Bash Shell Basics

Bash Shell Basics: References

- http://linuxcommand.org/lc3_lts0020.php
- man bash
- http://www.joshstaiger.org/archives/2005/07/bash_profile_vs.html
- https://coderwall.com/p/u003pa/bash-startup-scripts-on-linux-and-mac-os-x
- http://bencane.com/2013/09/16/understanding-a-little-more-about-etcprofile-and-etcbashrc/
- http://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html#sect_03_01
- http://www.tldp.org/LDP/abs/html/variables.html
- http://www.tldp.org/LDP/abs/html/testconstructs.html
- http://www.tldp.org/LDP/abs/html/testconstructs.html#DBLBRACKETS
- http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html
- http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-7.html
- http://www.tldp.org/LDP/abs/html/functions.html
- http://www.tldp.org/LDP/abs/html/options.html
- http://www.tldp.org/LDP/abs/html/debugging.html

Bash Shell Basics: References

- https://askubuntu.com/questions/29370/how-to-check-if-a-command-succeeded
- https://www.digitalocean.com/community/tutorials/how-to-use-bash-s-job-control-to-manage-foreground-and-background-processes
- https://askubuntu.com/questions/334994/which-one-is-better-using-or-to-execute-multiple-commands-in-one-line
- http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-3.html
- http://www.tldp.org/LDP/abs/html/io-redirection.html
- https://en.wikipedia.org/wiki/Shebang_(Unix)
- Bash Pocket Reference, 2nd Edition, by Arnold Robbins, *Published by Show publisher page link if publisher pages switch is on O'Reilly Media, Inc., 2016*
- Shell Scripting: Expert Recipes for Linux, Bash, and More, by Steve Parker, *Published by Show publisher page link if publisher pages switch is on Wrox, 2011*
- http://www.linuxjournal.com/content/bash-brace-expansion

Bash Shell Basics

- NOTE: there are some differences between the Bash shells in Windows, OS X, and various flavors of Linux.

Bash Shell Basics

- Shells can be interactive − i.e. you type commands in to run
  - By default when we open the bash shell on a Mac or in Cygwin they are interactive shells − and "login" shells
  - A Login shell runs under your current user credentials
- Or non-interactive − a program/script starts it

Interactive Bash Shell Init Files

- You can customize your shell by placing commands into a Bash Shell init file
- System-wide bash init files are in directory /etc
- Files in the /etc folder normally provide global, system-wide settings
- For the bash shell the global init file is “profile” in the /etc directory

Interactive Bash Shell: /etc/profile in Cygwin

```
-bash
/etc>ls profile
profile
/etc>
```

Interactive Bash Shell: /etc/profile

```
JeffsMacBookPro:etc jeffm$ ls profile
profile
JeffsMacBookPro:etc jeffm$
```

Interactive Bash Shell: /etc/profile

Within profile it says the following:

/etc/profile

 # System wide environment and startup programs, for login setup
 # Functions and aliases go in /etc/bashrc

It's NOT a good idea to change this file unless you know what you
are doing. It's much better to create a custom.sh shell script in
/etc/profile.d/ to make custom changes to your environment, as this
will prevent the need for merging in future updates.

Interactive Bash Shell: /etc/bashrc

Within bashrc it says the following:

System wide functions and aliases
Environment stuff goes in /etc/profile

It's NOT a good idea to change this file unless you know what you
 # are doing. It's much better to create a custom.sh shell script in
 # /etc/profile.d/ to make custom changes to your environment, as this
will prevent the need for merging in future updates.

NOTES:
a) my Cygwin bash did not have bashrc
b) My Mac OS X does have bashrc
c) Amazon Linux (installed in Docker) does have bashrc
d) Ubuntu Linux (installed in Docker) does NOT have /etc/bashrc

Bash Shell: /etc/profile.d

Profile.d is a directory where you can place custom initialization scripts for bash

From: http://bencane.com/2013/09/16/understanding-a-little-more-about-etcprofile-and-etcbashrc/

In addition to the setting environmental items the /etc/profile will execute the scripts within /etc/profile.d/*.sh.

If you plan on setting your own system wide environmental variables it is recommended to place your configuration in a shell script within /etc/profile.d.

NOTES:

a) My Cygwin installation has a /etc/profile.d directory
b) My Mac OS X does not have profile.d
c) Amazon Linux (installed in Docker) does have a /etc/profile.d
d) Ubuntu Linux (installed in Docker) has /etc/profile.d

Home Directory

- The “~” refers to your home directory.
  - On a Mac your home directory is under /Users/{Your-Login}
  - In Cygwin is is under /Home/{User-Name}
  - You can type “cd” by itself to get back to your home directory
  - You can us use “~” in a path name from anywhere in the system within a terminal to refer to directories/ files in or under your home directory

Interactive Bash Shell: ~/.bash_profile, ~./bash_login, ~./.profile

- There are several user specific bash initialization files that can reside in your home directory.

- The follow slide is an excerpt from “man bash” which displays information about the init files used by an interactive bash shell

Interactive Bash Shell

When bash is invoked as an **interactive login shell** , or as a non-interactive shell with the --login option, it first reads and executes commands from the file **/etc/profile**, if that file exists.

After reading that file, it looks **for ~/.bash_profile, ~/.bash_login**, and **~/.profile**, in that order, and reads and executes commands from the first one that exists and is readable.

You can run these commands in both the Mac terminal and Cygwin to see that both are login and interactive shells:

```
[[ $- == *i* ]] && echo 'Interactive' || echo 'Not interactive’

shopt -q login_shell && echo 'Login shell' || echo 'Not login shell'
```

Interactive Bash Shell

```
-bash
~>[[ $- == *i* ]] && echo 'Interactive' || echo 'Not interactive'
Interactive
~>shopt -q login_shell && echo 'Login shell' || echo 'Not login shell'
Login shell
~>
```

```
JeffsMacBookPro:~ jeffm$ [[ $- == *i* ]] && echo 'Interactive' || echo 'Not interactive'
Interactive
JeffsMacBookPro:~ jeffm$ shopt -q login_shell && echo 'Login shell' || echo 'Not login shell'
Login shell
```

Bash Basics

- The following pseudo-code from site:
  http://www.thegeekstuff.com/2008/10/execution-sequence-for-bash_profile-bashrc-bash_login-profile-and-bash_logout/

  show the order and logic of how the bash init files are executed for login shells:

```
execute /etc/profile
IF ~/.bash_profile exists THEN
   execute ~/.bash_profile
ELSE
   IF ~/.bash_login exist THEN
      execute ~/.bash_login
   ELSE
      IF ~/.profile exist THEN
        execute ~/.profile
      END IF
   END IF
END IF
```

