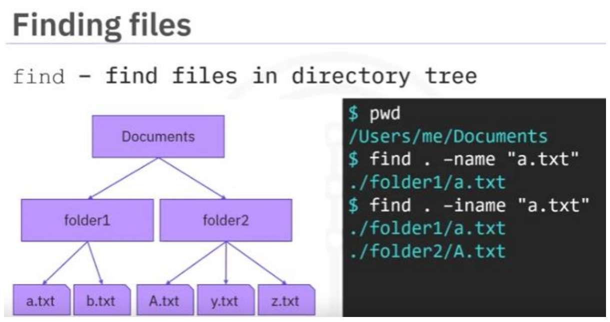
Data Engineering Day 12:

The credit for this course goes to Coursera. Click More

Another link : Azure data Engineer

Introduction to Linux Commands and Shell Scripting

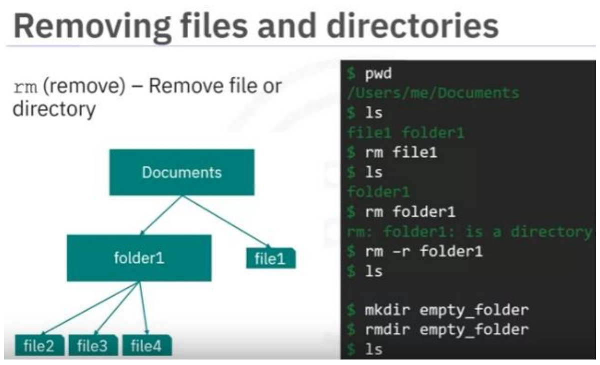
# File Directory and Navigations:



Finding files

find – find files in directory tree

```
$ pwd
/Users/me/Documents
$ find . -name "a.txt"
./folder1/a.txt
$ find . -iname "a.txt"
./folder1/a.txt
./folder2/A.txt
```

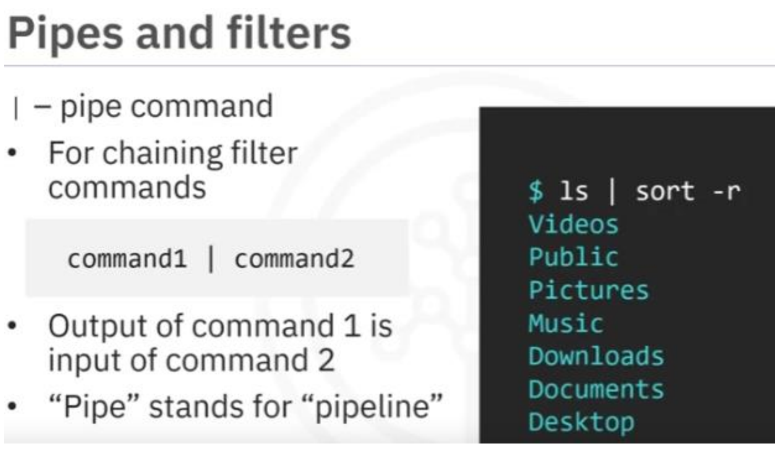


Removing files and directories

rm (remove) – Remove file or directory

```
$ pwd
/Users/me/Documents
$ ls
file1 folder1
$ rm file1
$ ls
folder1
$ rm folder1
rm: folder1: is a directory
$ rm -r folder1
$ ls

$ mkdir empty_folder
$ rmdir empty_folder
$ ls
```



