

# bash Tips &> Tricks

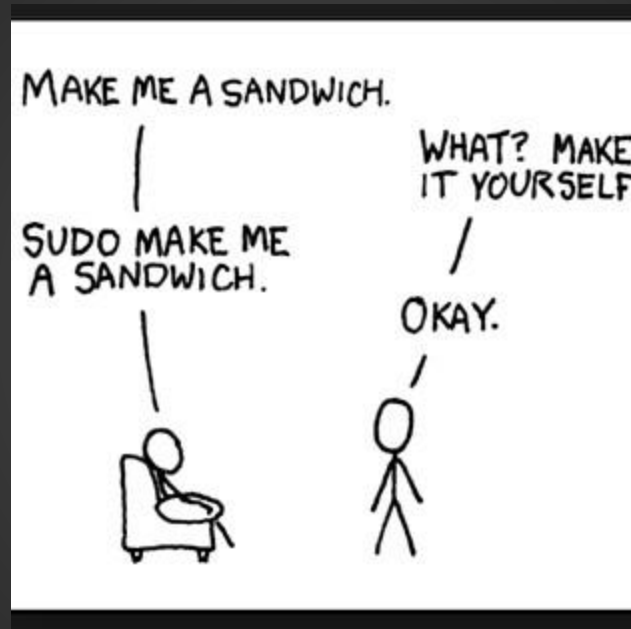
Using the shell for fun and profit!

Brian Gerard  
bgerard@gmail.com  
August, 2014

# What We'll Talk About Today

- Some Useful Commands and Options
- History Manipulation
- Gettin Loopy
- Aliases
- Functions

# Commands to Know and Love.



# Useful Commands and Options:

**man(1)** and **apropos(1)**

# Learn how to learn

```
bash$ man man
```

# Learn a heck of a lot more than you're

# going to in today's class

```
bash$ man bash
```

# See if there are any other man pages that

# would have useful info on the subject

```
bash$ apropos bash
```

# Useful Commands and Options:

## **man(1)** and **apropos(1)**

# Q: What do those numbers up there (↑)

# actually mean?

# A: It's the section of the manual where

# you will find the page in question.

# Find out about the user command printf(1)

```
bash$ man 1 printf
```

# Find out about the C standard library

# function named printf(3)

```
bash$ man 3 printf
```

# Useful Commands and Options:

## grep (1)

# Show the lines where a pattern occurs

```
bash$ grep pattern filename
```

# ...or the ones where it doesn't

```
bash$ grep -v pattern filename
```

# ...or anywhere in all the files under a

# directory; "r" == recursive

```
bash$ grep -r pattern directory
```

# Search case-insensitively

```
bash$ grep -i pattern filename
```

# Useful Commands and Options:

## grep (1)

# Show the lines where a pattern occurs,

# preceded by the line numbers

```
bash$ grep -n pattern filename
```

# Only report the \*count\* of lines on which

# a pattern occurs in the file; only counts

# lines, not instances of the pattern

```
bash$ grep -c pattern filename
```

# Useful Commands and Options:

## grep (1)

# The ABC's of grep...

# Show matching lines, plus two lines **A**fter

```
bash$ grep -A 2 pattern filename
```

# Show matching lines, plus two lines **B**efore

```
bash$ grep -B 2 pattern filename
```

# Show matching lines, with two lines before

# \*and\* two after (**C** == **C**ontext)

```
bash$ grep -C 2 pattern filename
```



# Useful Commands and Options:

## grep(1)

```
# grep(1) also has siblings it's worth  
# getting to know:
```

```
# Search for a fixed string (no regexes).  
# Can be much faster if you don't really  
# need a regex match (but -i still works)  
bash$ fgrep string filename
```

```
# Search for an extended regex  
bash$ egrep uberPattern filename
```

