Bulletproof Bash

Best practices for writing shell scripts

Sean Fisk
SEA 2014

Writing Scripts

- Script: a small program

GNU Bash is an interactive shell designed for UNIX-like operating systems.

Ubiquity:
- Standard install
- Compiles on a wide variety of OSes
- Commonly used

Aside from those:
- Standard implementation
- Extra features
- Generally faster than sh
- sh is frequently a more POSIX-compatible version of bash

A very easy way to script simple operations which require calling many external executables.

Bash syntax can get pretty ugly sometimes:
- Based on symbols rather than words
- Difficult to remember
- Hard to read
- Minor mistakes cause major headaches

Ultimately too inflexible to use for complex programs.

As I will talk about in a future presentation, complex Bash scripts should be converted to high-level general purpose programming languages like Python or Ruby.

The hash-bang instructs the operating system how to run this file.

Similar to Windows' "Open with…" and associated applications.
Less global, but in many ways more flexible.
Only works with "scripts", aka text files.

Use /usr/bin/env. Locates the first executable of that name on the PATH. PATH will be explained later.

Next, we'll talk about some necessary Bash boilerplate.

A simple Bash script. Now what if the directory ‘idontexist’, as implied, doesn’t exist? Will ‘hello there’ be echoed? Let’s try it!

Why is this: Bash doesn’t have exception handling.

Error Handling: What would Python do?

```
#!/usr/bin/env python

from __future__ import print_function
import os

print(*os.listdir('idontexist'), sep=' ')
print('hello there')
```

- Another simple and rather innocuous-looking Bash script.
- Remove the users' bin/ directories.
- Kinda mean ☺
- Scripts like this not recommended.
- Don't type this one.
- What would happen?

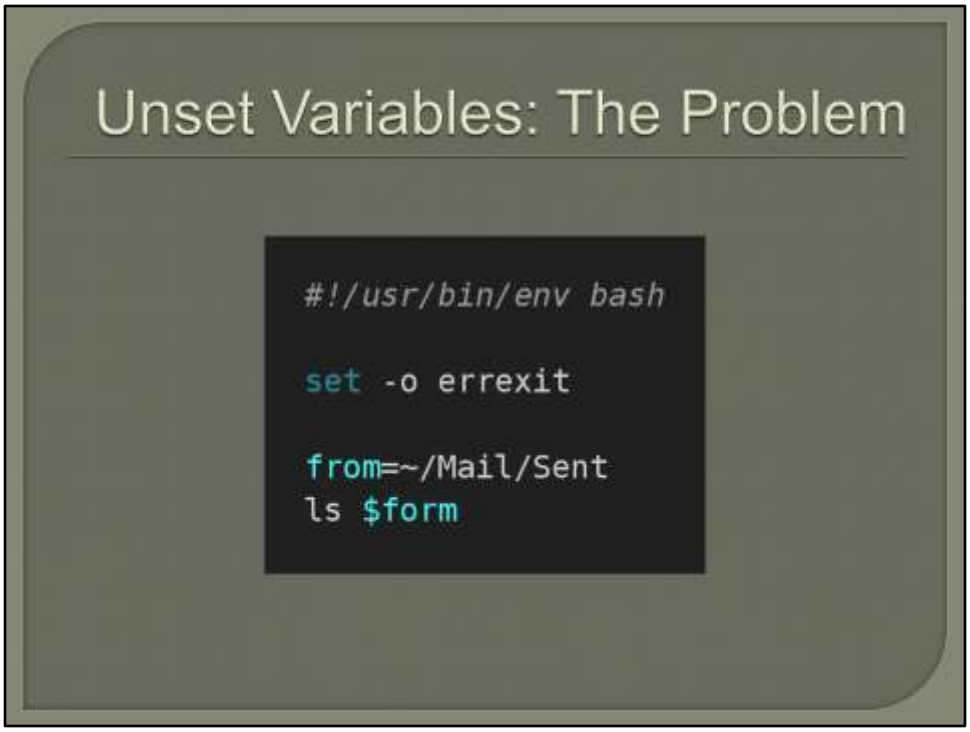
Error Handling: Cataclysm!

```
#!/usr/bin/env bash

for username in jacob hannah brad; do
    cd "/home/$username"
    rm -r bin
done
```

```
root@mybox:/# ls
...       bin
root@mybox:/# ls /home
jacob     brad
root@mybox:/# /root/script
cd: no such file or directory: /home/hannah
rm: bin: No such file or directory
rm: bin: No such file or directory
root@mybox:/# ls
...
root@mybox:/# # Oh no!
```

Error Handling: Solution

```
#!/usr/bin/env bash

set -o errexit

ls idontexist
echo 'hello there'
```



In Bash, you can create variables. Bash doesn't really have any data types, though… they are all just strings.

