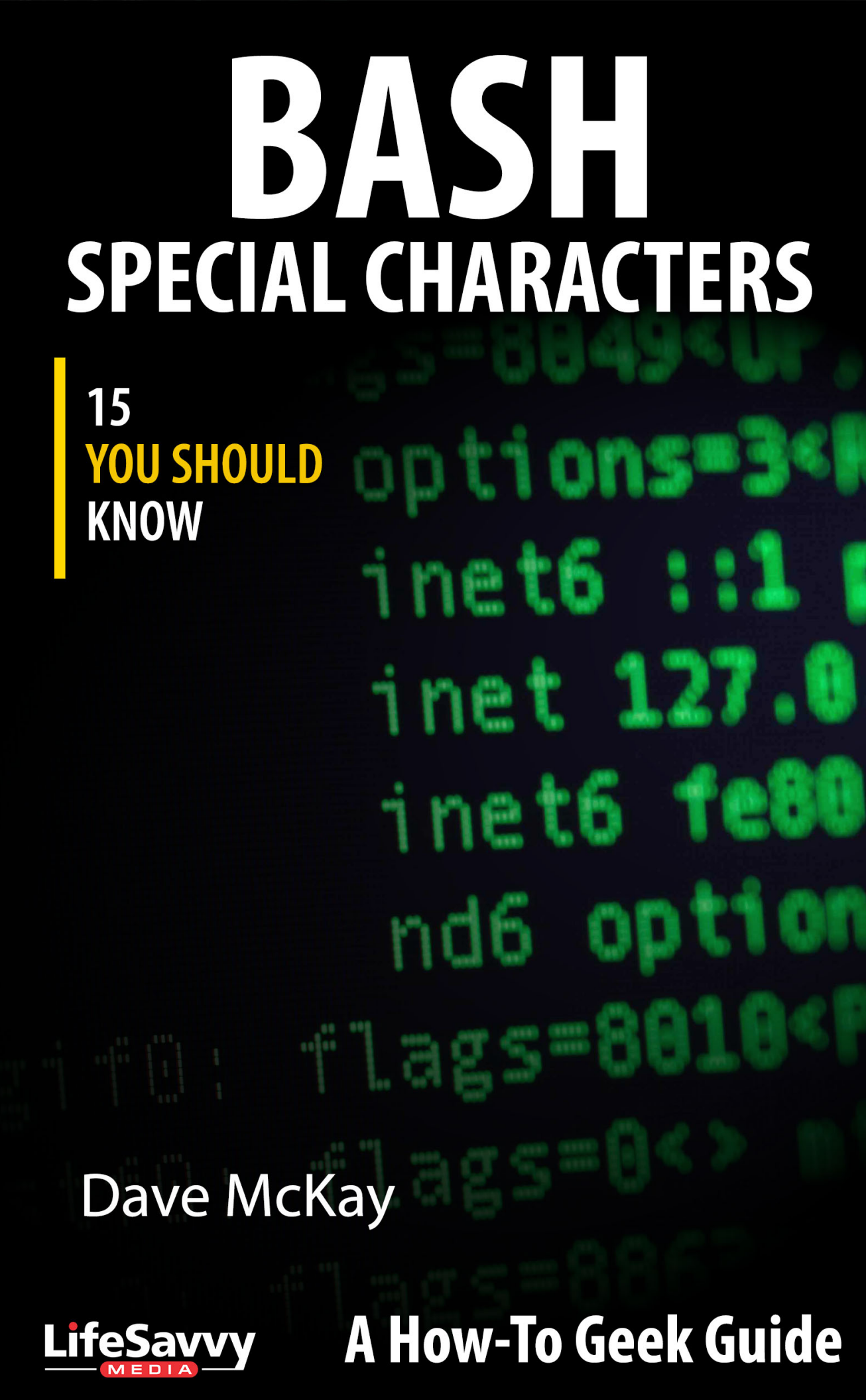
BASH
SPECIAL CHARACTERS

15 YOU SHOULD KNOW



Dave McKay

LifeSavvy MEDIA

A How-To Geek Guide

Bash Special Characters

15 Special Characters You Need to Know

Dave McKay

Cover Photo by Malll Themd/Shutterstock

LifeSavvy MEDIA

Contents

What Are Special Characters?