# Useful Commands and Options:

## **find(1)**

# All items with "something" in their names

```
bash$ find /somewhere -name '*something'
```

# All files larger than 2 MiB in size

```
bash$ find /somewhere -size +2M
```

# Or both...

```
bash$ find /somewhere -size +2M \  
      -a -name '*something'
```

# Useful Commands and Options:

## **find(1)**

# Things modified within the past day...

```
bash$ find /somewhere -mtime -1
```

# ...or those modified *\*more\** than a day ago

```
bash$ find /somewhere -mtime +0
```

# Stuff modified more recently than "foo"

```
bash$ find /somewhere -newer /path/to/foo
```

# Run something on each entry it finds

```
bash$ find /somewhere -name '*something*' \  
      -exec chmod 644 '{}' \;
```

# Useful Commands and Options:

**find(1)**

# Leading us inevitably to...

```
bash$ find /your_base -name '*' \  
      -exec chgrp us '{}' \;
```

# Useful Commands and Options:

## xargs (1)

```
# xargs(1) takes stdin and constructs a  
# command from it.
```

```
# Run grep(1) on a list of files  
bash$ cat file_list | xargs grep -H pattern
```

```
# Basically the same as running  
bash$ grep pattern file1 file2 file3 ...
```

```
# And more efficient than using find(1)
```

```
bash$ find . -name 'file*' \  
      -exec grep -H pattern '{}' \;
```

# Useful Commands and Options:

## **xargs (1)**

```
# The -n option tells xargs to only use a  
# certain number of items from stdin for  
# each constructed command
```

```
bash$ cat host_list | xargs -n 1 host
```

```
# Looks up each host in DNS, one at a time.  
# Without the '-n 1', host(1) would take the  
# first host as the name to look up, the  
# second as the DNS server to use, and would  
# simply be very confused about being given  
# anything more than that.
```

# Useful Commands and Options:

## xargs (1)

```
# The -I option allows xargs to Insert the  
# values it pulls from stdin somewhere other  
# than at the end of the command
```

```
bash$ cat files_to_copy | \  
    xargs -I THEM /bin/cp THEM /somewhere
```

```
# Useful with find(1), for example...
```

```
bash$ find /active -mtime +30 | \  
    xargs -I THEM /bin/mv THEM /archive
```

# Useful Commands and Options:

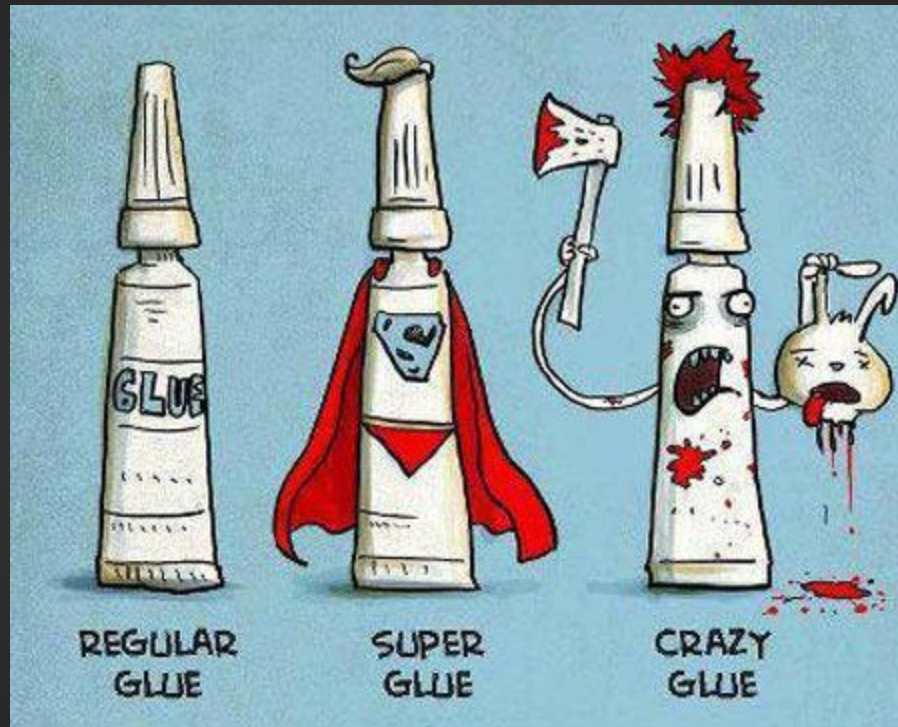
## xargs (1)

```
# While we're on the subject, *if* you use
# the find+xargs construct, you need two
# more options for safety's sake:
# find's -print0 (that's print<zero>)
# xargs's -0 (that's -<zero>), a.k.a. --null
bash$ find /my/logs -mtime +30 \
    -a -type f -print0 | xargs -0 /bin/rm

# Ensures that find(1) separates its results
# with NULLs, and that xargs(1) looks for a
# NULL to delimit its incoming arguments.
```