What happens when you refer to a variable that doesn't exist?

Do you see the mistake?

Unset Variables: Catastrophe!

```
#!/usr/bin/env bash

set -o errexit

readonly PREFIX=~/my-software/my-project
rm -r "$PREFI/etc"
```

```
root@mybox:/# ls
...       etc
root@mybox:/# ls /root/my-software/my-project
lib       etc       ...
root@mybox:/# /root/script
root@mybox:/# ls
...
root@mybox:/# # Whoops!
```

These basically do the same thing. So why use the double brackets? The main reason is quoting. The double brackets auto-quote their arguments and are recommended by almost every authority on Bash.

Quoting: Problems

```
#!/usr/bin/env bash

set -o errexit
set -o nounset

directory='My Documents'     # good
ls $directory                # bad

ls '$directory/Sent Faxes'   # nope

ls [ThisIsAnnoying]          # not what you think
```

Quoting: When

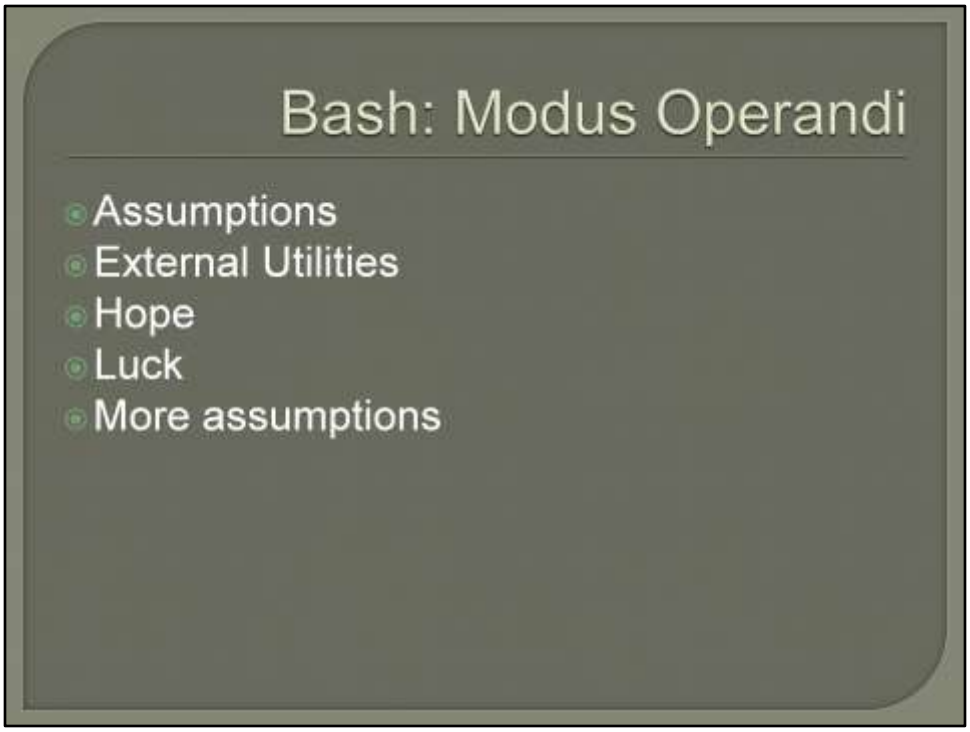
- Spaces in file names
- Double and single quotes
- Shell meta-characters
- Impossibility

Quoting: Solutions

```
#!/usr/bin/env bash

set -o errexit
set -o nounset

directory='My Documents'     # good
ls "$directory"              # yes

ls "$directory/Sent Faxes"   # awesome

ls '[ThisIsAnnoying]'        # perfect
```

User's Home Directory

- `$HOME` and ~ (tilde) are not the same thing!
- Tilde uses `$HOME` if set, otherwise looks up the user's home directory.
- Always use tilde.

