

Shell Scripting

Steps in Writing a Shell Script



- Write a script file using vi:
 - The first line identifies the file as a **bash** script.
`#!/bin/bash`
 - Comments begin with a **#** and end at the end of the line.
- give the user (and others, if (s)he wishes) permission to execute it.
 - `chmod +x filename`
- Run from local dir
 - `./filename`
- Run with a trace – echo commands after expansion
 - `bash -x ./filename`

Variables

- Create a variable
 - Variablename=value (no spaces, no \$)
 - read variablename (no \$)
- Access a variable's value
 - \$variablename
- Set a variable
 - Variablename=value (no spaces, no \$ before variablename)
- Sample:

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/playwithvar>

Positional Parameters



Positional Parameter	What It References
<code>\$0</code>	References the name of the script
<code>\$#</code>	Holds the value of the number of positional parameters
<code>\$*</code>	Lists all of the positional parameters
<code>\$@</code>	Means the same as <code>\$@</code> , except when enclosed in double quotes
<code>"\$*"</code>	Expands to a single argument (e.g., <code>"\$1 \$2 \$3"</code>)
<code>"\$@"</code>	Expands to separate arguments (e.g., <code>"\$1" "\$2" "\$3"</code>)
<code>\$1 .. \${10}</code>	References individual positional parameters
<code>set</code>	Command to reset the script arguments

wget

<http://home.adelphi.edu/~pe16132/csc271/notes/scripts/envvar>

Environment Variables



- set | more – shows all the environment variables that exist
- Change
 - PS1='\u>'
 - PATH=\$PATH:/home/pe16132/bin1
 - IFS=':'
 - IFS is **Internal Field Separator**
- **Sample**
 - wget
 - <http://home.adelphi.edu/~pe16132/csc271/note/scripts/envvar>

`$*` and `$@`

- `$*` and `$@` can be used as part of the list in a for loop or can be used as par of it.
- When expanded `$@` and `$*` are the same unless enclosed in double quotes.
 - `$*` is evaluated to a single string while `$@` is evaluated to a list of separate word.

Variable Scope & Processes



- Variables are shared only with their own process, unless exported
- `x=Hi` – define `x` in current process
- `sh` – launch a new process
- `echo $x` – cannot see `x` from parent process
- `x=bye`
- `<ctrl d>` -- exit new process
- `echo $x` -- see `x` in old process did not change
- `demoShare` – cannot see `x`
- `. demoShare` – run with dot space runs in current shell
- `export x` – exports the variable to make available to its children
- `demoShare` – now it can see `x`

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/demoShare>

The `read` Command

(continued)

Read from `stdin` (screen)

Read until new line

Format	Meaning
<code>read answer</code>	Reads a line from <code>stdin</code> into the variable <code>answer</code>
<code>read first last</code>	Reads a line from <code>stdin</code> up to the whitespace, putting the first word in <code>first</code> and the rest of the of line into <code>last</code>
<code>read</code>	Reads a line from <code>stdin</code> and assigns it to <code>REPLY</code>
<code>read -a arrayname</code>	Reads a list of word into an array called <code>arrayname</code>
<code>read -p prompt</code>	Prints a prompt, waits for input and stores input in <code>REPLY</code>
<code>read -r line</code>	Allows the input to contain a backslash.

Shortcut to Display Lots of Words



- Here file:
 - You give it the end token at the start
 - Type a list
 - Type the end token to end
 - cat << Here

words

Here

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/nosy>

Numbers

- Assumes variables are strings
- Math operations on strings are essentially ignored
 - Normalvar=1
 - 3+\$normalvar yields 3+1
- Must force consideration as number
 - Create variable with declare -i
 - Surround your mathematical statement with (())

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/numbers>

Different Base Num's: Octal, Hex



- Leading 0 in a number makes it be interpreted as octal so 017 represents the decimal # 15
- Leading 0x in a number makes it be interpreted as hex.
- Leading <Base># in a number makes it be interpreted as that base.

Floating Point Arithmetic



- **Bash** does not support floating point arithmetic but **bc**, **awk** and **nawk** utilities all do.

```
SIEGFRIE@panther:~$ n=`echo "scale=3; 13 / 2" | bc`
```

```
SIEGFRIE@panther:~$ echo $n
```

```
6.500
```

```
SIEGFRIE@panther:~$ product=`nawk -v x=2.45 -v
```

```
y=3.123 'BEGIN{printf "%.2f\n", x*y}'`
```

```
SIEGFRIE@panther:~$ echo $product
```

```
7.65
```

Test Command

- Command to test true or false:
 - test
 - [the comparison]
 - [means 'test'
 - Spaces around [
 -] for looks only
 - Logical
 - -o for OR
 - -a for AND

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/fscript>

Using **test** For Numbers And Strings – Old

Format



```
if test expression
```

```
then
```

```
    command
```

```
fi
```

or

```
if [ string/numeric expression]
```

```
then
```

```
    command
```

```
fi
```

```
wget
```

```
http://home.adelphi.edu/~pe16132/csc271/n  
ote/scripts/ifscript
```

Using `test` For Strings – New Format



```
if [string expression] ; then  
    command
```

```
elif
```

```
fi
```

or

```
if (numeric expression)
```

```
NOTE: new line for then or ; then
```

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/ifscript>

Testing Strings vs Numbers



Comparing numbers

- remember (())
- -eq , -ne, -gt, -ge, -lt, -le

Comparing strings

- Remember [[]]
- Remember space after [
- =
- !=
- Unary string tests
 - [string] (not null)
 - -z (0 length)
 - -n (some length)
 - -l returns the length of the string

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/ifscriptnum>

Echo

Echo command is well appreciated when trying to debug scripts.

Syntax: `echo {options} string`

Options: `-e` : expand \ (back-slash) special characters

`-n` : do not output a new-line at the end.

String can be a “weakly quoted” or a ‘strongly quoted’ string. In the weakly quoted strings the references to variables are replaced by the value of those variables before the output.

