**String Manipulation**

Since so much of working in bash is related to files and strings of text, the ability to manipulate strings is valuable.

While tools such as sed, awk, perl (and many others) are well worth learning, in this lesson I want to show you what is possible in bash - and it may be more than you think!

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/N0NjoAyRBoL#how-important-is-this-lesson)

This lesson is not essential. String manipulation in bash is only occasionally useful to know about, for example when the above-mentioned tools (sed, awk, perl et al) are not available.

The quoting tricks in this lesson are occasionally very useful, however.

### String Length [#](https://www.educative.io/courses/master-the-bash-shell/N0NjoAyRBoL#string-length)

One of the most common requirements when working with strings is to determine length:

1

2

3

A='12345678901234567890'

echo "${#A}"               # How long is 'A'?

echo "$#A"





Type the above code into the terminal in this lesson.

### String Editing [#](https://www.educative.io/courses/master-the-bash-shell/N0NjoAyRBoL#string-editing)

Bash provides a way to extract a substring from a string. The following example explains how to parse n characters starting from a particular position.

Work out what’s going on here. You may need to consult the manual:

1

2

echo ${A:2}

echo ${A:2:3}





Type the above code into the terminal in this lesson.

You can replace sections of scripts using search and replace. The first part enclosed in / signs represents what’s searched for, and the second what is replaced:

1

2

echo "${A/234/432}"

echo "${A//234/432}"





Type the above code into the terminal in this lesson.

What’s going on in the second command above? How does it differ from the first?

### String Editing [#](https://www.educative.io/courses/master-the-bash-shell/N0NjoAyRBoL#string-editing-2)

Another commonly-required string operation is getting a substring.

This outputs everything from the third character in the string:

1

echo ${A:2}





Type the above code into the terminal in this lesson.

and this outputs two numbers from the fourth character in the string:

1

echo ${A:3:2}





Type the above code into the terminal in this lesson.

You can tell from this that the strings are 0-indexed, that is the first character is numbered zero, the second is numbered one, and so on.

You can also search and replace within strings:

1

2

B=0000000000

echo ${B/0/1}





Type the above code into the terminal in this lesson.

The simple pattern replaces only the first 0 with a 1. If you want the pattern to go across all the matches in the string, then add another slash:

1

echo ${B//0/1}





Type the above code into the terminal in this lesson.

You can also convert lower case to upper case and vice versa. Using a single comma will turn the starting title case character into a lower case character:

1

2

C=AbCd

echo ${C,}





Type the above code into the terminal in this lesson.

whereas using two will lower-case the entire string:

1

2

echo ${C,,}

C=aAbCd





Type the above code into the terminal in this lesson.

Similarly, the caret (^) character will upper-case a string also:

1

2

echo ${C^}

echo ${C^^}





Type the above code into the terminal in this lesson.

### And More… [#](https://www.educative.io/courses/master-the-bash-shell/N0NjoAyRBoL#and-more)

There are many more string manipulation tricks you can use, but it is not worth trying to learn them all in one go.

If you want to study them in more detail and find out what’s available (or ever find yourself wondering what can be done in bash with strings) then have a look at the man page under Parameter Expansion.

### Extglobs and Removing Text [#](https://www.educative.io/courses/master-the-bash-shell/N0NjoAyRBoL#extglobs-and-removing-text)

A more advanced means of working with strings is possible by using bash’s **extglobs** functionality.

A word of warning here: although this functionality is useful to know, it is arguably less useful than learning the programs sed and perl for this purpose. It’s also quite confusing to have this extra type of glob syntax to learn in addition to regular expressions (and there’s even flavours of those too!). Feel free to skip this section if you feel it’s too obscure.

1

2

3

shopt -s extglob

A="12345678901234567890"

B="  ${A}  "





Type the above code into the terminal in this lesson.

You’ve ensured that the extglob option is on, and created a new variable B, which is A with two spaces in front and behind. The pipe (|) character is used in the echo output to show where the spaces are.

