Variables

In this lesson, you'll learn about variables in bash and some of their subtleties. It will cover basic variables and how they're quoted, the 'env' and 'export' commands, and simple and associative arrays. By the end, you will have a good overview of how variables work in bash and some of their pitfalls.

WE'LL COVER THE FOLLOWING

How Important is this Lesson?

Variables are fundamental to understanding bash commands, as much as they are in any programming language.

Basic Variables

Start by creating a variable and `echo`ing it.

```
MYSTRING=astring
echo $MYSTRING
```

Type the above code into the terminal in this lesson.

Terminal

> Note: continue to use this terminal during this lesson. Throughout this course it is assumed that you complete each lesson by typing in the commands in the provided terminal for that lesson in the order the commands are given.

Simple enough: you create a variable by

- Stating its name
- Immediately adding an equals sign
- Immediately stating the value

Variables don't need to be capitalized, but they generally are by convention.

To get the value out of the variable, you have to use the dollar sign to tell bash that you want the variable dereferenced.

Variables and Quoting

Things get more interesting when you start quoting.

Quoting can be used to group different *words* into a single variable value:

```
MYSENTENCE=A sentence     # This command will not work
MYSENTENCE="A sentence"   # The quotes group the words together
echo $MYSENTENCE
```

Type the above code into the terminal in this lesson.

Since (by default) the shell reads each *word* separated by a space, it thinks the word `sentence` is not related to the variable assigment, and treats it as a program. To store a sentence with space in it inside a variable, you can enclose it in the double quotes, as shown above.

Things get even more interesting when we embed other variables in the quoted string:

```
MYSENTENCE="A sentence with $MYSTRING in it"
echo $MYSENTENCE                              # Outputs the sentence with the variable deref
MYSENTENCE='A sentence with $MYSTRING in it'
```

```
echo $MYSENTENCE                    # Outputs the sentence with the variable not c
```

Type the above code into the terminal in this lesson.

If you were expecting similar behaviour to the previous lesson you may have gotten a surprise!

This illustrated an important point if you're reading shell scripts: the bash shell translates the variable into its value if it's in double quotes, but does not if it's in single quotes.

Remember from the previous lesson that this is not true when globbing!

Try out the code below in the terminal and see the output. Like always, make sure you think about the output you expect before you see it:

```
MYGLOB=*        # No quotes around the glob
echo $MYGLOB    # Glob is interpreted
MYGLOB="*"      # Double quotes around the glob
echo "$MYGLOB"  # Glob is not interpreted
MYGLOB='*'      # Single quotes around the glob
echo "$MYGLOB"  # Glob is not interpreted
echo '$MYGLOB'  # Variable is not interpreted
echo $MYGLOB    # Glob is interpreted
```

Type the above code into the terminal in this lesson.

Globs are not expanded when in either single or double quotes. Confusing isn't it?

Shell Variables

Some variables are special, and set up when bash starts:

```
echo $PPID
PPID=nonsense
echo $PPID
```

Type the above code into the terminal in this lesson.

- **Line 1** - `PPID` is a special variable set by the bash shell. It contains the bash's parent process id
- **Line 2** - Try and set the `PPID` variable to something else
- **Line 3** - Output `PPID` again

Can you work out what happened there?

You couldn't set the variable, because this is a `readonly` variable.

If you want to make a variable read-only, put `readonly` in front of it, like this:

```
readonly MYVAR=astring
MYVAR=anotherstring
```

Type the above code into the terminal in this lesson.

The `export` Command

Type in these commands, and try to predict what will happen:

```
MYSTRING=astring                # Set the MYSTRING variable
bash                            # Enter a fresh bash shell
echo $MYSTRING                  # Has the string made it to the new bash shell?
exit                            # Quit the bash shell
echo $MYSTRING                  # Is the string still there?
unset MYSTRING                  # Unset the string
echo $MYSTRING                  # Check it's not there
export MYSTRING=anotherstring   # Do the same, but export it this time
bash                            # Enter a fresh bash shell
echo $MYSTRING                  # Has it made it this time?
exit
```

Type the above code into the terminal in this lesson.

Based on this, what do you think `export` does?

You've already seen that a variable set in a bash terminal can be referenced later by using the dollar sign.

But what happens when you set a variable, and then start up another process?