# Putting the Pieces Together



# Say I wanna do something cool on a buncha hosts...

```
# Ok, here we go...
```

```
bash$ ssh host1 'do_something --cool'
```

```
bash$ ssh host2 'do_something --cool'
```

```
...
```

```
bash$ ssh hostN 'do_something --cool'
```

**Ugh. That really should have been a bit less... labor intensive.**

# Time to mess around with history!



## REVISIONIST HISTORY

TAKE THAT HISTORY CHANNEL!!!

[motifake.com](http://motifake.com)

# Revisionist History, Part One: Command Line Editing

```
# Default keybindings are emacs; let's  
# change that to the One True Editor™  
bash$ set -o vi
```

```
# So I hit the first host...  
bash$ ssh host1 'do_something --cool'  
# ...and now type (minus the spaces):  
# <esc> k w cw host2 <enter>  
# ...which will end up running:  
bash$ ssh host2 'do_something --cool'
```

# Revisionist History, Part... Two ...of Part One?: Command Line Editing

```
# There's also the 'fc' (fix command)
```

```
# shell built-in.
```

```
bash$ fc
```

```
# Yep, that's all there is to it. It will
```

```
# open your previous command in $EDITOR.
```

```
# When you exit, whatever you've changed
```

```
# that command to will then be executed.
```

```
# And I lied. There can be more to it than
```

```
# that. 'man bash' is your friend.
```

# Revisionist History, Part Two: Quick History Substitution

```
bash$ ssh host1 'do_something --cool'
```

```
# In order to re-run the previous command on  
# the next host...
```

```
bash$ ^1^2^
```

```
ssh host2 'do_something --cool'
```

```
# ...or just plain ^1^2 - the end '^' is  
# optional unless you want to append  
# something to the resulting command.
```

# How about another way...

```
## Quick substitution, alternate form.
```

```
bash$ ssh host1 'do_something --cool'
```

```
# Re-run the previous command on the next
```

```
# host...
```

```
bash$ !!:s/1/2/
```

```
ssh host2 'do_something --cool'
```

```
# And again, the trailing '/' is optional if
```

```
# you don't need to append anything to the
```

```
# resulting command.
```



# That seems a bit clunky; what does it give us?

```
## Quick substitution, but not on the  
## immediately previous command:
```

