Overview

Learners will be able to...

- **Locate and filter the contents of a file using ++Regular Expressions++.**
- **Use Regular Expressions and control structures to test and operate on specified text.**
- **Combine Regular expressions with common script utilities to locate, filter, and operate on the contents of a file.**

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.

Introduction

A series of characters is called an expression. Metacharacters are characters that have an interpretation that goes beyond their literal meaning. A quote symbol, for example, could represent a person's statement or something else.

A **regular expression** is a pattern that is used to match a set of characters in a file or stream of data. BREs, or basic regular expressions, and EREs, or extended regular expressions, are both defined in the POSIX Standard.

To get the most out of shell scripting, we'll have to learn how to use **Regular Expressions.** Some script commands and utilities, like `grep`, `expr`, `sed`, and `awk`, understand and use Regular Expressions because they are designed specifically to work with text. When used with these other tools, it gives us the ability to see customized data segments within a larger set of data.

We can also use regular expressions to perform actions on specific portions of a data set with string editing tools like `sed`

We'll begin by extending our knowledge into using regular expression characters, and advance by practicing extended regular expressions with `grep` and `egrep`.

Comparing Basic and Extended Regexes

Pattern matching in bash uses regular expressions that are compatible with POSIX. The POSIX standard defines two different kinds of regular expressions:
1. **Basic Regular Expressions** (BREs)
1. **Extended Regular Expressions** (EREs)

Of the two, Extended Regular Expressions (EREs) are more powerful, easier to read, and relatively easy to use.

BREs

With basic regular expressions, the following collection of characters are considered *“special”* and are normally reserved for specific functionality.

Function	Special Character
Match exactly one character	`.`
Character Escape	`\`
Character Set	`[...]`
Reference/repetition operator	`*`

The next collection of characters **do not** have any special functionality for basic regular expressions by default. These characters can be given special functionality by escaping them with the backslash `/`.

Function	Character
Reference/repetition operator	`\+`
Reference/repetition operator	`\?`
Reference/repetition operator	`\{...\}`
Pattern Group	`\(...\)`

EREs

With extended regular expressions, the following collection of characters are considered "*special"* and are normally reserved for specific functionality.

	Special Character

Function	
Match exactly one character	`.`
Character Escape	`\`
Character Set	`[...]`
Invert Pattern Match	`[^pattern]`
Pattern Group	`(...)`
Reference/repetition operators	`{...} + *`
anchor beginning of line	`^pattern`
anchor end of line	`pattern$`

If we want our expression to match any of these characters in a pattern instead of interpreting its function, we have to escape them using either the backslash `\` or square brackets `[...]`. For example:

- To escape the front line anchor `^` and ignore its special function using backslash, we can type `\^`.
- To escape the front line anchor `^` using square brackets `[..]`, we can type `[^]`.

Let's look at a couple of examples to compare the difference.

We'll take the following string as our first example:

```
#
                    ilysmjkjkjklololrolfbrb
```

Description	ERE Pattern	BRE Pattern	Matches
Match all occurrences of `jk`	`(jk)`	`\(jk\)`	ilysm==jkjkjk==lololrolfbrb
Match exactly 2 occurrences of `jk` and/or zero or one instance of `lol`	`(jk){2}|(lol){,1}`	`\(jk\)\{2\}|\(lol\)\{,1\}`	ilysm==jkjk==jk==lol==olrolfbrb

This kind of side-by-side comparison makes is easy to see how fast **Basic Regular Expressions** can become cumbersome and confusing compared to **Extended Regular Expressions**. EREs are much easier to read and they give us powerful options for pattern matching.

Regexes in the Command-Line

Different command-line tools support different regex types by default. Generally, we can have all of these tools support EREs by passing some flag option, commonly `-E`.

Use the table below as reference to which command-line tools support which regex type.

CL Tool	Default Regex Type
`grep, grep -G`	BREs
`sed`	BREs
`grep -E`	EREs
`sed -E`	EREs
`awk`	EREs
Bash: `[[ =~ ]]`	EREs

Matching Single Characters and Words

Let's explore matching exactly one instance of a character pattern or word using regular expressions

Each of the characters in the table below **matches exactly one single character or word** of the type that is described by the character.

Character Match	Function
`.`	Match one character of any type
`/d`	Match one number in range $0 - 9$
`/D`	Match non-number
`/w`	Match one "word" (including letters, digits, and _)
`/W`	Match one non-word character
`/s`	Match one whitespace character (space, tab, return, or newline)
`.`	Match one non-whitespace character
`[list]`	Match one character of a defined set.
`[^list]`	Match one character that's not included in a defined set.
`[0-9]`	Match one integer within a range
`^`	Match character or pattern at the beginning of a word
`$`	Match character or pattern at the end of a word.

