ITST 1136 - Searching, Extracting & Archiving Data

Name: _____

Step 1

Sign into a Pi UN = pi PW = raspberry

Step 2 - Grep - One of the most useful and versatile commands in a Linux terminal environment is the "grep" command. The name "grep" stands for "global regular expression print". This means that grep can be used to see if the input it receives matches a specified pattern.

In its simplest form, grep can be used to match literal patterns within a text file. This means that if you pass grep a word to search for, it will print out every line in the file containing that word.

A. Let's try an example. We will use grep to search for every line that contains the word "GNU" in the GNU General Public License version 3 on an Ubuntu system.

- Navigate to: [cd] **/usr/share/common-licenses**

- **grep "GNU" GPL-3**
  - how many GNU's do you see? _____

B. If we would want grep to ignore the "case" of our search parameter and search for both upper- and lower-case variations, we can specify the "-i" or "--ignore-case" option.

- **grep -i "license" GPL-3**
- How many uppercase Licenses? _____
- How many lowercase licenses? _____

As you can see, we have been given results that contain: "LICENSE", "license", and "License". If there was an instance with "LiCeNsE", that would have been returned as well.

C. We can search for every line that *does no*t contain the word "the" in the BSD license with the following command:

- **grep -v "the" BSD**
- Do you see the word "the"? _____

D. It is often useful to know the line number that the matches occur on. This can be accomplished by using the "-n" or "--line-number" option.

- **grep -vn "the" BSD**
- How many lines did you get? _____

Now we can reference the line number if we want to make changes to every line that does not contain "the".

Step 3

Regular Expressions - In the introduction, we stated that grep stands for "global regular expression print". A "regular expression" is a text string that describes a particular search pattern.

Different applications and programming languages implement regular expressions slightly differently. We will only be exploring a small subset of the way that grep describes its patterns.

Literal Matches - The examples above, when we searched for the words "GNU" and "the", we were actually searching for very simple regular expressions, which matched the exact string of characters "GNU" and "the".

It is helpful to always think of these as matching a string of characters rather than matching a word. This will become a more important distinction as we learn more complex patterns.

Patterns that exactly specify the characters to be matched are called "literals" because they match the pattern literally, character-for-character.

All alphabetic and numerical characters (as well as certain other characters) are matched literally unless modified by other expression mechanisms.

Anchor Matches - Anchors are special characters that specify where in the line a match must occur to be valid.

For instance, using anchors, we can specify that we only want to know about the lines that match "GNU" at the very beginning of the line. To do this, we could use the "^" anchor before the literal string.

a) This string example will only match "GNU" if it occurs at the very beginning of a line.

- **grep "^GNU" GPL-3**
- How many lines did you get? _____

b) Similarly, the "$" anchor can be used after a string to indicate that the match will only be valid if it occurs at the very end of a line.

We will match every line ending with the word "and" in the following regular expression:

- **grep "and$" GPL-3**
- How many lines did you get? _____

c) Matching Any Character. The period character (.) is used in regular expressions to mean that any single character can exist at the specified location.

For example, if we want to match anything that has two characters and then the string "cept", we could use the following pattern:

- **grep "..cept" GPL-3**
- How many except instances? ____
- How many accept instances? ____

As you can see, we have instances of both "accept" and "except" and variations of the two words. The pattern would also have matched "z2cept" if that was found as well.

d) Bracket Expressions. By placing a group of characters within brackets ("[" and "]"), we can specify that the character at that position can be any one character found within the bracket group.

This means that if we wanted to find the lines that contain "too" or "two", we could specify those variations succinctly by using the following pattern:

- **grep "t[wo]o" GPL-3**
- How many variables did it find? _____

We can see that both variations are found within the file.

e) Bracket notation also allows us some interesting options. We can have the pattern match anything **except** the characters within a bracket by beginning the list of characters within the brackets with a "^" character.