If you want to master the Bash shell on Linux, macOS, or another UNIX-like system, special characters (like ~, *, |, and >) are critical. We'll help you unravel these cryptic Linux command sequences and become a hero of hieroglyphics.

There are a set of characters the Bash shell treats in two different ways. When you type them at the shell, they act as instructions or commands and tell the shell to perform a certain function. Think of them as single-character commands.

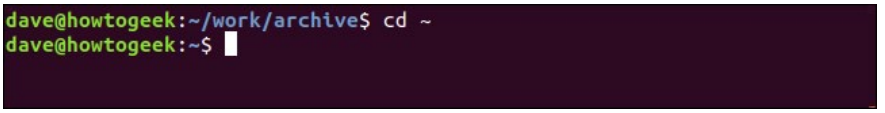
Sometimes, you just want to print a character and don't need it to act as a magic symbol. There's a way you can use a character to represent itself rather than its special function.

We'll show you which characters are "special" or "meta-" characters, as well as how you can use them functionally and literally.

~ Home Directory

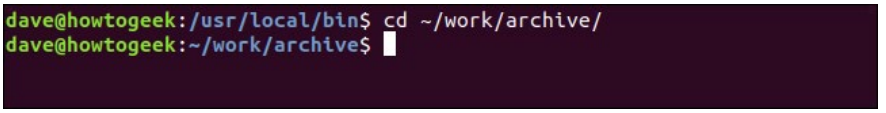
The tilde (~) is shorthand for your home directory. It means you don't have to type the full path to your home directory in commands. Wherever you are in the filesystem, you can use this command to go to your home directory:

```
cd ~
```



You can also use this command with relative paths. For example, if you're somewhere in the file system that's not under your home folder and want to change to the `archive` directory in your `work` directory, use the tilde to do it:

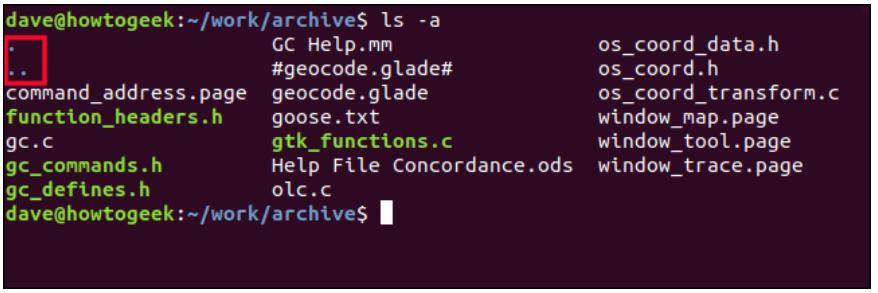
```
cd ~/work/archive
```



. Current Directory

A period (.) represents the current directory. You see it in directory listings if you use the `-a` (all) option with `ls`.

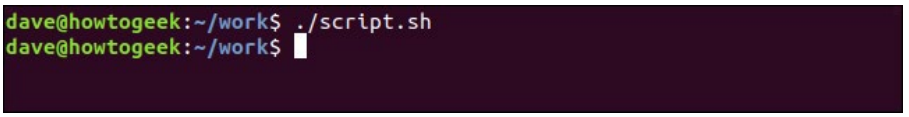
```
ls -a
```



You can also use the period in commands to represent the path to your current directory. For example, if you want to run a script from the current directory, you would call it like this:

```
./script.sh
```

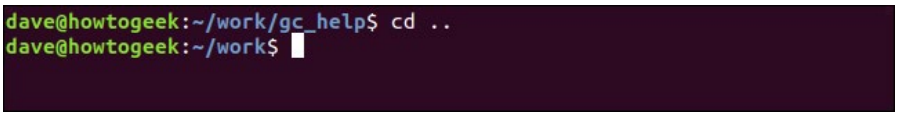
This tells Bash to look in the current directory for the `script.sh` file. This way, it won't search the directories in your path for matching executable or script.



