# Module 1 Cheat Sheet - Introduction to Linux

### Linux terminal tips

*Use****tab completion****to autocomplete pathnames and command names.*

*Scroll through your****command history****with the Up Arrow and Down Arrow keys to find and re-run a command you already used.*

### Getting information

##### Display the reference manual for the ls command:

1. 1
2. man ls

Copied!

### Browsing and navigating directories

#### Special paths

| **Symbol** | **Represents path to** |
| --- | --- |
| ~ | home directory |
| / | root directory |
| . | present working directory |
| .. | parent of present working directory |

##### List files and directories in the current directory:

1. 1
2. ls

Copied!

##### List files and directories in a directory:

1. 1
2. ls path\_to\_directory

Copied!

##### Return path to present working directory:

1. 1
2. pwd

Copied!

##### Change the current directory to a subdirectory:

1. 1
2. cd child\_directory\_name

Copied!

***Tip****: Because cd looks in the current directory for child\_directory\_name, you don’t need to type the entire path.*

##### Change the current directory:

***Up one level:****cd ../*

***To home:****cd ~ or cd*

***To some other directory:****cd path\_to\_directory*

##### Change the current directory to another one at the same level:

*Suppose you have two sibling directories within the same directory, dir\_1 and dir\_2, and your present working directory is dir\_1. To switch to dir\_2, enter:*

*cd ../dir\_2*

***Tip****: Using .., you don't need to know the path to the parent directory to switch to a sibling.*

##### Change the current directory back to the directory you were in previously:

*cd -*

### Upgrading and installing packages

##### Fetch and display up-to-date information about all upgradable packages:

1. 1
2. sudo apt update

Copied!

##### Upgrade to the latest supported version of nano:

1. 1
2. sudo apt upgrade nano

Copied!

##### Install Vim:

1. 1
2. sudo apt install vim

Copied!

### Creating and editing files

##### Create a new text file and open it with nano:

1. 1
2. nano file\_name.txt

Copied!

***Tip****: If the file already exists, nano simply opens it for editing.*

## Authors

# Getting Help for Linux Commands

There is huge value in spending time exploring and experimenting with commands, but there are many things you can't figure out just by experimenting. You need to see what's out there, see what's possible, and learn where to look to get answers. Let's take a look at some great ways to get the information you need to help you make progress.

In this reading, you may see links to external sources. You can open them by right-clicking and pressing "Open in new tab."

## 1. Use the built-in man command

The man command, which stands for "manual", provides the standard way to access help for Unix-like commands from the command prompt. It has been in development since 1971.

You can get a listing of all the commands on your system that have a manual page by entering:

1. 1
2. man -k .

Copied!

The resulting list includes a brief description of what each command does.

To see the man page for a command, simply enter:

1. 1
2. man command\_name

Copied!

All man pages are divided into several sections, including:

##### NAME

*The name of the command or feature and a brief description of what it does.*

##### SYNOPSIS

*A summary of the command syntax, including any options and arguments that can be used.*

##### DESCRIPTION

*A more detailed description of the command, including its function and behavior.*

##### OPTIONS

*All the available options and arguments that can be used with the command.*

##### EXAMPLES

*Some examples of how to use the command.*

##### SEE ALSO

*Related commands and documentation that may be helpful.*

You may also see other sections, including: EXIT STATUS, RETURN VALUE, ENVIRONMENT, BUGS, FILES, AUTHOR, REPORTING BUGS, HISTORY, and COPYRIGHT.

## 2. Install and use the tldr command

Similar to man pages, [TLDR Pages](https://tldr.sh/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMLX0117ENSkillsNetwork860-2023-01-01) is a free and open-source collaborative documentation effort. The goal is to create documentation that is more accessible than the traditional man pages, which tend to be quite verbose.

TLDR Pages, short for "Too Long; Didn't Read" and also known simply as tldr, provide examples for common use cases of various commands. The format of TLDR pages is similar to that of a cheatsheet.

You can install a command-line tool to access TLDR Pages from your terminal. Install it using the following command:

1. 1
2. npm install -g tldr

Copied!

Once you've installed the tool, you can use the tldr command to easily access the TLDR page of a command.

1. 1
2. tldr command\_name

Copied!

The tool will display a short, easy-to-understand summary of the command along with some examples of how to use it.

## 3. Search Stack Overflow

Stack Overflow is a popular community-driven question and answer platform for programmers, developers, and system administrators. It has a vast repository of questions and answers related to various programming languages, tools, and operating systems, including Linux.

To search for information about commands on Stack Overflow, you can use the search bar on the homepage and enter the name of the command you're looking for, along with any specific keywords or parameters. You can also refine your search by adding relevant tags, such as "linux" or "command-line".

Once you've entered your search query, Stack Overflow will display a list of relevant questions and answers that match your query. You can browse through the results to find the information you need, and even post your own question if you can't find an answer to your specific query.

When searching for information about commands on Stack Overflow, it's important to check the date of the answers to ensure that the information is still current and relevant. You should also read through the comments and discussion threads to get a better understanding of the context and any potential issues or limitations related to the command you're researching.