This example is like the pattern ".ode", but will not match the pattern "code":

- **grep "[^c]ode" GPL-3**
- What words did it find? _____ _____ _____

You will notice that in the second line returned, there is, in fact, the word "code". This is not a failure of the regular expression or grep.

Rather, this line was returned because earlier in the line, the pattern "mode", found within the word "model", was found. The line was returned because there was an instance that matched the pattern.

f) Another helpful feature of brackets is that you can specify a range of characters instead of individually typing every available character.

This means that if we want to find every line that begins with a capital letter, we can use the following pattern:

- **grep "^[A-Z]" GPL-3**
- Did you get a lot of data returned? _____

g) Repeat Pattern Zero or More Times. Finally, one of the most commonly used meta-characters is the "*", which means "repeat the previous character or expression zero or more times".

If we wanted to find each line that contained an opening and closing parenthesis, with only letters and single spaces in between, we could use the following expression:

- grep "([A-Za-z]*)" GPL-3
- How is the last line of what was returned different than what you got in step f ? _____

h) Escaping Meta-Characters. Sometimes, we may want to search for a literal period or a literal opening bracket. Because these characters have special meaning in regular expressions, we need to "escape" these characters to tell grep that we do not wish to use their special meaning in this case.

We can escape characters by using the backslash character (\) before the character that would normally have a special meaning.

For instance, if we want to find any line that begins with a capital letter and ends with a period, we could use the following expression. The ending period is escaped so that it represents a literal period instead of the usual "any character" meaning:

- **grep "^[A-Z].*\.$" GPL-3**
  - What are the last 4 words that were returned?
    _____

Step 4 - Extended Regular Expressions

Grep can be used with an even more extensive regular expression language by using the "-E" flag or by calling the "egrep" command instead of grep.

These options open the capabilities of "extended regular expressions". Extended regular expressions include all the basic meta-characters, along with additional meta-characters to express more complex matches.

A. *Grouping* - One of the easiest and most useful abilities that extended regular expressions open up is the ability to group expressions together to manipulate or reference as one unit.

Group expressions together using parentheses. If you would like to use parentheses without using extended regular expressions, you can escape them with the backslash to enable this functionality.

- grep "\(grouping\)" file.txt
- grep -E "(grouping)" file.txt
- egrep "(grouping)" file.txt

The above three expressions are functionally equivalent.

b) *Alternation* - Similar to how bracket expressions can specify different possible choices for single character matches, alternation allows you to specify alternative matches for strings or expression sets.

To indicate alternation, we use the pipe character "|". These are often used within parenthetical grouping to specify that one of two or more possibilities should be considered a match.

The following will find either "GPL" or "General Public License" in the text:

- **grep -E "(GPL|General Public License)" GPL-3**
- How many General Public Licenses did you get? ___
- How many GPL's _____

Alternation can select between more than two choices by adding additional choices within the selection group separated by additional pipe (|) characters.

c) Quantifiers - Like the "*" meta-character, that matched the previous character or character set zero or more times, there are other meta-characters available in extended regular expressions that specify the number of occurrences.

To match a character zero or one times, you can use the "?" character. This makes character or character set that came before optional, in essence.

The following matches "copyright" and "right" by putting "copy" in an optional group:

- **grep -E "(copy)?right" GPL-3**
- How many "right" did you get? _____

d) The "+" character matches an expression one or more times. This is almost like the "*" meta-character, but with the "+" character, the expression *must* match at least once.

   The following expression matches the string "free" plus one or more characters that are not whitespace:

   - **grep -E "free[^[:space:]]+" GPL-3**
   - How many did you get? _____

e) Specifying Match Repetition - If we need to specify the number of times that a match is repeated, we can use the brace characters ("{" and "}"). These characters are used to specify an exact number, a range, or an upper or lower bounds to the amount of times an expression can match.