As well as the variables some special backslash_escaped symbols are expanded during the output. If such expansions are required the `-e` option must be used.

test Command Operators – String Test



Test Operator	Tests True if
<code>[string1 = string2]</code>	String1 is equal to String2 (space surrounding = is necessary)
<code>[string1 != string2]</code>	String1 is not equal to String2 (space surrounding != is not necessary)
<code>[string]</code>	String is not null.
<code>[-z string]</code>	Length of string is zero.
<code>[-n string]</code>	Length of string is nonzero.
<code>[-l string]</code>	Length of string (number of character)

`[[]]` gives some pattern matching

`[[$name == [Tt]om]]` matches if \$name contains Tom or tom

`[[$name == [^t]om]]` matches if \$name contains any character but t followed by om

`[[$name == ?o]]` matches if \$name contains any character followed by o and then whatever number of characters after that.*

Just shell patterns, not regex

test Command Operators – Logical Tests



Test Operator	Test True If
[<i>string1</i> -a <i>string2</i>]	Both <i>string1</i> and <i>string2</i> are true.
[<i>string1</i> -o <i>string2</i>]	Both <i>string1</i> or <i>string2</i> are true.
[! <i>string</i>]	Not a <i>string1</i> match

Test operator	Tests True if
[[<i>pattern1</i> && <i>Pattern2</i>]]	Both <i>pattern1</i> and <i>pattern2</i> are true
[[<i>pattern1</i> <i>Pattern2</i>]]	Either <i>pattern1</i> or <i>pattern2</i> is true
[[! <i>pattern</i>]]	Not a <i>pattern</i> match

pattern1 and *pattern2* can contain metacharacters.

test Command Operators – Integer Tests



Test operator	Tests True if
[<i>int1</i> -eq <i>int2</i>]	$\text{int1} = \text{int2}$
[<i>int1</i> -ne <i>int2</i>]	$\text{int1} \neq \text{int2}$
[<i>int1</i> -gt <i>int2</i>]	$\text{int1} > \text{int2}$
[<i>int1</i> -ge <i>int2</i>]	$\text{int1} \geq \text{int2}$
[<i>int1</i> -lt <i>int2</i>]	$\text{int1} < \text{int2}$
[<i>int1</i> -le <i>int2</i>]	$\text{int1} \leq \text{int2}$

test Command Operators – File Tests



Test Operator	Test True If
[<i>file1</i> -nt <i>file2</i>]	True if file1 is newer than file2*
[<i>file1</i> -ot <i>file2</i>]	True if file1 is older than file2*
[<i>file1</i> -ef <i>file2</i>]	True if file1 and file2 have the same device and inode numbers.

* according to modification date and time

File Testing

Test Operator	Test True if:
-b filename	Block special file
-c filename	Character special file
-d filename	Directory existence
-e filename	File existence
-f filename	Regular file existence and not a directory
-G filename	True if file exists and is owned by the effective group id
-g filename	Set-group-ID is set
-k filename	Sticky bit is set
-L filename	File is a symbolic link

File Testing (continued)

Test Operator	Test True if:
-p filename	File is a named pipe
-O filename	File exists and is owned by the effective user ID
-r filename	file is readable
-S filename	file is a socket
-s filename	file is nonzero size
-t fd	True if fd (file descriptor) is opened on a terminal
-u filename	Set-user-id bit is set
-w filename	File is writable
-x filename	File is executable

Exit Status

- Every process running in Linux has an exit status code, where 0 indicates successful conclusion of the process and nonzero values indicates failure to terminate normally.
- Linux and UNIX provide ways of determining an exit status and to use it in shell programming.
- The `?` in **bash** is a shell variable that contains a numeric value representing the exit status.

Exit Status Demo

- All commands return something
- Standard 0 = success and 1 = failure
 - Backwards 0/1 from a true/false boolean

```
grep 'not there' myscript  
echo $?
```

1= failure

```
grep 'a' myscript  
echo $?
```

0 = success

`exit` Command and the `?` Variable



- `exit` is used to terminate the script; it is mainly to used to exit the script if some condition is true.
- `exit` has one parameter – a number ranging from 0 to 255, indicating if is ended successfully (0) or unsuccessfully (nonzero).
- The argument given to the script is stored in the variable `?`

`wget`

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/ifbigfiles>

Looping in Bash – The **for** Command

- Loop through a list – like java for each loop (pg 37)

```
for variable in word_list  
do  
    command(s)  
done
```
- **variable** will take on the value of each of the words in the list.
- To get a list, you can execute a subcommand that returns a list inside \$() ex \$(ls)

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/forscript>

while Command



- The while command evaluates the command following it and, if its exit status is 0, the commands in the body of the loop are executed.
- The loop continues until the exit status is nonzero.
- Format:

```
while command  
do  
    command (s)  
done
```

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/numm>

The **until** Command



- until works like the while command, except it execute the loop if the exit status is nonzero (i.e., the command failed).

- Format:

```
until command
```

```
do
```

```
command (s)
```

```
done
```

```
wget
```

```
http://home.adelphi.edu/~pe16132/csc271/note/scripts/hour
```

The `select` Command



- The `select` command allows the user to create menus in **bash**.
- A menu of numerically listed items is displayed to **stderr**, with **PS3** used to prompt the user for input.
- Format:

```
select var in wordlist
do
    command(s)
done
```

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/runit>

Commands Used With **select**



- **select** will automatically repeat and has do mechanism of its own to terminate. For this reason, the **exit** command is used to terminate.
- We use **break** to force an immediate exit from a loop (but not the program).
- We use **shift** to shift the parameter list one or more places to the left, removing the displaced parameters.

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/dater>

SELECT for a menu



- – creates menus that don't stop until you break out of the loop
 - Syntax:
 - PS3="Whatever you want your prompt to be for the menu "
 - select var in options list (and use ' ' to surround 2 word options)
 - do
 - Command(s)
 - done
 - Ex: select program in `ls -F` pwd date 'some other option' exit

File IO

- read command
 - Reads from stdin unless directed with < or |
ls | while read line
do
echo The line is "\$line"
done
 - Write to a file using redirection >
 - ls | while read line
do
echo The line is "\$line"
done > outputfile
 - Write to a temp file that is unique – use pid \$\$
 - done > tmp\$\$
- wget
<http://home.adelphi.edu/~pe16132/csc271/note/scripts/numberit>

Functions

- Define function before use
- Define function using: `functionname() { }`
- Call function using: `functionname parm1 parm2 ...`
- Function accesses parameters to it as `$1, $2 ..`
- Send back information with return statement

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/demofunction>

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/demofunction2>

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/demofunction3>

Trap an Interrupt



- Define the action that will happen when the interrupt occurs using: `trap 'the action to do when the interrupt occurs' the signal:`
 - `trap 'rm -f /tmp/my_tmp_file_$$' INT`
- When the signal arrives, that command will execute, and then it will continue with whatever statement it was processing.
- You can use a function instead of just one command.