.. Parent Directory

The double period or "double dot" (..) represents the parent directory of your current one. You can use this to move up one level in the directory tree.

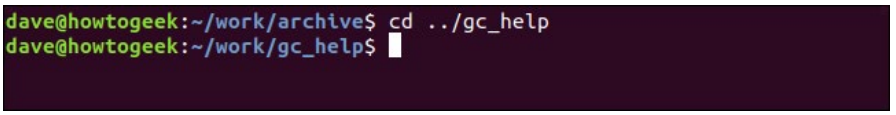
```
cd ..
```



You can also use this command with relative paths---for example, if you want to go up one level in the directory tree, and then enter another directory at that level.

You can also use this technique to move quickly to a directory at the same level in the directory tree as your current one. You hop up one level, and then back down one into a different directory.

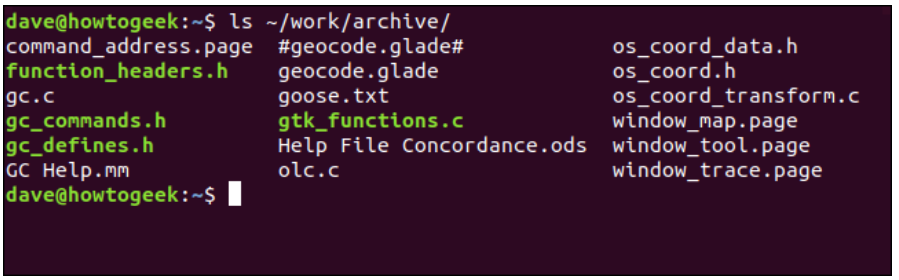
```
cd ../gc_help
```



/ Path Directory Separator

You can use a forward-slash (/)---often just called a slash---to separate the directories in a pathname.

```
ls ~/work/archive
```



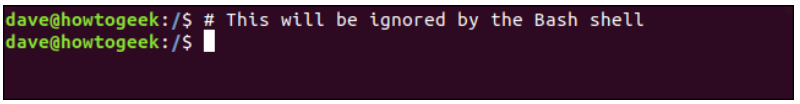
One forward-slash represents the shortest possible directory path. Because everything in the Linux directory tree starts at the root directory, you can use this command to move to the root directory quickly:

```
cd /
```

Comment or Trim Strings

Most often, you use the hash or number sign (#) to tell the shell what follows is a comment, and it should not act on it. You can use it in shell scripts and---less usefully---on the command line.

```
# This will be ignored by the Bash shell
```

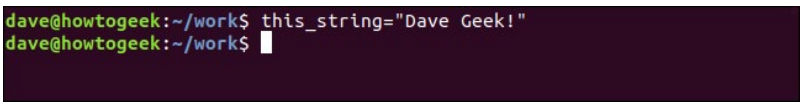


It isn't truly ignored, however, because it's added to your command history.

You can also use the hash to trim a string variable and remove some text from the beginning. This command creates a string variable called `this_string`.

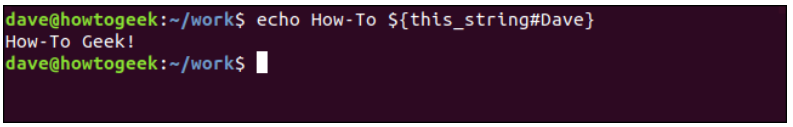
In this example, we assign the text "Dave Geek!" to the variable.

```
this_string="Dave Geek!"
```



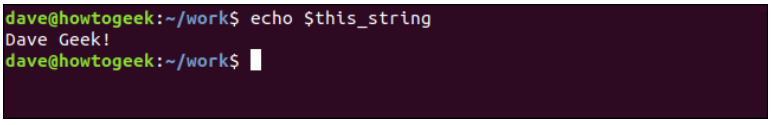
This command uses `echo` to print the words "How-To" to the terminal window. It retrieves the value stored in the string variable via a parameter expansion. Because we append the hash and the text "Dave," it trims off that portion of the string before it's passed to `echo`.

```
echo How-To ${this_string#Dave}
```