The same **character classes** that were useful for our globbing section, are useful with regular expressions as well.

Class	Definition
`[[:alnum:]]`	`[[:alpha:]]` and `[[:digit:]]` are alphanumeric characters; in the `C` locale and ASCII character encoding, this is the same as `[[0-9A-Za]-z]`.
`[[:alpha:]]`	Alphabetic characters: `[[:lower:]]` and `[[:upper:]]`; this is the same as `[[A-Za-z]]` in the `C` locale and ASCII character encoding.
`[[:blank:]]`	Space and tab are examples of blank characters.
`[[:cntrl:]]`	This includes control characters. These have the octal codes 000 through 037 in ASCII, and 177 in

	Unicode (DEL).
`[[:digit:]]`	`0 1 2 3 4 5 6 7 8 9`

It's important to remember that **these characters only match a single instance of a character or word** description. We'll explore matching more than one occurrence in the next section.

Checkpoint

Defining Number of Occurrences

Let's explore how to define exactly how many matches of a pattern we want our regular expression to find.

Character Match	Function
`.?`	Match zero or one instances
`.+`	Match one or more instances
`.*`	Match zero or more instances
`.{5}`	Match a specific number of instances
`.{,5}`	Match 0 to 5 instances
`.{5,}`	Match 5 or more instances
`.{2,5}`	Match at least 2 or at most 5 instances

Pattern Groups and Alternation

Grouping allows us to combine multiple patterns into a single match. This is useful because it lets us to define how many times we want to match a complete pattern group using occurrence operands.

Alternation is the regex equivalent of logical OR. It allows us to define one pattern or another. We can use alternation along with grouping to match either one pattern group or another.

Let's look at an example.

We can use alternation `|` to ask a regex to match `red` **or** `blue` using the syntax below.

#

```
red|blue
```

This expression will match:
- `red`
- `blue`

If we do not anchor the front and back of the expression with a `^` or `$`, respectively, the expression will enthusiastically match multiple occurrences of both options:

- `redred`
- `blueblue`

We can specifically define the number of instances we want to match by putting our regular expression inside parenthesis `()` and using an occurrence special character.

In the example below, we use the `?` character to define zero or one occurrences of the preceding pattern, `red` or `blue`.

#

```
^(red|blue)?$
```

This expression will only match:
- red
- blue

important

Notice

This expression is anchored with ^ in the front and $ in the back to prevent matching anything other than exactly what we ask for.

If we replace the ? character with the + symbol, we can require our expression to match zero or more instances of the preceding pattern, `red` or `blue`, intentionally matching multiple occurrences.
- red
- redred
- blue
- blueblue

Let's look at one more example.

How would we match all numbers 1-99?

We want to make sure we account for two things:
- All single digit numbers $1 - 9$, **OR**
- All double digit numbers $10 - 99$

We can cover both of these cases with alternation.

To cover the first requirement, we can use a character set that includes the range $1 - 9$.
#

```
[1-9]
```

For our second requirement, we can use two number character sets: one for the tens' place digits, and one for the ones' place digits.

#

```
[1-9][0-9]
```

Our completed alternation expression to match all numbers $1 - 99$ would look like the example below. Let's not forget to include our anchors so we get exactly the amount of matches we're looking for, and no more.

#

```
^([1-9]|[1-9][0-9])$
```

We can also cover this case by defining a specific number of instances a character set match can occur, by making the first character set mandatory, and using the `?` character to specify zero or one instances of the second match, as in the expression below.

```
#
                              ^([1-9][0-9]?)$
```

Checkpoint

Back References

A **back reference** is a pattern that is placed into a buffer so that it can be used later. This buffer can save up to nine unique patterns. These patterns can be recalled using a backslash followed by the pattern's position in the buffer `\1`, `\2`, etc. Back reference characters range from `\1`, to the limit of buffer references, `\9`.

Say we wanted to match words that contained more than one `ch`. We can use a back reference to find these words by:
- Specifying the pattern we're looking for in parenthesis to place it in a buffer: `(ch)`
- Followed by any number of any type of characters `.*`
- Recall and match the pattern that now lives in the first buffer position with a back reference character: `\1`

#

```
(ch).*\1
```

The result would be a pattern that matches words with two sets of `ch`:
- One match to the pattern in parenthesis
- Another match to the back reference character.