Login vs Non-Login Interactive Shells

- Summary: Bash documentation states that:
  - Login shells, which are what the Mac terminal and Cygwin bash shells are, look for config files in this order:
    - /etc/profile, ~/.bash_profile, ~./bash_login, ~/.profile
- Non-Login shells read ~/.bash_rc – if it exists
  - You can execute a non-login interactive shell by running “bash” at the command prompt in a Mac or Cygwin shell

Login vs Non-Login Interactive Shells

- Technically speaking, on Linux and Unix systems, opening a new bash terminal can open a non-login interactive shell
- However, by defaults on a Mac or in Cygwin – interactive bash shells are also login shells
- The following slide shows how the initial terminal opened is a login terminal, followed by opening another bash terminal which is not a login terminal

```
~ — -bash                                   /etc — less ◂ man bash
JeffsMacBookPro:~ jeffm$ shopt -q login_shell && echo 'Login shell' || echo 'Not login shell'
Login shell
JeffsMacBookPro:~ jeffm$ bash
JeffsMacBookPro:~ jeffm$  shopt -q login_shell && echo 'Login shell' || echo 'Not login shell'
Not login shell
JeffsMacBookPro:~ jeffm$ exit
exit
JeffsMacBookPro:~ jeffm$ shopt -q login_shell && echo 'Login shell' || echo 'Not login shell'
Login shell
JeffsMacBookPro:~ jeffm$
```

shopt is a shell builtin command to set and unset (remove) various Bash shell options

Notice – the initial Mac terminal is a login shell

Executing the “bash” command, open another Bash shell within the terminal – which is not an login terminal

Running exit, terminates the non-login bash shell and returns us to the login bash shell

```
-bash
~>
~>shopt -q login_shell && echo 'Login shell' || echo 'Not login shell'
Login shell
~>bash
~>shopt -q login_shell && echo 'Login shell' || echo 'Not login shell'
Not login shell
~>exit
exit
~>shopt -q login_shell && echo 'Login shell' || echo 'Not login shell'
Login shell
~>
```

Notice – like the Mac terminal on the previous page, the initial Cygwin terminal is a login shell

Executing the “bash” command, open another Bash shell within the terminal – which is not an login terminal

Running exit, terminates the non-login bash shell and returns us to the login bash shell

Login vs Non-Login Interactive Shells

- Mac and Cygwin also support .bashrc
- .bashrc is run every time you open a non-login terminal

Login vs Non-Login Interactive Shells

- If you get confused about which bash init file to use, site:

  http://www.joshstaiger.org/archives/2005/07/bash_profile_vs.html

  suggests a good technique for Mac and Cygwin:
  - Put your settings in your .bash_rc file
  - “source” (run) your .bash_rc file from your .bash_profile

What can you put in Bash init files?

- Environment variables like PATH, HOSTNAME
- Customized command prompts
- Command alias

- The following slide shows portions of the .bash_profile file from my Mac

What can you put in Bash init files?

```
export EDITOR=mvim
PATH=$PATH:.:/Users/jeffm/NOSQL-Class/MongoDB:/Users/jeffm/NOSQL-Class/MongoDB/mongodb-osx-x86_64-3.2.6/bin:/Applications
```

export is a builtin bash shell command that lets us set shell environment variables that are accessible to all programs run from a shell.

export EDITOR=mvim

This line sets the EDITOR system variable to mvim – a Mac version of VIM (an improvied version of VI for Macs)

PATH=… appends specific directories to the current PATH variable

What can you put in Bash init files?

```
JeffsMacBookPro:~ jeffm$ echo $EDITOR
mvim
JeffsMacBookPro:~ jeffm$ mvim
```

```
[No Name] - VIM1

~
~
~
~
~                    VIM - Vi IMproved
~
~                      version 8.0.52
~                  by Bram Moolenaar et al.
~          Vim is open source and freely distributable
~
~                Help poor children in Uganda!
~          type  :help iccf<Enter>       for information
~
~          type  :q<Enter>               to exit
~          type  :help<Enter>  or  <F1>  for on-line help
~          type  :help macvim<Enter>     for MacVim help
```

What can you put in Bash init files?

```
~ — -bash
JeffsMacBookPro:~ jeffm$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/usr/local/go/bin:.:/U
sers/jeffm/NOSQL-Class/MongoDB:/Users/jeffm/NOSQL-Class/MongoDB/mongodb-osx-x86_
64-3.2.6/bin:/Applications
JeffsMacBookPro:~ jeffm$
```

What can you put in Bash init files?

The image below show some alias's I added into the .bash_profile on my Mac:

```
alias py='python'
alias dk='docker'
alias run='open -a'
```

After making updates to any Bash init file the changes will not be active in the current bash shell. They will be active in any new Bash shells you open.

To make the changes active in the current Bash shell run the "source" command. For example:

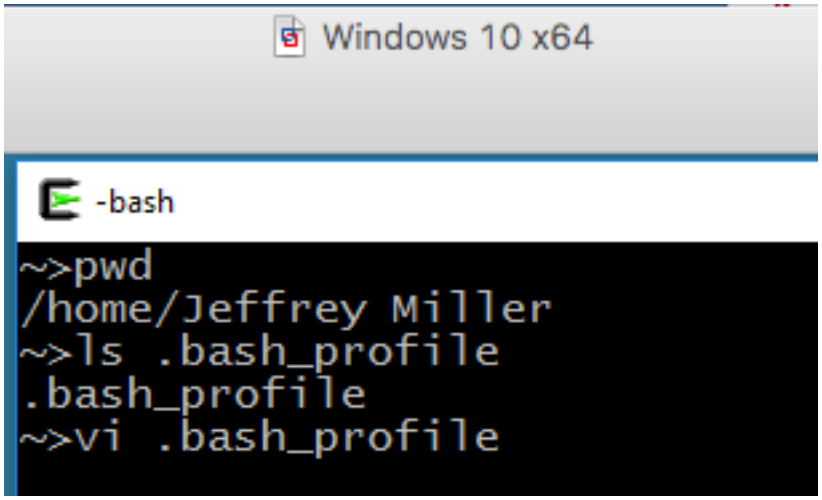
source .bash_profile

What can you put in Bash init files?

```
~ — python
JeffsMacBookPro:~ jeffm$ py
Python 2.7.10 (default, Jul 30 2016, 19:40:32)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

What can you put in Bash init files?

You can make the same Bash shell customizations in the Cygwin Bash shell



What can you put in Bash init files?

Windows 10 x64

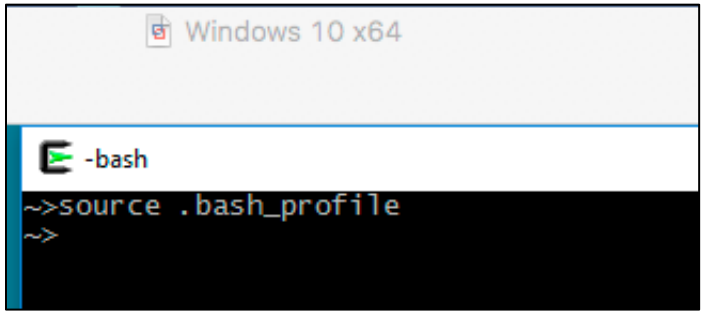
-bash

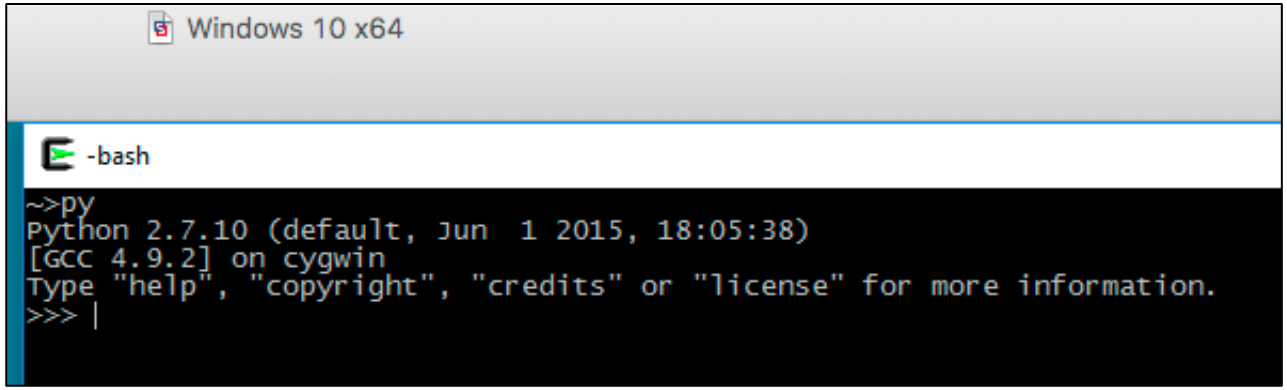
```
# Set PATH so it includes user's private bin
# if [ -d "${HOME}/bin" ] ; then
#   PATH="${HOME}/bin:${PATH}"
# fi

# Set MANPATH so it includes users' private
# if [ -d "${HOME}/man" ]; then
#   MANPATH="${HOME}/man:${MANPATH}"
# fi

# Set INFOPATH so it includes users' private
# if [ -d "${HOME}/info" ]; then
#   INFOPATH="${HOME}/info:${INFOPATH}"
# fi
alias py='python'
```

What can you put in Bash init files?





Bash Command Line Prompts: PS1, PS2

Environment variables: PS1, PS2, PS3, PS4 can be used to customize the Bash command line prompt – place them in a Bash init file (like .bash_profile):

```
PS1='\h:\W \u\$ '
PS2='> '
PS4='+ '
```

When executing interactively, bash displays the primary prompt **PS1** when it is ready to read a command,

and the secondary prompt **PS2** when it needs more input to complete a command.

The following slide from:

http://tldp.org/HOWTO/Bash-Prompt-HOWTO/bash-prompt-escape-sequences.html

list the PS1,PS2 settings you can use:

```
\a      an ASCII bell character (07)
\d      the date  in  "Weekday  Month  Date"  format
        (e.g., "Tue May 26")
\e      an ASCII escape character (033)
\h      the hostname up to the first `.'
\H      the hostname
\j      the  number of jobs currently managed by the
        shell
\l      the basename of the shell's terminal  device
        name
\n      newline
\r      carriage return
\s      the  name  of  the shell, the basename of $0
        (the portion following the final slash)
\t      the current time in 24-hour HH:MM:SS format
\T      the current time in 12-hour HH:MM:SS format
\@      the current time in 12-hour am/pm format
\u      the username of the current user
\v      the version of bash (e.g., 2.00)
\V      the release of bash,  version  +  patchlevel
        (e.g., 2.00.0)
\w      the current working directory
\W      the  basename  of the current working direc
        tory
\!      the history number of this command
\#      the command number of this command
\$      if the effective UID is 0, a #, otherwise  a
        $
\nnn    the  character  corresponding  to  the octal
        number nnn
\\      a backslash
\[      begin a sequence of non-printing characters,
        which could be used to embed a terminal con
        trol sequence into the prompt
\]      end a sequence of non-printing characters
```

```
JeffsMacBookPro:Code jeffm$ echo $PS1
\h:\W \u\$
JeffsMacBookPro:Code jeffm$ echo $PS2
>
```

Your instructor's settings for PS1 are:

\h – hostname/machine-name
: - add a colon after hostname
\W – name of current directory
a space
\u – name of current user
\$ - if user-id is 0 use #, otherwise $
a user-id of zero is root

Bash Command Line Prompts: PS1, PS2

On my system, PS1 was set in /etc/bashrc:

```
# System-wide .bashrc file for interactive bash(1) shells.
if [ -z "$PS1" ]; then
   return
fi

PS1='\h:\W \u\$ '
```

if [-z "$PS1"];

checks to see if $PS1 is zero length (not set)

Bash Commands

Bash commands return success or failure. Zero is success, non-zero is failure.

Built-in environment variable $? stores the value returned from a command.

The following image shows a simple scenario that illustrates command success or failure.