   If we want to find all of the lines that contain triple-vowels, we can use the following expression:

   - **grep -E "[AEIOUaeiou]{3}" GPL-3**
   - How many instances did it find? ____

f) If we want to match any words that have between 16 and 20 characters, we can use the following expression:

   - **grep -E "[[:alpha:]]{16,20}" GPL-3**
   - How many instances did it find? ____

Grep Summary –

There are many times when grep will be useful in finding patterns within files or within the file system hierarchy. It is worthwhile to become familiar with its options and syntax to save yourself time when you need it.

Regular expressions are even more versatile, and can be used with many popular programs. For instance, many text editors implement regular expressions for searching and replacing text.

Furthermore, most modern programming languages use regular expressions to perform procedures on specific pieces of data. Regular expressions are a skill that will be transferrable to many common computer-related tasks.

Step 5 – How to use Find and Locate

One problem users run into when first dealing with a Linux machine is how to find the files they are looking for. This lab will cover how to use the aptly named find command. This will help you search for files on Linux using a variety of filters and parameters. We will also briefly cover the locate command, which can be used to search for commands in a different way.

A. Finding by Name - The most obvious way of searching for files is by name.

- Navigate to the **/usr** directory

To find a file[s] by the name of pip, type:

- **find -name "pip"**
- Where are the files located? Write the path, *remember where you are*.

_____

_____

_____

Remember, this will be case sensitive, meaning a search for "file" is different than a search for "File". To find a file by name, but ignore the case the command would be:

```
find -iname "filename"
```

B. If you want to find all files that don't adhere to a specific pattern, you can invert the search with "-not" or "!". If you use "!", you must escape the character so that bash does not try to interpret it before find can act:

```
find -not -name "query_to_avoid"
```

Or

```
find \! -name "query_to_avoid"
```

Try it on the file "pip"

- **find -not -name "pip"**
- What happened? _____
  - Ctrl C is your friend

C. Finding by Type - You can specify the type of files you want to find with the "-type" parameter. It works like this:

find -type type_descriptor query

Some of the most common descriptors that you can use to specify the type of file are here:

- f: regular file
- d: directory
- l: symbolic link
- c: character devices
- b: block devices

For instance, if we wanted to find all of the character devices on our system, we could issue this command. Note: Do this from the **/etc** directory

- **find / -type c**
- How many files did it find? _____

We can search for all files that end in ".conf" like this. Note: do this from the **/usr** directory

- **find / -type f -name "*.conf"**
- What is the path and file name of the last file returned?

_____

Step 6 - Filtering by Time and Size

Find gives you a variety of ways to filter results by size and time.

A. Size - You can filter by size with the use of the "-size" parameter.

We add a suffix on the end of our value that specifies how we are counting. These are some popular options:

- c: bytes
- k: Kilobytes
- M: Megabytes
- G: Gigabytes
- b: 512-byte blocks

To find all files that are exactly 50 bytes, type: [still in **/usr**]

- **find -size 50c**
- How many files were returned? _____

B. To find all files less than 50 bytes, we can use this form instead:

- **find -size -50c**
- What is the name and path of the last file found?

_____

C. To Find all files more than 700 Megabytes, we can use this command:

- find / -size +700M
- Where were you searching? _____
- What is the name and path of the last file found?

_____

D. Time - Linux stores time data about access times, modification times, and change times.

- Access Time: Last time a file was read or written to.
- Modification Time: Last time the contents of the file were modified.
- Change Time: Last time the file's inode meta-data was changed.

We can use these with the "-atime", "-mtime", and "-ctime" parameters. These can use the plus and minus symbols to specify greater than or less than, like we did with size.

The value of this parameter specifies how many days ago you'd like to search.

To find files that have a modification time of a day ago, type:

- **find / -mtime 1**
- What is the name and path of the last file found?

_____

If we want files that were accessed in less than a day ago, we can type:

- **find / -atime -1**
- What is the name and path of the 2nd to the last file found?

