BROWN UNIVERSITY CENTER FOR COMPUTATION & VISUALIZATION

Automating tasks through scripting

HPC Certificate Program

Summary of preparation lectures

From "The Shell" at Software Carpentry:

Lecture 4: Pipes and Filters

The **redirect** operator saves command output to a file, or uses a file as input to a command.

The **pipe** operator uses one command's output as another's input.

A **filter** is a program that can accept input from stdin and output to stdout (e.g. almost all UNIX utilities).

Benefit of this approach: **modularity** and **reuse** of tools.

Summary of preparation lectures

From "The Shell" at Software Carpentry:

Lecture 6: Finding Things

The **grep** command searches for textual patterns.

The **find** command recursively finds files that match certain criteria (type, name, size, access date, permissions, etc.).

Lecture 8: Variables

Shell variables can hold strings, or lists of strings (conventional delimiter is a colon).

Values can be accessed using the **$** operator.

More advanced topics

The full syntax for **regular expressions**: useful for grep, as well as python, perl, and other languages that support 'regex'

Additional UNIX tools: **awk** and **sed**

Using control logic in the shell and in scripts: **if/else** and **for**

Writing your own shell scripts that use **getopts** to easily parse arguments from the command line

Regular expressions

A regular expression (RE) is a string that, through a precise syntax, defines a **set of all 'matching' strings**.

Many programming languages and programs can use a regular expression to search for and extract a set of matches (or lines containing matches) from textual data.

Useful reference pages:
Python: Regular Expression Syntax
PHP: PCRE Patterns
Perl regular expressions

Regular expressions

Each character in an RE is interpreted as either a **literal** (the actual character itself) or as a **meta-character** with a special meaning.

Examples of common meta-characters:

.	match any character except newline
|	or operator: match the string before or the string after
*	match the previous character 0 or more times
\	the 'escape' character that nullifies the special meaning of a meta-character after it

The RE '.*' is very general: it will match any number of any characters except newline (i.e. any line of text).

Simple expressions

Simple strings that don't contain any meta-characters are valid REs, e.g.

```
grep -P 'error' logfile.txt
```

Strings can also be or'd with the **|** operator: 'error|warning'

Use the \ escape character in literal strings that contain meta-characters, e.g. 'Dr\.' escapes the **.** meta-character so that it acts like a literal punctuation mark.

Disclaimer: the rest of these slides will use Perl-style regular expressions, which grep supports if you specify -P like above.

More useful meta-characters

\t	match the tab character
\d	match any digit character, 0-9
\w	match any 'word' character, a-z or A-Z or 0-9
\s	match a space, tab, or newline character
?	match the previous character 0 or 1 times
+	match the previous character 1 or more times
^	match the beginning of the line
$	match the end of the line

A good tutorial is available in Software Carpenty lectures:

- Regular Expressions: Operators
- Regular Expressions: Patterns

RE best practices

Constructing good REs is a balance between:

Generality - casting a wide enough net to find all matches you are interested in

Specificity - preventing unwanted matches, usually at the cost of increasing the complexity of the RE

Typical process:

- write a PE
- use it until an unexpected match (or lack of match) occurs
- revise PE
- iterate

RE best practices

Example of increasing specificity/complexity:

'.*-.*-.*' will match dates in YYYY-MM-DD form, but it will also match hyphenated text like 'In-N-Out'

'\d{4}-\d{2}-\d{2}' enforces the correct number of digits, but it will still match impossible dates like 3001-45-71 and will fail to match other formats like 2011-1-1 or 2011/1/1

'\d{4}[-/]\d{1,2}[-/]\d{1,2}' will match the other formats, but allows invalid dates and format combinations like 2011/1-1

'\d{4}([-/])([1-9]|0[1-9]|1[012])\1([1-9]|0[1-9]|2\d|3[01])' fixes most of these problems, but will still accept a few invalid dates like 2011/2/31, and is getting very complex!

RE limitations

RE can't handle every kind of validation, and can't always limit the matches to the exact subset you need.

Expect that you will typically do validation or trimming in your script or program on the set of matches from the RE.

In the YYYY-MM-DD example, February 29 is only valid during a leap year, when

- the year is divisible by 4 but not 100
- or the year is divisible by 400

It is far more efficient to perform these arithmetic tests in a scripting or programming language than in the RE itself.