This doesn't change the value stored in the string variable; it only affects what's sent to `echo`. We can use `echo` to print the value of the string variable once more and check this:

```
echo $this_string
```



? Single Character Wildcard

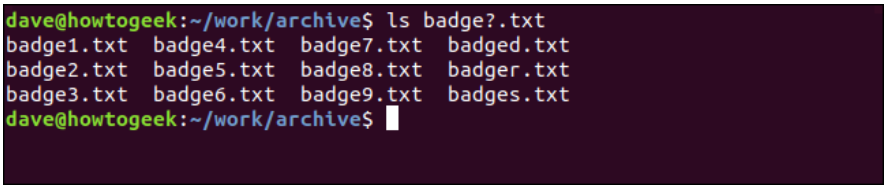
Bash shell supports three wildcards, one of which is the question mark (?). You use wildcards to replace characters in filename templates. A filename that contains a wildcard forms a template that matches a range of filenames, rather than just one.

The question mark wildcard represents *exactly one character*. Consider the following filename template:

```
ls badge?.txt
```

This translates as "list any file with a name that starts with 'badge' and is followed by any single character before the filename extension."

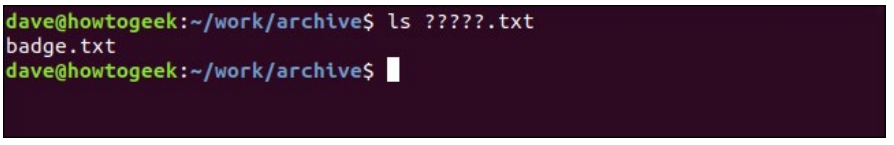
It matches the following files. Note that some have numbers and some have letters after the "badge" portion of the filename. The question mark wildcard will match both letters and numbers.



That filename template doesn't match "badge.txt," though, because the filename doesn't have a single character between "badge" and the file extension. The question mark wildcard must match a corresponding character in the filename.

You can also use the question mark to find all files with a specific number of characters in the filenames. This lists all text files that contain exactly five characters in the filename:

```
ls ?????.txt
```



* Character Sequence Wildcard

You can use the asterisk (*) wildcard to stand for any *sequence* of characters, including *no characters*. Consider the following filename template:

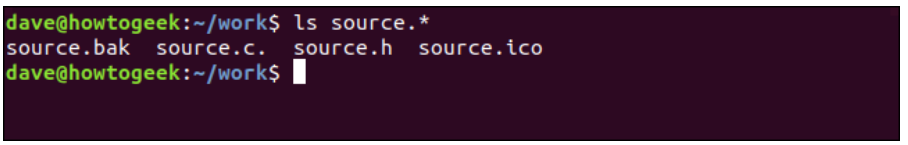
```
ls badge*
```

This matches all of the following:

It matches "badge.txt" because the wildcard represents any sequence of characters or no characters.

This command matches all files called "source," regardless of the file extension.

```
ls source.*
```



[] Character Set Wildcard

As covered above, you use the question mark to represent any single character and the asterisk to represent any sequence of characters (including no characters).

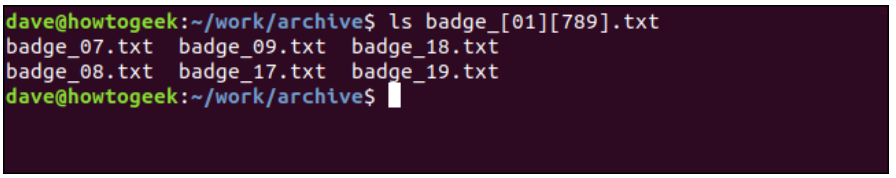
You can form a wildcard with the square brackets ([]) and the characters they contain. The relevant character in the filename must then match at least one of the characters in the wildcard character set.