`wget`

`http://home.adelphi.edu/~pe16132/csc271/note/scripts/trapper`

Case

If/elif/else construct

- Syntax:
 - case variable
 - value1)
 - commands
 - ;;
 - value2)
 - commands
 - ;;
 -) #default
 - Commands
 - ;;
 - esac

wget

<http://home.adelphi.edu/~pe16132/csc271/note/scripts/xcolors>

Summary

- Variables
- Decision - If / case / select (embedded while)
 - Numbers vs Strings
 - Unary tests
 - File tests
- Loop – for/ while / until
 - File IO
- Functions
- Trap

I/O and Redirection

Standard I/O

- Standard Output (stdout)
 - default place to which programs write
- Standard Input (stdin)
 - default place from which programs read
- Standard Error (stderr)
 - default place where errors are reported
- To demonstrate -- **cat**
 - Echoes everything you typed in with an <enter>
 - Quits when you press **Ctrl-d** at a new line -- (**EOF**)

Redirecting Standard Output

- `cat file1 file2 > file3`
 - concatenates file1 and file2 into file3
 - file3 is created if not there
- `cat file1 file2 >! file3`
 - file3 is clobbered if there
- `cat file1 file2 >> file3`
 - file3 is created if not there
 - file3 is appended to if it is there
- `cat > file3`
 - file3 is created from whatever user provides from standard input

Redirecting Standard Error



- Generally direct standard output and standard error to the same place:

`obelix[1] > cat myfile >& yourfile`

- If `myfile` exists, it is copied into `yourfile`
- If `myfile` does not exist, an error message

`cat: myfile: No such file or directory` is copied in `yourfile`

- In `tcsh`, to write standard output and standard error into different files:

`obelix[2] > (cat myfile > yourfile) >& yourerrorfile`

- In `sh` (for shell scripts), standard error is redirected differently

– `cat myfile > yourfile 2> yourerrorfile`

Redirecting Standard Input



- `obelix[1] > cat < oldfile > newfile`
- A more useful example:
 - `obelix[2] > tr string1 string2`
 - Read from standard input.
 - Character *n* of `string1` translated to character *n* of `string2`.
 - Results written to standard output.
 - Example of use:
`obelix[3] > tr aeiou eoiaa`
`obelix[4] > tr a-z A-Z < file1 > file2`

/dev/null

- /dev/null
 - A virtual file that is always empty.
 - Copy things to here and they disappear.
 - `cp myfile /dev/null`
 - `mv myfile /dev/null`
 - Copy from here and get an empty file.
 - `cp /dev/null myfile`
 - Redirect error messages to this file
 - `(ls -l > recordfile) >& /dev/null`
 - Basically, all error messages are discarded.

Filters (1)

- Filters are programs that:
 - Read stdin.
 - Modify it.
 - Write the results to stdout.
- Filters typically do not need user input.
- Example:
 - **tr** (translate):
 - Read stdin
 - Echo to stdout, translating some specified characters
- Many filters can also take file names as operands for input, instead of using stdin.

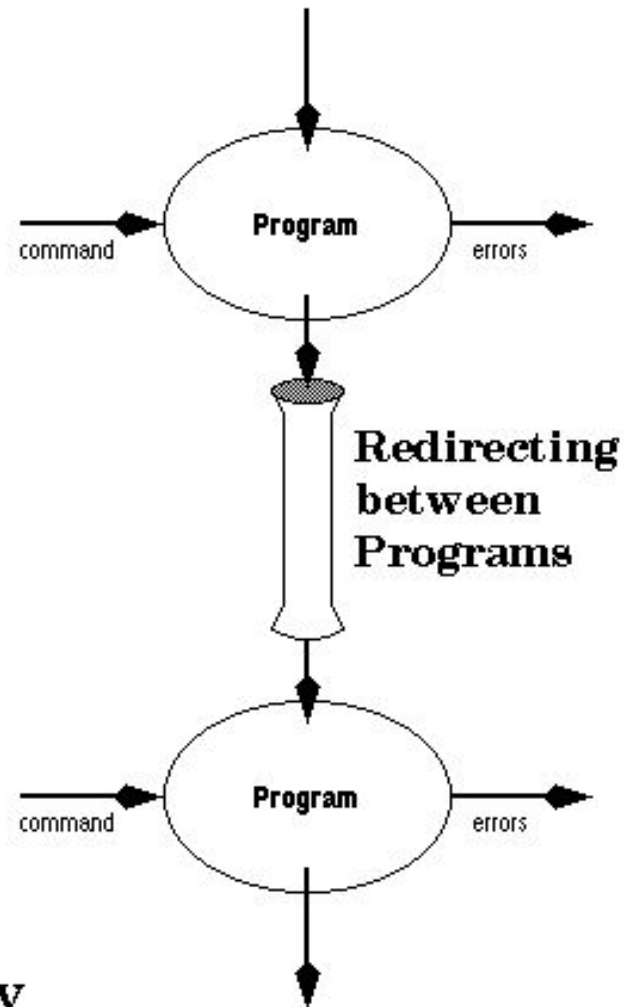
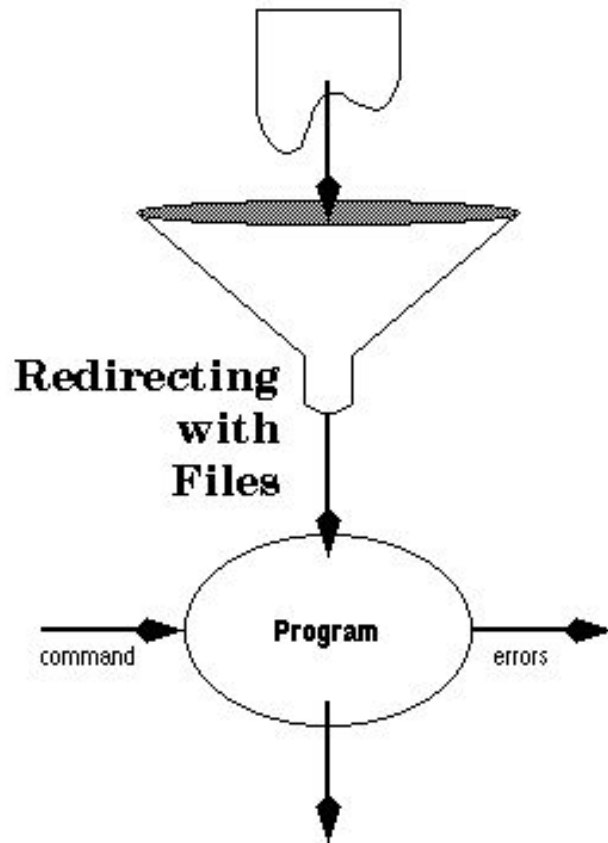
Filters (2)