Pipes and filters

| – pipe command

- For chaining filter commands

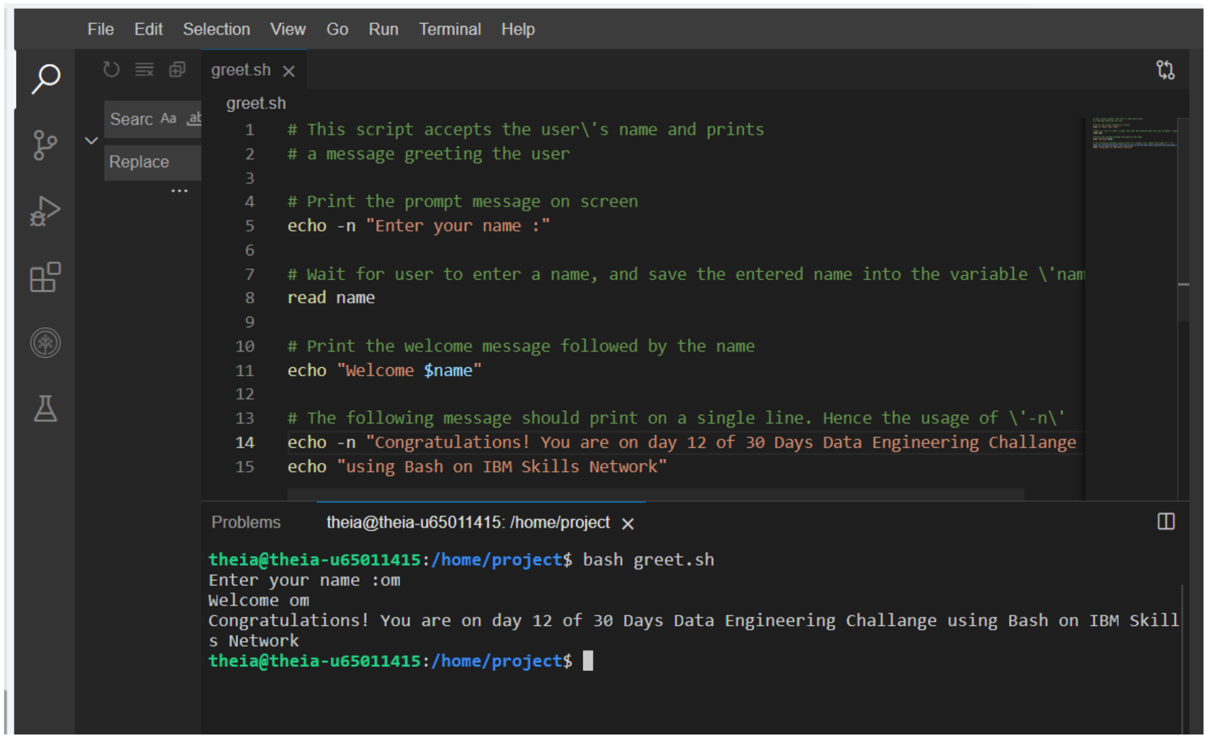
```
command1 | command2
```

- Output of command 1 is input of command 2
- "Pipe" stands for "pipeline"

```
$ ls | sort -r
Videos
Public
Pictures
Music
Downloads
Documents
Desktop
```

⇨ Command 1 becomes the input for command 2.

# Scripting can be beneficial in automating some tasks like ETL process.


File Edit Selection View Go Run Terminal Help
greet.sh
Problems
theia@theia-u65011415: /home/project

```
# This script accepts the user\'s name and prints
# a message greeting the user

# Print the prompt message on screen
echo -n "Enter your name :"

# Wait for user to enter a name, and save the entered name into the variable \'nam
read name

# Print the welcome message followed by the name
echo "Welcome $name"

# The following message should print on a single line. Hence the usage of \'-n\'
echo -n "Congratulations! You are on day 12 of 30 Days Data Engineering Challange
echo "using Bash on IBM Skills Network"
```

```
theia@theia-u65011415:/home/project$ bash greet.sh
Enter your name :om
Welcome om
Congratulations! You are on day 12 of 30 Days Data Engineering Challange using Bash on IBM Skill
s Network
theia@theia-u65011415:/home/project$
```

Module 2 Cheat Sheet - Introduction to Linux Commands

Getting information

Return your user name:

```
whoami
```

Return your user and group id:

```
id
```

Return operating system name, username, and other info:

```
uname -a
```

Display reference manual for a command:

```
man top
```

List available `man` pages, including a brief description for each command:

```
man -k .
```

Get help on a command:

```
curl --help
```

Return the current date and time:

```
date
```

Navigating and working with directories

```
ls -lrt
```

Find files in directory tree that end in `.sh` :

```
find -name \'\*.sh\'
```

Return path to present working directory:

```
pwd
```

Make a new directory:

```
mkdir new_folder
```

Change the current directory:

> Up one level:

```
cd ../
```

> To home:

```
cd ~` or `cd
```

> To some other directory: `cd path_to_directory`

Remove directory verbosely:

```
rmdir temp_directory -v
```

Monitoring system performance and status

List selection of/all running processes and their PIDs:

```
ps
```

Display resource usage:

```
top
```

List mounted file systems and usage:

```
df
```

Creating, copying, moving, and deleting files:

Create an empty file or update existing file's timestamp:

```
touch a_new_file.txt
```

Copy a file:

```
cp file.txt new_path/new_name.txt
```

Change file name or path:

```
mv this_file.txt that_path/that_file.txt
```

Remove a file verbosely:

```
rm this_old_file.txt -v
```

Working with file permissions

Change/modify file permissions to 'execute' for all users:

```
chmod  +x  my_script.sh
```

Change/modify file permissions to 'execute' only for you, the current user:

Remove 'read' permissions from group and other users:

```
chmod go-r
```

Displaying file and string contents

Display file contents:

```
cat my_shell_script.sh
```

Display file contents page-by-page:

```
more ReadMe.txt
```

Display first 10 lines of file:

```
head -10 data_table.csv
```

Display last 10 lines of file:

```
tail -10 data_table.csv
```

Display string or variable value:

```
echo "I am not a robot"
echo "I am $USERNAME"
```

Basic text wrangling

Sorting lines and dropping duplicates:

Sort and display lines of file alphanumerically:

```
sort text_file.txt
sort -r text_file.txt
```

Drop consecutive duplicated lines and display result:

```
uniq list_with_duplicated_lines.txt
```

Displaying basic stats:

Display the count of lines, words, or characters in a file:

> **Lines:**

```
wc  -l table_of_data.csv
```

> **Words:**

```
wc  -w my_essay.txt
```

> **Characters:**

```
wc  -m some_document.txt
```

Extracting lines of text containing a pattern:

Some frequently used options for `grep` :

Option	Description
`-n`	Print line numbers along with matching lines
`-c`	Get the count of matching lines
`-i`	Ignore the case of the text while matching
`-v`	Print all lines which do not contain the pattern
`-w`	Match only if the pattern matches whole words

Extract lines containing the word "hello", case insensitive and whole words only:

```
grep  -iw hello  a_bunch_of_hellos.txt
grep  -l hello  *.txt
```

Merge two or more files line-by-line, aligned as columns:

Suppose you have three files containing the first and last names of your customers, plus their phone numbers.

Use `paste` to align file contents into a Tab-delimited table, one row for each customer:

```
paste first_name.txt last_name.text phone_number.txt
```

Use a comma as a delimiter instead of the default Tab delimiter:

```
paste -d "," first_name.txt last_name.text phone_number.txt
```

Use the `cut` command to extract a column from a table-like file:

Suppose you have a text file whos rows consist of first and last names of customers, delimited by a comma.

Extract first names, line-by-line:

```
cut -d "," -f 1 names.csv
```

Extract the second to fifth characters (bytes) from each line of a file:

```
cut -b 2-5 my_text_file.txt
```

Extract the characters (bytes) from each line of a file, starting from the 10th byte to the end of the line:

```
cut -b 10- my_text_file.txt
```

Compression and archiving

Archive a set of files:

```
tar -cvf my_archive.tar.gz file1 file2 file3
```

```
zip my_zipped_files.zip file1 file2
zip my_zipped_folders.zip directory1 directory2
```

Extract files from a compressed zip archive:

```
unzip my_zipped_file.zip
unzip my_zipped_file.zip -d extract_to_this_direcory
```

Working with networking commands

Print hostname:

```
hostname
```

Send packets to URL and print response:

```
ping  www.google.com
```

Display or configure system network interfaces:

```
ifconfig
ip
```

Display contents of file at a URL:

```
curl  <url>
```

Download file from a URL:

```
wget  <url>
```

Authors

Jeff Grossman
Sam Propupchuk

Reading: A Brief Introduction to Shell Variables

Learning Objectives

After completing this reading, you will be able to:

- Describe shell variables
- Create shell variables

What is a shell variable?

Shell variables offer a powerful way to store and later access or modify information such as numbers, character strings, and other data structures by name. Let's look at some basic examples to get the idea.

Consider the following example.

```
$ firstname=Jeff
$ echo $firstname
Jeff
```

The first line assigns the value `Jeff` to a new variable called `firstname`. The next line accesses and displays the value of the variable, using the `echo` command along with the special character `$` in front of the variable name to extract its value, which is the string `Jeff`.

Thus, we have created a new shell variable called `firstname` for which the value is `Jeff`.

This is the most basic way to create a shell variable and assign it to a value all in one step.

