Basic variables in Bash

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

Assigning variables

Similar to other languages, you can assign variables with the equals notation.

```
var1="Moon"
```

Then reference with `$` notation.

```
echo $var1
```

```
Moon
```

Assigning string variables

Name your variable as you like (something sensible!):

```
firstname='Cynthia'
lastname='Liu'
echo "Hi there" $firstname $lastname
```

```
Hi there Cynthia Liu
```

Both variables were returned - nice!

Missing the $ notation

If you miss the `$` notation - it isn't a variable!

```
firstname='Cynthia'
lastname='Liu'
echo "Hi there " firstname lastname
```

```
Hi there firstname lastname
```

(Not) assigning variables

Bash is not very forgiving about spaces in variable creation. Beware of adding spaces!

```
var1 = "Moon"
echo $var1
```

```
script.sh: line 3: var1: command not found
```

Single, double, backticks

In Bash, using different quotation marks can mean different things. Both when creating variables and printing.

- Single quotes (`'sometext'`) = Shell interprets what is between literally
- Double quotes (`"sometext"`) = Shell interprets literally **except** using `$` and backticks

The last way creates a 'shell-within-a-shell', outlined below. Useful for calling command-line programs. This is done with backticks.

- Backticks (`sometext`) = Shell runs the command and captures STDOUT back into a variable

Different variable creation

Let's see the effect of different types of variable creation

```
now_var='NOW'
now_var_singlequote='$now_var'
echo $now_var_singlequote
```

```
$now_var
```

```
now_var_doublequote="$now_var"
echo $now_var_doublequote
```

```
NOW
```

The date program

The `Date` program will be useful for demonstrating backticks

Normal output of this program:

```
date
```

```
Mon  2 Dec 2019 14:07:10 AEDT
```

Shell within a shell

Let's use the shell-within-a-shell now:

```
rightnow_doublequote="The date is `date`."
echo $rightnow_doublequote
```

```
The date is Mon 2 Dec 2019 14:13:35 AEDT.
```

The date program was called, output captured and combined in-line with the `echo` call.

We used a shell within a shell!

Parentheses vs backticks

There is an equivalent to backtick notation:

```
rightnow_doublequote="The date is `date`."
rightnow_parentheses="The date is $(date)."
echo $rightnow_doublequote
echo $rightnow_parentheses
```

```
The date is Mon 2 Dec 2019 14:54:34 AEDT.
The date is Mon 2 Dec 2019 14:54:34 AEDT.
```

Both work the same though using backticks is older. Parenthesis is used more in modern applications. (See http://mywiki.wooledge.org/BashFAQ/082)

Let's practice!

INTRODUCTION TO BASH SCRIPTING

Numeric variables in Bash

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

Numbers in other languages

Numbers are not built in natively to the shell like most REPLs (console) such as R and Python

In Python or R you may do:

```
>>> 1 + 4
```

```
5
```

It will return what you want!

Numbers in the shell

Numbers are not natively supported:

(In the terminal)

```
1 + 4
```

```
bash: 1: command not found
```

Introducing expr

`expr` is a useful utility program (just like `cat` or `grep`)

This will now work (in the terminal):

```
expr 1 + 4
```

```
5
```

Nice stuff!

expr limitations

`expr` cannot natively handle decimal places:

(In terminal)

```
expr 1 + 2.5
```

```
expr: not a decimal number: '2.5'
```

Fear not though! (There is a solution)

Introducing bc

`bc` (basic calculator) is a useful command-line program.

You can enter it in the terminal and perform calculations:

```
~$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
5 + 7
12
quit
~$
```

Getting numbers to bc

Using `bc` without opening the calculator is possible by piping:

```
echo "5 + 7.5" | bc
```

```
12.5
```

bc scale argument

`bc` also has a `scale` argument for how many decimal places.

```
echo "10 / 3" | bc
```

```
3
```

```
echo "scale=3; 10 / 3" | bc
```

Note the use of `;` to separate 'lines' in terminal

```
3.333
```

Numbers in Bash scripts

We can assign numeric variables just like string variables:

```
dog_name='Roger'
dog_age=6
echo "My dog's name is $dog_name and he is $dog_age years old"
```

Beware that `dog_age="6"` will work, but makes it a string!

```
My dog's name is Roger and he is 6 years old
```

Double bracket notation

A variant on single bracket variable notation for numeric variables:

```
expr 5 + 7
echo $((5 + 7))
```

```
12
12
```

Beware this method uses `expr` , not `bc` (no decimals!)

