Overview

Learners will be able to...

- **Locate and filter files that match a specific pattern using ++Basic Glob Expressions++.**
- **Locate and filter files that match a specific pattern using ++Extended Glob Expressions++.**
- **Combine Bash commands with Glob Expressions to perform actions on files.**
- **Describe the difference between Globs and Regular Expressions**

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.

Order of Expansions

In previous sections, we discussed the different types of expansion available in the shell.

Much like the order of operations in mathematics, Bash evaluates these shell expansions in a specific order.

Take the following command, for example:

```
echo ~/"$USER"/$(who | tail -n10 | awk '{print $2}') \\
$(($USERID + 12 ))/userReport_{1..8}.*
```

1. Brace Expansion
   - `{1..8}`
2. Tilde Expansion
   - `~/`
3. Parameter Expansion
   - `$USER , $USERID`
4. Variable Expansion
   - `$(who | tail -n10 | awk '{print $2}')`
5. Arithmetic Expansion & Command Substitution (left to right)
   - `$(($USERID + 12 ))`
6. Word Splitting
7. Filename Expansion
   - `.*`
8. Quote Removal
   - `"$USER"`

```
echo ~/"$USER"/$(who | tail -n10 | awk '{print $2}') \\
        $(($USERID + 12 ))/userReport_{1..8}.*
```

Checkpoint

Globbing Review

In this section, we'll increase our efficiency in the terminal with some advanced bash scripting. begin this section with a light review on **pattern matching** with Bash.

Bash has several pattern-matching options. There are globs, expanded globs, brace expansion, and basic and extended regular expressions.

Globs are a collection of bash features that match or extend specific patterns. Globs can look and function like regular expressions. It's common to confuse globs with regular expressions. **Regular expressions are used to match text, while globs are used to match file names.**

Click the button below to explore globbing.

It's common to use globbing with the `ls` command. The command below uses `ls` to find:
- `file_`: Any file beginning with this string
- `[0-9]`: Followed by any number
- `?`: Exactly one of any character
- `*`: Any number of any characters

Each glob character will expand to include files that match the characters' descriptions.

Copy and paste the snippet below into the terminal to see this in action.

```
ls file_[0-9]?.*
```

We receive a lot of files that follow this pattern, including files that end in `.txt`, `.pdf`, and `.jpg`. We can modify this glob to return only files ending in `.pdf` with the following command.

```
ls file_[0-9]?.pdf
```

Let's narrow down our file search even more by returning files that begin with `file_1`, following one of any character, and ending in `.pdf`.

```
ls file_1?.pdf
```

▼ **Changing Glob's Behavior**
Bash's interpretation of special characters in globbing can be changed. Globbing is disabled by using the `set -f` command, and the `nocaseglob`nullglob options to shopt alter globbing behavior.

When you are finished exploring, click the button below to clean up the workspace for the next section.

Checkpoint

Wildcards and Character Sets

In file request dialogs, we may often use wildcards to match patterns. They are also very helpful on the command line. The **asterisk** is the most commonly used wildcard character.

Click the button below to explore wildcards.

Copy and paste the snippet into the terminal to use the `ls` command to search for filenames containing any number of any type of character.

```
ls *
```

This is commonly used to search files with a specific extension.

Use the snippet below in the terminal to search for all files with the `.txt` extension.

```
ls *.txt
```

The question mark `?` is similar to the asterisk because it also matches any type of character, but it only matches one of any character.

Character Sets

Character sets are like a specialized wildcard. We can list the probable matches for them. Use single square brackets to construct a character set. The character set matches one character, like the question mark wildcard, but has a shorter list.

Copy the code snippet below into the terminal to match `filei7.txt` through `filek7.txt`.

```
ls file[ijk]7.png
```

Use the snippet below to match against a range of numbers.

```
ls file[0-9]a.png
```

This matches `file0a.png` through `file9a.png`. Because character classes **only match one character**. `file09a.png` and `file4b.png` would not match this expression.

When you are finished exploring, click the button below to clean up the workspace for the next section.

Checkpoint

Character Classes

As you may recall from previous lessons, **character classes** are predefined groups of characters included with bracket expansion capabilities. Their meaning relies on the locale specified by LC_CTYPE. The brackets in these class names are symbolic names, and they must be provided in addition to the brackets that delimit the expression.

Check out the table below for the character classes and their definitions.