*Newest questions on Stack Overflow tagged "Linux":*[*https://stackoverflow.com/questions/tagged/linux*](https://stackoverflow.com/questions/tagged/linux)

## 4. Search Stack Exchange

Stack Exchange is a network of question and answer communities, similar to Stack Overflow, but covering a broader range of topics beyond just programming. There are several Stack Exchange communities that specialize in topics related to Linux and open source software, such as Unix & Linux, Ask Ubuntu, and Server Fault.

Visit the relevant community to search for information on Stack Exchange. Like Stack Overflow, you can use the search bar to enter the name of the command you're looking for, along with any keywords or parameters.

*Unix and Linux community on Stack Exchange:*[*https://unix.stackexchange.com/*](https://unix.stackexchange.com/)

## 5. Just google it!

Google is a powerful tool that can provide you the answer to almost any question. Learn how to enter the right queries and filter your results, such as by including "Wikipedia", "Stack Overflow", or "Linux" as part of your search. However, use at your own risk. Never blindly trust what you find on the web - there's a lot of noise out there!

## 6. Use the cheat sheets from this course

Throughout this course, you will encounter "cheat sheets" that condense the information you've learned into easy-to-reference guides. They are great for reviewing the material you've learned and can also help you out with your graded assignments.

## 7. Refer to Wikipedia's list of Unix commands:

Finally, Wikipedia maintains a list of commands that can be found on Unix operating systems, along with a short description. You can check the page to quickly reference a Unix command: <https://en.wikipedia.org/wiki/List_of_Unix_commands>

# Security: Managing File Permisions and Ownership

## Learning Objectives

After completing this reading, you will be able to:

* Explain file ownership and permissions
* View file and directory permissions
* Make a file private

### Why do we need file permissions and ownership?

Linux is a multi-user operating system. This means that by default, other users will be able to view any files you store on the system. However, you may have some files - such as your personal tax documents or your employer's intellectual property documents - that are private or confidential. How can you protect these sensitive documents from being viewed or modified by others?

### File ownership and permissions

There are three possible levels of file ownership in Linux: user, group, and other.  
Whoever creates a file, namely the **user** at the time of creation, becomes the owner of that file by default. A **group** of users can also share ownership of a file. The **other** category essentially refers anyone in the universe with access to your Linux machine - careful when assigning ownership permission to this level!

Only an official owner of a file is allowed to change its permissions. This means that only owners can decide who can read the file, write to it, or execute it.

### Viewing file permissions

Let's say you've entered the following lines of code:

1. 1
2. 2
3. 3
4. 4
5. 5
6. $ echo "Who can read this file?" > my\_new\_file
7. $ more my\_new\_file
8. Who can read this file?
9. $ ls -l my\_new\_file
10. -rw-r--r-- 1 theia users 25 Dec 22 17:47 x

Copied!

Here we've echoed the string "Who can read this file?" into a new file called my\_new\_file. The next line uses the more command to print the contents of the new file. Finally, the ls command with the -l option displays the file's (default) permissions: rw-r--r--

The first three characters (rw-) define the **user** permissions, the next three (r--) the **group** pemissions, and the final three (r--) the **other** permissions.

So you, being the user, have the permission rw-, which means you have read and write permissions by default, but do not have execution permissions. Otherwise there would be an x in place of the last -.

Thus by looking at the entire line, rw-r--r--, you can see that anyone can read the file, nobody can execute it, and you are the only user that can write to it.

***Note:****The - at the very beginning of the line in the terminal means that the permissions are referring to a file. If you were getting the permissions to a directory, you would see a d in the front for "directory".*

### Directory permissions

The permissions for directories are similar but distinct for files. Though directories use the same rwx format, the symbols have slightly different meanings.

The following table illustrates the meanings of each permission for directories:

| **Directory Permission** | **Permissible action(s)** |
| --- | --- |
| r | List directory contents using ls command |
| w | Add or remove files or directories |
| x | Enter directory using cd command |

Setting appropriate permissions on directories is a best practice for both security and stability reasons. Though this reading focuses on security, you will learn more about other reasons for setting file permissions and ownership later in this course.

### Making a file private

You can revoke read permissions from your group and all other users by using the chmod command. Ensure successful modification by using the ls -l command again:

1. 1
2. 2
3. 3
4. chmod go-r my\_new\_file
5. ls -l my\_new\_file
6. -rw------- 1 theia users 24 Dec 22 18:49 my\_new\_file

Copied!

In the chmod command, go-r is the permission change to be applied, which in this case means removing for the group (g) and others (o) the read (r) permission. The chmod command can be used with both files and directories.

### Executable files - looking ahead

You've learned what it means to read or write to a file, but what does it mean to have permissions to **execute** a file in Linux?

A Linux file is executable if it contains instructions that can be directly interpreted by the operating system. Basically, an exectuable file is a ready-to-run program. They're also referred to as **binaries** or **executables**.