- **grep patternstr:**
 - Read stdin and write lines containing **patternstr** to stdout
 - obelix[1] > grep "unix is easy" < myfile1 > myfile2**
 - Write all lines of **myfile1** containing phrase ***unix is easy*** to **myfile2**
- **wc:**
 - Count the number of chars/words/lines on stdin
 - Write the resulting statistics to stdout
- **sort:**
 - Sort all the input lines in alphabetical order and write to the standard output.

Pipes

- The pipe:
 - Connects stdout of one program with stdin of another
 - General form:
`command1 | command2`
 - stdout of command1 used as stdin for command2
 - Example:
`obelix[1] > cat readme.txt | grep unix | wc -l`
- An alternative way (not efficient) is to:
`obelix[2] > grep unix < readme.txt > tmp`
`obelix[3] > wc -l < tmp`
- Can also pipe stderr: `command1 |& command2`

Redirecting and Pipes (1)



Plumbing the UNIX Way

Redirecting and Pipes (2)

- Note: The name of a command always comes first on the line.
- There may be a tendency to say:
`obelix[1] > readme.txt > grep unix | wc -l`
 - This is WRONG!!!
 - Your shell will go looking for a program named `readme.txt`
- To do it correctly, many alternatives!
`obelix[1] > cat readme.txt | grep unix | wc -l`
`obelix[2] > grep unix < readme.txt | wc -l`
`obelix[3] > grep unix readme.txt | wc -l`
`obelix[4] > grep -c unix readme.txt`

The 'grep' Command

What is grep?

- grep is a command that searches through the input file for a specified pattern
- When grep finds a match to the pattern, it prints the entire line to standard output
- grep general structure:
 - grep options pattern input_file_names

Options

- grep has a variety of options that can execute a wide range of operations once a match is found
- This presentation is an overview of some of the more basic/common options
- For all available options and more in depth explanations, please see Unix resource 3 on the course website titled “Grep command documentation” or check the man page for grep in your terminal

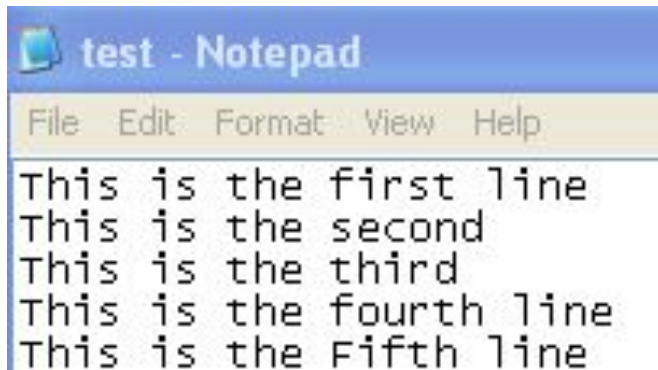
Matching Control



- `'-i'`
 - Ignores case.
- `'-v'`
 - Inverts the matching. When used, grep will print out lines that do not match the pattern
- `'-e pattern'`
 - Pattern is the pattern. This can be used to specify multiple patterns, or if the pattern starts with a `'-'`. A line only has to contain one of the patterns to be matched.

Examples using '-i', '-v', and '-e'

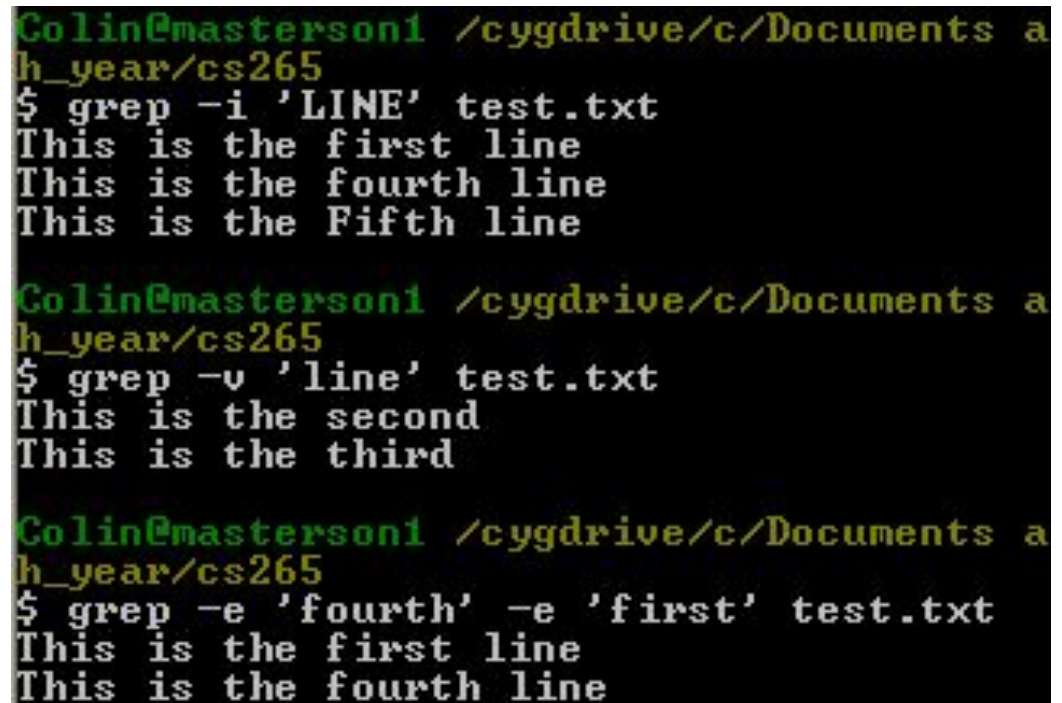
- The file below was used for these examples



