Bash Basics

Joseph Vantassel, The University of Texas at Austin

Sources:

chmod
string manipulation color and styling

Command Line Basics

```
man <command>              # Display manual information for command
man bash                   # Display manual on bash
echo $BASH_VERSION         # Echo bash version number to screen
ls -l                      # List directory contents (-l is for long listing)
rmdir <dirname>            # Remove directory
clear                      # Clear terminal
cat <file>                 # Concatenate file(s) and print to stdout
head <file>                # Show the first few lines of a file
tail <file>                # Show the last few lines of a file
more <file>                # Show entire file page by page
```

Tilde

```
~                          # Variable for home directory (e.g. cd ~)
~-                         # Variable for previous directory
```

Brace Expansion

Below is an example script illustrating the usefulness of brace expansion.

```
#!/bin/bash
mkdir 0_examples
cd 0_examples
touch {1,2,3}.foo            # Simple array
touch {5..10}.bar            # Array with the interpolation done for you
touch {20..15..2}.spam       # Different steps
touch {01..10}.num           # Append zero to start
touch {A..Z..5}.letter          # Also works for letters
```

Note that brace expansion is inclusive. Meaning `{1..3}` will produce `1 2 3`.

Pipes and Redirects

Pipes and redirects allow you to string together bash commands for more complex behavior.

```
ls | more                  # List out current directory, page by

ls 1> file.ouput           # Pass stdout to file.output
ls 2> file.error           # Pass stderr to file.error
ls &> file.both            # Pass stdout and stderr to file.both
ls &> dev/null             # Pass stdout and stderr to nowhere

ls>file.append             # > Write file
ls>>file.append            # >> Appends to the file
```

File Permissions

Each file has a permission (e.g., read, write, and execute).

```
chmod [ugoa][+-=][rwxst]     # [ugoa] User, Group, Other, All
# [+=-] Add, Equal, Remove
# [rwxst] Read, Write, Execute, Set User/Group ID on Execute, Restricted deletion flag or sticky bit.
```

Can also use binaries to set the permissions

```
# _ _ _ ---> Treat like binary, so use numbers 0 to 7
# r w x
chmod 777                  # chmod a+rwx
chmod 700                  # chmod a-rwx, chmod u+rwxs
chmod 710                  # chmod u=rwx, chmod g+
```

grep

grep is a command line tool for finding patterns in plain text.

```
grep <find> <filename>      # Searches a particular filename
grep <find> *               # Search all files in the current directory
grep <find> -r *            # Search all files recursively

grep -i <find> <filename>   # Case insensitive
grep -e <find> <filename>   # Find regular expression
```

awk

TODO

Scripting Basics

Setting up a shell script.

```
#!/bin/bash

bash <script.sh>            # Send script to bash
./<script.sh>               # Current directory is not part of path
```

echo

echo can be used to print information to stdout.

```
echo statement              # Prints as it finds it (can be ambiguous)
echo 'statement'            # Not interpreted (like a string in other Languages)
echo "statement"            # Strong quotes which is interpreted
```

Variables

May declare variables, but it is not required.

```
declare -i                  # Declare as integer
declare -r                  # Declare as readonly
declare -l                  # Declare as lowercase
declare -u                  # Declare as UPPPERCASE
```

Some standard variables.

```
echo $HOME                  # Echo home directory
echo $PWD                   # Echo current working directory
echo $MACHTYPE              # Echo machine type
echo $SECONDS               # Echo seconds since program started
echo $BASH_VERSION          # Echo bash version
```

Simple script

```
#/bin/bash
echo $HOME                  # Echo home directory
echo $PWD                   # Echo current working directory
echo $MACHTYPE              # Echo machine type
echo $SECONDS               # Echo seconds
echo $BASH_VERSION          # Echo bash version
touch {1..10000}.txt
rm *.txt
echo $SECONDS
touch {1..10000}.txt
rm *.txt
echo $SECONDS
```

Strings

In general bash treats everything as a string unless you explicitly tell it to treat it as something else (e.g., integer).

Concatenation

```
#!/bin/bash
a="Ta"                      # Define string Ta
b="Da"                      # Define string Da
c=$a$b                      # Define concatenation of a and b
echo $c                     # Show the concatenation
```

Slicing Strings by Position

```
#!/bin/bash
string="This is a long string" # Define a String

# Length of string
echo ${#string}               # Echo the length of the String

# String Slicing
# ${string:START:NUMBER} -> START can be negative
echo ${string:0:4}            # Echo the first 4 characters of the String
echo ${string:0:${#string}-6}  # Echo the string minus the last 6 characters
echo ${string: -6}            # Echo the last 6 characters -> Note the space!
```

Slicing Strings By Pattern (Parameter Expansion)

```
#!/bin/bash
x=abcABC123ABCabc
echo ${x%b*c}   # --> abcABC123ABCabc Delete shortest match from back
echo ${x%%b*c}  # --> a               Delete longest match from back
echo ${x#a*b}   # --> cABC123ABCabc   Delete shortest match from front
echo ${x##a*b}  # --> c               Delete longest match from front
```

