

# BASH

## SPECIAL CHARACTERS

15

**YOU SHOULD  
KNOW**

Dave McKay

# Bash Special Characters

---

15 Special Characters You Need to Know

Dave McKay

©2019 by LifeSavvy Media. All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means without permission in writing from the publisher, except by a reviewer, who may quote brief passages in a review.

Cover Photo by [Malll Themd/Shutterstock](#)



# Contents

What Are Special Characters?.....	1
~ Home Directory.....	2
. Current Directory.....	3
.. Parent Directory.....	4
/ Path Directory Separator.....	5
# Comment or Trim Strings.....	6
? Single Character Wildcard.....	7
* Character Sequence Wildcard.....	8
[] Character Set Wildcard.....	9
; Shell Command Separator.....	10
& Background Process.....	11
< Input Redirection.....	12
> Output Redirection.....	13
Pipe.....	14
! Pipeline logical NOT and History Operator.....	14
\$ Variable Expressions.....	16
Quoting Special Characters.....	18

## 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 ~
```

```
dave@howtogeek:~/work/archive$ cd ~  
dave@howtogeek:~$ █
```

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
```

```
dave@howtogeek:/usr/local/bin$ cd ~/work/archive/  
dave@howtogeek:~/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
```

```
dave@howtogeek:~/work/archive$ ls -a
.          GC Help.mm          os_coord_data.h
..         #geocode.glade#    os_coord.h
command_address.page geocode.glade       os_coord_transform.c
function_headers.h  goose.txt            window_map.page
gc.c               gtk_functions.c     window_tool.page
gc_commands.h      Help File Concordance.ods window_trace.page
gc_defines.h       olc.c
dave@howtogeek:~/work/archive$
```

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.

```
dave@howtogeek:~/work$ ./script.sh
dave@howtogeek:~/work$
```

## .. 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 ..
```

```
dave@howtogeek:~/work/gc_help$ cd ..  
dave@howtogeek:~/work$
```

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
```

```
dave@howtogeek:~/work/archive$ cd ../gc_help  
dave@howtogeek:~/work/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
```

```
dave@howtogeek:~$ ls ~/work/archive/
command_address.page  #geocode.glade#      os_coord_data.h
function_headers.h    geocode.glade         os_coord.h
gc.c                  goose.txt              os_coord_transform.c
gc_commands.h         gtk_functions.c        window_map.page
gc_defines.h          Help File Concordance.ods window_tool.page
GC Help.mm            olc.c                  window_trace.page
dave@howtogeek:~$ █
```

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 /
```

```
dave@howtogeek:~$ cd /
dave@howtogeek:/$ █
```

## # 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
```

```
dave@howtogeek:/$ # This will be ignored by the Bash shell
dave@howtogeek:/$ █
```

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!"
```

```
dave@howtogeek:~/work$ this_string="Dave Geek!"
dave@howtogeek:~/work$ █
```

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}
```

```
dave@howtogeek:~/work$ echo How-To ${this_string#Dave}
How-To Geek!
dave@howtogeek:~/work$ █
```

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
```

```
dave@howtogeek:~/work$ echo $this_string
Dave Geek!
dave@howtogeek:~/work$ █
```

## ? 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.

```
dave@howtogeek:~/work/archive$ ls badge?.txt
badge1.txt badge4.txt badge7.txt badged.txt
badge2.txt badge5.txt badge8.txt badger.txt
badge3.txt badge6.txt badge9.txt badges.txt
dave@howtogeek:~/work/archive$
```

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
```

```
dave@howtogeek:~/work/archive$ ls ??????.txt
badge.txt
dave@howtogeek:~/work/archive$
```

## \* 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:

```
dave@howtogeek:~/work/archive$ ls badge*
badge1.txt  badge4.txt  badge7.txt  badged.txt  badge.txt
badge2.txt  badge5.txt  badge8.txt  badger.txt
badge3.txt  badge6.txt  badge9.txt  badges.txt
dave@howtogeek:~/work/archive$
```

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.*
```

```
dave@howtogeek:~/work$ ls source.*
source.bak  source.c.  source.h  source.ico
dave@howtogeek:~/work$
```

## [] 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
```

```
dave@howtogeek:~/work/archive$ ls badge_0[246].txt
badge_02.txt badge_04.txt badge_06.txt
dave@howtogeek:~/work/archive$
```

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

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

```
dave@howtogeek:~/work/archive$ ls badge_[01][789].txt
badge_07.txt badge_09.txt badge_18.txt
badge_08.txt badge_17.txt badge_19.txt
dave@howtogeek:~/work/archive$
```

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
```

```
dave@howtogeek:~/work/archive$ ls badge_[23][1-5].txt
badge_21.txt badge_23.txt badge_25.txt badge_32.txt badge_34.txt
badge_22.txt badge_24.txt badge_31.txt badge_33.txt badge_35.txt
dave@howtogeek:~/work/archive$
```

## ; 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
```

```
dave@howtogeek:~/work/archive$ ls > count.txt; wc -l count.txt ; rm count.txt
58 count.txt
dave@howtogeek:~/work/archive$
```