In this example, the command translates to: "any file with a ".png" extension, a filename beginning with "pipes_0," and in which the next character is *either* 2, 4, or 6."

```
ls badge_0[246].txt
```

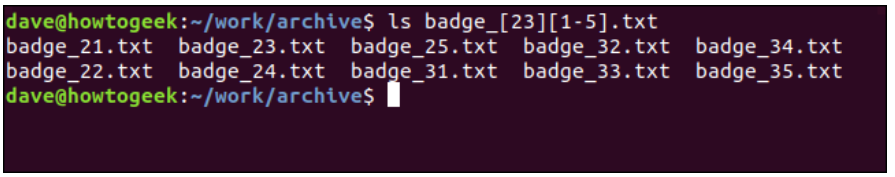
You can use more than one set of brackets per filename template:

```
ls badge_[01][789].txt
```



You can also include ranges in the character set. The following command selects files with the numbers 21 to 25, and 31 to 35 in the filename.

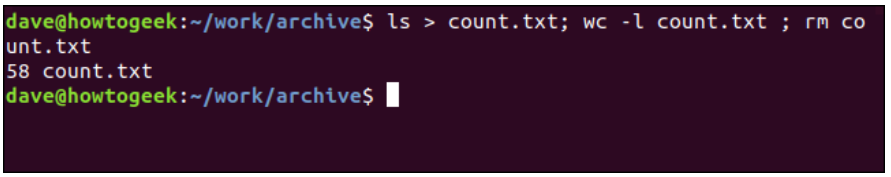
```
ls badge_[23][1-5].txt
```



; Shell Command Separator

You can type as many commands as you like on the command line, as long as you separate each of them with a semicolon (;). We'll do this in the following example:

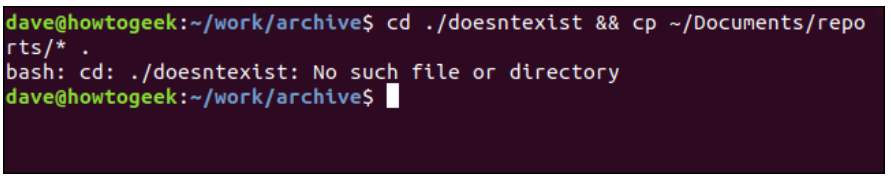
```
ls > count.txt; wc -l count.txt; rm count.txt
```



Note that the second command runs even if the first fails, the third runs even if the second fails, and so on.

If you want to stop the sequence of execution if one command fails, use a double ampersand (&&) instead of a semicolon:

```
cd ./doesntexist && cp ~/Documents/reports/* .
```



& Background Process

After you type a command in a terminal window and it completes, you return to the command prompt. Normally, this only takes a moment or two. But if you launch another application, such as `gedit`, you cannot use your terminal window until you close the application.

You can, however, launch an application as a background process and continue to use the terminal window. To do this, just add an ampersand to the command line:

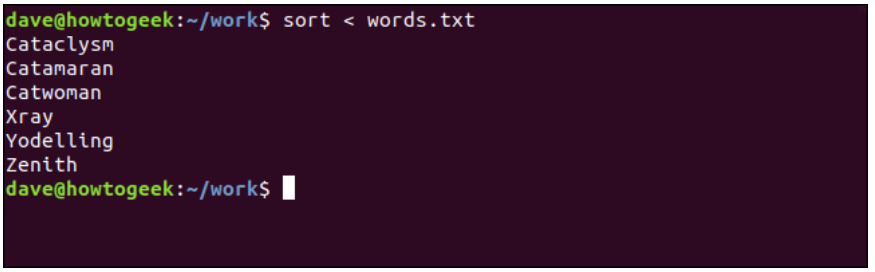
```
gedit command_address.page &
```

Bash shows you the process ID of what launched, and then returns you to the command line. You can then continue to use your terminal window.

< Input Redirection

Many Linux commands accept a file as a parameter and take their data from that file. Most of these commands can also take input from a stream. To create a stream, you use the left-angle bracket (<), as shown in the following example, to redirect a file into a command:

```
sort < words.txt
```

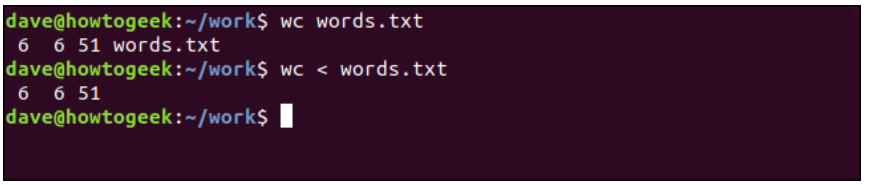


When a command has input redirected into it, it might behave differently than when it reads from a named file.

