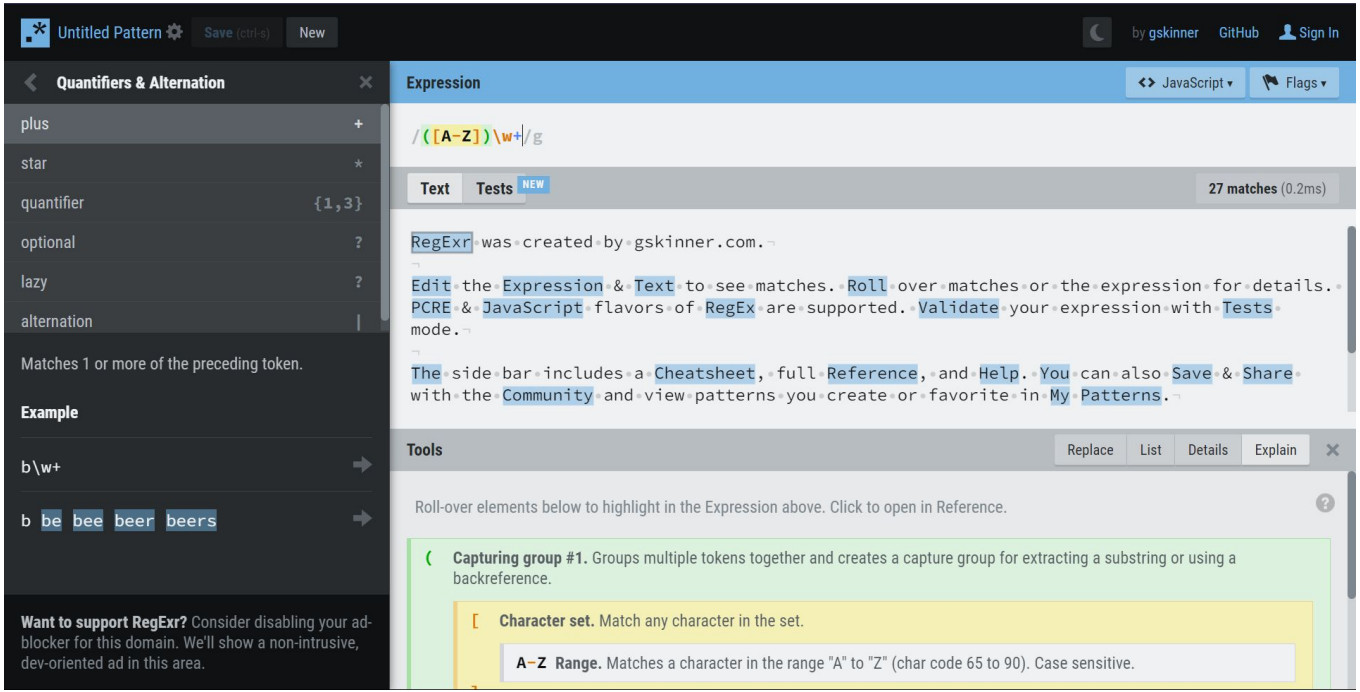
Will `grep -E '[a-z]*'` match some substring of the following strings: `"foo bar"`, `"123"`

Hint: `*` matches 0 or more of the previous pattern, `[a-z]` matches any lowercase letter

"foo bar" = no "123" = no	"foo bar" = yes "123" = no
"foo bar" = no "123" = yes	"foo bar" = yes "123" = yes

Try out Regexes online!

https://regexr.com/



Assignment Reminders

EX4 due Monday 7/6 **@10:49am**

HW1: Bash Commands is due **Friday @11:59pm**

- Try not to use late days, but max late deadline is Monday @11:59pm (two late days)

HW2 will be released soon and is **due Friday 7/10 @11:59pm**

- Use what you've learned about Bash commands and scripting to start writing your own short scripts!

Make sure to submit LP5 before the next class as well.

All of these assignments can be found on our course's Gradescope page.

NO lecture or office hours on Friday, July 4th.
Enjoy the long weekend! 🎆🎇

Next week 👀: Learning version control and C programming!