1

2

3

4

5

6

7

echo "B      |${B}|"

echo "B#+( ) |${B#+( )}|"

echo "B#?( ) |${B#?( )}|"

echo "B#\*( ) |${B#\\*( )}|"

echo "B##+( )|${B##+( )}|"

echo "B##\*( )|${B##\*( )}|"

echo "B##?( )|${B##?( )}|"





Type the above code into the terminal in this lesson.

Using the bash man page, and experimenting, can you figure out the differences between the above extglobs? They are:

* ?( )
* +( )
* \*( )

Now try % instead of # above. What happens?

One potentially handy application of this is when having to remove leading zeroes from dates:

1

2

TODAY=$(date +%j)

TODAY=${TODAY##+(0)}





Type the above code into the terminal in this lesson.

Remember: # is to the left, and % is to the right on a (US) keyboard. Or ‘hash’ is before ‘per cent’ in the alphabet.

### Quoting Hell [#](https://www.educative.io/courses/master-the-bash-shell/N0NjoAyRBoL#quoting-hell)

Quoting - as I’m sure you’ve seen - can get very complicated in bash very quickly.

Try this:

1

echo 'I really want to echo $HOME and I can't avoid it'





Type the above code into the terminal in this lesson.

Uh-oh. You’re now stuck.

Can you see why?

Hit CTRL-c to get out of it.

So you might think to try this:

1

echo "I really want to echo $HOME and I can't avoid it"





Type the above code into the terminal in this lesson.

but this time the $HOME variable is evaluated and the output is not what is wanted.

Try this:

1

echo 'I really want to echo $HOME and I can'"'"'t avoid it'





Type the above code into the terminal in this lesson.

That works. Remember this trick as it can save you a lot of time!

# Autocomplete

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/gkP2Kww6QDD#how-important-is-this-lesson)

In this lesson you will build on what you learned previously about:

* Bash arrays
* Terminal escape codes
* Built-ins
* Bash functions
* Bash startup scripts

to comprehend more deeply a feature you will probably use almost every time you use bash.

### Autocompleting Commands [#](https://www.educative.io/courses/master-the-bash-shell/gkP2Kww6QDD#autocompleting-commands)

If you hit the TAB key on your keyboard twice at the default prompt, then you should see something like this:

Display all 2266 possibilities (y or n)?

If you’re brave enough to hit y then you will see a list of all the commands available in your path.

To escape that list, hit q.

Now if you try typing z on a fresh shell command line, and then hit TAB twice, you will likely immediately see the commands available that begin with z.

That’s autocomplete in bash - by default, you get the commands available that match the letters you’ve typed in so far. This is handy if you can only remember part of a command, or you just want to see what commands start with a (try it!).

### Autocompleting Arguments [#](https://www.educative.io/courses/master-the-bash-shell/gkP2Kww6QDD#autocompleting-arguments)

In addition to looking for commands, bash’s autocomplete can be made to be context-aware. A simple example of this is to see what happens when you type ls, and then a space, and then TAB twice on a fresh shell line. What happens? You should see that this time commands are not listed, but something else…

Listing local files is something of a default in bash - it assumes that arguments are likely to be files. What I’m going to show you next is how you can get bash to be even more context dependent with auto-complete to the point where you can advise the user of your specific program about what is available to them.

### A Simple Program [#](https://www.educative.io/courses/master-the-bash-shell/gkP2Kww6QDD#a-simple-program)

First you’re going to create a program called myecho. As you type it in, try and work out what it’s doing.

If you don’t understand a line or command in there, then try and find out what it does by playing with it in a toy script, or try and find out from man bash.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

cat > myecho << 'END'

#!/usr/bin/env bash

function usage() {

    echo "Usage: $0 [red|green|blue] [message]"

    exit $1

}

if [ -z "$1" ]

then

    usage 1

fi

typeset -l COLOR="$1"

shift

RED='\033[0;31m'

GREEN='\033[0;32m'

BLUE='\033[0;34m'

MSG="$@"

COLOR=${COLOR,,}

if [[ "${COLOR}" = 'red' ]]

then

    MSG="${RED}${@}${RED}"

elif [[ "${COLOR}" = 'green' ]]

then

    MSG="${GREEN}${@}${GREEN}"

elif [[ "${COLOR}" = 'blue' ]]

then

    MSG="${BLUE}${@}${BLUE}"

else

    usage 1

fi

echo -e "${MSG}"

END





Type the above code into the terminal in this lesson.