Reading user input into a shell variable at the command line

Here's another way to create a shell variable, using the `read` command. After entering

```
$ read lastname
```

on the command line, the shell waits for you to enter some text:

```
$ read lastname
Grossman
$
```

Now we can see that the value `Grossman` has just been stored in the variable `lastname` by the `read` command:

```
$ read lastname
Grossman
$ echo $lastname
Grossman
```

By the way, notice that you can echo the values of multiple variables at once.

```
$ echo $firstname $lastname
Jeff Grossman
```

As you will soon see, the `read` command is particularly useful in shell scripting. You can use it within a shell script to prompt users to input information, which is then stored in a shell variable and available for use by the shell script while it is running. You will also learn about **command line arguments**, which are values that can be passed to a script and automatically assigned to shell variables.

Summary

In this reading, you learned that:

- Shell variables store values and allow users to later access them by name
- You can create shell variables by declaring a shell variable and value or by using the `read` command

Authors

Jeff Grossman

Other Contributors

Rav Ahuja

Change Log

Examples of Pipes

Learning Objectives

After completing this reading, you will be able to:

- Describe pipes
- Use pipes to combine commands when working with strings and text file contents
- Use pipes to extract information from URLs

What are pipes?

Put simply, pipes are commands in Linux which allow you to use the output of one command as the input of another.

Pipes `|` use the following format:

```
[command 1] | [command 2] | [command 3] ... | [command n]
```

There is no limit to the number of times you can chain pipes in a row!

In this lab, you'll take a closer look at how you can use pipes and filters to solve basic data processing problems.

Pipe examples

Combining commands

Let's start with a commonly used example. Recall the following commands:

- `sort` - sorts the lines of text in a file and displays the result

- `uniq` - prints a text file with any consecutive, repeated lines collapsed to a single line

With the help of the pipe operator, you can combine these commands to print all the unique lines in a file.

Suppose you have the file `pets.txt` with the following contents:

```
$ cat pets.txt
goldfish
dog
cat
parrot
dog
goldfish
goldfish
```

If you *only* use `sort` on `pets.txt`, you get:

```
$ sort pets.txt
cat
dog
dog
goldfish
goldfish
goldfish
parrot
```

The file is sorted, but there are duplicated lines of "dog" and "goldfish".

On the other hand, if you *only* use `uniq`, you get:

```
$ uniq pets.txt
goldfish
dog
cat
parrot
dog
goldfish
```

This time, you removed consecutive duplicates, but non-consecutive duplicates of "dog" and "goldfish" remain.

But by combining the two commands in the correct order - by first using `sort` then `uniq` - you get back:

```
$ sort pets.txt | uniq
cat
```

```
dog
goldfish
parrot
```

Since `sort` sorts all identical items consecutively, and `uniq` removes all consecutive duplicates, combining the commands prints only the unique lines from `pets.txt` !

Applying a command to strings and files

Some commands such as `tr` only accept *standard input* - normally text entered from your keyboard - but not strings or filenames.

- `tr` (translate) - replaces characters in input text

```
tr [OPTIONS] [target characters] [replacement characters]
```

In cases like this, you can use piping to apply the command to strings and file contents.

With strings, you can use `echo` in combination with `tr` to replace all the vowels in a string with underscores `_` :

```
$ echo "Linux and shell scripting are awesome\!" | tr "aeiou" "_"
L_n_x _nd sh_ll scr_pt_ng _r_ _w_s_m_!
```

To perform the complement of the operation from the previous example - or to replace all the *consonants* (any letter that is not a vowel) with an underscore - you can use the `-c` option:

```
$ echo "Linux and shell scripting are awesome\!" | tr -c "aeiou" "_"
_i_u__a_____e______i__i___a_e_a_e_o_e_
```

With files, you can use `cat` in combination with `tr` to change all of the text in a file to uppercase as follows:

```
$ cat pets.txt | tr "[a-z]" "[A-Z]"
GOLDFISH
DOG
CAT
PARROT
DOG
GOLDFISH
GOLDFISH
```

The possibilities are endless! For example, you could add `uniq` to the above pipeline to only return unique lines in the file, like so:

```
$ sort pets.txt | uniq | tr "[a-z]" "[A-Z]"
CAT
DOG
GOLDFISH
PARROT
```

Extracting information from JSON Files:

Let's see how you can use this json file to get the current price of Bitcoin (BTC) in USD, by using grep command.

```
{
  "coin": {
    "id": "bitcoin",
    "icon": "https://static.coinstats.app/coins/Bitcoin6l39t.png",
    "name": "Bitcoin",
    "symbol": "BTC",
    "rank": 1,
    "price": 57907.78008618953,
    "priceBtc": 1,
    "volume": 48430621052.9856,
    "marketCap": 1093175428640.1146,
    "availableSupply": 18877868,
    "totalSupply": 21000000,
    "priceChange1h": -0.19,
    "priceChange1d": -0.4,
    "priceChange1w": -9.36,
    "websiteUrl": "http://www.bitcoin.org",
    "twitterUrl": "https://twitter.com/bitcoin",
    "exp": [
      "https://blockchair.com/bitcoin/",
      "https://btc.com/",
      "https://btc.tokenview.com/"
    ]
  }
}
```

Copy the above output in a file and name it as `Bitcoinprice.txt` .

The JSON field you want to grab here is `"price": [numbers].[numbers]"` . To get this, you can use the following `grep` command to extract it from the JSON text:

```
grep -oE "\"price\"\s*:\s*[0-9]*?\.[0-9]*"
```

Let's break down the details of this statement:

- `-o` tells `grep` to *only* return the matching portion
- `-E` tells `grep` to be able to use extended regex symbols such as `?`
- `\"price\"` matches the string `"price"`
- `\s*` matches any number (including 0) of whitespace (`\s`) characters
- `:` matches `:`
- `[0-9]*` matches any number of digits (from 0 to 9)
- `?\.` optionally matches a `.`

Use the cat command to get the output of the JSON file and pipe it with the grep command to get the required output.

```
cat Bitcoinprice.txt | grep -oE "\"price\"\s*:\s*[0-9]*?\.[0-9]*"
```

You can also extract information directly from URLs and retreive any specific data using such grep commands.

▶ Click here to see the process of extracting information directly from URLs and retreiving specific data:

Summary

In this reading, you learned that:

- Pipes are commands in Linux which allow you to use the output of one command as the input of another
- You can combine commands such as `sort` and `uniq` to organize strings and text file contents
- You can pipe the output of a `curl` command to `grep` to extract components of URL data

Authors

Jeff Grossman Sam Prokopchuk

Skills Network

IBM

Examples of Bash Shell Features

Learning Objectives

After completing this reading, you will be able to:

- List examples of metacharacters
- Use quoting to specify literal or special character meanings
- Implement input and output redirection
- Apply command substitution
- Describe applications for command line arguments

Metacharacters

Metacharacters are characters having special meaning that the shell interprets as instructions.

Metacharacter	Meaning
`#`	Precedes a comment
`;`	Command separator
`*`	Filename expansion wildcard
`?`	Single character wildcard in filename expansion

Pound `#`

The pound `#` metacharacter is used to represent comments in shell scripts or configuration files. Any text that appears after a `#` on a line is treated as a comment and is ignored by the shell.

```
#!/bin/bash

# This is a comment
echo "Hello, world!"  # This is another comment
```

Comments are useful for documenting your code or configuration files, providing context, and explaining the purpose of the code to other developers who may read it. It's a best practice to include comments in your code or configuration files wherever necessary to make them more readable and maintainable.

Semicolon `;`

The semicolon `;` metacharacter is used to separate multiple commands on a single command line. When multiple commands are separated by a semicolon, they are executed sequentially in the order they appear on the command line.

```
$ echo "Hello, "; echo "world!"
Hello,
world!
```

As you can see from the example above, the output of each `echo` command is printed on separate lines and follows the same sequence in which the commands were specified.

The semicolon metacharacter is useful when you need to run multiple commands sequentially on a single command line.

Asterisk `*`

The asterisk `*` metacharacter is used as a wildcard character to represent any sequence of characters, including none.

```
ls *.txt
```

In this example, `*.txt` is a wildcard pattern that matches any file in the current directory with a `.txt` extension. The `ls` command lists the names of all matching files.

Question mark `?`

The question mark `?` metacharacter is used as a wildcard character to represent any single character.

```
ls file?.txt
```

In this example, `file?.txt` is a wildcard pattern that matches any file in the current directory with a name starting with `file`, followed by any single character, and ending with the `.txt` extension.

Quoting

Quoting is a mechanism that allows you to remove the special meaning of characters, spaces, or other metacharacters in a command argument or shell script. You use quoting when you want the shell to interpret characters literally.