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`
`[[:graph:]]`	Graphic characters including `[[:alnum:]]` and `[[:punct:]]`
`[[:lower:]]`	Lower-case letters `a b c d e f g h I j k l m n o p q r s t u v w x y z`
`[[:print:]]`	Printable characters including `[[:alnum:]]`, `[[:punct:]]`, and space
`[[:punct:]]`	Characters for punctuation including `! " # $ % &' () * +, -. /: ; = >? @ [] ' .`
`[[:space:]]`	Tab, newline, vertical tab, form feed, carriage return, and space are all space characters
`[[:upper:]]`	Upper-case letters: `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
`[[:xdigit:]]`	Hexadecimal digits: `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`

Click the button below to explore character classes.

```
ls file[[:alnum:]]8.png
```

```
ls file1[[:lower:]].png
```

```
ls file[[:digit:]][[:lower:]].png
```

```
ls file[![:lower:]][[:digit:]].png
```

Intro to Extended Globbing

Bash also supports extended globs. Extended globs make regular expressions more useful for globbing. Unlike character sets or classes, patterns can include more than one character and many occurrences. We can't say exactly how many, but we can say zero or one, exactly one, one or more, or zero or more.

It adds more capability to `if` conditional statements and scripts, as well as gives extra power to `case` statements.

Why is this useful, since this kind of `if` conditional allows us to match regular expressions? The short answer is: Because it's faster.

We also can't use regular expressions in `case` statements, and standard globs aren't very useful here. `Case` is also faster than `if` because it only evaluates the variable once, where `if...then...elif` evaluates for every test.

Enabling Extended Globbing

Extended globbing is a built-in option supported by Bash. Often, this option is not enabled by default.

We can enable this option by entering `shopt -s extglob` in the terminal.

Click the button below to enable extended globbing in our terminal workspace.

We can make sure extended globs are enables by entering `shopt -p` in the terminal.

Click the button below to check for extended globbing functionality in our terminal workspace.

We'll explore working with extended globs more in the following sections. Extended globs aren't a magic solution to all our problems, but they're a valuable resource.

Regular expressions are sometimes necessary when extended globs just won't do, but keep in mind that regular expressions may take a lot longer.

Checkpoint

Extended Glob Characters

In this section, we'll explore extended glob characters and their definitions.

Click the button below to experiment with the examples below.

@

Using extended globbing, we can **match exactly one** item to a pattern by using `@` before our matching criteria. **The text to the left of the extended glob has to match the rest of the expression as well.**

```
ls file2d@(.png)
```

We can also use a **logical OR** `|` to search for exactly one occurrence of pattern 1 OR pattern 2. Because we included the `@` symbol, the expression below will match exactly one occurrence of `file2d.png` or `file2d.jpg`

```
ls file2d@(.png|.jpg)
```

This expression will **not** match:
- `file2d`
- `file2d.png.jpg`
- `file2d.txt`

?

We can use the `?` character to say that we want to match **0 or 1 occurrence** of a specified pattern.

```
ls file3c?(.txt|.png)
```

This expression would also match `file3c` without an extension because it matches **zero** occurrences of the pattern after the glob while matching everything before the extended glob.

*

Matches **zero or more** occurrences of the given patterns. This pattern will return everything matched by `?` and `+` combined.

```
ls file3c*(.txt|.png)
```

The pattern above matches items with zero occurrences of the pattern, like `file3c`, as well as items with more than zero occurrences of the pattern, like `file3c.txt` and `file3c.txt.png`. Because everything before the extended glob character must also match the result, `file3b` or `file3b.txt` will not match this pattern.

+

The `+` character allows us to match **one or more** occurrences of the given patterns.

```
ls file3c+(.txt|.png)
```

Notice that this pattern matches multiple patterns that come after the extended glob expression, including `file3c.png.txt` and `file3c.txt.txt`.

This expression will not match `file3c` because the `+` needs for there to be at least one match to the pattern after the glob.

!

The `!` symbol inverts the search pattern selection, matching anything **except** one of the given patterns

```
ls file5e!(.txt|.png)
```

We can use this inversion technique to invert exiting glob expressions by **nesting** glob expressions.