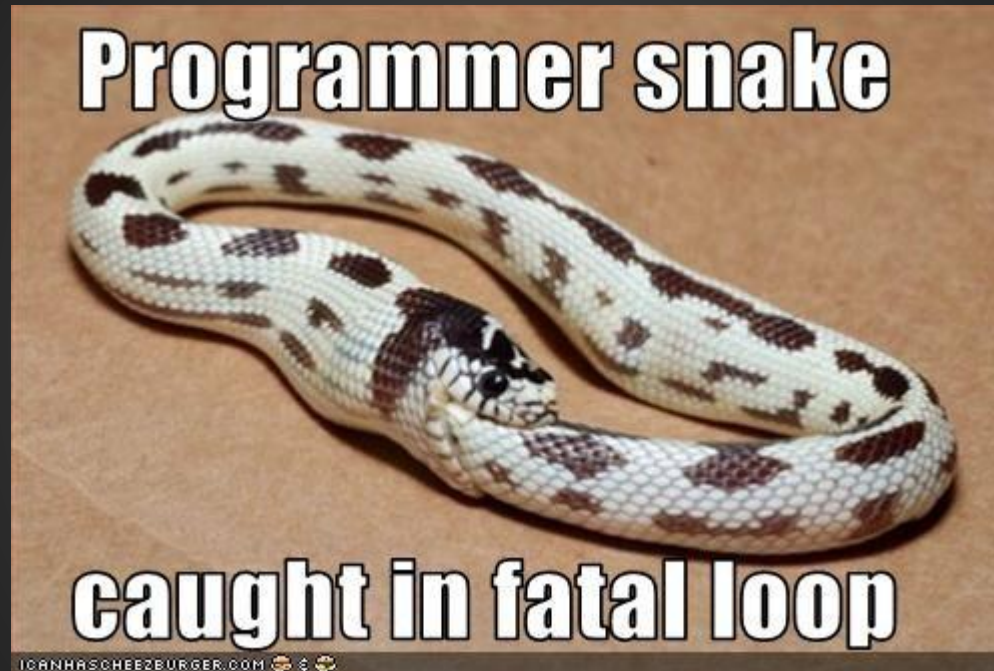
```
# ...later on in the shell session...  
bash$ history | grep 'ssh host1 '  
    1234  ssh host1 'do_something --cool'
```

```
# Re-execute command number 1234, going to  
# a new host this time.
```

```
bash$ !1234:s/host1/a-new-host/  
ssh a-new-host 'do_something --cool'
```

**Cool, but isn't that *still* a bit manual?**

# Good News; Bash Has Loops!



# Gettin All Loopy

```
# There's while...
bash$ while (true); do
> echo "hello world"
> sleep 1
> done
hello world
hello world
hello world
^C
```

# Gettin All Loopy

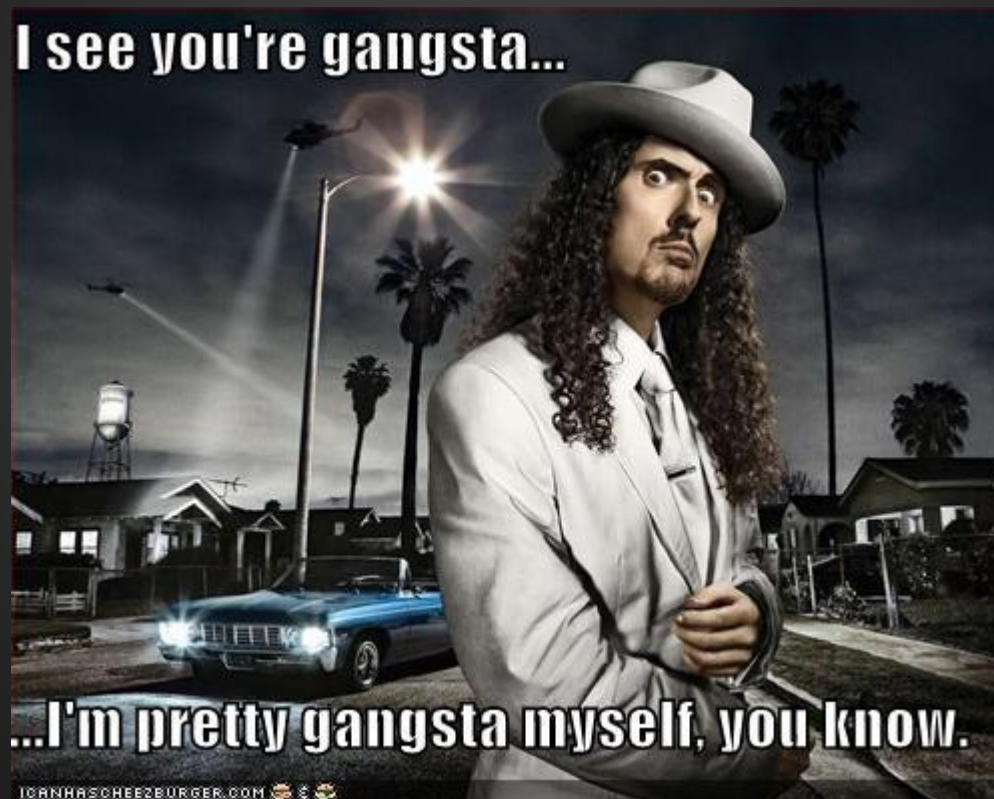
```
# ...and also for...  
bash$ for item in ...list...; do  
> echo $item  
> done  
thing1  
thing2  
...
```