In this course, you will become very familiar with a particular kind of executable called a **script**, which is a program written in a scripting language. You'll learn all about **shell scripting**, or more specifically **Bash scripting**, which is writing scripts in Bash (born-again shell), a very popular shell scripting language. A shell script is a plain text file that can be interpreted by a shell.

Formally speaking, for a text file to be considered an executable shell script for a given user, it needs to have two things:

1. Execute permissions set for that user
2. A directive, called a "shebang", in its first line to declare itself to the operating system as a binary

All of this will become more clear to you soon when we get to the topic of shell scripting.

## Summary

In this reading, you learned that:

* There are three possible levels of file ownership in Linux - user, group, and other - which determine who can read, write to, and execute a file
* You can use the ls -l command to view file and directory permissions
* You can change permissions on a file by using the chmod command

# A Brief Introduction to Networking

Here we introduce some key concepts related to computer networking. This is an optional but recommended reading which is designed to help you build context for what you will be learning later about networking and informational commands.

## Learning Objectives

After completing this reading, you will be able to:

* Describe computer networks, network resources, and network nodes
* Explain hosts, clients, and servers
* Describe packets and pings
* Explain URLs and IP addresses

## Computer Networks

A **computer network** is a set of computers that are able to communicate with each other and share **resources** provided by **network nodes**.

*Examples of computer networks include Local Area Networks (LANs), Wide Area Networks (WANs), and the entire Internet. The Internet, or World Wide Web, is essentially a giant network of computer networks.*

A network **resource** is any object, such as a file or document, which can be identified by the network.

*An object is identifiable if it can be assigned a unique name and address that the network can use to identify and access it.*

A **network node** is any device that participates in a network.

*A network can include any device which is not necessarily a computer but is part of the network’s infrastructure. Examples of network nodes include modems, network switches, hubs, and wifi hotspots.*

## Hosts, Clients, and Servers

A **host** is a special type of node in a computer network - it is a computer that can function as a **server** or a **client** on a network.

A **server** is a host computer that is able to accept a connection from a **client** host and fulfill certain resource requests made by the client.

Many hosts can perform either role, acting as both client and server.

## Packets and Pings

A **network packet** is a formatted chunk of data that can be transmitted over a network.

Today's computer networks typically use communication protocols that are based on such packets of information. Every packet consists of two types of data: 1. the **control information**, and 2. the **payload**. The control information is data about how and where to deliver the payload, such as the source and destination network addresses, while the payload is the message being sent.

The ping command works by sending special 'echo request' packets to a host and waiting for a response from the host.

*ping is a utility available on most operating systems that have networking capability. Linux has its own implementation of the ping command that's used to test and troubleshoot connectivity to other network hosts.*

## URLs and IP Adresses

IP stands for "Internet Protocol" which defines the format of data transmitted over the internet or a local network.

An **IP address** is a code used to uniquely identify any host on a network.

*An IP address can be used to establish a connection to a host and exchange packets with it, for example using the ping command. In addition to their payload, IP packets - a type of network packet that conforms to the Internet Protocol - contain the IP addresses of the source and destination hosts.*

A **URL**, more commonly known as a web address, stands for Uniform Resource Locator.

*A URL uniquely identifies a web resource and enables access to that resource. Typically the resource that a URL points to is a web page, but it can also be used for tasks such as transferring files, sending emails, and accessing databases.*

*For example, the URL to the Wikipedia page for URL is https://en.wikipedia.org/wiki/URL. Just like for a typical URL, its format indicates a protocol (https), a hostname (en.wikipedia.org), and a file name (/wiki/URL).*

# Module 2 Cheat Sheet - Introduction to Linux Commands

### Getting information

##### Return your user name:

1. 1
2. whoami

Copied!

##### Return your user and group id:

1. 1
2. id

Copied!

##### Return operating system name, username, and other info:

1. 1
2. uname -a

Copied!

##### Display reference manual for a command:

1. 1
2. man top

Copied!

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

1. 1
2. man -k .

Copied!

##### Get help on a command:

1. 1
2. curl --help

Copied!

##### Return the current date and time:

1. 1
2. date

Copied!

### Navigating and working with directories

##### List files and directories by date, newest to last:

1. 1
2. ls -lrt

Copied!

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

1. 1
2. find -name \'\\*.sh\'

Copied!

##### Return path to present working directory:

1. 1
2. Pwd

Copied!

##### Make a new directory:

1. 1
2. mkdir new\_folder

Copied!

##### Change the current directory:

***Up one level:***

1. 1
2. cd ../

Copied!

***To home:***

1. 1
2. cd ~` or `cd

Copied!

***To some other directory:****cd path\_to\_directory*

##### Remove directory verbosely:

1. 1
2. rmdir temp\_directory -v

Copied!

### Monitoring system performance and status

##### List selection of/all running processes and their PIDs:

1. 1
2. ps

Copied!

1. 1
2. ps -e

Copied!

##### Display resource usage:

1. 1
2. top

Copied!

##### List mounted file systems and usage:

1. 1
2. df

Copied!

### Creating, copying, moving, and deleting files:

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

1. 1
2. touch a\_new\_file.txt

Copied!