We can also recall this pattern more than once to match words with three sets of `ch`.

#

```
(ch).*\1.*\1
```

info

Note

Back references are not allowed in the POSIX Standard for EREs. If you use POSIX tools, you need to change EREs to BREs for this to work.

BASH_REMATCH

BASH_REMATCH is an additional functionality, similar to back references, that is included with Bash's in process regular expressions.

The major difference between the two is while back references are limited to nine patterns being saved into the buffer, **`BASH_REMATCH` does not have a limit to the number of patterns you can store inside the array.**

If a Bash regular expression matches a string, the test statement returns a code of `0` for `TRUE`, and the portion that was matched is put into an array named `BASH_REMATCH`.

Let's take the following string for example.

#

```
hippopotamus
```

We can test to see if the pattern `po` followed by exactly one of any character exists within the string above by using double square brackets `[[...]]` and comparing the string to the regex with the `=~` operator.

Then, we can display the return code held inside the `$?` variable with the `echo` command.

Copy and paste the statement below into the terminal and press `return`.

```
[[ hippopotamus =~ po. ]]; echo $?
```

This test returns a code of `0` for `TRUE`. The matching pattern should now be stored in the `$BASH_REMATCH` variable.

Use the statement below in the terminal to display the first item stored inside the `$BASH_REMATCH` array.

```
echo $BASH_REMATCH[0]
```

We can see that first instance of the pattern match is now stored inside the `$BASH_REMATCH` array.
#

If our regular expression test does not match a pattern in our specified text, the test returns a code of `1` for `FALSE` and the `$BASH_REMATCH` array returns empty or `NULL`.

Copy and paste the statement below into the terminal and press `return` to see this in action.

```
[[ hippopotamus =~ ze. ]]; echo $?; echo "${BASH_REMATCH[0]}"
```

Because `BASH_REMATCH`is an array, it can also store the results of matches that are grouped together. Just like with back references, patterns that are enclosed in parenthesis `()` are stored in the array. patterns that are not wrapped in parenthesis `()`, do not get stored.

The regular expression below includes 3 groups wrapped in parenthesis `()` ### ### Copy and paste the statement below into the terminal and press `return` to test this expression

```
[[ "hippopotamus" =~ (p)o(t)(.) ]]; echo $?
```

We receive a return code of `0` for a match. Now, if we display the contents of the array, `BASH_REMATCH[0]` contains the entire matched pattern and `BASH_REMATCH[1]` contains the match to the first group.

```
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
echo "${BASH_REMATCH[2]}"
```

`BASH_REMATCH[2]` contains `t` instead of `o` because that match was not included in a group, therefore it did not get stored. Because the following character was included in a group, it was stored in the next available space.

The values stored in the `BASH_REMATCH` array are only stored until we process another regular expression. At that point the values in the array are replaced with the results of the most recent expression.

Regexes with IF statements

Bash only lets you use regular expressions with the double square bracket if statement `[[ =~ ]]`. In just about any other case, extended globs provide us with the best option pattern matching.

Bash's in-process regular expressions only work with EREs, and only do so when they are inside double square brackets and compared with `=~` operator.

Let's look at the example below.

Remember our filename patterns from our globbing exercises.

#

`Logfile_03_10_2022.txt.csv`

We can use an `IF` statement and regular expressions to check if:
- The filename stored in the variable `$FILENAME` matches `=~`
- A pattern beginning with `Logfile`
- Followed by any number of any type of character `.*`
- Ending in `.txt.csv`, and
- Anchoring the expression in the front `^` and back `$` to prevent any unwanted behaviors.

If this test statement is `TRUE`, the test will return a `0` and execute the `echo` command in the body of the statement.

```
if [[$FILENAME =~ ^Logfile.*.txt.csv$ ]];
then
  echo "$FILENAME matches our expression"
fi
```

important

IMPORTANT

We ++**Do not**++ want to use quotes around the regex inside the if statement. This will break our regular expression.

Checkpoint

Regexes with grep

Grep stands for **G**lobal **R**egular **E**xpression **P**rint and it's Linux's most used regular expressions tool. It gives us access to some powerful options that allow us a lot of flexibility with regular expressions.

`grep` has BREs enabled by default, but it also allows us the use of EREs with the `-E` option or `egrep`. For our exercises, we'll be using `grep -E` to work with EREs.