# Ooo... that looked promising.

```
# A for loop's list can just be the contents
# of a file, broken on whitespace, and so...
bash$ for host in $(cat list_of_hosts.txt)
> do
> echo "Running on $host"
> ssh $host 'do_something --cool'
> done
Running on host1
Running on host2
...
```

**...but what if I want to do that  
again sometime?**

# Aliases: They're Not Just for Gangsters Anymore!





# What Is a Bash Alias?

```
# An alias is a string which will be  
# substituted for a word when that word  
# appears first in a simple command
```

```
bash$ alias sayfoo='echo $foo'
```

```
bash$ type sayfoo
```

```
sayfoo is aliased to `echo $foo`
```

```
bash$ sayfoo
```

```
bash$ foo=bar
```

```
bash$ sayfoo
```

```
bar
```

# What Is a Bash Alias?

```
# It is worth noting that aliases are only  
# expanded once, so something like this...  
bash$ alias ls='ls -AFC'  
# ...will not result in an infinite  
# re-expansion.
```

```
# And this, given the declaration above  
bash$ ls /foo  
# Is exactly the same as this, without it  
bash$ ls -AFC /foo
```

# Putting That to Good Use

```
# So we can turn that loop into a command
# that can be used again whenever needed.
bash$ alias do_cool_stuff='for host in $(cat
list_of_hosts.txt); do echo "Running on
$host"; ssh $host "do_something --cool";
done'

bash$ type do_cool_stuff
do_cool_stuff is aliased to `for host in
$(cat list_of_hosts.txt); do echo "Running
on $host"; ssh $host "do_something --cool";
done'
```

# Step 3: Profit!

```
# And now...
```

```
bash$ do_cool_stuff
```

```
Running on host1
```

```
Running on host2
```

```
...
```

```
Running on hostN
```

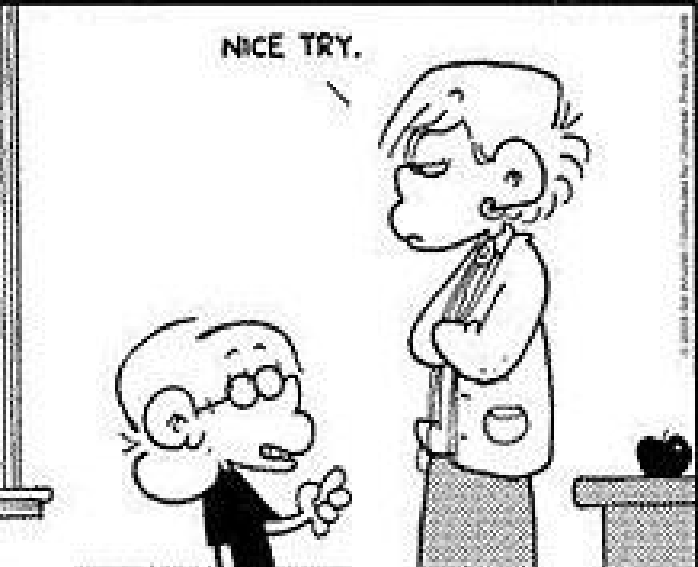
**Any other ways to do that?**

# Even Better - Functions!

```
#include <stdio.h>
int main(void)
{
    int count;

    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

WMD 10-3



# Function Version of do\_cool\_stuff

