Overview

Learners will be able to…

- **Retrieve and display information about files including:**
  - **Display N lines from the beginning or end of a file with `head` and `tail`**
  - **Display the word, line, character, and byte counts for files with `wc`**
  - **Locate lines inside a file matching a defined pattern with `grep` and `egrep`**
- **Change information within files from the command line by:**
  - **Transforming and deleting characters in a file with `tr`**
  - **Changing, formatting, and reporting data in a file with `awk`**
  - **Locating and modifying information inside files with `sed`, `cut`, and `find`**
- **Receive input and send it to multiple locations with `tee`**
- **Transform standard input into command-line arguments with `xargs`**

info

Make Sure You Know

This assignment assumes the learner has no previous experience with Bash scripting.

Limitations

This assignment is an overview of important scripting concepts that learners should be familiar with before attempting upcoming assignments.

head and tail

# head

The `head` command prints the first n lines of a standard input or file. If n is not specified, the command prints the first 10 lines by default.

When multiple file names are provided, data from each file is preceded by its name.

We can use the following command to print the first 6 lines of the results of the `ls -l` command.

```
ls -l | head -6
```

Let's explore a few examples using two files, `pres_first.txt` and `pres_last.txt` that contain the first and last names of the US presidents in alphabetical order. You can see the contents of these files by using the `cat` command.

```
cat pres_first.txt
```

Let's use `head` to view the first few lines of `pres_first.txt`. If the number of lines is not specified, the command prints the first 10 lines by default.

```
head pres_first.txt
```

We have several options to choose from when using `head`.
- `n`: Rather than printing the first 10, it prints the first 'num' lines. If num is not specified in command, it displays an error.
- `c`: Prints the first 'num' bytes from the given file. A newline counts as a single character, so printing one counts as a byte. If num is not provided, the command fails.
- `v`: When this option is used, data from the chosen file is always preceded by the name of the file.
- `q`: If more than one file is specified, it is used. By performing this operation, each file's data is not preceded by the file name.

Let's try a few of these in examples. We can display the first 10 bytes of a file.

```
head -c 10 pres_first.txt
```

Now with the `-q` option.

```
head -q  pres_first.txt pres_last.txt
```

# **`tail`**

The `tail` command prints the last n lines of a standard input or file. We can use the following command to print the last 3 lines of the results of the `ls -l` command.

```
ls -l | tail -3
```

We can also combine the `head` and `tail` commands to retrieve the lines in between. Let's pull lines $6 - 10$ of the `ls -l` command by:
- Retrieving the first 10 lines with `head -10`, then
- Get the last 5 lines of that result with `tail -5`, giving us lines $6 - 10$

We can combine these requests into the following single command:

```
ls -l |head -10 |tail -5
```

We can also look at the last 5 lines of one of our example files with the command below.

```
tail -5 pres_first.txt
```

`tail` is compatible with all of the same options as `head`. For example, we can similarly print the last 12 bytes of a file with the following command.

```
tail -c 12 pres_first.txt
```

Checkpoint

wc

The word count command `wc` prints out counts for lines, words, bytes, and characters in a specified file. This command displays four columns for output. Each column contains:

1. Number of lines in the file
2. Number of words in the file
3. Number of characters in the file
4. The file name

Let's explore a few examples with our two files, `pres_last.txt` and `pres_first.txt`. Use the button below to explore the contents of the each file.

Pass one file, `pres_last` into `wc` and examine each output column for the files line, word, and character counts along with the file's name.

```
wc pres_last.txt
```

Now, let's pass both of our files to `wc`. Notice the difference in output.

```
wc pres_last.txt pres_first
```

When given more than one file, `wc` displays count information for all of the files as well as the total across the files.

We have access to several options to use with `wc` to strengthen the tool. Let's explore a few using the buttons below.

`-w`: Displays only the word count and file name.

```
wc -w pres_first.txt
```

`-c`: Displays only the byte count and file name.

```
wc -c pres_first.txt
```

`-l`: Displays only the line count and file name.

```
wc -l pres_first.txt
```

`-m`: Displays only the character count and file name.

```
wc -m pres_first.txt
```

You can learn more about `wc` and all of its capabilities by entering `man wc` in the terminal. Click the button below to explore the `wc` manual pages.