We can refer to the table below for some useful regex options with `grep`. For a more extensive list, we can type `man grep` in the terminal and press `return`

Option	Description
`grep -E`	Interpret PATTERN as an extended regular expression
`grep -G`	Interpret PATTERN as a basic regular expression
`-e PATTERN`	Use PATTERN as the pattern
`-f FILENAME`	Obtain patterns from FILE, one per line.
`-o`	Only return matched portion
`-p`	Return the return code instead of the match
`-w`	Only match whole words
`-x`	Only match whole lines
`-q`	Do not print to standard output

Let's explore using `grep` with regular expressions to locate text in a file named `jabberwocky`.

Click the button below to display the contents of the file `jabberwocky` to the terminal.

Let's use `grep` to search for lines that contain `ll` inside this file, remembering to include `-E` to enable extended regexes.

Copy and paste the code below into the terminal and press `return`.

```
grep -E '(ll)' jabberwocky
```

Notice that this match returns the entire line that contains the matching pattern. Let's take another look at a pattern using special character matches.

The following pattern will match:
- One character of any type `.`
- Followed by the letter `u`
- Followed by one or more characters of any type `.+`
- The letter `u`
- One character of any type `.`

Paste the code below into the terminal to search for matches to this pattern in the `jabberwocky` file.

```
grep -E '(.u.+u.)' jabberwocky
```

Let's explore the difference in output for this pattern match when we incorporate some of the options available with `grep`.

`-w`

Only match whole words that fit this pattern

```
grep -E -w '(.u.+u.)' jabberwocky
```

`-o`

Only return the portion of the line that matches this pattern

```
grep -E -o '(.u.+u.)' jabberwocky
```

`-x`

Only return entire lines that match this pattern.

```
grep -E -q '(.u.+u.)' jabberwocky
```

`-c`

We can also ask `grep` to count how many matches there are with `-c`.

```
grep -E -q '(.u.+u.)' jabberwocky
```

`grep` with regular expressions can be very useful for finding patterns in files or in the hierarchy of the file system, so it's worth taking the time to learn its options and syntax.

Checkpoint

sed review

Sed is a stream editor utility in Bash. Stream editors allow us to modify text as it is fed through the editor. Sed uses BREs by default, but if the-e option gives us the ability to use EREs.

We will explore three modes while using regexes with sed:
1. **Print**
1. **Delete**, and
1. **Substitute**

Print

This mode is very similar to `grep` in that it finds matches to a pattern and displays them in the terminal. The syntax for this command can be found in the example below. The `p` at the end of the expression indicates that this is a print action.

#

```
sed -n -E '/PATTERN /p'
```

Let's print all of the lines in the file `greeneggs.txt` that contain the pattern `Sam` using `sed`

```
sed -n -E '/Sam/p' greeneggs.txt
```

We can limit this to only match lines beginning with `Sam` by anchoring the pattern with a `^` at the beginning.

```
sed -n -E '/^Sam/p' greeneggs.txt
```

Let's try again, finding all lines that contain `eggs`.

```
sed -n -E '/eggs/p' greeneggs.txt
```

In the example above, we use the `-n` option to disable sed's default behavior of printing all process output.

The pattern in this syntax need not only be a regex. Any pattern, such as globs, character sets, character classes, or regexes, can be entered here.

We have two options for getting information into sed:

- We can specify a file and use a pipe `|` to stream any text data from our source into `sed`, like in the example below.

#
```
cat greeneggs.txt | sed -n -E '/eggs /p'
```

- We can also define a file directly in the command line after the `sed` expression, as in the example below.

#
```
sed -n -E '/eggs /p' greeneggs.txt
```

When data is piped into `sed`, **only the standard output** is changed. The original file is left untouched.

If we want to keep this edited text, we'll have to redirect the resulting output to a new file.

Use the command below in the terminal to redirect the lines that contain the pattern `eggs` into a file called `onlyeggs.txt`

```
sed -n -E '/eggs/p' greeneggs.txt > onlyeggs.txt
```

If we try to redirect the output back to the original file, it will erase the existing data, leaving us with a blank file.

Delete

We can use `sed` to delete a specified pattern from a set of text. The syntax to delete a pattern with `sed` it is very similar to that for printing. The only difference is that we change the trailing `p` to a `d` at the end of the expression.

#
```
sed -n -E '/PATTERN/d'
```

Let's explore this by deleting the pattern `not` from `greeneggs.txt` with the example below.

Copy and paste the command below into the terminal and press `return`.

```
sed -E '/egg/d' greeneggs.txt > noeggs
```

Substitution

Using `sed` with substitution allow us to replace a pattern with another pattern. We specify substitution mode by placing an `s` at the beginning of the expression. In this mode, we have to supply two different patterns:
- The old pattern that we want to find in the text, and
- The new pattern that we want to replace the old pattern.