```
# Translating...
bash$ function do_cool_stuff () {
> for host in $(cat list_of_hosts.txt); do
> echo "Functioning on $host"
> ssh $host "do_something --cool"
> done
> }
bash$
```

# Function Version of do\_cool\_stuff

```
# Here goes...  
bash$ do_cool_stuff  
Running on host1  
Running on host2  
...  
Running on hostN
```



# What the wha?



# Aliases are expanded before functions

```
# Meh.
```

```
bash$ type do_cool_stuff
```

```
do_cool_stuff is aliased to `for host in  
$(cat list_of_hosts.txt); do echo "Running  
on $host"; ssh $host "do_something --cool";  
done`
```

```
# Must fix.
```

```
bash$ unalias do_cool_stuff
```

# **\*Now\* let's see how it looks**

```
bash$ type do_cool_stuff
do_cool_stuff is a function
do_cool_stuff ()
{
    for host in $(cat list_of_hosts.txt);
    do
        echo "Functioning on $host";
        ssh $host "do_something --cool";
    done
}
```

# Aaaah... much better.

```
# One mo time!  
bash$ do_cool_stuff  
Functioning on host1  
Functioning on host2  
...  
Functioning on hostN
```

**But why use functions rather than aliases?**

# 1) Arguments!



# Positional Parameters

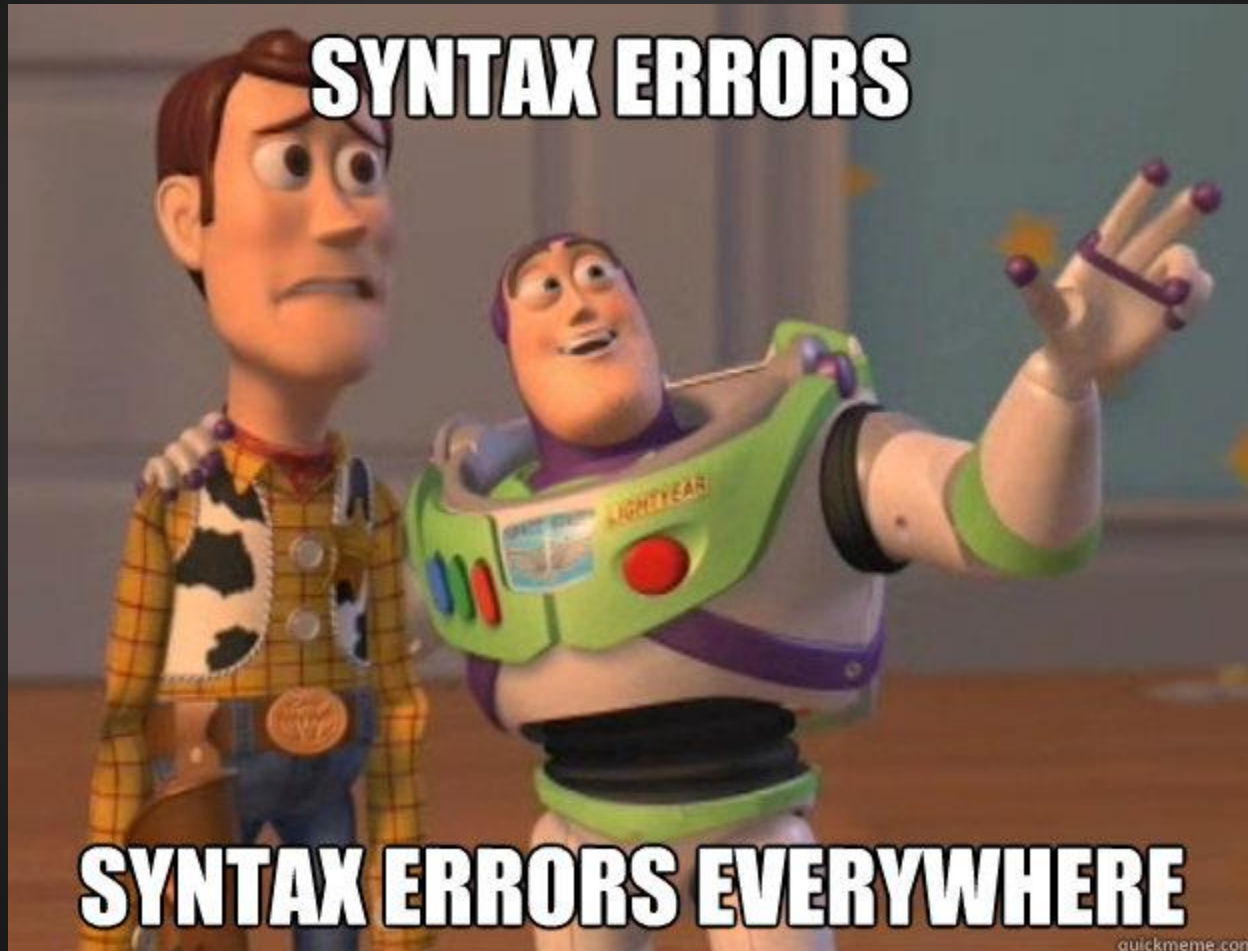
```
# We can modify the function to pick a host
bash$ function do_cool_stuff () {
> if [[ $# -gt 0 ]]
> then the_hosts=($@)
> else the_hosts=$(cat list_of_hosts.txt)
> fi
> for host in ${the_hosts[*]}; do
> echo "Functioning on $host"
> ssh $host "do_something --cool"
> done
> }
```

# Positional Parameters