In this case, you set a variable (`MYSTRING`) to the value `astring`, and then start up a new bash shell process. Within that bash shell process, `MYSTRING` does not exist, so an error is thrown. In other words, the variable was not inherited by the bash process you just started.

After exiting that bash session, and unsetting the `MYSTRING` variable to ensure it's gone, you set it again, but this time `export` the variable, so that any processes started by the running shell will have it in their environment. You show this by starting up another bash shell, and it `echoes` the new value

`anotherstring` to the terminal.

It's not just shells that have environment variables! All processes have environment variables.

Outputting Exported and Shell Variables

Wherever you are, you can see the exported variables that are set by running `env`:

```
env
```

Type the above code into the terminal in this lesson.

The output of `env` will likely be different wherever you run it.

That isn't all the variables that are set in your shell, though. It's just the *environment* variables that are exported to processes that you start in the shell.

If you want to see *all* the variables that are available to you in your shell, type:

```
compgen -v
```

Type the above code into the terminal in this lesson.

`compgen` is a command that generates list of possible 'word completions' in bash when you hit tab repeatedly. The `-v` flag shows all the variables that could be completed from where you type. Hence, it lists all variables, exported and local to the shell, in the environment where you are.

Arrays

Also worth mentioning here are *arrays*. One such built-in, read only array is `BASH_VERSINFO`. As in other languages, arrays in bash are zero-indexed.

Type out the following commands, which illustrate how to reference the version information's major number:

```
echo $BASH_VERSINFO
echo $BASH_VERSINFO[0]   # Does not work
echo ${BASH_VERSINFO[0]} # Will work
```

```
echo ${BASH_VERSINFO}
```

Type the above code into the terminal in this lesson.

Arrays can be tricky to deal with, and bash doesn't give you much help!

The first thing to notice is that the array will output the item at the index `0` if no index is given.

The second thing to notice is that simply adding `[0]` to a normal variable reference does not work. Bash treats the square bracket as a character not associated with the variable and appends it to the end of the array.

You have to tell bash to treat the whole string `BASH_VERSION[0]` as the variable to be dereferenced. You do this by using the curly braces.

These curly braces can be used on simple variables too:

```
echo $BASH_VERSION_and_some_string     # Won't find the variable
echo ${BASH_VERSION}_and_some_string   # The curly quotes tell bash where the variable is
```

Type the above code into the terminal in this lesson.

In fact, 'simple variables' can be treated as arrays with one element!

```
echo ${BASH_VERSION[0]}
```

Type the above code into the terminal in this lesson.

So all bash variables are 'really' arrays!

Bash has 6 items (0-5) in its `BASH_VERSINFO` array:

```
  echo ${BASH_VERSINFO[1]}
  echo ${BASH_VERSINFO[2]}
  echo ${BASH_VERSINFO[3]}
  echo ${BASH_VERSINFO[4]}
  echo ${BASH_VERSINFO[5]}
  echo ${BASH_VERSINFO[6]}
```

Type the above code into the terminal in this lesson.

As ever with variables, if the item does not exist then the output will be an empty line.

Associative Arrays

Bash also supports *associative arrays*.

With associative arrays, you use a string instead of a number to reference the value:

```
declare -A MYAA=([one]=1 [two]=2 [three]=3)  # declare the associative array
MYAA[one]="1"                                # set the 'one' array element
MYAA[two]="2"
echo $MYAA                                   # outputs nothing
echo ${MYAA[one]}                            # outputs '1'
WANT=two
echo ${MYAA[$WANT]}                          # outputs '2'
```

Type the above code into the terminal in this lesson.

Variables Quiz

1 What will this output?

```
echo '$HOME'
```

COMPLETED 0%

1 of 2

As well as not being compatible with versions less than 4, associative arrays

are quite fiddly to create and use, hence are not seen very often.

What You Learned

- Basic variable usage in bash
- Variables and quoting
- Variables set up by bash
- `env` and `export`
- Bash arrays

What Next?

Next you will learn about another core language feature implemented in bash: **functions**.

Exercises

1) Take the output of `env` in your terminal and work out why each item is there and what it might be used for. You may want to use `man bash`, or google to figure it out. You could try resetting it to see what happens.

2) Find out what the items in `BASH_VERSINFO` mean.