##### Copy a file:

1. 1
2. cp file.txt new\_path/new\_name.txt

Copied!

##### Change file name or path:

1. 1
2. mv this\_file.txt that\_path/that\_file.txt

Copied!

##### Remove a file verbosely:

1. 1
2. rm this\_old\_file.txt -v

Copied!

### Working with file permissions

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

1. 1
2. chmod +x my\_script.sh

Copied!

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

1. 1
2. chmod u+x my\_file.txt

Copied!

##### Remove 'read' permissions from group and other users:

1. 1
2. chmod go-r

Copied!

### Displaying file and string contents

##### Display file contents:

1. 1
2. cat my\_shell\_script.sh

Copied!

##### Display file contents page-by-page:

1. 1
2. more ReadMe.txt

Copied!

##### Display first 10 lines of file:

1. 1
2. head -10 data\_table.csv

Copied!

##### Display last 10 lines of file:

1. 1
2. tail -10 data\_table.csv

Copied!

##### Display string or variable value:

1. 1
2. 2
3. echo "I am not a robot"
4. echo "I am $USERNAME"

Copied!

### Basic text wrangling

#### Sorting lines and dropping duplicates:

##### Sort and display lines of file alphanumerically:

1. 1
2. sort text\_file.txt

Copied!

##### *In reverse order:*

1. 1
2. sort -r text\_file.txt

Copied!

##### Drop consecutive duplicated lines and display result:

1. 1
2. uniq list\_with\_duplicated\_lines.txt

Copied!

#### Displaying basic stats:

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

***Lines:***

1. 1
2. wc -l table\_of\_data.csv

Copied!

***Words:***

1. 1
2. wc -w my\_essay.txt

Copied!

***Characters:***

1. 1
2. wc -m some\_document.txt

Copied!

#### 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:

1. 1
2. grep -iw hello a\_bunch\_of\_hellos.txt

Copied!

##### Extract lines containing the pattern "hello" from all files in the current directory ending in .txt:

1. 1
2. grep -l hello \*.txt

Copied!

#### 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:

1. 1
2. paste first\_name.txt last\_name.text phone\_number.txt

Copied!

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

1. 1
2. paste -d "," first\_name.txt last\_name.text phone\_number.txt

Copied!

#### 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:

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

Copied!

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

1. 1
2. cut -b 2-5 my\_text\_file.txt

Copied!

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

1. 1
2. cut -b 10- my\_text\_file.txt

Copied!

### Compression and archiving

##### Archive a set of files:

1. 1
2. tar -cvf my\_archive.tar.gz file1 file2 file3

Copied!

##### Compress a set of files:

1. 1
2. 2
3. zip my\_zipped\_files.zip file1 file2
4. zip my\_zipped\_folders.zip directory1 directory2

Copied!

##### Extract files from a compressed zip archive:

1. 1
2. 2
3. unzip my\_zipped\_file.zip
4. unzip my\_zipped\_file.zip -d extract\_to\_this\_direcory

Copied!

### Working with networking commands

##### Print hostname:

1. 1
2. hostname

Copied!

##### Send packets to URL and print response:

1. 1
2. ping www.google.com

Copied!

##### Display or configure system network interfaces:

1. 1
2. 2
3. ifconfig
4. ip

Copied!

##### Display contents of file at a URL:

1. 1
2. curl <url>

Copied!

##### Download file from a URL:

1. 1
2. wget <url>

Copied!

## Authors

## 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.

1. 1
2. 2
3. 3
4. $ firstname=Jeff
5. $ echo $firstname
6. Jeff

Copied!

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

1. 1
2. $ read lastname

Copied!

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

1. 1
2. 2
3. 3
4. $ read lastname
5. Grossman
6. $

Copied!

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

1. 1
2. 2
3. 3
4. 4
5. $ read lastname
6. Grossman
7. $ echo $lastname
8. Grossman

Copied!

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

1. 1
2. 2
3. $ echo $firstname $lastname
4. Jeff Grossman

Copied!

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.

## 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.

A diagram of a pipe

Description automatically generated  
  
Pipes | use the following format:

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

Copied!

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](https://man7.org/linux/man-pages/man1/sort.1.html?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMLX0117ENSkillsNetwork860-2023-01-01) - sorts the lines of text in a file and displays the result
* [uniq](https://man7.org/linux/man-pages/man1/uniq.1.html?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMLX0117ENSkillsNetwork860-2023-01-01) - 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:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. $ cat pets.txt
10. goldfish
11. dog
12. cat
13. parrot
14. dog
15. goldfish
16. goldfish

Copied!

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

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. $ sort pets.txt
10. cat
11. dog
12. dog
13. goldfish
14. goldfish
15. goldfish
16. parrot

Copied!

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

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

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. $ uniq pets.txt
9. goldfish
10. dog
11. cat
12. parrot
13. dog
14. goldfish

Copied!

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:

1. 1
2. 2
3. 3
4. 4
5. 5
6. $ sort pets.txt | uniq
7. cat
8. dog
9. goldfish
10. parrot

Copied!