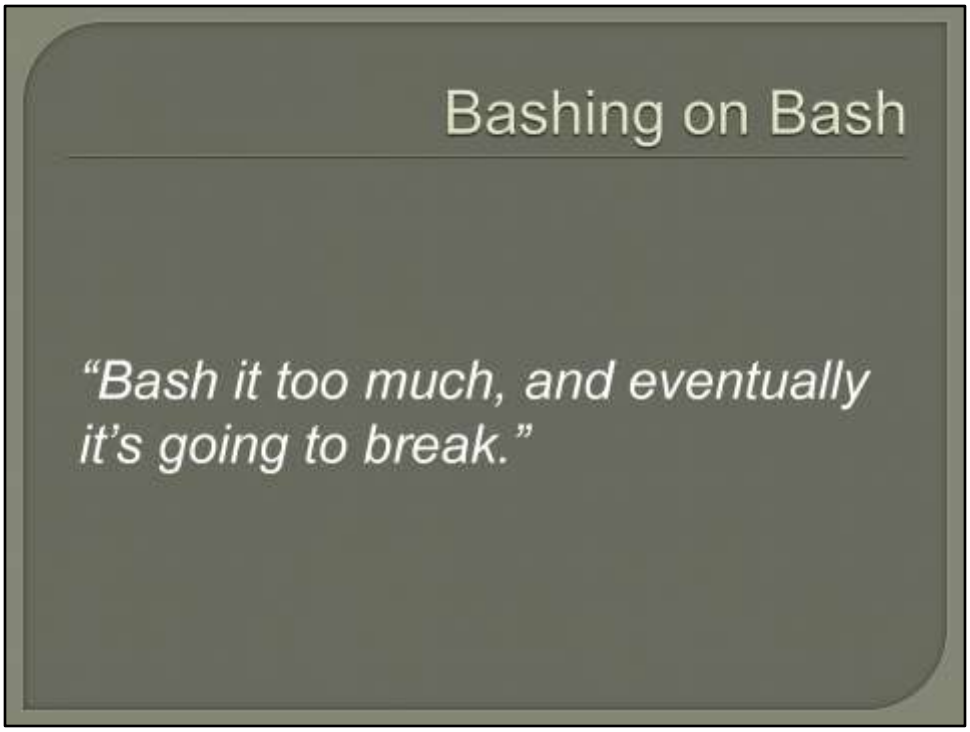


Now that we've gone through this, what makes Bash scripts run? I'll tell you.

Bash doesn't prevent you from doing things incorrectly, but it certainly doesn't help you.

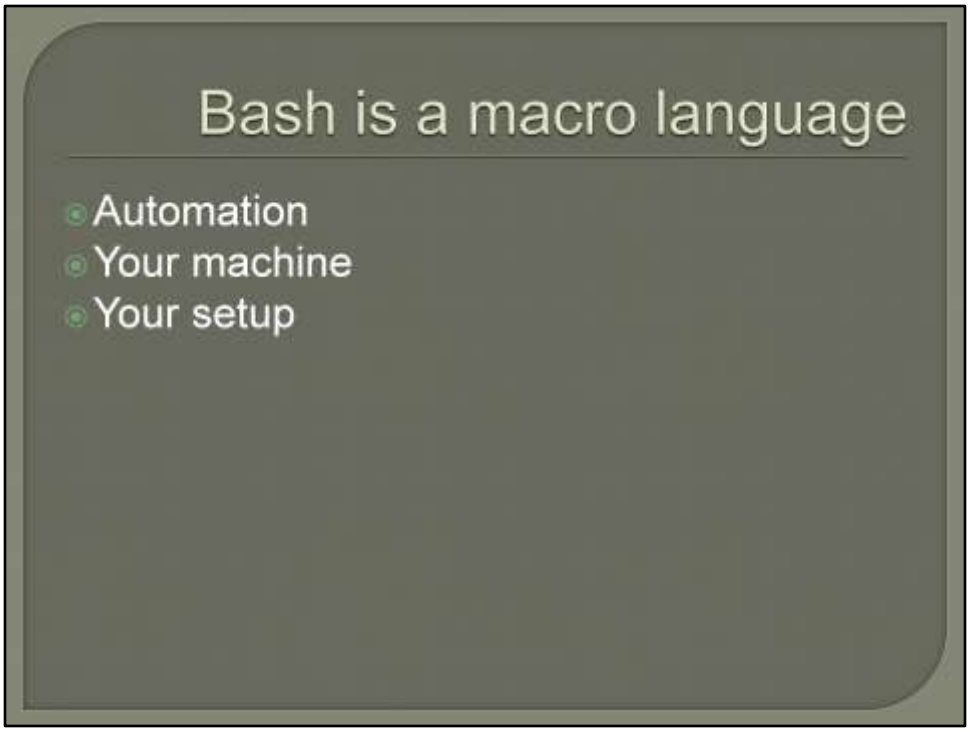
Assumptions:

Correct version
Utilities present on system
Files that exist
Paths that exist
Machine-specific configurations



When your task is critical… don't use Bash.
When your task involves lots of possible conditions… don't use Bash.
When you need dependable error handling… don't use Bash.
When you are using lot's of non-standard utilities… don't use Bash.
When your process has many steps… don't use Bash.

Key to remember… Bash is a macro language.



Bash is most useful for automating processes that are typically performed manually.

Your scripts may be specific to your own machine, your company's setup, your distribution, or your own configuration.

Keep these in mind always, but especially with Bash.

Conclusions

- Know when to use Bash.
- Know when to use something else.
- When you use Bash, follow these best practices.

Bash Bible:
`http://www.gnu.org/software/bash/manual/bash.html`