```
JeffsMacBookPro:ubuntu jeffm$ ls .. > cats.txt
JeffsMacBookPro:ubuntu jeffm$ ls cat.txt
ls: cat.txt: No such file or directory
JeffsMacBookPro:ubuntu jeffm$ echo $?
1
JeffsMacBookPro:ubuntu jeffm$ ls cats.txt
cats.txt
JeffsMacBookPro:ubuntu jeffm$ echo $?
0
```

Bash Commands

Multiple commands can be issued on the same line.

#runs cmd-2 regardless of cmd-1; cmd-2

cmd-a**;** cmd-b

#runs cmd-2 only of cmd-1 was successful

cmd-1 **&&** cmd-2

#runs cmd-2 if cmd-1 fails

cmd-1 **||** cmd-2

runs cmd-1 in the background (more on this later on)
cmd-1 &

The following slide demonstrates running multiple commands

Bash Standard Input, Output, Error

The Bash shell has 3 "standard" places you can take input from and send out out too.

stdout −called standard out − is the terminal you are running in by default
stdout is referenced as "**2**"

stdin − called standard input − is , by default what you type into the current terminal − i.e. the keyboard

stderr − where error messages are displayed

stdout, stdin, stderr can be "redirected" to come from or got to different places using **I/O redirection**

Bash I/O Redirection

> redirects the output of a command into a file

>> redirects the output of a command into a file, if the file exists, the file is appended to

```
JeffsMacBookPro:ubuntu jeffm$ ls > lsout.txt
JeffsMacBookPro:ubuntu jeffm$ cat lsout.txt
cats.txt
lsout.txt
JeffsMacBookPro:ubuntu jeffm$ ls .. >> lsout.txt
JeffsMacBookPro:ubuntu jeffm$ cat lsout.txt
cats.txt
lsout.txt
amazonlinux
cassandra
d2
mongo
py01
ubuntu
```

stdout is assigned "1" in the shell. You can also use the commands in this format:

ls 1> lsout.txt

…

ls 1>>lsout.txt

Bash I/O Redirection

2> redirects the error of a command into a file

2>> redirects the output of a command into a file, if the file exists, the file is appended to

```
JeffsMacBookPro:ubuntu jeffm$ ls file-does-not-exists.txt 2> err.txt
JeffsMacBookPro:ubuntu jeffm$ cat err.txt
ls: file-does-not-exists.txt: No such file or directory
JeffsMacBookPro:ubuntu jeffm$ ls and-this-file-does-not-exists.txt 2>> err.txt
JeffsMacBookPro:ubuntu jeffm$ cat err.txt
ls: file-does-not-exists.txt: No such file or directory
ls: and-this-file-does-not-exists.txt: No such file or directory
```

see http://www.tldp.org/LDP/abs/html/io-redirection.html for more details about Bash shell I/O redirection

Bash I/O Redirection

You can also use I/O redirection, <, with **stdin:**

```
JeffsMacBookPro:ubuntu jeffm$ cat < cats.txt
amazonlinux
cassandra
d2
mongo
py01
ubuntu
```

Bash Commands

```
JeffsMacBookPro:ubuntu jeffm$ ls .. > cats.txt
JeffsMacBookPro:ubuntu jeffm$ ls cat.txt; echo '2nd cmd'
ls: cat.txt: No such file or directory
2nd cmd
JeffsMacBookPro:ubuntu jeffm$ ls cat.txt && echo '2nd cmd'
ls: cat.txt: No such file or directory
JeffsMacBookPro:ubuntu jeffm$ ls cat.txt || echo '2nd cmd'
ls: cat.txt: No such file or directory
2nd cmd
```

The ">" redirection operator is used. It takes the output of the command on the left and places it into whatever is in the right. For this example, the output of the ls command is placed into file cats.txt.

Bash Job Control

- When using Bash (and Linux/Unix) processes/ commands can run in the foreground, the background, be suspended, resumes
- The Bash shell also contains several job/ process control commands
- To start a process – run any command in a Bash shell

Bash Job Control

- A process and be **terminated** from the shell using **control-c** – if that process was launched from the same shell

```
JeffsMacBookPro:ubuntu jeffm$ ping -i 5 ucsc.edu
PING ucsc.edu (128.114.109.5): 56 data bytes
64 bytes from 128.114.109.5: icmp_seq=0 ttl=44 time=17.728 ms
64 bytes from 128.114.109.5: icmp_seq=1 ttl=44 time=23.598 ms
^C
--- ucsc.edu ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 17.728/20.663/23.598/2.935 ms
```

Bash Job Control

- A process can be **suspended** using **control-z**
- A **suspended process** can be restarted using **fg**
- A **suspended process** can be restarted in the background using **bg**

Bash Job Control

```
JeffsMacBookPro:ubuntu jeffm$ ping -i 3 ucsc.edu
PING ucsc.edu (128.114.109.5): 56 data bytes
64 bytes from 128.114.109.5: icmp_seq=0 ttl=44 time=110.339 ms
64 bytes from 128.114.109.5: icmp_seq=1 ttl=44 time=37.738 ms
^Z
[1]+  Stopped                 ping -i 3 ucsc.edu
JeffsMacBookPro:ubuntu jeffm$ fg
ping -i 3 ucsc.edu
64 bytes from 128.114.109.5: icmp_seq=2 ttl=44 time=31.063 ms
64 bytes from 128.114.109.5: icmp_seq=3 ttl=44 time=134.794 ms
^C
--- ucsc.edu ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 31.063/78.483/134.794/44.985 ms
JeffsMacBookPro:ubuntu jeffm$
```

Bash Job Control

- The **jobs** command can be used to list all stopped and background processes
- The kill command terminates a processed using a process id (see ps below) or %n when "n" is the number on the jobs command output
- The number listed next to the jobs command output can be used with fg, bg, and kill
- fg and bg used without an arguments operates on the last process you put into the background/foreground

Bash Job Control