Let's revisit our previous + expression to match every file that:

- Begin with `file3c`
- end in one more more instances of `.txt` **or** `.png`

```
ls file3c+(.txt|.png)
```

Now let's ++invert++ this expression using the `!` glob to match every file that:

- Begin with `file3c` (because it comes before the glob expression)
- Does **NOT** end in one more more instances of `.txt` **or** `.png`

```
ls file3c!(+(.txt|.png))
```

Grouping Globs

We can string several extended globs together and treat them as a single match by grouping them together with parenthesis `()`.

Let's look at an example where we match one or more files that:

- Starts with one or more instances of`filea` **or** `filec`: `+(filea|filec)`
- Has any number of characters: `*`
- Ends with one or more instances of `.png` **or** `.jpg`: `+(.png|.jpg)`
- Then, we'll wrap it all in parenthesis and invert the results to match all files that do not match the criteria above: `!(...)`

```
ls !(+(filea|filec)*+(.png|.jpg))
```

When you're finished exploring extended glob characters, click the button below to clean up the workspace.

Matching Filename Patterns

Let's look at some pattern matching options with extended globs.

Navigate to the `PracticeFiles` directory using the `cd` command.

```
cd PracticeFiles
```

Click the button below to generate the files that we'll explore for this section.

Use the `ls` command or the button below to list all of the files in our current directory.

There are several thousand files inside this directory whose filenames are similar to the pattern below.

`Logfile_03_10_2022.txt.csv`

This format includes:

1. The word `Logfile_` or `Backup_`
2. The month: `00_`
3. The date: `0_` or `00_`
4. The year: `0000`
5. A file extension combination: `.log.csv`

Let's build an extended glob expression to check for filenames matching this pattern.

1. First, let's build an expression to match filenames that contain either the word `Logfile` **OR** `|` the word `Backup` followed by an underscore _ separator.

```
@(Logfile|Backup)_
```

2. We'll add to this expression a pattern to match two digits for the month portion of our date. We can do this using **number character sets** `[0-9]` to check for one occurrence of the numbers $0 - 9$. Since our day format has **two numbers**, we'll need **two number character sets**, followed by an underscore _ separator.

```
@(Logfile|Backup)_[0-9][0-9]_
```

3. Now let's add an expression to match either one **OR** two digits for the day using the `@` extended glob character, character sets, and the logical OR `|` pipe. This statements requires that this part of the filename pattern contain one or more occurrences of a match to one number set, or a match to two number sets. We'll follow this with an underscore _ separator.

```
@(Logfile|Backup)_[0-9][0-9]_@([0-9]|[0-9][0-9])_
```

4. Now, we can add four character sets to our expression to match the format of the year.

```
ls @(Logfile|Backup)_[0-9][0-9]_@([0-9]|[0-9][0-9])_[0-9][0-9]
        [0-9][0-9]
```

5. The only thing left for us to include is our expression to match the creative combination of filename extensions.

- **First, let's check to see what filenames extensions exist in our current working directory.**
    - We can use a `for` loop to cycle through all of the filenames and display each `filename` after removing everything `#*` that comes before the first `.`.
    - Then, we'll pipe the results of the for loop into the `sort` command to place these extensions into alphabetical order and include `-u` to only show each unique occurrence, excluding any duplicates.

```
for filename in *; do echo ${filename#*.}; done | sort -u
```

Looking at our list of extensions, we should see two types of filename extensions.

- Files that end in the single-level extensions. All of our single extension practice file end in either `.log` or `.txt`.
- Files that end in the double-level extensions that end with either `.csv`, `.pdf`, or `.xlsx`.

.log	.txt
.log.csv	.txt.csv
.log.pdf	.txt.pdf
.log.xlsx	.txt.xlsx

- **Finally, let's add to our expression to match the patterns found in our filename extensions.**
    - For the first level extensions, let's use `@` to check for one or more occurrences of `.log` or `.txt` with the logical or `|` pipe.
    - For the second level extensions, we'll use `?` to check for zero or more occurrences of `.csv`, `.pdf`, or `.xlsx`

Our complete extended glob expression should look like the example below.
Copy and paste this extended glob into the terminal to list all of the filenames that match this pattern.

```
ls @(Logfile|Backup)_[0-9][0-9]_@([0-9]|[0-9][0-9])_[0-9][0-9]
        [0-9][0-9]@(.log|.txt)?(.csv|.pdf|.xlsx)
```

There are a lot of files! We can pipe the results of this expression into the word count utility `wc` with the line option `-l` enabled to see exactly how many files match this pattern.