test - Notepad

File Edit Format View Help

This is the first line
This is the second
This is the third
This is the fourth line
This is the Fifth line



```
Colin@masterson1 /cygdrive/c/Documents and Settings/Colin/My Documents/h_year/cs265
$ grep -i 'LINE' test.txt
This is the first line
This is the fourth line
This is the Fifth line

Colin@masterson1 /cygdrive/c/Documents and Settings/Colin/My Documents/h_year/cs265
$ grep -v 'line' test.txt
This is the second
This is the third

Colin@masterson1 /cygdrive/c/Documents and Settings/Colin/My Documents/h_year/cs265
$ grep -e 'fourth' -e 'first' test.txt
This is the first line
This is the fourth line
```

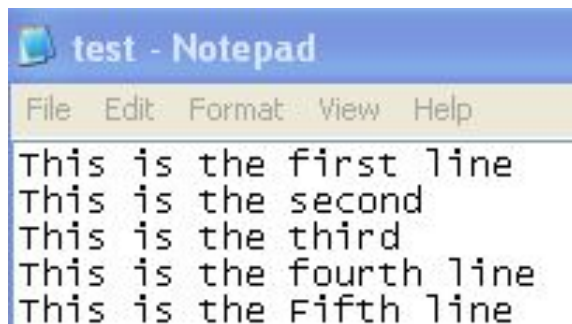
General Output Control



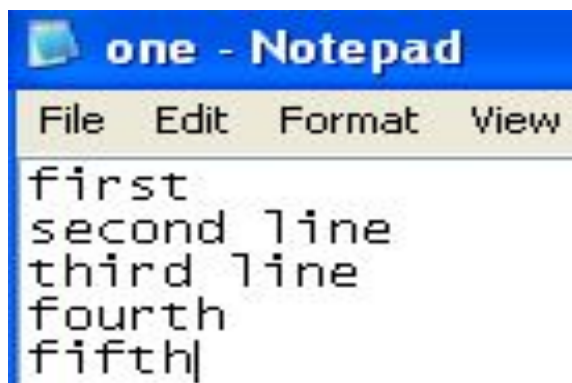
- **'-c'**
 - Suppress normal output and instead print out a count of matching lines for each input file
- **'-l'**
 - Suppress normal output and print the name of each file that contains at least one match
- **'-L'**
 - Suppress normal output. Print the name of each file that does not contain any matches
- Note: both the **'-l'** and **'-L'** options will stop searching a file once a match is found

Examples using '-c', '-l', and '-L'

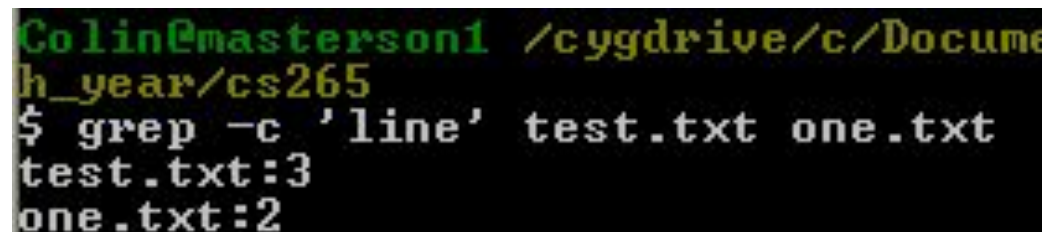
- Two files were used and specified below




```
test - Notepad
File Edit Format View Help
This is the first line
This is the second
This is the third
This is the fourth line
This is the Fifth line
```



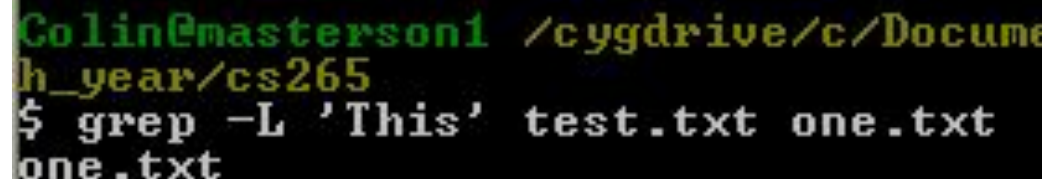
```
one - Notepad
File Edit Format View
first
second line
third line
fourth
fifth|
```



```
Colin@masterson1 /cygdrive/c/Documents and Settings/h_year/cs265
$ grep -c 'line' test.txt one.txt
test.txt:3
one.txt:2
```



```
Colin@masterson1 /cygdrive/c/Documents and Settings/h_year/cs265
$ grep -l 'This' test.txt one.txt
test.txt
```



```
Colin@masterson1 /cygdrive/c/Documents and Settings/h_year/cs265
$ grep -L 'This' test.txt one.txt
one.txt
```

Output Line Prefix Control



- ‘-n’
 - Prefixes each line of output with the line number from the input file the match was found on
- ‘-H’
 - Prefix each line of output with the input file name that the match was found in
- ‘-T’
 - Makes sure that the actual line content (or whatever content comes after the ‘-T’) lands on a tab stop

Examples using '-H', '-n', and '-T'