_____

To get files that last had their meta information changed more than 3 days ago, type:

- **find / -ctime +3**
- What is the name and path of the 2nd to the last file found?

_____

There are also some companion parameters we can use to specify minutes instead of days:

- **find / -mmin -1**

This will give the files that have been modified type the system in the last minute.

Find can also do comparisons against a reference file and return those that are newer:

The syntax would be: find / -newer myfile

Step 7 - Find Files Using Locate

An alternative to using find is the locate command. This command is often quicker and can search the entire file system with ease.

A. You can install the command with apt-get:

- **sudo apt-get update**
- **sudo apt-get install mlocate**

The reason locate is faster than find is because it relies on a database of the files on the filesystem.

B. The database is usually updated once a day with a cron script, but you can update it manually by typing:

- **sudo updatedb**

Run this command now. Remember, the database must always be up-to-date if you want to find recently acquired or created files.

C. To find files with locate, simply use this syntax: Note do this at the **/usr** directory

- **locate query**
- What is the name and path of the last file found?

  _____
- Was locate faster than find? _____

D. You can filter the output in some ways.

For instance, to only return files containing the query itself, instead of returning every file that has the query in the directories leading to it, you can use the "-b" for only searching the "basename":

- **locate -b query**
- What is the name and path of the 2nd to the last file found?

  _____

E. To have locate only return results that still exist in the filesystem (that were not remove between the last "updatedb" call and the current "locate" call), use the "-e" flag:

The syntax would be: locate -e query

F. To see statistics about the information that locate has cataloged, use the "-S" option:

- **locate -S**
- how many directories? _____
- how many files? _____

Find and Locate Summary

Both find and locate are good ways to find files on your system. It is up to you to decide which of these tools is appropriate in each situation.

Find and locate are powerful commands that can be strengthened by combining them with other utilities through pipelines.

Step 8 – The wc Command, Logical Operators

A. Instead of searching for information within a file, now you will be digging out information *about* a file using the wc command.

- Navigate to: **/usr/share/common-licenses**
- **wc GPL-3**

The `wc` command produces output that shows three statistics for the file: line count, word count, and byte count. Record the numbers you received from the command.

- Line count = _____Word Count = _____Byte Count = _____.

B. Try another file with `wc`:

- **wc GPL-3**

- How many lines are in the file? Record your answer here:_____.

C. Let's take a look at logical operators. Logical operators allow you to control what commands are issued. They are more useful in shell scripts than on the command line. First try out this `echo` command: type

- **echo "Hello"**
- What happened? _____

Try a logical operator by typing:

- **echo "Hello" && echo "2nd Hello"**
- What happened? _____
  _____

Both `echo` commands should have executed and shown their messages to the screen. This happens because the `&&` symbols are logical operations. When the first command finishes, then the second command will execute (run).

Step 9 - Exploring Redirection and Pipes

A. Type **echo "Hello"** and press Enter. The word Hello should have displayed to your computer screen (Standard Output or STDOUT).

B. Try redirecting STDOUT to a file by typing
- **echo "Hello" > new_file**

Notice nothing was displayed on the screen! This is because standard output, which normally displays to the screen, was redirected (using the >) into a file. The word Hello should now be stored in the file new_file.

C. See what is in the file new_file.
- **cat new_file**
- What do you see? _____

You should see the word Hello, because the word was redirected into the file in the previous step by using the > redirection operator.

D. Redirect output again.
- **cat new_file > second_file**
  Was the output redirected in the step above in the file second_file?
- **cat second_file**.
- What do you see? _____

E. See what happens to the original contents of new_file when additional data is redirected into it.
- **echo "I am redirected" > new_file**
  This will redirect the output of "I am redirected" into the file new_file instead of to the screen. Now see if the redirect above worked.
- **cat new_file**
- Did it work? _____

F. You can redirect standard output and append the output to a file, using the symbols >> together. Try appending to the file

- **echo "I appended this" >> new_file**
  Now see if the redirect above worked
- **cat new_file**
  Using two redirection symbols, >>, appends the output to the end of a file.
- What does the text in the "new_file" say? _____

G. So far you have just been redirecting output. Now you will try redirecting Standard Error (STDERR). First, take a look at where error messages are directed to by default, by typing a command that won't work.