#

```
sed -E 's/OLDPATTERN/NEWPATTERN/g'
```

We have the option to end this expression with a parameter to further define the behavior of the command.

In this example, we used the letter `g` to define this as a **global** substitution. This tells `sed` that we want to replace every occurrence of this pattern. Without this option, sand will only match and replace the first occurrence of the match.

Let's change every instance of the word `green` to the word `old` using substitution and redirect the result to a file called `oldeggs`.

Run the command below to perform the substitution.

```
sed -E 's/green/old/g' greeneggs.txt > oldeggs
```

info

Notice

You'll probably notice that we're separating our patterns in this statement with forward slashes `/`. We can replace this character with a colon `:` or any other character that doesn't have a specific significance for regular expressions if it becomes difficult to read.

Ranges

`sed` also let's us define specific lines or a range of lines in a file to work within. For example, if we wanted to only **delete** lines $23 - 47$, we can use the `d` option to specify a delete action and specify the range of lines using the syntax below.

```
#
                        sed -E '23,47d'
```

Similarly, if we only wanted to print lines $12 - 24$, we can specify the `p` option and define our desired range.

```
#
                        sed -E '12,24d'
```

If we wanted to declare a range of lines to use for a substitution, we'd put it in front of the expression before the pattern, and before the `s`.

```
#
              sed -E '12,24s/OLDPATTERN/NEWPATTERN/g'
```

If you want to know more of what this powerful tool is capable of, click the button below to open the manual pages fr `sed`.

Checkpoint

Regexes with sed

Now that we're familiar with `sed`, let's explore how to use it with regular expressions. The file `data.csv` contains $1,000$ unformatted phone numbers.

Click the button below to display the contents of the file `data.csv`.

We can use regular expressions with `sed` to edit the numbers to follow the format below.

#

```
(xxx)-xxx-xxxx
```

The first thing we have to do is isolate the first three digits `[0-9]{3}`, representing the area code, and surround these numbers with parenthesis `()`.

The ampersand `%` character along with substitution can help us with this task. Ampersand `%` saves a match to a pattern and allows us to recall it to make modifications.

Run the command below in the terminal to match the first three digits in each line and replace them with the same digits, wrapped in parenthesis `()`

```
sed -E 's/[0-9]{3}/(&)/' data.csv
```

We can use backreferences to modify this approach and add a hyphen between the remaining three and four characters. Let's create three backreference groups by wrapping our patterns to match in parenthesis `()`.

- `\1`: 3-digit Area Code: `([0-9]{3})`
- `\2`: 3-digit Exchange Code: `([0-9]{3})`
- `\3`: 4-digit Line Number: `([0-9]{4})`

Now, let's recall these references with our formatting between each group, and replace the original pattern.

Copy and paste the completed expression into the terminal to see the final formatted phone numbers

```
sed -E 's/([0-9]{3})([0-9]{3})([0-9]{4})/(\1)\2-\3/' data.csv
```

These examples are just a few of the useful actions we can perform with `sed` and regular expressions.

Checkpoint

Regexes with awk

Awk is a text processing programming language that is commonly used as a data extraction and data reporting tool. Awk can perform a variety of tasks, including:

- Text processing
- Producing structured text reports
- Performing mathematical operations
- String manipulations

Because it is an entire programming language, complete scripts can be written using `awk`. This language only supports EREs, so BREs won't work here.

One of the most common uses of `awk` with regular expressions is to operate on a file, look for lines that match, and then do something to the matched portions.

We'll explore this with a data file, `people.csv`

Use the command below to print the contents of the data file.

```
awk '{print}' people.csv
```

We can specify our field separator with the `-F` flag. Since we are working with a `.csv` file, our values are separated by commas `,`.

Awk saves each column of information in a variable, allowing us the ability to print specified columns by recalling the appropriate variable.

Let's explore this by find each line in the file that contains the pattern `Male`, and print the 1st and 4th columns of data for each match.

Run the command below to find these matches in `people.csv`.

```
awk -F"," '/Male/{print $1,$4}' people.csv
```

Using regular expressions with `awk`, we can search for rows of data that match a specific pattern.

For example, we can check the column holding first names `$2` and match all first names that begin and end with `A` with any number of any type of characters in between.

Once we locate these matches, we can print only the first name, last name, and gender columns for these matches.

```
awk -F"," '$2 ~ /^A.*a$/{print $2,$3,$5}' people.csv
```

Checkpoint