Shell within a shell revisited

Remember how we called out to the shell in the previous lesson?

Very useful for numeric variables:

```
model1=87.65
model2=89.20
echo "The total score is $(echo "$model1 + $model2" | bc)"
echo "The average score is $(echo "($model1 + $model2) / 2" | bc)"
```

```
The total score is 176.85
The average score is 88
```

Let's practice!

INTRODUCTION TO BASH SCRIPTING

Arrays in Bash

INTRODUCTION TO BASH SCRIPTING

Alex Scriven
Data Scientist

datacamp

What is an array?

Two types of arrays in Bash:

- An array
  - 'Normal' numerical-indexed structure.
  - Called a 'list' in Python or 'vector' in R.

In Python: `my_list = [1,3,2,4]`

In R: `my_vector <- c(1,3,2,4)`

Creating an array in Bash

Creation of a numerical-indexed can be done in two ways in Bash.

1. Declare without adding elements

```
declare -a my_first_array
```

2. Create and add elements at the same time

```
my_first_array=(1 2 3)
```

Remember - no spaces around equals sign!

Be careful of commas!

Commas are not used to separate array elements in Bash:

This is **not** correct:

```
my_first_array=(1, 2, 3)
```

This **is** correct:

```
my_first_array=(1 2 3)
```

Important array properties

- All array elements can be returned using `array[@]` . Though do note, Bash requires curly brackets around the array name when you want to access these properties.

```
my_array=(1 3 5 2)
echo ${my_array[@]}
```

```
1 3 5 2
```

- The length of an array is accessed using `#array[@]`

```
echo ${#my_array[@]}
```

```
4
```

Manipulating array elements

Accessing array elements using square brackets.

```
my_first_array=(15 20 300 42)
echo ${my_first_array[2]}
```

```
300
```

- Remember: Bash uses zero-indexing for arrays like Python (but unlike R!)

Manipulating array elements

Set array elements using the index notation.

```
my_first_array=(15 20 300 42 23 2 4 33 54 67 66)
my_first_array[0]=999
echo ${my_first_array[0]}
```

```
999
```

- Remember: don't use the `$` when overwriting an index such as `$my_first_array[0]=999`, as this will not work.

Manipulating array elements

Use the notation `array[@]:N:M` to 'slice' out a subset of the array.

- Here `N` is the starting index and `M` is how many elements to return.

```
my_first_array=(15 20 300 42 23 2 4 33 54 67 66)
echo ${my_first_array[@]:3:2}
```

```
42 23
```

Appending to arrays

Append to an array using `array+=(elements)`.

For example:

```
my_array=(300 42 23 2 4 33 54 67 66)
my_array+=(10)
echo ${my_array[@]}
```

```
300 42 23 2 4 33 54 67 66 10
```

(Not) appending to arrays

What happens if you do not add parentheses around what you want to append? Let's see.

For example:

```
my_array=(300 42 23 2 4 33 54 67 66)
my_array+=10
echo ${my_array[@]}
```

```
30010 42 23 2 4 33 54 67 66
```

The string `10` will just be added to the first element. Not what we want!

Associative arrays

- An **associative** array
  - Similar to a normal array, but with key-value pairs, not numerical indexes
  - Similar to Python's dictionary or R's list
  - Note: This is only available in Bash 4 onwards. Some modern macs have old Bash! Check with `bash --version` in terminal

In Python:

```
my_dict = {'city_name': "New York", 'population': 14000000}
```

In R:

```
my_list = list(city_name = c('New York'), population = c(14000000))
```

Creating an associative array

You can only create an associative array using the declare syntax (and uppercase `-A`).

You can either declare first, then add elements or do it all on one line.

- Surround 'keys' in square brackets, then associate a value after the equals sign.
  - You may add multiple elements at once.

Associative array example

Let's make an associative array:

```
declare -A city_details # Declare first
city_details=([city_name]="New York" [population]=14000000) # Add elements
echo ${city_details[city_name]} # Index using key to return a value
```

```
New York
```

Creating an associative array

Alternatively, create an associative array and assign in one line

- Everything else is the same

```
declare -A city_details=([city_name]="New York" [population]=14000000)
```

Access the 'keys' of an associative array with an `!`

```
echo ${!city_details[@]} # Return all the keys
```

```
city_name population
```

Let's practice!

INTRODUCTION TO BASH SCRIPTING