- Two files were used and specified below

test - Notepad

File Edit Format View Help

This is the first line
This is the second
This is the third
This is the fourth line
This is the Fifth line

one - Notepad

File Edit Format View

first
second line
third line
fourth
fifth|

```
Colin@masterson1 /cygdrive/c/Documents and Settings/Colin/My Documents/h_year/cs265
$ grep -HTnT 'line' test.txt one.txt
test.txt: 1 :This is the first line
test.txt: 4 :This is the fourth line
test.txt: 5 :This is the Fifth line
one.txt: 2 :second line
one.txt: 3 :third line
```

```
Colin@masterson1 /cygdrive/c/Documents and Settings/Colin/My Documents/h_year/cs265
$ grep -Hn 'line' test.txt one.txt
test.txt:1:This is the first line
test.txt:4:This is the fourth line
test.txt:5:This is the Fifth line
one.txt:2:second line
one.txt:3:third line
```

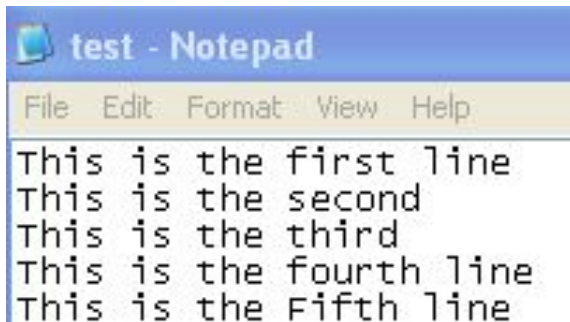
Context Line Control



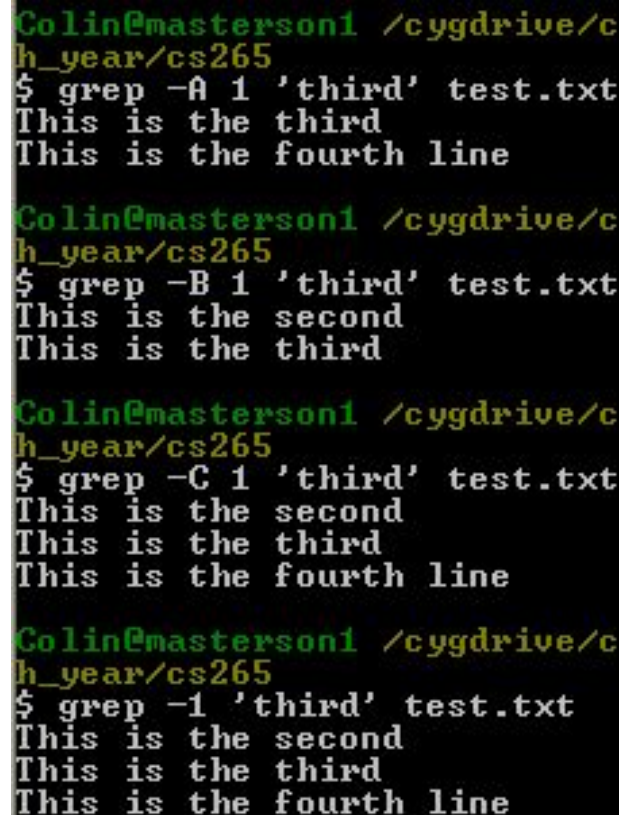
- *'-A num'*
 - Print *num* lines of trailing context after matching lines
- *'-B num'*
 - Print *num* lines of leading context before matching lines
- *'-C num'* or *'-num'*
 - Print *num* lines of leading and trailing output context

Examples using '-A', '-B', and '-C'

- The file below was used for these examples



```
test - Notepad
File Edit Format View Help
This is the first line
This is the second
This is the third
This is the fourth line
This is the Fifth line
```



```
Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -A 1 'third' test.txt
This is the third
This is the fourth line

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -B 1 'third' test.txt
This is the second
This is the third

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -C 1 'third' test.txt
This is the second
This is the third
This is the fourth line

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -l 'third' test.txt
This is the second
This is the third
This is the fourth line
```

Special Characters



- '.' The period '.' matches any single character.
- '?' The preceding item is optional and will be matched at most once.
- '*' The preceding item will be matched zero or more times.
- '+' The preceding item will be matched one or more times.
- '{n}' The preceding item is matched exactly n times.
- '{n,}' The preceding item is matched n or more times.
- '{,m}' The preceding item is matched at most m times.
- '{n,m}' The preceding item is matched at least n times, but not more than m times.

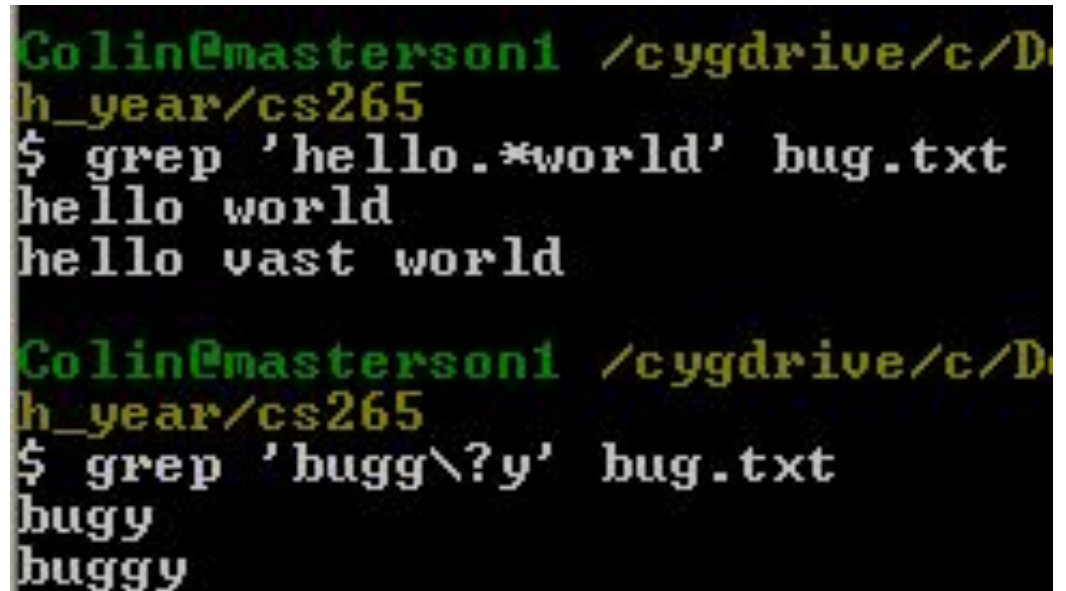
Examples using '.', '*', and '?'