If we use `wc` to count the words, lines, and characters in a file, it prints the values, and then the filename. If we redirect the contents of the file to `wc`, it prints the same numeric values but doesn't know the name of the file from which the data came. It cannot print a filename.

Here are some examples of how you can use `wc`:

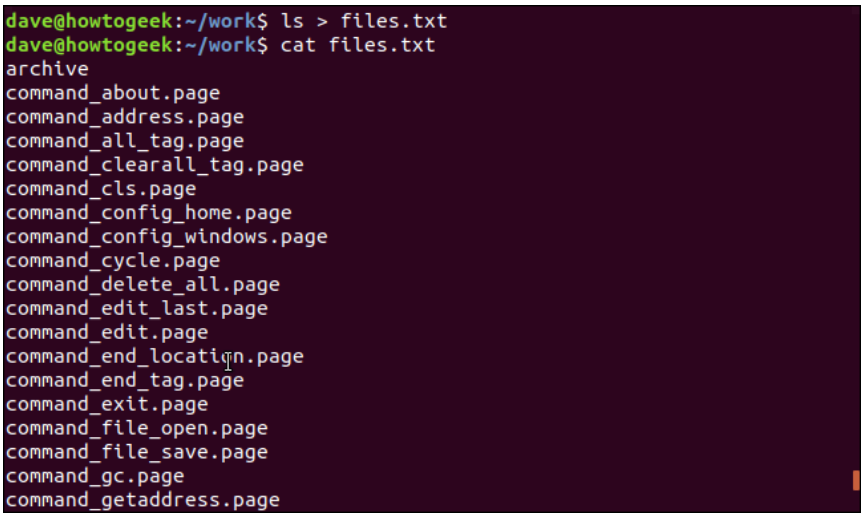
```
wc words.txt
wc < words.txt
```



> Output Redirection

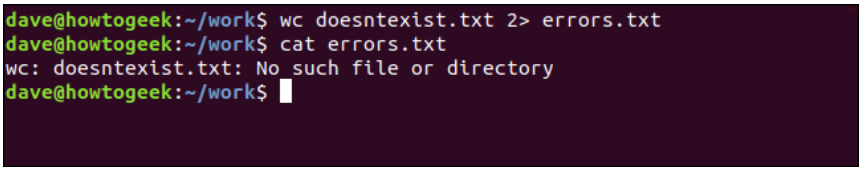
You can use the right-angle bracket (>) to redirect the output from a command (typically, into a file); here's an example:

```
ls > files.txt
cat files.txt
```



Output redirection can also redirect error messages if you use a digit (2, in our example) with >. Here's how to do it:

```
wc doesntexist.txt 2> errors.txt
cat errors.txt
```



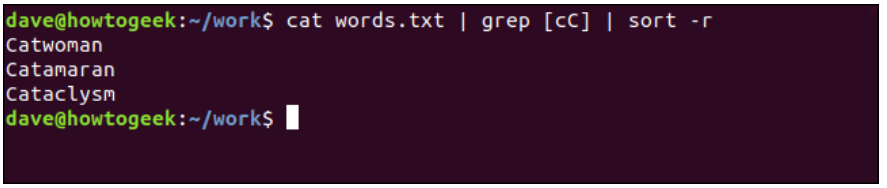
| Pipe

A "pipe" chains commands together. It takes the output from one command and feeds it to the next as input. The number of piped commands (the length of the chain) is arbitrary.

Here, we'll use `cat` to feed the contents of the words.txt file into `grep`, which extracts any line that contains either a lower- or uppercase "C." `grep` will then pass these lines to `sort`. `sort` is using the `-r` (reverse) option, so the sorted results will appear in reverse order.

We typed the following:

```
cat words.txt | grep [cC] | sort -r
```



! Pipeline logical NOT and History Operator

The exclamation point (!) is a logical operator that means NOT.