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/* .
```

```
dave@howtogeek:~/work/archive$ cd ./doesntexist && cp ~/Documents/reports/* .
bash: cd: ./doesntexist: No such file or directory
dave@howtogeek:~/work/archive$
```

## & 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 &
```



```
dave@howtogeek:~/work/archive$ gedit command_address.page &  
[1] 3831  
dave@howtogeek:~/work/archive$ █
```

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
```

```
dave@howtogeek:~/work$ sort < words.txt
Cataclysm
Catamaran
Catwoman
Xray
Yodelling
Zenith
dave@howtogeek:~/work$ █
```

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`:

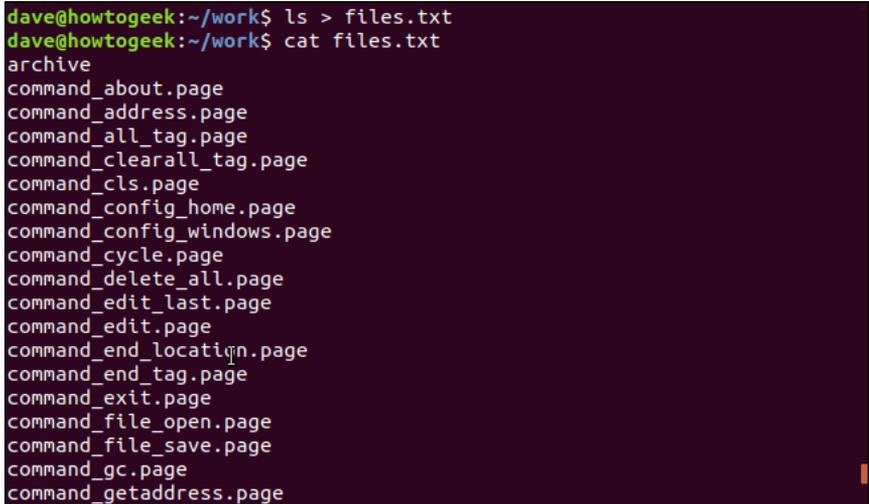
```
wc words.txt
wc < words.txt
```

```
dave@howtogeek:~/work$ wc words.txt
 6  6 51 words.txt
dave@howtogeek:~/work$ wc < words.txt
 6  6 51
dave@howtogeek:~/work$ █
```

## > Output Redirection

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

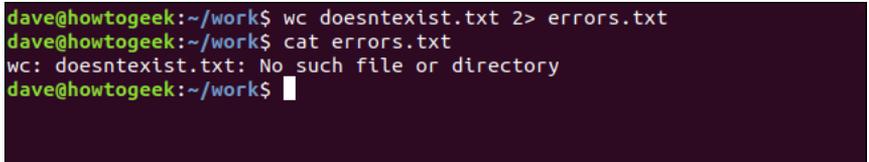
```
ls > files.txt
cat files.txt
```

A terminal window with a dark purple background. The prompt is 'dave@howtogeek:~/work\$'. The user enters 'ls > files.txt' and then 'cat files.txt'. The output of 'cat' is a list of files: 'archive', 'command\_about.page', 'command\_address.page', 'command\_all\_tag.page', 'command\_clearall\_tag.page', 'command\_cls.page', 'command\_config\_home.page', 'command\_config\_windows.page', 'command\_cycle.page', 'command\_delete\_all.page', 'command\_edit\_last.page', 'command\_edit.page', 'command\_end\_location.page', 'command\_end\_tag.page', 'command\_exit.page', 'command\_file\_open.page', 'command\_file\_save.page', 'command\_gc.page', and 'command\_getaddress.page'.

```
dave@howtogeek:~/work$ ls > files.txt
dave@howtogeek:~/work$ cat files.txt
archive
command_about.page
command_address.page
command_all_tag.page
command_clearall_tag.page
command_cls.page
command_config_home.page
command_config_windows.page
command_cycle.page
command_delete_all.page
command_edit_last.page
command_edit.page
command_end_location.page
command_end_tag.page
command_exit.page
command_file_open.page
command_file_save.page
command_gc.page
command_getaddress.page
```

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
```

A terminal window with a dark purple background. The prompt is 'dave@howtogeek:~/work\$'. The user enters 'wc doesntexist.txt 2> errors.txt' and then 'cat errors.txt'. The output of 'cat' is an error message: 'wc: doesntexist.txt: No such file or directory'.

```
dave@howtogeek:~/work$ wc doesntexist.txt 2> errors.txt
dave@howtogeek:~/work$ cat errors.txt
wc: doesntexist.txt: No such file or directory
dave@howtogeek:~/work$
```