RE sub-expressions

You can bracket parts of your RE with parentheses to indicate they are "sub" expressions so you can extract the sub-matches.

For example, '(\d+)-(\d+)-(\d+)' will match a date like YYYY-MM-DD, and will contain 3 sub-matches { YYYY, MM, DD }.

A simple Python script called **subgrep** is available on Oscar to print sub-matches in tabular form.

```
usage: subgrep [-h] [--separator SEP] PATTERN [FILE]

Matches lines in <stdin> (or in FILE if specified) against the regular
expression PATTERN and prints all matching subexpressions as columns separated
by SEP (default is tab).

$ echo "Wednesday 2011-02-29" | subgrep "(\d+)-(\d+)-(\d+)"
2011	02	29
```

Filtering and reporting with awk

awk is a UNIX utility and mini-programming language for manipulating rows and columns of text.

awk:

- understands the same arithmetic operators as C
- has functions for string manipulation
- provides associative arrays
- is line-oriented (like grep and most other utilities)

The basic template of an awk program is:

pattern { action }

which is applied to each line in input data.

Extracting columns

The columns of each line are accessible with the special variables *$1*, *$2*, ...

Example: extract the 4th and 7th column of a tab-separated text file with

```
awk '{print $4"\t"$7}' data.txt
```

The variable *$0* contains the entire line.

The variable *NS* contains the number of fields. You can access the last field with *$NS*, the second to last field with *$(NS-1)*, etc.

Changing the delimiter

By default, awk uses any whitespace as the column delimiter.
To change, e.g. to a colon, use the -F flag

```
awk -F: ...
```

or set the special variable *FS* in a BEGIN statement:

```
awk 'BEGIN { FS=":"; } ... '
```

Example: extract the 4th and 7th column of a **comma**-separated text file with

```
awk -F, '{print $4","$7}' data.csv
```

Selectively printing lines

The built-in NR variable contains the current line number.

To print a single line, say line 12, of a text file 'file.txt', use

```
awk 'NR==12' file.txt
```

To print a range of lines, say 8 through 12, use

```
awk 'NR==8,NR==12' file.txt
```

Stream editing with sed

sed is a UNIX filter that can perform operations like matching, printing and substitution on a stream of text.

The basic usage is:

sed '*commands*' input >output

where *commands* is a list of operations to perform on the text from the file 'input', with the results written to the file 'output'.

Often, the input/output will be a pipe instead of a file.

Substitution with sed

BROWN UNIVERSITY CENTER FOR COMPUTATION & VISUALIZATION

Probably the most used command with sed is *s*,
for **substitution**:

```
sed 's/search/replace/'
```

This example uses a literal string for the search, but you can also use regular expressions.

The character after the *s* command is the delimiter, which be convention is often '/'. If you are searching for a string with '/' in it (f instance a file path), you may want to use ':' or '|':

```
sed 's:/home/alice:/home/bob:g'
```

Using the matched value

You may want to add to a string rather than replace it completely, e.g. add quotes around a string:

```
sed 's/abc/"abc"/'
```

But what if you could match multiple strings? What do you specify as the replacement value?

The & operator expands to the matched value:

```
sed 's/[a-z]*/"&"/'
```

Global replacement and ranges

By default, a substitution will only replace the first match in a line of text. To replace all matches in a line, add the *g* flag:

```
sed 's/search/replace/g'
```

By default, all lines in the input are processed. You can restrict this to a range of lines, e.g. the first 10 lines:

```
sed '1,10 s/search/replace/'
```

The *$* operator means "end of file", or the last line in the input:

```
sed '100,$ s/search/replace/'
```

Note: line numbers are cumulative if several files are edited.

Deleting lines

Another useful operation is *d* for **delete**:

```
sed '1,10 d'
```

This command deletes the first 10 lines of the input and prints the rest.

Obviously, this is very similar to the **tail** command, except that sed can handle more complicated ranges.

The *q* operation (**quit**) stops printing at a given line:

```
sed '10 q'
```

This is another way that sed is similar to head/tail.