Checkpoint

grep

`grep` searches a file for character patterns and displays all lines matching the pattern and uses regular expressions search for patterns in files. `grep` stands for global search for regular expression and printout.

Let’s practice with a few examples.

```
cat greeneggs.txt
```

If `-i` is specified, the search in the given file will be case-insensitive. No matter what the case of the characters, we can find matches to strings.

```
grep -i "eGgS" greeneggs.txt
```

The `-w` options allows us to search for entire words within a file.

```
grep -w "in" greeneggs.txt
```

We can also display the number of lines that match our given string with the `-c` option.

```
grep -c "like" greeneggs.txt
```

We can show files containing a specified string or pattern.

```
grep -l "Sam" *
```

You can learn more about `grep` and all of its options by entering `man grep` in the terminal. Click the button below to explore the `grep` manual pages.

Checkpoint

egrep

`egrep` is a pattern searching command from the `grep` family that works like `grep -E`. A regular expression is used to match the pattern. If the pattern matches multiple files, it displays the file names on each line.

`egrep` uses many of the same options as `grep`. However, unlike `grep`, `egrep` does not require meta-characters to be escaped. Having fewer characters to replace makes `egrep` faster than `grep`.

```
egrep [ options ] 'PATTERN' files
```

Let’s look at a few examples.

`-c`: Counts and prints the number of lines that match the pattern, without printing lines.

```
egrep -c eggs greeneggs.txt
```

`-v`: Prints the lines that don’t match the pattern.

```
egrep -v eggs greeneggs.txt
```

`-w`: Only prints the lines containing the entire word.

- Numbers, letters, and underscores are word-constituting characters.
- To separate matching substrings, non-word component characters must be used.

```
egrep -w eggs greeneggs.txt
```

`-o`: For each match, only the matching bits of the line are printed, not the complete line.

```
egrep -o eggs greeneggs.txt
```

You can learn more about `egrep` and all of its options by entering `man egrep` in the terminal. Click the button below to explore the `egrep` manual pages.

tr

UNIX’s **translate**`tr` command can translate or delete characters. With it, we can convert uppercase to lowercase, remove repetitive characters, delete specific characters, and perform basic search and replace operations.

Let’s do a few translations using our file `greeneggs.txt` for practice. Use `cat` to remind yourself of the contents of the file.

```
cat greeneggs.txt
```

We can pipe the results of `cat` into `tr` to translate all of the characters in a file to uppercase characters. Both of the options below accomplish the same task.

```
cat greeneggs.txt | tr "[a-z]" "[A-Z]"
```

```
cat greeneggs.txt | tr "[:lower:]" "[:upper:]"
```

We also have the ability to convert whitespace to tab characters with the following command.

```
cat greeneggs.txt | tr [:space:] '\t'
```

`tr` gives us the power to delete characters that we specify using the `-d` option.

```
cat greeneggs.txt | tr -d 'I'
```

We can use `tr` to remove numbers from a string…

```
echo "Call 489-4608 and I'll be here" | tr -d [:digit:]
```

Or to remove everything EXCEPT the numbers from a string.

```
echo "Call 489-4608 and I'll be here" | tr -cd [:digit:]
```

There are several other options available to extend the functionality of `tr`. Click the button below to explore the `tr` manual pages.

# awk

Awk is a utility for manipulating and reporting data, allowing us to use variables, numeric functions, string functions, and logical operators.

Awk is commonly used to look for patterns and change them. It looks through one or more files for lines that match the patterns you set. If it finds a match, it performs your command.

This utility is useful for:
- Changing data files
- Formatting lines of output
- Creating formatted reports

Syntax for awk commands include the following criteria as described in the image below.

# awk options 'selection _criteria {action}' input-file > output-file

Let's explore a few examples of what we can do with `awk`.

Print all of the lines of data in a file called `students.txt` with the following command.

```
awk '{print}' students.txt
```

In this example, we didn't input a selection criteria, so the actions apply to all lines. The `print` action prints the entire line by default, so the entire file is printed.

We can also print lines that match a specific pattern.