```
JeffsMacBookPro:ubuntu jeffm$ ping -i 8 google.com
PING google.com (172.217.4.142): 56 data bytes
64 bytes from 172.217.4.142: icmp_seq=0 ttl=47 time=47.602 ms
^Z
[1]+  Stopped                 ping -i 8 google.com
JeffsMacBookPro:ubuntu jeffm$ ping -i 8 yahoo.com
PING yahoo.com (98.138.253.109): 56 data bytes
64 bytes from 98.138.253.109: icmp_seq=0 ttl=41 time=73.613 ms
^Z
[2]+  Stopped                 ping -i 8 yahoo.com
JeffsMacBookPro:ubuntu jeffm$ jobs
[1]-  Stopped                 ping -i 8 google.com
[2]+  Stopped                 ping -i 8 yahoo.com
JeffsMacBookPro:ubuntu jeffm$ fg
ping -i 8 yahoo.com
64 bytes from 98.138.253.109: icmp_seq=1 ttl=41 time=65.860 ms
^Z
[2]+  Stopped                 ping -i 8 yahoo.com
JeffsMacBookPro:ubuntu jeffm$ jobs
[1]-  Stopped                 ping -i 8 google.com
[2]+  Stopped                 ping -i 8 yahoo.com
JeffsMacBookPro:ubuntu jeffm$ kill %1
[1]-  Terminated: 15          ping -i 8 google.com
JeffsMacBookPro:ubuntu jeffm$ kill %2
[2]+  Terminated: 15          ping -i 8 yahoo.com
JeffsMacBookPro:ubuntu jeffm$ jobs
JeffsMacBookPro:ubuntu jeffm$
```

fg %1
will bring the 1st process in the jobs list into the foreground (not shown)

kill %2
terminates the 2nd processes in the jobs list

Bash Job Control

- A process can be run in the **background** using **&**
- When a process is in the background, it output will still appear in the terminal.
- However, you can still enter commands into to terminal

Bash Job Control

```
JeffsMacBookPro:ubuntu jeffm$ ping -i 10 yahoo.com &
[1] 34764
PING yahoo.com (98.139.180.149): 56 data bytes
JeffsMacBookPro:ubuntu jeffm$ 64 bytes from 98.139.180.149: icmp_seq=0 ttl=39 time=105.990 ms

JeffsMacBookPro:ubuntu jeffm$ jobs
[1]+  Running                 ping -i 10 yahoo.com &
JeffsMacBookPro:ubuntu jeffm$ kill %1
[1]+  Terminated: 15          ping -i 10 yahoo.com
```

Bash Job Control

```
JeffsMacBookPro:ubuntu jeffm$ ping -i 20 yahoo.com &
[1] 34841
PING yahoo.com (98.139.180.149): 56 data bytes
JeffsMacBookPro:ubuntu jeffm$ 64 bytes from 98.139.180.149: icmp_seq=0 ttl=39 time=97.252 ms

JeffsMacBookPro:ubuntu jeffm$ ping -i 20 google.com &
[2] 34859
JeffsMacBookPro:ubuntu jeffm$ PING google.com (172.217.4.142): 56 data bytes
64 bytes from 172.217.4.142: icmp_seq=0 ttl=47 time=32.856 ms

JeffsMacBookPro:ubuntu jeffm$ jobs
[1]-  Running                 ping -i 20 yahoo.com &
[2]+  Running                 ping -i 20 google.com &
JeffsMacBookPro:ubuntu jeffm$ kill %64 bytes from 98.139.180.149: icmp_seq=1 ttl=39 time=94.220 ms

kill: usage: kill [-s sigspec | -n signum | -sigspec] pid | jobspec ... or kill -l [sigspec]
JeffsMacBookPro:ubuntu jeffm$ kill %2
[2]+  Terminated: 15          ping -i 20 google.com
JeffsMacBookPro:ubuntu jeffm$ kill %1
[1]+  Terminated: 15          ping -i 20 yahoo.com
JeffsMacBookPro:ubuntu jeffm$ jobs
```

The **jobs** command lists suspended and background processes associated with the current terminal

Processes can be terminated with the **kill** command. You can use "kill %n" where "n" the the number listed in the jobs command. For example **kill %2** above.

Bash Job Control

- The **ps** command is used to list running processes.

- The **pid** value from the **ps** command can be used with the **kill** command to terminate a process.

```
ubuntu — -bash — 64×11
~/DevOps-Tech/docker/ubuntu — -bash
JeffsMacBookPro:ubuntu jeffm$ ping -i 20 yahoo.com
PING yahoo.com (98.138.253.109): 56 data bytes
64 bytes from 98.138.253.109: icmp_seq=0 ttl=41 time=107.794 ms
Terminated: 15
JeffsMacBookPro:ubuntu jeffm$
```

```
ubuntu — -bash — 65×11
~/DevOps-Tech/docker/ubuntu — -bash
JeffsMacBookPro:ubuntu jeffm$ ps -e | grep ping
35660 ttys001    0:00.00 ping -i 20 yahoo.com
35666 ttys003    0:00.00 grep ping
JeffsMacBookPro:ubuntu jeffm$ kill 35660
JeffsMacBookPro:ubuntu jeffm$
```

ping was run in the top terminal

The output of the **ps −e** command was sent into **grep ping** to limit the output of ps −e.

The "**|**" is called a **pipe**. It takes the out of the command on the left and "pipes" into the command on the right as it's input.

kill 35660 was used to terminate the process running ping

I/O Redirection,Pipes, Job Control Summary

< redirect stdin

> redirect stdout , >> redirect and append stdout ,

2> redirect stderr , **2>>** redirect and append stderr

| pipes the output of one command into another command as its input

control-c – terminate a process

control-z – suspend

jobs – list processes associated with current terminal

fg , fg %n, fg pid – bring a background or suspended process into the foreground

bg, bg %n, bg pid – put foreground or suspended process into background

kill %n, kill pid – terminate a process

Shell Builtins

- The bash shell has a set of internal commands called “builtins”. Following is a partial list of these. For a complete list see:

http://www.tldp.org/LDP/abs/html/internal.html#BUILTINREF

Shell Builtins

- Builtins:
  - I/O: echo, printf, read
  - File system: cd, pwd, pushd, popd, dirs

Shell Builtins

- Builtins:
  - Variable evaluation: let
    - eval: The eval command can be used for code generation from the command-line or within a script. For example:

```
06_Bash-01.sh cats dogs txt

y=`eval ls *.$3`
echo $y   #linefeed not included
echo "$y" #linefeed included
```

Shell Builtins

- Builtins:
  - Variable evaluation continued:
    - set, unset
    - export − export the value of a variable as an environment variable from the script
    - getopts − command line parsing
    - source
      - Execute a script (typically used for init scripts)
    - . filename − similar to a #include statement

Shell Builtins

- Builtins:
  - Scripts
    - exit
    - exec − execute another command from script and exit
    - shopt − change shell options (see Shell options)
    - caller − used in a script function to print the caller to stdout
  - See http://www.tldp.org/LDP/abs/html/internal.html#BUILTINREF for more details

Shell Settings

- The Bash shell has many different settings you can use.