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](https://man7.org/linux/man-pages/man1/tr.1p.html?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMLX0117ENSkillsNetwork860-2023-01-01) (translate) - replaces characters in input text

1. 1
2. tr [OPTIONS] [target characters] [replacement characters]

Copied!

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 \_:

1. 1
2. 2
3. $ echo "Linux and shell scripting are awesome\!" | tr "aeiou" "\_"
4. L\_n\_x \_nd sh\_ll scr\_pt\_ng \_r\_ \_w\_s\_m\_!

Copied!

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:

1. 1
2. 2
3. $ echo "Linux and shell scripting are awesome\!" | tr -c "aeiou" "\_"
4. \_i\_u\_\_a\_\_\_\_\_e\_\_\_\_\_\_i\_\_i\_\_\_a\_e\_a\_e\_o\_e\_

Copied!

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

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. $ cat pets.txt | tr "[a-z]" "[A-Z]"
10. GOLDFISH
11. DOG
12. CAT
13. PARROT
14. DOG
15. GOLDFISH
16. GOLDFISH

Copied!

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

1. 1
2. 2
3. 3
4. 4
5. 5
6. $ sort pets.txt | uniq | tr "[a-z]" "[A-Z]"
7. CAT
8. DOG
9. GOLDFISH
10. PARROT

Copied!

## Extracting information from URLs

You can also use curl in combination with the grep command to extract components of URL data by piping the output of curl to grep.

Let's see how you can use this pattern to get the current price of Bitcoin (BTC) in USD.

First, find a public URL API. In this example, you will use one provided by [CoinStats](https://coinstats.app/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMLX0117ENSkillsNetwork860-2023-01-01" \t "_blank).

CoinStats provides a public API with no key required at https://api.coinstats.app/public/v1/coins/bitcoin\?currency\=USD, which returns some JSON about the current BTC price in USD.

You can see what this looks like by entering the above link in your browser.

Entering the following command returns the BTC price data, displayed as a JSON object:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
22. 22
23. 23
24. 24
25. 25
26. 26
27. $ curl -s --location --request GET https://api.coinstats.app/public/v1/coins/bitcoin\?currency\=USD
28. {
29. "coin": {
30. "id": "bitcoin",
31. "icon": "https://static.coinstats.app/coins/Bitcoin6l39t.png",
32. "name": "Bitcoin",
33. "symbol": "BTC",
34. "rank": 1,
35. "price": 57907.78008618953,
36. "priceBtc": 1,
37. "volume": 48430621052.9856,
38. "marketCap": 1093175428640.1146,
39. "availableSupply": 18877868,
40. "totalSupply": 21000000,
41. "priceChange1h": -0.19,
42. "priceChange1d": -0.4,
43. "priceChange1w": -9.36,
44. "websiteUrl": "http://www.bitcoin.org",
45. "twitterUrl": "https://twitter.com/bitcoin",
46. "exp": [
47. "https://blockchair.com/bitcoin/",
48. "https://btc.com/",
49. "https://btc.tokenview.com/"
50. ]
51. }
52. }

Copied!

***Note:****For the purpose of this reading, we've reformatted the output to make it easier to interpret. The actual output is a continuous stream of text.*

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:

1. 1
2. grep -oE "\"price\"\s\*:\s\*[0-9]\*?\.[0-9]\*"

Copied!

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 .

Now that you have the grep statement that you need, you can pipe the BTC data to it using the curl command from above:

1. 1
2. 2
3. 3
4. $ curl -s --location --request GET https://api.coinstats.app/public/v1/coins/bitcoin\?currency\=USD |\
5. grep -oE "\"price\":\s\*[0-9]\*?\.[0-9]\*"
6. "price": 57907.78008618953

Copied!

***Tip:****The backslash \ character used here after the pipe | allows you to write the expression on multiple lines.*

Finally, to get only the value in the price field and drop the "price" label, you can use chaining to pipe the same output to another grep:

1. 1
2. 2
3. 3
4. 4
5. $ curl -s --location --request GET https://api.coinstats.app/public/v1/coins/bitcoin\?currency\=USD |\
6. grep -oE "\"price\":\s\*[0-9]\*?\.[0-9]\*" |\
7. grep -oE "[0-9]\*?\.[0-9]\*"
8. 57907.78008618953

Copied!

This now displays only the numerical price without the label.

# 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.

1. 1
2. 2
3. 3
4. 4
5. #!/bin/bash
6. # This is a comment
7. echo "Hello, world!" # This is another comment

Copied!

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.

1. 1
2. 2
3. 3
4. $ echo "Hello, "; echo "world!"
5. Hello,
6. world!

Copied!

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.

1. 1
2. ls \*.txt

Copied!

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.

1. 1
2. ls file?.txt

Copied!

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 |
| " " | 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:

1. 1
2. touch file\ with\ space.txt

Copied!

### 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:

1. 1
2. 2
3. $ echo "Hello $USER"
4. Hello <username>

Copied!

### 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:

1. 1
2. 2
3. $ echo 'Hello $USER'
4. Hello $USER

Copied!