There are two commands in this command line:

```
[ ! -d ./backup ] && mkdir ./backup
```

- The first command is the text within the square brackets;
- The second command is the text that follows the double ampersands `&&`.

The first command uses `!` as a logical operator. The square brackets indicate a test is going to be made. The `-d` (directory) option tests for the presence of a directory called backup. The second command creates the directory.

Because double ampersands separate the two commands, Bash will only execute the second if the first *succeeds*. However, that's the opposite of what we need. If the test for the "backup" directory succeeds, we *don't* need to create it. And if the test for the "backup "directory fails, the second command won't be executed, and the missing directory won't be created.

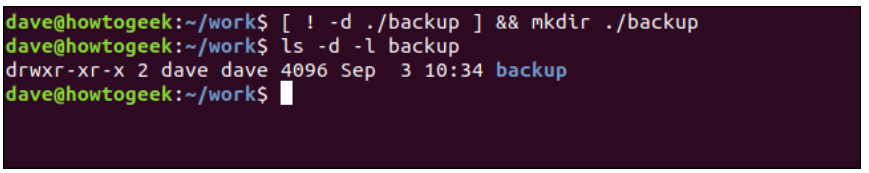
This is where the logical operator `!` comes in. It acts as a logical NOT. So, if the test succeeds (i.e., the directory exists), the `!` flips that to "NOT success," which is *failure*. So, the second command *isn't* activated.

If the directory test fails (i.e., the directory doesn't exist), the `!` changes the response to "NOT failure," which is *success*. So, the command to create the missing directory *is* executed.

That little `!` packs a lot of punch when you need it to!

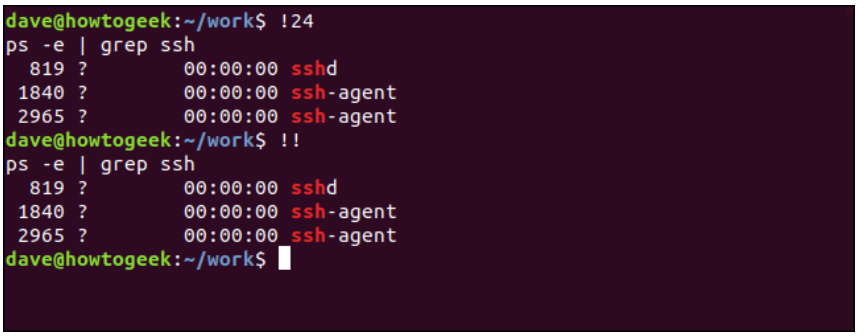
To check the status of the backup folder, you use the `ls` command and the `-l` (long listing) and `-d` (directory) options, as shown below:

```
ls -l -d backup
```



You can also run commands from your command history with the exclamation point. The `history` command lists your command history, and you then type the number of the command you wish to re-run with `!` to execute it, as shown below:

```
!24
```



The following re-runs the previous command:

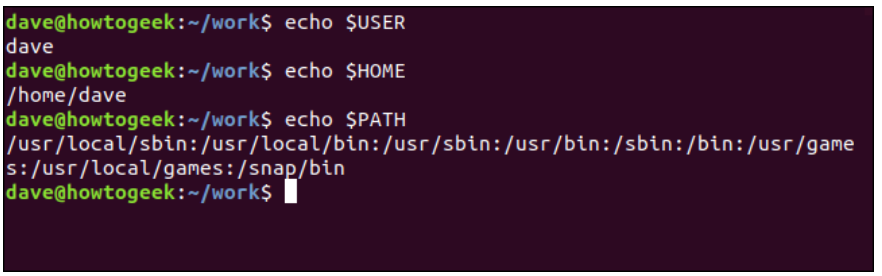
```
!!
```

$ Variable Expressions

In the Bash shell, you create variables to hold values. Some, like environment variables, always exist, and you can access them any time you open a terminal window. These hold values, such as your username, home directory, and path.