Copy and paste the command below into the terminal to see how many files we're working with.

```
ls @(Logfile|Backup)_[0-9][0-9]_@([0-9]|[0-9][0-9])_[0-9][0-9]
[0-9][0-9]@(.log|.txt)?(.csv|.pdf|.xlsx) | wc -l
```

We can also invert this expression with `!` to list all of the files in this directory that do **NOT** match this pattern.

```
ls !(*@(Logfile|Backup)_[0-9][0-9]_@([0-9]|[0-9][0-9])_[0-9][0-
        9][0-9][0-9]@(.log|.txt)?(.csv|.pdf|.xlsx)*)
```

Making changes to and sorting through $17,280$ files can be a daunting task. Thankfully, we can leverage the power of extended globbing to work with multiple files matching a particular pattern at the same time.

Checkpoint

Commands using Extended Globs

Instead of just matching file names, Let's use extended globs with commands to do some destructive pattern matching.

In the terminal, navigate to the practice folder called `PracticeFiles` using the `cd` command.

```
cd PracticeFiles
```

Type `ls` in the terminal to make sure all of our practice files are present. If the directory is empty, click the button below to generate our practice files.

Let's organize our files by separating them into different directories. We'll start by creating two directories named `Logs` and `Backups`

Create two directories, `Logs` and `Backups` using the `mkdir` command and brace expansion `{...}`.

```
mkdir {Logs,Directories}
```

Now, let's match all of the files that begin with `Logfile`, followed by any number of any characters, and ending with any combination of our file extensions.

Use the `ls` command to list the files that match this pattern

```
ls @(Logfile)*@(@(.log|.txt)@(.csv|.xlsx|.pdf))
```

Now that we've matched the proper files, we can move them to the `Logs` directory using them `mv` command.

```
mv @(Logfile)*@(@(.log|.txt)@(.csv|.xlsx|.pdf)) Logs
```

Let's list `ls` the contents of the `Logs` directory to be sure our command was successful.

```
ls Logs/
```

We can do the same for all of the files beginning with `Logfile`, followed by any number of any characters, and ending with any combination of our file extensions.

Let's move the `Backup` files to the `Backups` directory with the `mv` command.

```
mv @(Backup)*@(@(.log|.txt)@(.csv|.xlsx|.pdf)) Backups
```

List the contents of the `Backups` directory to make sure our command was successful.

```
ls Backups/
```

If we list the contents of our `PracticeFiles` directory using `ls`, we'll see that we only have 3 image files, all named `corgi` with different extensions.

Let's move all of the files that start with `corgi` and end in any number of any characters to a directory called `Images`.

```
mv corgi* Images
```

At this point, we've successfully organized all of our files into different directories without having to drag and drop a single one. This is just the very beginning of how useful these expressions can be.

Extended Globs vs. Regular Expressions

Globs can look very similar to regular expressions.

We'll explore regular expressions in more detail in a later section. For now, let's review and clarify some of the differences between globs and regular expressions.

Globs

Globs are typically used to **match files**. Although we may use them in case statements, if conditionals, for loops, and variable pattern matching, their only function is to enhance globbing to match files.

Regular expressions

Regular expressions find matches in text. That text may be the output of a command that lists the contents of a file, like `cat`, or lists files in a directory, like `ls`. Regexes work as long as the output is in text format.

Regular expressions are not meant to match files and we can't use them as globs on the command-line or in for loops. Regular expressions must be done by the command that is taking care of the text processing, like sed, grep or awk.

Regular expressions can also be done with Bash's regular expression matching using double square brackets: `if [[ =~ ]]`

If we look at matching expressions for both globs and regular expressions, we can see just how similar the expressions look. However, we can also easily see the difference between the two. This can be helpful in remembering one if you remember the other.

Unlike extended globs, extended regular expressions allow us to precisely define the number of instances of a pattern that we wish to match, for example, five.

Definition	Extended Glob	Regular Expression
Match any number of occurrences of any		

characters	`*(pattern)`	`(pattern)*`
Match exactly one occurrences of pattern	`@(pattern)`	`(pattern)`
Match one or more occurrences of pattern	`+(pattern)`	`(pattern)+`
Match zero or one occurrences of pattern	`?(pattern)`	`(pattern)?`
Invert match criteria	`!(pattern)`	`(!pattern)`
Match defined number of instances	**Impossible**	`(pattern){5}*`

Checkpoint