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 |
| 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:

1. 1
2. 2
3. 3
4. $ here=$(pwd)
5. $ cd path\_to\_some\_other\_directory
6. $ cd $here

Copied!

## 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:

1. 1
2. $ ./MyBashScript.sh arg1 arg2

Copied!

## 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

# Introduction to Advanced Bash Scripting



Estimated time needed: **5** minutes

In the hands-on lab portion of the final project, you will be using more advanced scripting commands and concepts that the course has not covered yet. This reading will familiarize you with these more advanced concepts, so you can complete the lab with confidence.

## Objectives

After completing this reading, you will be able to create Bash scripts that:

* use conditional statements to run a set of commands only if a specified condition is true
* apply logical operators to create true/false comparisons
* perform basic arithmetic calculations
* create list-like arrays and access their elements
* implement for loops to execute operations repeatedly, based on a looping index

## Conditionals

**Conditionals**, or if statements, are a way of telling a script to do something only under a specific condition.

Bash script conditionals use the following if-then-else syntax:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. if [ condition ]
8. then
9. statement\_block\_1
10. else
11. statement\_block\_2
12. fi

Copied!

If the condition is true, then Bash executes the statements in statement\_block\_1 before exiting the conditional block of code. After exiting, it will continue to run any commands after the closing fi.

Alternatively, if the condition is false, Bash instead runs the statements in statement\_block\_2 under the else line, then exits the conditional block and continues to run commands after the closing fi.

***Tips:***

* *You must always put spaces around your condition within the square brackets [ ].*
* *Every if condition block must be paired with a fi to tell Bash where the condition block ends.*
* *The else block is optional but recommended. If the condition evaluates to false without an else block, then nothing happens within the if condition block. Consider options such as echoing a comment in statement\_block\_2 to indicate that the condition was evaluated as false.*

In the following example, the condition is checking whether the number of command-line arguments read by some Bash script, $#, is equal to 2.

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. if [[ $# == 2 ]]
8. then
9. echo "number of arguments is equal to 2"
10. else
11. echo "number of arguments is not equal to 2"
12. fi

Copied!

Notice the use of the double square brackets, which is the syntax required for making integer comparisons in the condition [[ $# == 2 ]].

You can also make string comparisons. For example, assume you have a variable called string\_var that has the value "Yes" assigned to it. Then the following statement evaluates to true:

1. 1
2. `[ $string\_var == "True" ]`

Copied!

Notice you only need single square brackets when making string comparisons.

You can also include multiple conditions to be satified by using the "and" operator && or the "or" operator ||. For example:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. if [ condition1 ] && [ condition2 ]
8. then
9. echo "conditions 1 and 2 are both true"
10. else
11. echo "one or both conditions are false"
12. fi

Copied!

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. if [ condition1 ] || [ condition2 ]
8. then
9. echo "conditions 1 or 2 are true"
10. else
11. echo "both conditions are false"
12. fi

Copied!

## Logical operators

The following logical operators can be used to compare integers within a condition in an if condition block.

==: is equal to

If a variable a has a value of 2, the following condition evaluates to true; otherwise it evalutes to false.

1. 1
2. $a == 2

Copied!

!=: is not equal to

If a variable a has a value different from 2, the following statement evaluates to true. If its value is 2, then it evalutes to false.

1. 1
2. a != 2

Copied!

***Tip****: The ! logical negation operator changes true to false and false to true.*

<=: is less than or equal to

If a variable a has a value of 2, then the following statement evaluates to true:

1. 1
2. a <= 3

Copied!

and the following statement evalutates to false:

1. 1
2. a <= 1

Copied!

Alternatively, you can use the equivalent notation -le in place of <=:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. a=1
10. b=2
11. if [ $a -le $b ]
12. then
13. echo "a is less than or equal to b"
14. else
15. echo "a is not less than or equal to b"
16. fi

Copied!

We've only provided a small sampling of logical operators here. You can explore resources such as the [Advanced Bash-Scripting Guide](https://tldp.org/LDP/abs/html/comparison-ops.html) to find out more.

## Arithmetic calculations

You can perform integer addition, subtraction, multiplication, and division using the notation $(()).  
For example, the following two sets of commands both display the result of adding 3 and 2.

1. 1
2. echo $((3+2))

Copied!

or

1. 1
2. 2
3. 3
4. 4
5. a=3
6. b=2
7. c=$(($a+$b))
8. echo $c

Copied!

Bash natively handles integer arithmetic but does not handle floating-point arithmetic. As a result, it will always truncate the decimal portion of a calculation result.

For example:

1. 1
2. echo $((3/2))

Copied!

prints the truncated integer result, 1, not the floating-point number, 1.5.

The following table summarizes the basic arithmetic operators:

| **Symbol** | **Operation** |
| --- | --- |
| + | addition |
| - | subtraction |
| \* | multiplication |
| / | division |

*Table:****Arithmetic operators***

## Arrays

The **array** is a Bash built-in data structure. An array is a space-delimited list contained in parentheses.ʊTo create an array, declare its name and contents:

1. 1
2. my\_array=(1 2 "three" "four" 5)

Copied!