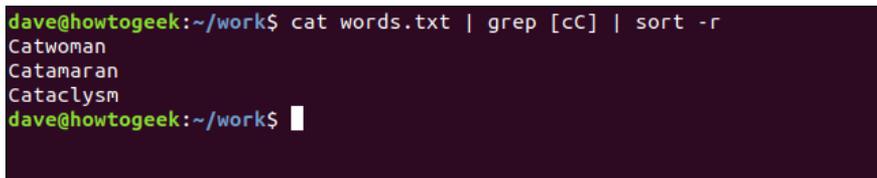
## | Pipe

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
```



```
dave@howtogeek:~/work$ cat words.txt | grep [cC] | sort -r
Catwoman
Catamaran
Cataclysm
dave@howtogeek:~/work$ █
```

## ! 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.

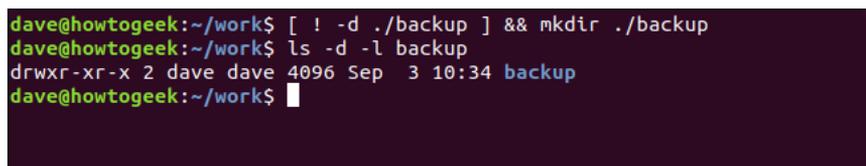
This is where the logical operator `!` comes in. It acts as a logical NOT. So, if the test succeeds (i.e., the directory exists), the `!` flips that to "NOT success," which is *failure*. So, the second command *isn't* activated.

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
```



```
dave@howtogeek:~/work$ [ ! -d ./backup ] && mkdir ./backup
dave@howtogeek:~/work$ ls -d -l backup
drwxr-xr-x 2 dave dave 4096 Sep  3 10:34 backup
dave@howtogeek:~/work$
```

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
```



```
dave@howtogeek:~/work$ !24
ps -e | grep ssh
 819 ?          00:00:00 sshd
1840 ?          00:00:00 ssh-agent
2965 ?          00:00:00 ssh-agent
dave@howtogeek:~/work$ !!
ps -e | grep ssh
 819 ?          00:00:00 sshd
1840 ?          00:00:00 ssh-agent
2965 ?          00:00:00 ssh-agent
dave@howtogeek:~/work$
```

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
```

```
dave@howtogeek:~/work$ echo $USER
dave
dave@howtogeek:~/work$ echo $HOME
/home/dave
dave@howtogeek:~/work$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
dave@howtogeek:~/work$ █
```

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
```

```
dave@howtogeek:~/work$ ThisDistro=Ubuntu
dave@howtogeek:~/work$ MyNumber=2001
dave@howtogeek:~/work$ echo $ThisDistro
Ubuntu
dave@howtogeek:~/work$ echo $MyNumber
2001
dave@howtogeek:~/work$ █
```

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):

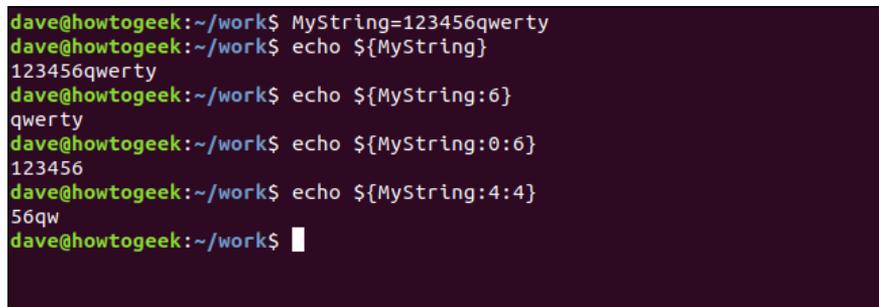
```
echo ${myString:6}
```

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}
```



```
dave@howtogeek:~/work$ MyString=123456qwerty
dave@howtogeek:~/work$ echo ${MyString}
123456qwerty
dave@howtogeek:~/work$ echo ${MyString:6}
qwerty
dave@howtogeek:~/work$ echo ${MyString:0:6}
123456
dave@howtogeek:~/work$ echo ${MyString:4:4}
56qw
dave@howtogeek:~/work$ █
```

## 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:

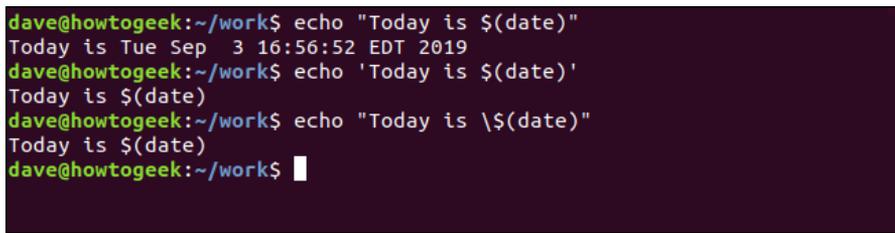
```
echo "Today is $(date)"
```

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)"
```



```
dave@howtogeek:~/work$ echo "Today is $(date)"
Today is Tue Sep  3 16:56:52 EDT 2019
dave@howtogeek:~/work$ echo 'Today is $(date)'
Today is $(date)
dave@howtogeek:~/work$ echo "Today is \$(date)"
Today is $(date)
dave@howtogeek:~/work$ █
```

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.