```
set: set [--abefhkmnptuvxBCHP] [-o option] [arg ...]
        -a  Mark variables which are modified or created for export.
        -b  Notify of job termination immediately.
        -e  Exit immediately if a command exits with a non-zero status.
        -f  Disable file name generation (globbing).
        -h  Remember the location of commands as they are looked up.
        -k  All assignment arguments are placed in the environment for a
            command, not just those that precede the command name.
        -m  Job control is enabled.
        -n  Read commands but do not execute them.
        -o option-name
            Set the variable corresponding to option-name:
                allexport    same as -a
                braceexpand  same as -B
                emacs        use an emacs-style line editing interface
                errexit      same as -e
                errtrace     same as -E
                functrace    same as -T
                hashall      same as -h
                histexpand   same as -H
                history      enable command history
:
```

help set | more

There are many, many settings.

options are enabled using:

set −X …
#where X is the option

set +X …
#turns off a shell feature/option

Shell Settings

The current settings can be found in the "$-" environment variable

```
JeffsMacBookPro:devops-notes jeffm$ echo $-
himBHT
```

-h Remember the location of commands as they are looked up. This causes Bash to remember the locations of previous commands/programs it has run, instead of searching $PATH again

-i Is interactive shell

-B – next slide

-m Job control is enabled. Allows use of job control (bg, fg) from the shell

-H Enable ! style history substitution. Set section below

-T If set, the DEBUG trap is inherited by shell functions. Provides enlaced debugging support (not covered in class)

Shell Settings

-B the shell will perform brace expansion

Bash brace expansion is used to generate stings at the command line or in a shell script.

#brace-expand.sh
#!/bin/sh

echo first-param-expansion=$1
echo second-param-expansion=$2
echo **{aa,bb,cc}**
echo **{0..11}**
echo **{10..-5}**
echo **{a..m}**
echo **{z..l}**
echo cat-**{5..8}** and **{8..4}**-mouse

Shell Settings

```
~/DevOps-Tech/devops-notes/Code — -bash
JeffsMacBookPro:Code jeffm$ brace-expand.sh /Users/jeffm/DevOps-Tech/devops-docker/py01/{py01.sh,py02.sh}
first-param-expansion=/Users/jeffm/DevOps-Tech/devops-docker/py01/py01.sh
second-param-expansion=/Users/jeffm/DevOps-Tech/devops-docker/py01/py02.sh
aa bb cc
0 1 2 3 4 5 6 7 8 9 10 11
10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5
a b c d e f g h i j k l m
z y x w v u t s r q p o n m l
cat-5 cat-6 cat-7 cat-8 and 8-mouse 7-mouse 6-mouse 5-mouse 4-mouse
JeffsMacBookPro:Code jeffm$
```

brace-expand.sh /Users/jeffm/DevOps-Tech/devops-docker/py01/**{py01.sh,py02.sh}**

Bash Command Line History

- Bash can save a list of the commands you run as "history"
- The "fc" command can be used to find and edit "history"

Bash Command Line History

```
JeffsMacBookPro:py01 jeffm$ help fc
fc: fc [-e ename] [-nlr] [first] [last] or fc -s [pat=rep] [cmd]
    fc is used to list or edit and re-execute commands from the history list.
    FIRST and LAST can be numbers specifying the range, or FIRST can be a
    string, which means the most recent command beginning with that
    string.

       -e ENAME selects which editor to use.  Default is FCEDIT, then EDITOR,
          then vi.

       -l means list lines instead of editing.
       -n means no line numbers listed.
       -r means reverse the order of the lines (making it newest listed first).

    With the `fc -s [pat=rep ...] [command]' format, the command is
    re-executed after the substitution OLD=NEW is performed.

    A useful alias to use with this is r='fc -s', so that typing `r cc'
    runs the last command beginning with `cc' and typing `r' re-executes
    the last command.
```

You can also use the up and down arrow keys in most shells to scroll through command history

Bash Command Line History

```
~/DevOps-Tech/devops-docker/py01
JeffsMacBookPro:py01 jeffm$ fc -l -10
505      mn history
506      c
507      man history
508      c
509      help fc
510      c
511     *
512      c
513      fc -l 10
514      c
```

#list the last 10 commands
fc −l -10

#edit command 505 in vi
fc −e vi 505

Bash Command Line History

```
~/DevOps-Tech/devops-docker/py0
JeffsMacBookPro:py01 jeffm$ fc -l -10
505      mn history
506      c
507      man history
508      c
509      help fc
510      c
511     *
512      c
513      fc -l 10
514      c
```

```
py01 — vi /var/folders/9m/9gyxss
~/DevOps-Tech/devops-docker/py01 — vi /var/
man histo
~/DevOps-Tech/devops-docker/py01 — vi /var/folders/9m/9gyxsscj3h5494pdbgh4lys40000gn/T//bash-fc-1499691684
~
~
~
-- INSERT --
```

#edit command 505 in vi
fc −e vi 505

```
JeffsMacBookPro:py01 jeffm$ fc -l -5
512      c
513      fc -l 10
514      c
515      fc -l -10
516      man history
```

after editing and saving the command, the command became the latest command and was executed

Bash Command Line History

```
JeffsMacBookPro:py01 jeffm$ fc -l -5
512      c
513      fc -l 10
514      c
515      fc -l -10
516      man history
```

Using "!', you can run previous commands

```
JeffsMacBookPro:py01 jeffm$ ls
Dockerfile      cmd02.sh        cmd04.sh        py01.sh         py02.sh
cmd01.sh        cmd03.sh        import.sh       py01.tar
JeffsMacBookPro:py01 jeffm$ !ls D*
ls D*
Dockerfile
```

!ls – run the last command that "ls" was used in

Bash Command Line History

```
JeffsMacBookPro:py01 jeffm$ ls
Dockerfile      cmd02.sh        cmd04.sh        py01.sh         py02.sh
cmd01.sh        cmd03.sh        import.sh       py01.tar
JeffsMacBookPro:py01 jeffm$ cat py01.sh
#!/bin/sh

docker run -it buildpack-deps-jessie:py01 /bin/sh
JeffsMacBookPro:py01 jeffm$ ls
Dockerfile      cmd02.sh        cmd04.sh        py01.sh         py02.sh
cmd01.sh        cmd03.sh        import.sh       py01.tar
JeffsMacBookPro:py01 jeffm$ !ca
cat py01.sh
#!/bin/sh

docker run -it buildpack-deps-jessie:py01 /bin/sh
```

In this example: "!ca" ran the last command with substring "ca" in it

Bash Command Line History