```
# And that means we can do this...
bash$ do_cool_stuff host1 host4
Functioning on host1
Functioning on host4
# While this still does the default...
bash$ do_cool_stuff
Functioning on host1
Functioning on host2
...
Functioning on hostN
```



## 2) Syntax Checked While You Wait



# bash will catch some errors when you define your function

# such as a missing semicolon

```
bash$ function bad_syntax_yo () {  
> echo "This is okay"  
> for i in 1 2 3 4; do  
> echo "Not ok - no semicolon here ->" done  
> }
```

```
bash: syntax error near unexpected token `}'
```

```
bash$ type bad_syntax_yo
```

```
bash: type: bad_syntax_yo: not found
```

# Some stuff still slips through

```
# Missing command? Runtime error.
bash$ function good_syntax_bad_command () {
> echo "This is okay"
> this is not a command
> }
bash$ good_syntax_bad_command
This is okay
-bash: this: command not found
bash$
```

### 3) Output Redirection



# Bash Redirection

# bash makes it easy to redirect stdout...

```
bash$ ls -l /var/log > list_of_logfiles
```

# ...stderr...

```
bash$ grep -r foo /etc 2> /dev/null
```

# ...or both...

```
bash$ my.script &> ~/output.and.errors
```

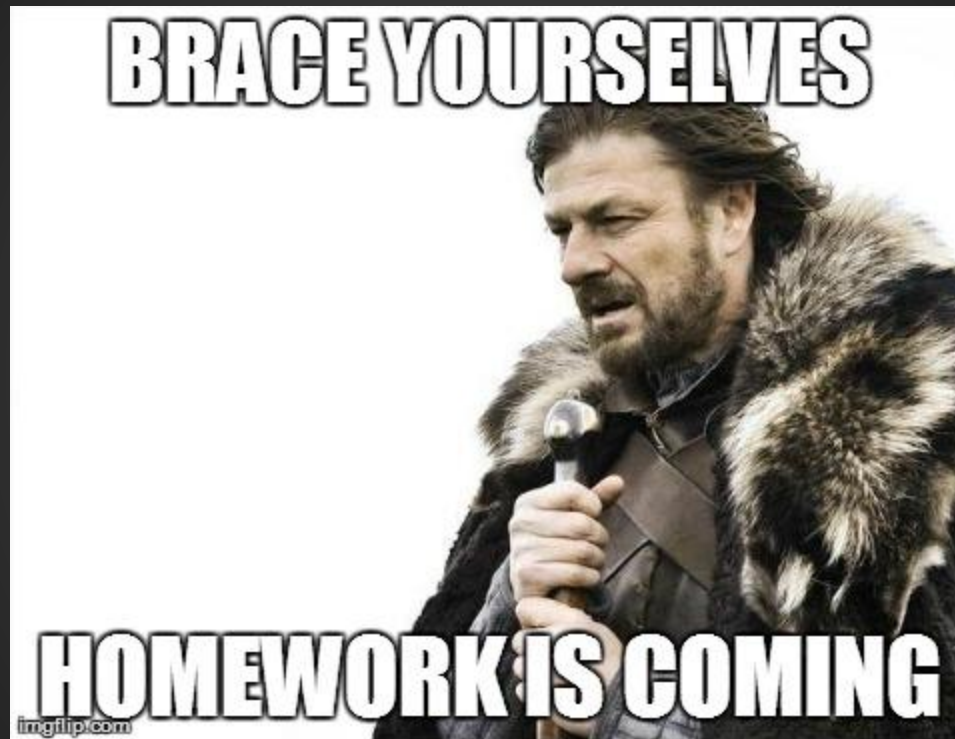
# (same thing)