If you’re still struggling, then other lessons in this course will help enlighten you. The only new command here is shift. You may be able to work out what shift does from context, but just in case it’s not obvious, it takes the first argument from your command line and removes it from the list of arguments passed in. This is handy for processing command line arguments, as you can keep calling shift until all the arguments are processed. Such a looping construct is used in many bash scripts.

You may also have not come across the variable $@ before. Try finding out from the man bash page what it holds.

If you can’t find it, don’t worry, it’s not easy to find in there. Look for the section on Special Parameters.

If you still can’t figure it out, then try adding echo statements to figure out what’s going on. Keep at it.

Finally, can you figure out what the two commas in ${COLOR,,} do? If not, play with it on the command line. This playing is how you learn and embed the bash knowledge as you get more advanced.

Now you’ve created the myecho script, make it executable and available on your path, and run it:

1

2

3

4

chmod +x myecho

PATH=.:${PATH}

myecho red WARNING this is dangerous

myecho green OK good to go





Type the above code into the terminal in this lesson.

Try changing the case of the colour argument (eg ‘GreeN’ and see what happens. Was that what you expected?). What part of the code is responsible for this?

### Adding Autocomplete Functionality [#](https://www.educative.io/courses/master-the-bash-shell/gkP2Kww6QDD#adding-autocomplete-functionality)

Now let’s say that it is some time later, and you’ve forgotten about your brilliant myecho script.

You type myecho on the command line, followed by a space and then hit TAB twice.

Wouldn’t it be good if autocomplete could present the three colour options to you when auto-completing?

You can do that. Type out this script:

1

2

3

4

5

6

7

8

9

    cat > myechocomplete << END

    myecho\_completion()

    {

      COMPREPLY+=("red")

      COMPREPLY+=("blue")

      COMPREPLY+=("green")

    }

    complete -F myecho\_completion myecho

    END





Type the above code into the terminal in this lesson.

To make your shell aware of it, type:

1

source myechocomplete





Type the above code into the terminal in this lesson.

Now if you type myecho, followed by a space and then two 'Tab’s, you should see the three options.

The complete command is a builtin, and the -F flag expects a function to be given to it, and will run it when the last argument to complete is typed on the command line and autocomplete is triggered.

When the function is run bash expects you to add to an array called COMPREPLY to give the options that should be made available to the user when they try to autocomplete. The three colours added above are the displayed to the user from the array.

### Advanced Autocomplete Functionality [#](https://www.educative.io/courses/master-the-bash-shell/gkP2Kww6QDD#advanced-autocomplete-functionality)

If you want to get a bit more sophisticated with bash’s autocomplete, then you have other variables than COMPREPLY available to you. For this simple program you’re going to provide a hint to input a message if the user has already input a colour argument to myecho.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

cat > myechocomplete << END

myecho\_completion()

{

  if [ ${COMP\_CWORDS -ge 2 ]

  then

    COMPREPLY+=("Input")

    COMPREPLY+=("message!")

  else

    COMPREPLY+=("red")

    COMPREPLY+=("blue")

    COMPREPLY+=("green")

  fi

}

complete -F myecho\_completion myecho

END

source myechocomplete





Type the above code into the terminal in this lesson.

Now if you type myecho red, followed by a space and then two 'Tab’s, you should see the instruction on the line.

This is not exactly a neat way to get this information across. Unfortunately, the completion output functionality is not very flexible. You might want to try implementing a better solution with echos or longer COMPREPLY entries to see its limitations.

### Automate It [#](https://www.educative.io/courses/master-the-bash-shell/gkP2Kww6QDD#automate-it)

Although this is useful, it is not realistic to source the completion scripts every time you might want to use the particular command you have an autocomplete script for.

For this reason, you can add these functions to your bash startup scripts (see the lesson on [Scripts and Startups](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6245552490545152) in Part I) so that they are available whenever you use bash on that machine.