This statement creates and populates the array my\_array with the items in the parentheses: 1, 2, "three", "four", and 5.

You can also create an empty array by using:

1. 1
2. declare -a empty\_array

Copied!

If you want to add items to your array after creating it, you can add to your array by appending one element at a time:

1. 1
2. 2
3. my\_array+=("six")
4. my\_array+=(7)

Copied!

This adds elements "six" and 7 to the array my\_array.

By using indexing, you can access individual or multiple elements of an array:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. # print the first item of the array:
10. echo ${my\_array[0]}
11. # print the third item of the array:
12. echo ${my\_array[2]}
13. # print all array elements:
14. echo ${my\_array[@]}

Copied!

***Tip:****Note that array indexing starts from 0, not from 1.*

## for loops

You can use a construct called a for loop along with indexing to iterate over all elements of an array.

For example, the following for loops will continue to run over and over again until every element is printed:

1. 1
2. 2
3. 3
4. for item in ${my\_array[@]}; do
5. echo $item
6. done

Copied!

or

1. 1
2. 2
3. 3
4. for i in ${!my\_array[@]}; do
5. echo ${my\_array[$i]}
6. done

Copied!

The for loop requires a ; do component in order to cycle through the loop. Additionally, you need to terminate the for loop block with a done statement.

Another way to implement a for loop when you know how many iterations you want is as follows. For example, the following code prints the number 0 through 6.

1. 1
2. 2
3. 3
4. 4
5. N=6
6. for (( i=0; i<=$N; i++ )) ; do
7. echo $i
8. done

Copied!

You can use for loops to accomplish all sorts of things. For example, you could count the number of items in an array or sum up its elements, as the following Bash script does:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. #!/usr/bin/env bash
17. # initialize array, count, and sum
18. my\_array=(1 2 3)
19. count=0
20. sum=0
21. for i in ${!my\_array[@]}; do
22. # print the ith array element
23. echo ${my\_array[$i]}
24. # increment the count by one
25. count=$(($count+1))
26. # add the current value of the array to the sum
27. sum=$(($sum+${my\_array[$i]}))
28. done
29. echo $count
30. echo $sum

Copied!

Go ahead and try running this script, so you get a sense of how this loop works.

## Summary

In this lab, you learned that:

* Conditional statements can be used to run commands based on whether a specified condition is true
* Logical operators do true/false comparisons
* Arithmetic operators perform basic arithmetic calculations
* You can create list-like arrays and access their individual elements
* for loops execute operations repeatedly, based on a looping index

Congratulations! You are now ready to practice your newly acquired knowledge in the following hands-on lab.

 +-

Play

00:00

06:06

Mute

Settings



# Module 3 Cheat Sheet - Introduction to Shell Scripting

### Bash shebang

1. 1
2. #!/bin/bash

Copied!

### Get the path to a command

1. 1
2. which bash

Copied!

### Pipes, filters, and chaining

##### Chain filter commands together using the pipe operator:

1. 1
2. ls | sort -r

Copied!

##### Pipe the output of manual page for ls to head to display the first 20 lines:

1. 1
2. man ls | head -20

Copied!

##### Use a pipeline to extract a column of names from a csv and drop duplicate names:

1. 1
2. cut -d "," -f1 names.csv | sort | uniq

Copied!

### Working with shell and environment variables:

##### List all shell variables:

1. 1
2. set

Copied!

##### Define a shell variable called my\_planet and assign value Earth to it:

1. 1
2. my\_planet=Earth

Copied!

##### Display value of a shell variable:

1. 1
2. echo $my\_planet

Copied!

##### Reading user input into a shell variable at the command line:

1. 1
2. read first\_name

Copied!

***Tip:****Whatever text string you enter after running this command gets stored as the value of the variable first\_name.*

##### List all environment variables:

1. 1
2. env

Copied!

##### Environment vars: define/extend variable scope to child processes:

1. 1
2. 2
3. export my\_planet
4. export my\_galaxy='Milky Way'

Copied!

### Metacharacters

##### Comments #:

1. 1
2. # The shell will not respond to this message

Copied!

##### Command separator ;:

1. 1
2. echo 'here are some files and folders'; ls

Copied!

##### File name expansion wildcard \*:

1. 1
2. ls \*.json

Copied!

##### Single character wildcard ?:

1. 1
2. ls file\_2021-06-??.json

Copied!

### Quoting

##### Single quotes '' - interpret literally:

1. 1
2. echo 'My home directory can be accessed by entering: echo $HOME'

Copied!

##### Double quotes "" - interpret literally, but evaluate metacharacters:

1. 1
2. echo "My home directory is $HOME"

Copied!

##### Backslash \ - escape metacharacter interpretation:

1. 1
2. echo "This dollar sign should render: \$"

Copied!

### I/O Redirection

##### Redirect output to file and overwrite any existing content:

1. 1
2. echo 'Write this text to file x' > x

Copied!

##### Append output to file:

1. 1
2. echo 'Add this line to file x' >> x

Copied!

