Autocomplete

In this lesson, you will cover how autocomplete works in bash, the bash 'shift' builtin, and changing case in variables.

WE'LL COVER THE FOLLOWING

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

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)?
```

Terminal

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

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

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`.

```
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:

```
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

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:

```
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:

```
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

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`.

```
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 `echo`s or longer `COMPREPLY` entries to see its limitations.

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 in Part I) so that they are available whenever you use bash on that machine.

What You Learned

- How *autocomplete* works in bash
- How to create your own *autocomplete* function
- How to handle arguments to bash scripts and functions using `shift`
- How to change the case of string variables

Exercises

1) Write autocomplete scripts for already-existing programs you use often.

2) Look for autocomplete scripts on GitHub, and install and use them locally.

3) Read the bash manual section and figure out what all the `COMP_` variables store when autocompleting. Write example programs to use them.