bug - Notepad

File Edit Format View

bugy
buggy
buggyy
hello world
hello vast world



```
Colin@masterson1 /cygdrive/c/D
h_year/cs265
$ grep 'hello.*world' bug.txt
hello world
hello vast world

Colin@masterson1 /cygdrive/c/D
h_year/cs265
$ grep 'bugg\?y' bug.txt
bugy
buggy
```

Basic vs Extended Regular Expressions



- `'-G'`
 - Interpret pattern as basic regular expression (BRE). This is the default.
- `'-E'`
 - Interpret pattern as extended regular expression (ERE)
- When using basic regular expression some special characters (like `'?'` in the previous example) lose their special meaning and must have a `'\'`, the escape character, before them

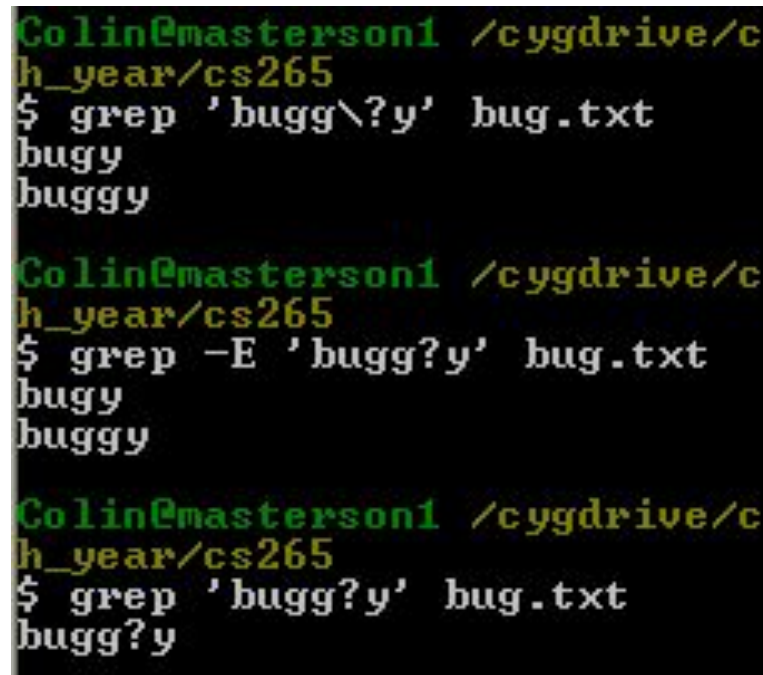
BRE and ERE Difference



bug - Notepad

File Edit Format View

bugy
buggy
buggyy
buggy?
hello world
hello vast world



```
Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep 'bugg\?y' bug.txt
bugy
buggy

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep -E 'buggy?y' bug.txt
bugy
buggy

Colin@masterson1 /cygdrive/c
h_year/cs265
$ grep 'buggy?y' bug.txt
buggy?y
```

- Note that without the `\` in the BRE call (example 3), the `?` is seen as a normal character

Bracket Expressions



- A bracket expression is a list of characters enclosed by '[' and ']'. It matches any single character in the list
- However, if the first character in the list is '^', it matches any character not in the list
- A range can be done by using '-' in a bracket expression
 - [0-5] is the same as [012345]
- Some ranges are pre-defined in character classes

Bracket Expression Example

test2 - Notepad

File Edit Format

first line
2nd line
3rd
fourth line
5th line

```
Colin@masterson1 /cygdrive/c/Do
h_year/cs265
$ grep '[rl]' test2.txt
first line
3rd
fourth line

Colin@masterson1 /cygdrive/c/Do
h_year/cs265
$ grep '[rdl]' test2.txt
first line
2nd line
3rd
fourth line

Colin@masterson1 /cygdrive/c/Do
h_year/cs265
$ grep '[:digit:]' test2.txt
2nd line
3rd
5th line
```

Cut Command

- Linux cut command is useful for selecting a specific column of a file. It is used to cut a specific sections by byte position, character, and field and writes them to standard output. It cuts a line and extracts the text data. It is necessary to pass an argument with it; otherwise, it will throw an error message.
- To cut a specific section, it is necessary to specify the delimiter. A delimiter will decide how the sections are separated in a text file. Delimiters can be a space (' '), a hyphen (-), a slash (/), or anything else. After '-f' option, the column number is mentioned.
- Syntax:
- `cut OPTION... [FILE]...`

Cut Options

- **-c, --characters=LIST:** It is used to select the specified characters.
- **-d, --delimiter=DELIM:** It is used to cut a specific section by a delimiter.
- **-f, --fields=LIST:** It is used to select the specific fields. It also prints any line that does not contain any delimiter character, unless the -s option is specified.
- **-n:** It is used to ignore any option.
- **--complement:** It is used to complement the set of selected bytes, characters or fields
- **-s, --only-delimited:** It is used to not print lines that do not have delimiters.
- **--output-delimiter=STRING:** This option is specified to use a STRING as an output delimiter; The default is to use "input delimiter".
- **-z, --zero-terminated:** It is used if line delimiter is NUL, not newline.

Introduction

AWK Programming

AWK is a great language. Awk is geared towards text processing and report generation, yet features many well-designed features that allow for serious programming. And, unlike some languages, awk's syntax is familiar, and borrows some of the best parts of languages like C, python, and bash (although, technically, awk was created before both python and bash). Awk is one of those languages that, once learned, will become a key part of one's strategic coding arsenal.

Awk stands for the names of its authors "**A**ho, **W**einberger, and **K**ernighan"

Syntax 1 - Generic Syntax:

```
awk 'BEGIN {start_action} {action} END {stop_action}' filename
```

Syntax 2 – Pattern Matching :

```
awk '/search pattern1/ {Actions} /search pattern2/ {Actions}' file
```

[Note: Either search pattern or action(s) are optional, but not both.]