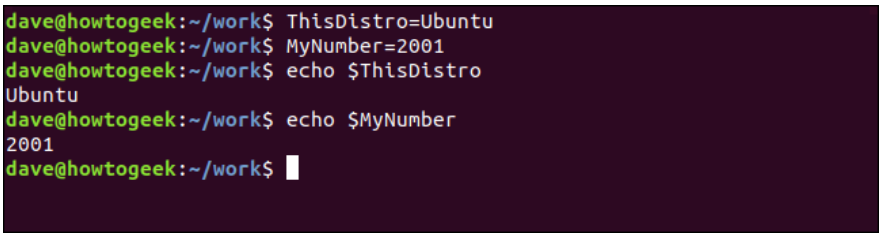
You can use `echo` to see the value a variable holds---just precede the variable name with the dollar sign ($), as shown below:

```
echo $USER
echo $HOME
echo $PATH
```



To create a variable, you must give it a name and provide a value for it to hold. You *do not* have to use the dollar sign to create a variable. You only add `$` when you reference a variable, such as in the following example:

```
ThisDistro=Ubuntu
MyNumber=2001
echo $ThisDistro
echo $MyNumber
```



Add braces ({}) around the dollar sign and perform a parameter expansion to obtain the value of the variable and allow further transformations of the value.

This creates a variable that holds a string of characters, as shown below:

```
MyString=123456qwerty
```

Use the following command to echo the string to the terminal window:

```
echo ${MyString}
```

To return the substring starting at position 6 of the whole string, use the following command (there's a zero-offset, so the first position is zero):

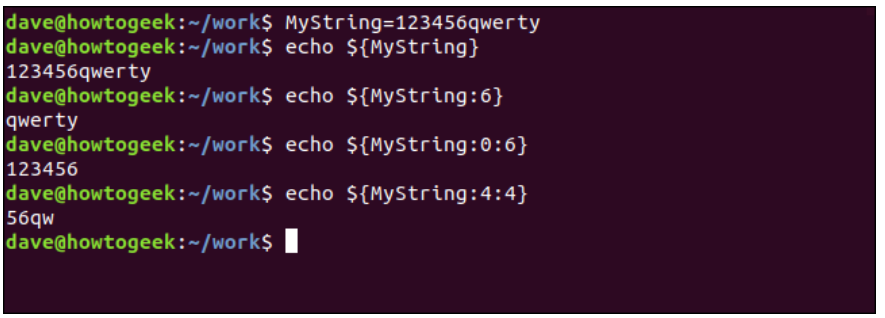
```
echo ${myString:6}
```

If you want to echo a substring that starts at position zero and contains the next six characters, use the following command:

```
echo ${myString:0:6}
```

Use the following command to echo a substring that starts at position four and contains the next four characters:

```
echo ${myString:4:4}
```



Quoting Special Characters

If you want to use a special character as a literal (non-special) character, you have to tell the Bash shell. This is called quoting, and there are three ways to do it.

If you enclose the text in quotation marks ("..."), this prevents Bash from acting on most of the special characters, and they just print. One notable exception, though, is the dollar sign ($). It still functions as the character for variable expressions, so you can include the values from variables in your output.

For example, this command prints the date and time:

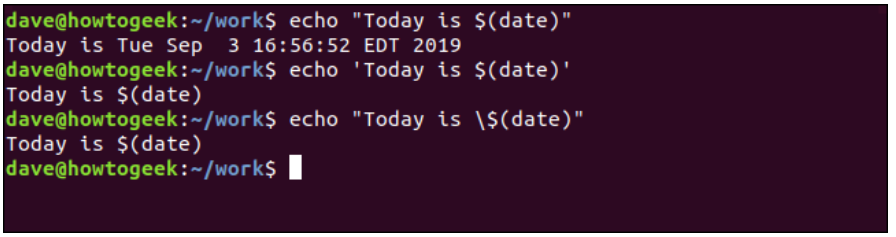
```
echo "Today is $(date)"
```

If you enclose the text in single quotes ('...') as shown below, it stops the function of *all* the special characters:

```
echo 'Today is $(date)'
```

You can use a backslash (\ ) to prevent the following character from functioning as a special character. This is called "escaping" the character; see the example below:

```
echo "Today is \$(date)"
```



Just think of special characters as very short commands. If you memorize their uses, it can benefit your understanding of the Bash shell---and other people's scripts---immensely.