##### Redirect standard error to file:

1. 1
2. bad\_command\_1 2> error.log

Copied!

##### Append standard error to file:

1. 1
2. bad\_command\_2 2>> error.log

Copied!

##### Redirect file contents to standard input:

1. 1
2. $ tr “[a-z]” “[A-Z]” < a\_text\_file.txt

Copied!

##### The input redirection above is equivalent to:

1. 1
2. $cat a\_text\_file.txt | tr “[a-z]” “[A-Z]”

Copied!

### Command Substitution

##### Capture output of a command and echo its value:

1. 1
2. 2
3. THE\_PRESENT=$(date)
4. echo "There is no time like $THE\_PRESENT"

Copied!

##### Capture output of a command and echo its value:

1. 1
2. echo "There is no time like $(date)"

Copied!

### Command line arguments

1. 1
2. ./My\_Bash\_Script.sh arg1 arg2 arg3

Copied!

### Batch vs. concurrent modes

##### Run commands sequentially:

1. 1
2. start=$(date); ./MyBigScript.sh ; end=$(date)

Copied!

##### Run commands in parallel:

1. 1
2. ./ETL\_chunk\_one\_on\_these\_nodes.sh & ./ETL\_chunk\_two\_on\_those\_nodes.sh

Copied!

### Scheduling jobs with cron

##### Open crontab editor:

1. 1
2. crontab -e

Copied!

##### Job scheduling syntax:

1. 1
2. m h dom mon dow command

Copied!

*(minute, hour, day of month, month, day of week)*

***Tip:****You can use the \* wildcard to mean "any".*

##### Append the date/time to a file every Sunday at 6:15 pm:

1. 1
2. 15 18 \* \* 0 date >> sundays.txt

Copied!

##### Run a shell script on the first minute of the first day of each month:

1. 1
2. 1 0 1 \* \* ./My\_Shell\_Script.sh

Copied!

##### Back up your home directory every Monday at 3:00 am:

1. 1
2. 0 3 \* \* 1 tar -cvf my\_backup\_path\my\_archive.tar.gz $HOME\

Copied!

##### Deploy your cron job:

*Close the crontab editor and save the file.*

##### List all cron jobs:

1. 1
2. crontab -l

Copied!

### Conditionals

##### if-then-else syntax:

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. if [[ $# == 2 ]]
8. then
9. echo "number of arguments is equal to 2"
10. else
11. echo "number of arguments is not equal to 2"
12. fi

Copied!

##### 'and' operator &&:

1. 1
2. if [ condition1 ] && [ condition2 ]

Copied!

##### 'or' operator ||:

1. 1
2. if [ condition1 ] || [ condition2 ]

Copied!

### Logical operators

| **Operator** | **Definition** |
| --- | --- |
| == | is equal to |
| != | is not equal to |
| < | is less than |
| > | is greater than |
| <= | is less than or equal to |
| >= | is greater than or equal to |

### Arithmetic calculations

##### Integer arithmetic notation:

1. 1
2. $(())

Copied!

##### Basic arithmetic operators:

| **Symbol** | **Operation** |
| --- | --- |
| + | addition |
| - | subtraction |
| \* | multiplication |
| / | division |

##### Display the result of adding 3 and 2:

1. 1
2. echo $((3+2))

Copied!

##### Negate a number:

1. 1
2. echo $((-1\*-2))

Copied!

### Arrays

##### Declare an array that contains items 1, 2, "three", "four", and 5:

1. 1
2. my\_array=(1 2 "three" "four" 5)

Copied!

##### Add an item to your array:

1. 1
2. 2
3. my\_array+="six"
4. my\_array+=7

Copied!

##### Declare an array and load it with lines of text from a file:

1. 1
2. my\_array=($(echo $(cat column.txt)))

Copied!

### for loops

##### Use a for loop to iterate over values from 1 to 5:

1. 1
2. 2
3. 3
4. for i in {0..5}; do
5. echo "this is iteration number $i"
6. done

Copied!

##### Use a for loop to print all items in an array:

1. 1
2. 2
3. 3
4. for item in ${my\_array[@]}; do
5. echo $item
6. done

Copied!

##### Use array indexing within a for loop, assuming the array has seven elements:

1. 1
2. 2
3. 3
4. for i in {0..6}; do
5. echo ${my\_array[$i]}
6. done

Copied!

## Authors

Jeff Grossman  
Sam Propupchuk

### Other Contributors

Rav Ahuja

## Change Log

| **Date (YYYY-MM-DD)** | **Version** | **Changed By** | **Change Description** |
| --- | --- | --- | --- |
| 2023-06-07 | 2.0 | Jeff Grossman | Added advanced scripting examples |
| 2023-05-17 | 1.3 | Nick Yi | Added content |
| 2023-05-09 | 1.2 | Nick Yi | Add code blocks, update title |
| 2023-04-26 | 1.1 | Nick Yi | ID Review |
| 2023-02-14 | 1.0 | Jeff Grossman | Update to reflect module content |

© Copyright IBM Corporation 2023. All rights reserved.