THE first awk

Command

- `$ awk '{ print }' /etc/passwd`

Output

- Print the contents of the `/etc/passwd` file

Explanation

- In awk, curly braces are used to group blocks of code together, similar to C. In awk, when a print command appears by itself, the full contents of the current line are printed.

Note/Remarks

- The following command accomplishes the same thing.
 - `awk '{ print $0 }' /etc/passwd`
- `$0` stands for the entire line, for each record.

Printing Multiple fields

Purpose

- Printing multiple columns.

Command

- `awk '{print $k}' input_file`
- `awk '{print $p,$q}' input_file`
- `awk '{print pq$r}' input_file`

Output

- Prints the kth column of a space separated file
- Prints pth and qth column separated by space.
- Prints p,q & rth column of a file without any space

Explanation

- Space is considered as the default OFS i.e. Output Field Separator.

Note/Remarks

- The following command won't retrieve the kth column of the CSV file.
 - `awk '{print $k}' input_csv_file`

Printing Multiple fields of specially delimited files

Purpose

- Printing multiple columns of files with delimiters other than space, say comma. Files having comma as the delimiter are called CSV files.

Command

- `awk 'BEGIN{FS=","} {print $p}' input_file` (OR)
- `awk -F"," '{print $p}' input_file`
- `awk 'BEGIN {OFS="|"} {print $4,$5}' input_file`

Output

- Prints the pth column of a CSV file
- Separates the 4th and 5th fields by pipe while printing to stdout.

Explanation

- FS and OFS respectively mean (Input) Field Separator & Output Field Separator.

Note/Remarks

- We can also use any number or string or even Regex as FS.

The begin and end blocks

BEGIN BLOCK

- If we need to execute initialization code *before* awk begins processing the text from the input file, a BEGIN block is used.
- Because the BEGIN block is evaluated before awk starts processing the input file, it's an excellent place to initialize the FS (field separator) variable, print a heading, or initialize other global variables that we'll reference later in the program.

END BLOCK

- Awk executes this block after all lines in the input file have been processed. Typically, the END block is used to perform final calculations or print summaries that should appear at the end of the output stream.

Regular expressions & blocks



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Awk allows the use of regular expressions to selectively execute an individual block of code, depending on whether or not the regular expression matches the current line. Here's an example script that outputs only those lines that contain the character sequence foo:

```
/foo/ { print }
```

Of course, we can use more complicated regular expressions. Here's a script that will print only lines that contain a floating point number:

```
/[0-9]+\.[0-9]*/ { print }
```

AWK BUILT IN VARIABLES



SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

NF

- NF stands for Number of field.
- Example:
 - Purpose: Number of columns in each row.
 - Code: `awk '{print NF}' input_file`

NR

- NR stands for Number of Record.
- Example:
 - Purpose: Display line numbers for each row.
 - Code: `awk '{print NR}' input_file`
 - Purpose: Total lines in a file
 - Code: `awk 'END {print NR}' input_file`

FS

- Discussed in earlier slides.
- Additionally `FS="\t+"` is used to process fields separated by one or more tabs.
- `FS="foo[0-9][0-9][0-9]"` is used to consider a string comprising of foo appended by three digits as the field separator.

OFS

- Discussed in earlier slides.

Conditional statements

Checks if the 9th
field of the input
file contains
HONDA

- `awk '{ if($9 == "HONDA") print $0;}' input_file`

Prints only those
records where the
first field contains
value more than 4

- `awk -F"," '{if($1>4) print $0;}' list1.txt`

Prints records if the
third field holds
Jadavpur

- `awk -F, '{if($3=="Jadavpur")print $0;}' list1.txt`
- `awk -F, '$3~/Jadavpur/' list1.txt`

Prints records if any
field contains
Jadavpur

- `awk -F, '/Jadavpur/' list1.txt`

Print lines matching
string A or B

- `$ awk '/Shibpur/ > /Jadavpur/' list1.txt`

Conditional statements(contd.)

Count all
employees who
belong to Ohio

```
awk -F, 'BEGIN{count=0;}  
$3~/Ohio/{count=count+1;}  
END{print count}'  
list1.txt
```

Awk also allows the use of boolean operators "||" (for "logical or") and "&&" (for "logical and") to allow the creation of more complex boolean expressions:

```
( $1 == "foo" ) && ( $2 == "bar" ) { print }
```

This example will print only those lines where field one equals foo and field two equals bar.

NUMERIC variables

(find number of blank lines in a file)

So far, we've either printed strings, the entire line, or specific fields. However, awk also allows us to perform both integer and floating point math. Using mathematical expressions, it's very easy to write a script that counts the number of blank lines in a file. Here's one that does just that:

```
BEGIN { x=0 }  
/^$/ { x=x+1 }  
END { print "I found " x " blank lines. :)" }
```

In the BEGIN block, we initialize our integer variable x to zero. Then, each time awk encounters a blank line, awk will execute the `x=x+1` statement, incrementing x. After all the lines have been processed, the END block will execute, and awk will print out a final summary, specifying the number of blank lines it found.

Plenty of operators

Another nice thing about awk is its full package of mathematical operators. In addition to standard addition, subtraction, multiplication, and division, awk allows us to use the exponent operator "^", the modulo (remainder) operator "%", and a bunch of other handy assignment operators borrowed from C.

These include pre- and post-increment/decrement (`i++`, `--foo`), add/sub/mult/div assign operators (`a+=3`, `b*=2`, `c/=2.2`, `d-=6.2`). But that's not all -- we also get handy modulo/exponent assign ops as well (`a^=2`, `b%=4`).

Looping:

Calculate square
of numbers from
1 to 10.

- `awk 'BEGIN { for(i=1;i<=10;i++) print "square of", i, "is", i*i; }'`

String functions in awk

String Functions used in AWK

- `Index(search,string)`
- `length(string)`
- `split(string,array,separator)`
- `substr(string,position)`
- `substr(string,position,max)`
- `tolower(string)`
- `toUpper(string)`



Employee Details with Educational Qualifications

```
awk -F","  
'BEGIN {print "EmpID\tQualification\tInstitute";}  
      {print $1,"\t",$2,"\t",$3;}  
END{print "REPORT GENERATED.\n";}'  
list1.txt
```

THANK YOU