Symbol	Meaning
\	Escape metacharacter interpretation

Symbol	Meaning
`" "`	Interpret metacharacters within string
`' '`	Escape all metacharacters within string

Backslash `\`

The backslash character is used as an escape character. It instructs the shell to preserve the literal interpretation of special characters such as space, tab, and `$`. For example, if you have a file with spaces in its name, you can use backslashes followed by a space to handle those spaces literally:

```
touch file\ with\ space.txt
```

Double quotes `" "`

When a string is enclosed in double quotes, most characters are interpreted literally, but metacharacters are interpreted according to their special meaning. For example, you can access variable values using the dollar `$` character:

```
$ echo "Hello $USER"
Hello <username>
```

Single quotes `' '`

When a string is enclosed in single quotes, all characters and metacharacters enclosed within the quotes are interpreted literally. Single quotes alter the above example to produce the following output:

```
$ echo 'Hello $USER'
Hello $USER
```

Notice that instead of printing the value of `$USER`, single quotes cause the terminal to print the string `"$USER"`.

Input/Output redirection

Symbol	Meaning
`>`	Redirect output to file, overwrite
`>>`	Redirect output to file, append

Symbol	Meaning
`2>`	Redirect standard error to file, overwrite
`2>>`	Redirect standard error to file, append
`<`	Redirect file contents to standard input

Input/output (IO) redirection is the process of directing the flow of data between a program and its input/output sources.

By default, a program reads input from *standard input*, the keyboard, and writes output to *standard output*, the terminal. However, using IO redirection, you can redirect a program's input or output to or from a file or another program.

Redirect output `>`

This symbol is used to redirect the standard output of a command to a specified file.

> `ls > files.txt` will create a file called `files.txt` if it doesn't exist, and write the output of the `ls` command to it.

> Warning: When the file already exists, the output overwrites all of the file's contents!

Redirect and append output `>>`

This notation is used to redirect and append the output of a command to the end of a file. For example,

> `ls >> files.txt` appends the output of the `ls` command to the end of file `files.txt`, and preserves any content that already existed in the file.

Redirect standard output `2>`

This notation is used to redirect the standard error output of a command to a file. For example, if you run the ls command on a non-existing directory as follows,

> `ls non-existent-directory 2> error.txt` the shell will create a file called `error.txt` if it doesn't exist, and redirect the error output of the `ls` command to the file.

> Warning: When the file already exists, the error message overwrites all of the file's contents!

Append standard error `2>>`

This symbol redirects the standard error output of a command and appends the error message to the end of a file without overwriting its contents.

> `ls non-existent-directory 2>> error.txt` will append the error output of the `ls` command to the end of the `error.txt` file.

Redirect input `<`

This symbol is used to redirect the standard input of a command from a file or another command. For example,

> `sort < data.txt` will `sort` the contents of the `data.txt` file.

Command Substitution

Command substitution allows you to run command and use its output as a component of another command's argument. Command substitution is denoted by enclosing a command in either backticks (`**command**`) or using the `$()` syntax. When the encapsulate command is executed, its output is substituted in place, and it can be used as an argument within another command. This is particularly useful for automating tasks that require the use of a command's output as input for another command.

For example, you could store the path to your current directory in a variable by applying command substitution on the `pwd` command, then move to another directory, and finally return to your original directory by invoking the `cd` command on the variable you stored, as follows:

```
$ here=$(pwd)
$ cd path_to_some_other_directory
$ cd $here
```

Command Line Arguments

Command line arguments are additional inputs that can be passed to a program when the program is run from a command line interface. These arguments are specified after the name of the program, and they can be used to modify the behavior of the program, provide input data, or provide output locations. Command line arguments are used to pass arguments to a shell script.

For example, the following command provides two arguments, arg1, and arg2, that can be accessed from within your Bash script:

```
$ ./MyBashScript.sh arg1 arg2
```

Summary

In this reading, you learned that:

- Metacharacters such as `#`, `;`, `*`, and `?` are characters that the shell interprets with special meanings
- Quoting allows you to ensure any special characters, spaces, or other metacharacters are interpreted literally by the shell
- Input/output redirection redirects a program's input or output to/from a file
- Command substitution allows you to use the output of a command as an argument for another command
- Command line arguments can be used to pass information to a shell script