Print all of the lines in the file `students.txt` that match the pattern `freshman` using the command below.

```
awk '/freshman/ {print}' students.txt
```

By default, `awk` divides each line by the whitespace character and stores it in $n variables. Our lines have four values. Each one is saved in within a field, and each field be accessed with `$1`, `$2`, `$3`, and `$4`. `$0` is the variable for the entire line.

Let's print the first and third values on each line with the command below.

```
awk '{print $1,$3}' students.txt
```

Awk also supports several built in variables to make it an even more powerful utility.

- `FS`: Specifies the field separator character.
  - The default is “white space”, which includes spaces and tabs. We can redefine the field separator to another character (like a comma) using `BEGIN { FS="," }`
- `OFS`: Holds the output field separator used by Awk to print the fields.
  - The default is a blank space. If print contains more than one parameter separated by a comma, the value of OFS between each parameter is printed.
- `NR`: Keeps count of input records, or lines.
  - Awk runs pattern/action statements once for each record in a file.
- `NF`: Keeps track of how many fields are in the current input record.
  - The fields of a line are separated by field separators.
- `RS`: Holds the current record separator character.
  - The default record separator character is a newline, since a line of input is considered to be a record.
- `ORS`: stores the output record separator that’s used by Awk to separate output lines.
  - A newline character is the default. The contents of ORS is automatically printed at the end of whatever else it is instructed to print.

Click the button below to learn more about `awk` and all of its options.

Checkpoint

sed

SED stands for stream editor and it can search, find and replace, insert or delete information in a file. The SED command is commonly for substitution or find and replace tasks. SED allows you to modify files without opening them, which is faster than opening them and then editing them.

It doesn't change a file by default, but rather produces an output based on the filtering of a file.

- `sed` substitute `s` allows us to replace instances of text in a file with different text and print the resulting output.

The following example text:
- Switches the first occurrence of `old` to `new` for every line in filename.
- Displays the output to standard output `stdout`, without changing the file.

```
sed 's/old/new/' filename
```

This only changes the first occurrence on a line. to change every instance in a single line, we would have to use the global indicator `G`.

```
sed 's/old/new/G' filename
```

Let's explore a few examples.

We can run multiple `sed` commands at once using `;`.

`sed 's/freshman/sophomore/; s/junior/senior/'`

We could also write an entire script of `sed` commands and run them as a file using `-f`

Let's explore with a file called `greeneggs.txt`.

```
cat greeneggs.txt
```

We can use `sed` to substitute text in a file. Let's use it to replace the word `like` with `eat`.

```
sed 's/like/eat/' greeneggs.txt
```

We can replace strings on a specific line number. Let’s replace the string only on line number 3 with the following command:

```
sed '3 s/like/eat/' greeneggs.txt
```

We can also replace strings on line numbers within a specific range. Let’s replace strings between lines 4 and 7 with the following command:

```
sed '4,7 s/like/eat/' greeneggs.txt
```

SED can also be used to delete certain lines from a file. The SED command is used to delete a file without having to open it first

Let’s delete the $3rd$ line from `greeneggs.txt`.

```
sed '3d' greeneggs.txt
```

We can also delete lines that match a pattern. Let’s delete all of the lines in the file that contain `with`.

```
sed '/with/d' greeneggs.txt
```

cut

The cut command extracts portions from each line of a file and outputs them as standard output. Clipping can be done by byte, character, or field. Cut splits a line and extracts its contents.

Let's explore a few examples using our two files, `pres_first.txt` and `pres_last.txt`.

The `cut` utility has several options to use with it. If we try to use `cut` without specifying an option with the command, we will receive an error.

```
cut pres_first.txt
```

To cut by character, use the `-c` option and specify the characters to trim. This can be a list of numbers separated by commas or hyphens (-). They are treated as characters. This option requires a list of character numbers otherwise it fails.

Let's use `cut` to print the first, third, and fifth character from each line of the file `pres_first.txt`.

```
cut -c 1,3,5 pres_first.txt
```

We can also cut a range of characters by providing a start and end position. Let's print characters 2-6 from each line of `pres_last.txt`.

```
cut -c 2-6 pres_last.txt
```

When multiple file names are provided, the data is not preceded by the file name for each file.

The `-b` options allows us to cut specific bytes from each line. We can specify either:
- A list of bytes separated by commas `,`, or
- A range of bytes using a hyphen `-`.