```
JeffsMacBookPro:py01 jeffm$ fc -l -4
532      ls
533      cat py01.sh
534      fc -l -4
535      c
JeffsMacBookPro:py01 jeffm$ !!
fc -l -4
533      cat py01.sh
534      fc -l -4
535      c
536      fc -l -4
```

"!!" runs the last command

Shell Settings: Command Line Editing

A useful Bash shell settings is:

```
#set command line editing mode to emacs
set −o emacs
```

or

```
#set command line editing mode to vi
set −o vi
```

Shell Settings: Command Line Editing

As we saw above, you can use

fc −e vi LineNumber

to bring up VI to edit:

```
JeffsMacBookPro:py01 jeffm$ fc -l -8
586      brace-expand.sh /Users/jeffm/DevOps-Tech/devops-docker/py01/{py01.sh,py02.sh}
587      help set
588      echo $-
589      cd ..
590      cd docker
591      ls
592      cd py01
593      c
JeffsMacBookPro:py01 jeffm$ fc -e vi 586
```

Shell Settings: Command Line Editing

You can also do in-line editing of command history by hitting the escape key.

The following chart from − Bash Shell Pocket Reference, 2nd Edition, Arnold Robbins, O'Reilly shows some of the in-line editing commands for both vi and emacs (next slide):

vi	Emacs	Result
k	CTRL-p	Get previous command
j	CTRL-n	Get next command
/*string*	CTRL-r *string*	Get previous command containing *string*
h	CTRL-b	Move back one character
l	CTRL-f	Move forward one character
b	ESC-b	Move back one word
w	ESC-f	Move forward one word
x	DEL	Delete previous character
x	CTRL-d	Delete character under cursor
dw	ESC-d	Delete word forward
db	ESC-h	Delete word backward
xp	CTRL-t	Transpose two characters

Bash shell in-line editing commands for VI and Emacs.

Hit the escape key to enter line-edit mode. The editor used will be what is set in:

set −o vi

or set −o emacs

```
JeffsMacBookPro:py01 jeffm$ brace-expand.sh /Users/jeffm/DevOps-Tech/devops-docker/py01/{py01.sh,py02.sh}
```

up arrow keys was used to get to command
'b' was hit 7 times to move cursor to point depicted above

```
JeffsMacBookPro:py01 jeffm$ brace-expand.sh /Users/jeffm/DevOps-Tech/devops-docker/py01/{py02.sh}
```

dw was hit 4 times to delete py01.sh

```
JeffsMacBookPro:Code jeffm$ brace-expand.sh /Users/jeffm/DevOps-Tech/devops-docker/py01/{py02.sh}
first-param-expansion=/Users/jeffm/DevOps-Tech/devops-docker/py01/{py02.sh}
second-param-expansion=
aa bb cc
0 1 2 3 4 5 6 7 8 9 10 11
10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5
a b c d e f g h i j k l m
z y x w v u t s r q p o n m l
cat-5 cat-6 cat-7 cat-8 and 8-mouse 7-mouse 6-mouse 5-mouse 4-mouse
```

hit return when finished editing to execute the command

Shell Settings: Command Line Editing

Summary VI Command Line Editing Commands:

- k -get previous command
- j - get next command
- /string - get previous command contains string
- h - move backwards 1 character
- l - move forwards 1 character
- b - move back one word
- w - move forward one word
- X - delete previous character
- x - delete character under cursor
- dw - delete forward one word
- db - delete backwards one word
- xp - tranpose two characters

Command Line Editing

- When you start spending a lot of time using the Bash shell command line, learning how to use its basic line editing features can be very useful.

Writing Bash Scripts

- Bash scripts have

  #!/bin/bash

  or

  #!/bin/sh

  as the first line. This is called "shebang" or "hashbang"

- NOTE: PowerPoint turns a regular double quote into a reverse double quote unless your turn off "smart quotes". So if you copy and paste from your instructor's pdfs, some of the commands with quotes may not work.

Writing Bash Scripts

- No spaces are permitted on either side of the = sign in an assignment statement!
- $variable-name or ${variable-name} to reference the value of a variable

```
#!/bin/bash
a=11
echo $a
#The next line will cause an error if uncommented
#b= 12
```

Writing Bash Scripts

- You can place a variable reference within double quotes:

```
b=abc
echo "the value of b is $b”
```

- Placing a variable reference within single quotes produces a literal value

```
c='Cats Like Stuff'
echo $c
echo 'Cats $c'
```

Writing Bash Scripts

- Bash variables are un-typed − basically they are strings
- However, Bash supports arithmetic operations/ comparisons if the string value in the variable only contains digits

Writing Bash Scripts

- Math expressions use:
  - expr expressions
  - ` back-quotes
  - ((…)) double parens
  - Double quotes
  - $[…] − square brackets
  - let

Writing Bash Scripts

```
d=100 #assignment from literal value
e=$d  #assignment from another variable

f1=$(expr $e + 100)
f2=`expr $e + 101`
f3=$(($e + 102))
let f4=f3+500
let "f5 = f4 + 600"

echo "d is $d, e is $e, f1 is $f1, f2 is $f2, f3 is $f3, f4 is $f4, f5 is $f5"
echo " -- $f3 + 1"
echo $(($f3 + 2))
echo $((f3 + 2))
echo $[$f3 + 3]
echo $[f3 + 3]
```

Writing Bash Scripts

- Variables can also be assigned via command substitution using back-quotes or $(…):

```
fn=`ls *.sh`
echo "sh files in cur-dir: $fn "

curdir=$(pwd)
echo "cur-dir=$curdir"
```

Writing Bash Scripts

- Scripts can access environment variables:

  echo "current-user is **$USER**”

- You can type – set – by itself in a bash terminal to see all environment variables in the shell’s process

Writing Bash Scripts

- Parameters can be passed into a Bash script by position:
  - $0 is the name of the script
  - $1 is the first parameter
  - $2 the 2nd
  - etc.
    - this works for $1 to $9 for more than 10 command line arguments use ${n} – for example ${10}
  - $* or $@ references all parameters
  - $# is the number of arguments
  - ${!#} is the last argument
  - $? Is the status of the last command executed, 0 is OK,

Writing Bash Scripts

- For example: 06_Bash-01.sh cats dogs

```
echo "arguments are: $1 , $2"
echo "all args are: $*”
echo "number of arguments is $#"
echo "last arg is ${!#}”
```

- To test for the existence of a positional parameter:

Writing Bash Scripts

- To test for the existence of a positional parameter:

  For example: 06_Bash-01.sh cats dogs