```
bash$ my.script > ~/output.and.errors 2>1
```

# Redirections can be part of the function definition

```
bash$ function here_n_there () {  
> echo "This goes to stdout"  
> ls /no/such/file  
> } > ~/herethere.out 2> ~/herethere.err  
bash$ here_n_there  
bash$ cat ~/herethere.out  
This goes to stdout  
bash$ cat ~/herethere.err  
ls: cannot access /no/such/file: No such  
file or directory
```

# Stuff What Was Skipped



# Left As an Exercise to the Reader:

## Arrays

```
bash$ my_list=(first second third)
```

```
bash$ echo ${my_list[1]}
```

```
second
```

```
bash$ hosts=$(cat my.host.list)
```

```
bash$ echo ${hosts[3]}
```

```
host4
```

```
bash$ echo ${hosts[${#hosts}]}]
```

```
hostN
```



# Left As an Exercise to the Reader:

## Conditionals

```
bash$ if [[ ${my_list[1]} = "second" ]]
> then echo "yep, it is second"
> else echo "no, it is not"
> fi
```

yep, it is second

```
bash$ if [[ ${#my_list} -ge 42 ]]
> then echo "big list"
> else echo "leetle list"
> fi
```

leetle list

# Your Mission, Should You Choose to Accept It...

Look through your own history (by running 'history') and see what kinds of things you repeat regularly.

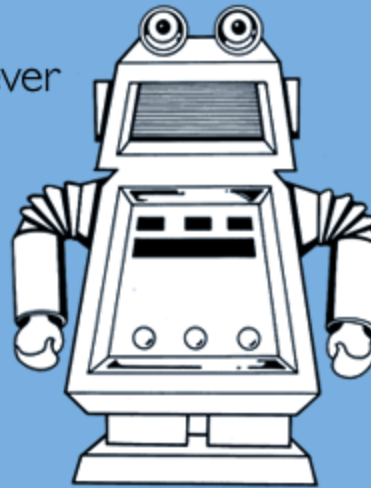
What would make a good alias or function?

Pick one, turn it into one or the other, test it, tweak it until you're happy. Then add it to your `.bashrc` so you have it handy when you log in.

Lather, rinse, repeat.

# Thank You!

Thank you, blah,  
blah...something clever  
and hilarious, blah.



your  cards  
someecards.com

# References

The `bash(1)` man page, natch.

Read through these sections for more:

- ALIASES
- FUNCTIONS
- REDIRECTION
- CONDITIONAL EXPRESSIONS

Also feel free to peruse `~bgerard/.bashrc` on pilot.

Cheat sheets, sample code, etc:

[https://github.com/briangerard/bash\\_class](https://github.com/briangerard/bash_class)

[https://github.com/briangerard/my\\_env](https://github.com/briangerard/my_env)