Appending and inserting lines

The *i* command (**insert**) adds a line before the matching line:

```
sed '/WORD/ i Insert this line before lines containing WORD'
```

The *a* command (**append**) adds a line after:

```
sed '/WORD/ a Append this line after lines containing WORD'
```

The c command (**change**) replaces an entire line:

```
sed '/WORD/ c This line previously contained WORD'
```

Combining commands

You can specify a sequence of commands with the -e flag:

```
sed -e '$ d' -e 's/search/replace/'
```

This command would perform a substitution on all but the last line of input.

Writing a shell script

A shell script is a text file that starts with the special line

```
#!/bin/bash
```

This specifies that the bash shell is the *interpreter* for the file.

A script is different from a compiled program because it is interpreted at run-time, rather than at compile-time.

If your script is a file called 'myscript', you can *execute* it similar to how you execute a program with:

```
./myscript
```

(Note: first, you many need to "chmod 755 myscrpt".)

Control flow

The typical constructs like conditionals and for and while loops are available in bash.

```
if test condition; then
      clause
fi

for variable in list; do
    clause
done

while test condition; do
    clause
done
```

If statements

There are many built-in operators for test operations, ie.

```
if test $i -gt 10; then ...
```

As a shortcut, you can put the test in brackets:

```
if [ $i -gt 10 ]; then ...
```

Note: you must have spaces around the brackets!

You'll also run into double brackets: these allow for more sophisticated bash-specific tests, e.g. matching an RE:

```
if [[ "$email" =~ "\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}\b" ]]; then
    echo "\$email contains a valid e-mail address."
fi
```

Test operators

Numeric:

[	-eq	-ne	-gt	-lt	-ge	-le	]
((	==	!=	<	>	<=	>=	))

Logical:

```
[   -a   -o   ]        [[   &&   ||   ]]
```

String:

==	equal	-n	length > 0
!=	not equal	-z	empty (length == 0)

File:

-e	exists	-r	is readable
-f	is a regular file	-w	is writable
-d	is a directory	-x	is executable
-h	is a symbolic link	-s	is non-empty

For loops

Example: loop through a list of numbers.

```
for i in {0..10}; do echo $i; done
```

Example: loop through a list of strings.

```
for s in a b c; do echo $s; done
```

Example: loop through a list of text files.

```
for f in *.txt; do echo $f; done
```

Similar to: `find . -name '*.txt' -exec echo {} \;`

Arithmetic

Use double parentheses for arithmetic operations.

Example:

j=$((i*2))

Example: increment the variable $i by 2.

((i += 2))

incorrect: $((i += 2))

String manipulation

String length of $var:

${#var}

Extract a substring of $var:

${var:*start*}
${var:*start*:*length*}

Delete a substring pattern from $var (shortest match):

delete from beginning: ${var#*pattern*}
delete from end: ${var%*pattern*}

For longest match, use ## and %% instead.

String manipulation

Search and replace inside $var (first match):

${var/*search*/*replace*}

or **all** matches with ${var***//****search*/*replace*}

or only at beginning with ***/#***, or end with ***/%***

Note: this is **bash** syntax, and won't work with csh, etc.

Special variables

The command line arguments passed to a script are accessible in the special variables *$1*, *$2*, *$3*, etc.

$# total number of command line arguments
$@ a list of all command line arguments

You can remove arguments from the front of $@ using the 'shift' command.

$$ is the process id of the current shell
$!   process id of the previously run command
$? exit status of the previously run command

Parsing command-line arguments

A convenient way to handle arguments is to use the built-in **getopts** command:

```
getopts "ab:" OPTION
```

The "ab:" descriptor specifies the acceptable argument flags. The ":" indicates that "b" is a flag that takes an argument, while "a" is a stand-alone argument flag.

The usage for these options would be:

```
./myscripts [-a] [-b value]
```

Parsing command-line arguments

OPTION is the variable name that getopts will fill as it parses the arguments.

Typically, you loop over the parsed values with:

```
while getopts "ab:" OPTION
do
    (parse $OPTION here)
done
```

Parsing command-line arguments

BROWN UNIVERSITY CENTER FOR COMPUTATION & VISUALIZATION

```
usage() {
        echo "usage: myscript [-a] [-b value]"
        exit 0
}

while getopts "ab:" OPTION
do
        case $OPTION in
                a ) AVAL=1 ;;
                b ) BVAL=$OPTARG ;;
                ? ) usage ;;
        esac
done
shift $((OPTIND-1))
```