```
#the spaces in [ -z $2 ] and [ -z $3 ] are needed
if [ -z $2 ]
then
  echo "missing 2dn parameter"
else
  echo 'have 2nd parameter'
fi

if [ -z $3 ]
then
  echo "no 3rd parameter"
fi
```

Writing Bash Scripts

- Tests and Conditionals
  - If/then tests as follows
    - A zero return is true
    - A 1 is false
    - Square brackets [ ] is used to test comparisons or on file expressions (i.e. exists, not-exists)
    - This is what we did in the previous example:

```
if [ -z $3 ]
```

Writing Bash Scripts

- Tests and Conditionals
  - The […] has many limitations
  - The [[…]] construct (which is newer)

Using the **[[...]]** test construct, rather than **[...]** can prevent many logic errors in scripts. For example, the &&, ||, <, and > operators work within a [[ ]] test, despite giving an error within a [ ] construct.

Writing Bash Scripts

- Tests and Conditionals
  - If you do not like using […] or [[…]] with if statements you can use “test”:

```
if test -z $3
then
  echo "still no 3rd parameter"
fi

#to put statement on same line use ;
if test -z $3; then echo "not yet 3rd parameter"; fi
```

Writing Bash Scripts

- If/then also supports:
  - elif [condition]
  - else
  - ends with fi

Writing Bash Scripts

- If statements have a set of built in expression that can be used to test files, strings, and more. The next slide has a partial list.
- Reference

  http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

  has a complete list.

Writing Bash Scripts

- If statement expressions
  - [-a FILE] True if FILE exists
  - [-d FILE] True if FILE exists and is a directory.
  - [-e FILE] True if FILE exists.
  - [-f FILE] True if FILE exists and is a regular file.
  - [-h FILE] True if FILE exists and is a symbolic link.
  - [-r FILE] True if FILE exists and is readable.
  - [-w FILE] True if FILE exists and is writable.
  - [-x FILE] True if FILE exists and is executable.
  - [-N FILE] True if FILE exists and has been modified since it was last read

Writing Bash Scripts

- If statement expressions continued
  - [FILE1 -nt FILE2] True if FILE1 has been changed more recently than FILE2, or if FILE1 exists and FILE2 does not.
  - [FILE1 -ot FILE2] True if FILE1 is older than FILE2, or is FILE2 exists and FILE1 does not.
  - [FILE1 -ef FILE2] True if FILE1 and FILE2 refer to the same device and inode numbers

  - [-z STRING] True of the length if "STRING" is zero.
  - [-n STRING] or [STRING] True if the length of "STRING" is non-zero.
  - [STRING1 == STRING2] True if the strings are equal. "=" may be used instead of "==" for strict POSIX compliance.
  - [STRING1 != STRING2] True if the strings are not equal.
  - [STRING1 < STRING2] True if "STRING1" sorts before "STRING2" lexicographically in the current locale.
  - [STRING1 > STRING2] True if "STRING1" sorts after "STRING2" lexicographically in the current locale.

Writing Bash Scripts

- If statement expressions continued
  - [ARG1 OP ARG2] "OP" is one of -eq, -ne, -lt, -le, -gt or -ge. These arithmetic binary operators return true if "ARG1" is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to "ARG2", respectively. "ARG1" and "ARG2" are integers

Writing Bash Scripts

- If statement examples from: http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

```
if [ -f /var/log/messages ]
  then
    echo "/var/log/messages exists."
fi
```

Writing Bash Scripts

- If statement examples continued:

```
#file that does not exist
ls backupzz
If [ $? −ne 0 ] ;  then echo "does not exist" ;  fi

num=200
if [ "$num" −gt "199"]; then echo "greater than"; fi


if [ "$(whoami)" != 'root' ]; then
      echo "You are not root user."
      exit 1;
fi
```

Writing Bash Scripts

- Loops: for

```
for i in $( ls ); do
  echo item: $i
done

for i in `seq 1 10`;
do
  echo $i
done
```

Writing Bash Scripts

- Loops: while, until

```
COUNTER=0
while [  $COUNTER -lt 10 ]; do
   echo The counter is $COUNTER
   let COUNTER=COUNTER+1
Done

COUNTER=20
until [  $COUNTER -lt 10 ]; do
   echo COUNTER $COUNTER
   let COUNTER-=1
done
```

Writing Bash Scripts

- Functions – see http://www.tldp.org/LDP/abs/html/functions.html for more details
  - Syntax:

```
function_name() {
  …
}
```

Writing Bash Scripts

Function example:

```
sayhello() {
  echo Hello
}

sayhello
```

Writing Bash Scripts

- Function example with parameters:
- Bash shell functions do NOT list their parameters in the function
- Parameters passed to functions are access like command line parameters as $1, $2, etc.

Writing Bash Scripts

- Functions with arguments example:

```
func01() {
  echo "$# arguments passed in"
  if [ -n "$1" ]
  then
    echo "1st parameter is: \"$1\" "
    if [ -n "$2" ]
    then
      echo "2nd parameter is: \"$2\"  "
    fi
  fi
}

func01
func01 Cats
func01 "mice are nice" 27
```

Writing Bash Scripts

- Shell Options
- The shell can be started with several options, or these options can be turned on and off with set, unset
- The following slide lists a few options
- See reference http://www.tldp.org/LDP/abs/html/options.html for more details

Writing Bash Scripts

- Shell Options

-a allexport Export all defined variables
-e errexit Abort script at first error, when a command exits with non-zero status (except in until or while loops, if-tests, list constructs)
-n noexec Read commands in script, but do not execute them (syntax check)
-v verbose Print each command to stdout before executing it
-x xtrace Similar to -v, but expands commands

Writing Bash Shell Scripts

- Debugging Techniques

  See http://www.tldp.org/LDP/abs/html/debugging.html for more details

Writing Bash Shell Scripts

- Debugging Techniques Continued
- Insert echo commands to monitor script
- When piping multiple commands together use "tee" to see the output of intermediate commands in the pipeline

Writing Bash Shell Scripts

- Debugging Techniques Continued
- Echo out $LINENO in the script:

```
echo "***** $LINENO ***"
func01
func01 Cats
func01 "mice are nice" 27
```

Writing Bash Shell Scripts

- Debugging Techniques Continued

- Use the "caller" builtin:

```
func01() {
    caller #print out script name and line #
    …
}
```