- **cat /etc/PASSWORD**

  Note: The file /etc/PASSWORD does not exist and therefore will generate an error message. By default, error messages are displayed to your terminal screen. Now, try redirecting the error message.

- **cat /etc/PASSWORD 2>> new_file**

There are two things to notice in this command:

1) The number 2 was used to redirect the error message. You did not have to put a number in front of the two redirection symbols (>>) when redirecting output earlier. However, redirecting error messages requires a 2.

2) Since two redirection symbols (>>) were used instead of one, the error message will be appended to the end of the file new_file.

Take a look at how the file new_file looks now

- **cat new_file**
- Is the error message should now be appended at the end of new_file's contents? _____

H. Now try "wiping out" the contents of new_file and just have an error message in the file, by using only one redirection symbol.

- **cat /etc/PASSWORD 2> new_file**

Remember, just using a single redirection symbol will wipe out the current contents of the file and put in the new contents. In this case, the new contents will be an error message.
See if your error message was properly redirected

- **cat new_file**
- Did it work? _____

I. Before you start exploring pipes, clean up the two files you used in redirecting output and an error message:
   - **rm -i *file**
   - Type **y** and press Enter for the questions concerning deleting these two files.

J. Create some new files in your home directory [home/pi]

   - **touch fileA.dat fileB.dat fileC.dat fileD.dat**

     Make sure all four files were created by typing

   - **ls file?.dat** Create any files you missed in the preceding step.

K. Use the pipe symbol to display the files in alphabetically sorted order. The | (pipe) symbol can appear either as a solid line or as a vertical dash. A pipe redirects the standard output of the first command into input for the second command. Try it out.

   - **ls file?.dat | sort**

   That's not too interesting, because files are typically shown in alphabetically sorted order.

L. Reverse the sort order by typing

   - **ls file?.dat | sort -r**

   The pipe symbol took the ls command's standard output and sent it as standard input into the sort command, which then reverse-sorted the file names and displayed them to standard output (the screen).
   - Did it work? _____

M. You can display piped output *and* save it to a file using the **tee** command. Try this out

   - **ls file?.dat | sort -r | tee fileE.dat**

   See if the output was saved to the fileE.dat file

   - **cat fileE.dat**
   - Did it work? _____

N. Instead of doing a regular exercise file cleanup with the **rm** command, you will use the **xargs** command. The xargs command will take information from standard output piped to it, build a command from it, and execute it.

- **ls file?.dat | xargs /bin/rm**

It looks like nothing happened, but all the file?.dat files were deleted. The **xargs** commands uses an absolute directory reference for the commands it builds. That's why /bin/rm was used, instead of simply **rm.**

See if the files were deleted by typing

- **ls file?.dat**
- What message did you receive? _____

Other commands to know:

less is a terminal pager program on Unix, Windows, and Unix-like systems used to view (but not change) the contents of a text file one screen at a time. It is similar to more, but has the extended capability of allowing both forward and backward navigation through the file.

The ***head*** command reads the first few lines of any text given to it as an input and writes them to *standard output* (which, by default, is the display screen). head's basic syntax is: *head [options] [file(s)]*

tail prints the last 10 lines of each FILE to standard output. With more than one FILE, it precedes each set of output with a header giving the file name. If no FILE is specified, or if FILE is specified as a dash ("-"), **tail** reads from standard input. tail syntax: tail [*OPTION*]... [*FILE*]...

Try these commands out

- Navigate to: **/usr/share/common-licenses**
- **less GPL-3** [q to quit]
- **head GPL-3**
  - How many lines were returned? _____
- **tail GPL-3**
  - How many lines were returned? _____