A more practical example.

```
#!/bin/bash
file=/user/johndoe/c/interesting/info.txt   # We have the path to our file
path=${file%/*}                              # Return file path
echo ${path}   # --> /user/.../interesting
name=${file##*/}                             # Delete file path, leave file name
echo ${name}   # --> info.txt
end=${name##*.}
echo ${end}    # --> txt
short=${name%%.*}
echo %{short}   # --> info
```

Replacing Text

```
#!/bin/bash
string="This is a long string" # Define a String
echo ${string/i/!}            # Replace first instance of i with !
echo ${string//i/!}           # Replace all instances of i with !
echo ${string/#T/B}           # Replace T at start of string with B
echo ${string/%ng/foo}        # Replace ng at end of string with foo
echo ${string/s*/blah}        # Replace s followed by char(s) blah
```

Command Substitution -> TODO

```
cdir=$(pwd)
#     |___|  -> runs commands
#   |_____| -> specifies to variable
```

Arithmetic Operations

Bash has some limited integer arithmetic operations. Do not use Bash for floating point arithmetic or serious scientific computations. There libraries for handling floating point calculations such as `bc` which you can pipe to if its absolutely necessary (e.g., `echo 1/3 | bc -l`). This, however, is not recommended.

```
#!/bin/bash
val=23
echo $val       # --> val=23

val=$((++val))  # Pre-increment does effect assigned value
echo $val       # --> val=24

val=$((--val))  # Pre-decrement does effect assigned value
echo $val       # --> val=23

aa=$((val++))   # Post-increment does not effect assigned value
echo $aa        # --> aa=23
echo $val       # --> val=24

bb=$((val--))   # Post-decrement does not effect assigned value
echo $bb        # --> bb=24
echo $val       # --> val=23

val=$((val+=3))
echo $val       # --> val=26

val=$((val/=2))
echo $val       # --> val=13

val=$((val*=4))
echo $val       # --> val=52

val=$((val-=2))
echo $val       # --> val=50
```

Comparison Operations

Perform comparisons between two variables.
String Comparisons
`[[ A symbol B ]]` ← Note the spaces, FALSE==1, TRUE==-0!

Symbols

- **<** ← Less than
- **>** ← Greater than
- **<=** ← Less than or equal to
- **>=** ← Greater than or equal to
- **==** ← Equal
- **!=** ← Not equal

Integer Comparison
`[[ A symbol B ]]` ← Note the spaces, FALSE==1, TRUE==-0!

Symbols

- **-lt** ← Less than
- **-gt** ← Greater than
- **-le** ← Less than or equal to
- **-ge** ← Greater than or equal to
- **-eq** ← Equal
- **-ne** ← Not equal

More Complex Conditions
Symbols

- **&&** ← And
- **||** ← Or
- **!** ← Not
- **-z** ← Null?
- **-n** ←Not null?

```
#!/bin/bash
MakeAnExample="TODO"
```

Styling

Coloring Text

You can color and style text in Bash shell for error messages etcetra.
Here is a simple example:

```
#!/bin/bash
echo -e '\033[34;42;4m Blue on Green \033[0m'
```

`-e` informs echo that the following string contains an escape sequence.
`\033` is the escape string other options include `\e`, `\x1B`.
`[34;42;4m` denotes the formating 34=foreground, 42=background, 4=format.
`Blue on Green` denotes the text to be formatted.
`\033` is again an escape string.
`[0m` is a formatting string to return later text to default.
Some more example:

```
#!/bin/bash
echo -e '\e[90;49mGray on Black\e[0m'
echo -e '\e[34;100mBlue on Gray\e[0m'
echo -e '\033[38;34;4mUnderlined Blue on Black\033[0m'
echo -e '\e[34;100;5mBlinking Blue on Gray\e[0m'
```

A full list of coloring and styling options can be found here.

Character Usage

Many of the same characters are used throughout Bash for various purposes that only become obvious based on their context. This section will include commonly used characters and what they may be used for. This is certainly not an exhaustive list.

Braces { }

Braces can be used to:

- Delimit variables unambiguously. `a="Ta"; b="Da"; ab="Foo"; c=${a}${b} not c=ab`.
- Slice strings: `lstring="This That These"; this=${lstring:0:4}`.
- Make substitutions, like sed:
- Use a default value.
- Brace expansion.

Parentheses ( )

Parentheses **( )** can be used to:

- Define an array of values. `x=( 1 2 3 4 5 6 ); echo ${x[3]}`.

Double Parentheses **(( ))** creates a sub-shell and can be used to:

- Define arithmetic operations.
- Allow omission of dollar signs on integer Variables.
- Allow the inclusion of spaces around operators.
- `a=5; b=10; c=$(( a * b )); echo ${c}`

Bracket [ ]

Brackets **[ ]** can be used to:

- Define an if conditions. `x=1; if [[ x -lt 5 ]]; then echo "..."; fi`.
- Call a program name.

Double Brackets **[[ ]]** is very similar to single brackets **[ ]**, but double brackets **[[ ]]** is a bash built-in and is therefore saver and less general than single brackets **[ ]** and should generally be preferred. A more thorough discussion code be found here.