Whitespaces and tabs count as 1-byte characters.

Let's extract bytes 3, 4, and 5 from each line of `pres_last` with the following command.

```
cut -b 3,4,5 pres_last.txt
```

Let’s extract bytes 1-4 using a hyphen `-`.

```
cut -b 1-4 pres_last.txt
```

There are several other options to explore using `cut`. We can type `cut --help` in the terminal for a deeper look into what `cut` has to offer.

find

The `find` utility walks a file structure to find files and folders and then operate on them. Filtering is done by file or folder name as well as creation and update dates.

Let’s take a look at our file hierarchy by clicking the button below.

We can search for a file within a directory by name using `-name` flag and the following syntax.

INSERT IMAGE

Let’s look for a file, `foundIt.txt` in the directory `findIt`.

```
find ./findIt -name foundIt.txt
```

This returns the **absolute pathname** to the specified file.

We can also look for empty files and directories using the `-empty` option.

```
find ./findIt -empty
```

Many UNIX commands can be executed on specified files or folders using `-exec`. Using it with `find` gives us the ability to search for text within multiple files.

Enter the following command to:
- Search within the directories `./`
- For a file input type `-type f`
- By name `-name`
- Files with any combination of characters `*`followed by the `.txt`file extension
- Search within files for lines matching `music`

```
find ./ -type f -name "*.txt" -exec grep 'music'  {} \;
```

You can learn more about `find` and all of its options by entering `man find` in the terminal. Click the button below to explore the `find` manual pages.

tee

The tee command receives standard input and writes it to standard output as well as one or more files. It breaks a program's output so it can be shown and saved. It performs both jobs simultaneously, saving the output to files or variables and displaying it.

Let's use the `df` command to get information about available disk space.
The output is piped to `tee`, which does two things:
1. Shows the output of the command in the terminal
1. writes the same output to `disk_use.txt`.

```
df -h | tee disk_use.txt
```

We can see the output in the terminal. We should be able to see a copy of the same content within `disk_use.txt`.

```
cat disk_use.txt
```

The tee utility can write to many files simultaneously by passing a space-separated list of files as arguments.

```
echo "A message for many eyes" | tee message1.txt message2.txt
message3.txt
```

`tee` also has several options to explore.

We an append information to one or more files with the `-a` option.

```
echo "that only few can see" | tee -a message1.txt message2.txt
```

We can also hide the output of `tee` from the standard output by redirecting the results to `/dev/null`.

```
echo "and even fewer will hear" | tee -a message1.txt >
/dev/null
```

If we're ever stuck on how to use `tee`, we can type `tee --help` in the terminal for guidance.

xargs

`xargs` takes standard input and turns it into command-line arguments. It takes a list of items separated by spaces and executes a command once for each item in the list. It can be used anywhere you want to send command output, line-by-line, to another command or pipeline.

.guides/img/xargs

Some tools, like grep, take standard input. Others can't. The `xargs` utility allows tools like `echo` and `rm` to use standard input as arguments.

Let's explore a few examples.

We'll use the output of `echo` as arguments into the `mkdir` command, creating directories based on `echo`'s output.

```
echo 'red blue yellow' | xargs mkdir
```

Now, list the contents of the current directories using `ls`. We should now be able to see three new directory folders: red, blue, and yellow.

```
ls
```

`xargs` is commonly used with the `find` command to process results or operate on files that match certain criteria. Common examples include relocating files and modifying the ownership of files.

Let's use the following command to:
- Find all `.txt` files in our current directory, and
- Send the results as input to the `cat` command to display the contents of every `.txt` file.

```
find -type f -name "*.txt" | xargs cat
```

When using `xargs`, it might be difficult to visualize the precise command that will be executed. The `-t` option allows us to print the command that will be run right before executing. This is useful for debugging scripts.

```
find -type f -name "*.txt" | xargs -t echo
```

We should see that the following command will execute as a result.

```
echo ./pres_last.txt ./students.txt ./greeneggs.txt
./pres_first.txt
```

`xargs` can be challenging to master. We can learn more about `xargs` and all of its options by entering `man xargs` in the terminal. Click the button below to explore the `xargs` man pages.