Target Audience: intermediate to advanced bash scripters
Expectations/Pre-Requisites

- Have written scripts Using
  - for
  - while
  - echo
  - cat
  - grep
  - if-elif-else-fi
  - stream forwarding 1>&2, 2>&1, etc.
- Have written or managed at least one script over 100 lines.

- preferred:
  - awk
  - sed
  - tr
  - cut
  - sort
  - uniq
  - wc
  - case

- bonus:
  - jq
About Me: Michael Knapp

• Senior Software engineer for Capital One, since 2016.
• CKA and CKAD
• AWS certified Systems Associate.
• Have written many scripts for job.
• **Is Not** biased in favor of bash.
• **Is** biased in favor of automating *everything*. 
Agenda

• Discuss Challenge/Practice Code
• Why Use Bash?
• Bash IDE Tools
• What is a process?
• Common simplifications
• Variable Scope
• Order of Operations
• Returning Values from Functions
• Writing Effective Functions
• General advice
Challenge/Practice

• https://github.com/msknapp/maintainable-bash
• Part 1: When shown the code, tell me what happens if it’s run
• Part 2: For various tasks, you must implement them in a function.
• Part 3: Given a poorly written shell script, you must refactor it to follow the principles taught in this presentation.
• Solutions are provided.
But… Why Bash?

- Fastest way to get something running in docker and/or Kubernetes
  - Don't need to write a Dockerfile.
- Don't need to worry about installing anything, no dependencies.
- Easiest way to integrate with other command line tools.
- Easiest thing to update, don't need to rebuild.
- Complements your ad-hoc troubleshooting skills.
- Enables your peers.
- Legacy code.
- So easy to maintain and test.
But… Why Bash?

- In your favorite programming language:
  - How many lines of code would this take?
  - How much time would you lose with dependency management?
  - How much time would you lose to unit testing, writing build scripts, jenkins jobs, creating artifacts, PRs?

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

# This lists all the orphaned persistent volumes
kubectl get pv -o json | jq -r '.items[] | select(.spec.claimRef.kind == "PersistentVolumeClaim") | .metadata.name +" ".spec.awsElasticBlockStore.volumeID' | sed 's|aws.*||g' |

while read line; do
  pv_name=$(cut -f1 <<< $line); vol=$(cut -f2 <<< $line)
  aws ec2 describe-volumes --volume-ids $vol &> /dev/null || echo "$pv_name DNE!"
done
```
Helpful Tools

• Turn colors on in vi
  - vi ~/.vimrc
  - syntax on
  - syntax off

• Use an IDE with color/context highlighting.
  - intelliJ has a bash plugin.
What Is a Process?

Environment

Process

Standard Input

Standard Output

Standard Error

Subprocess 1

Sub-process 1

Subprocess 2

Command Line Arguments

Status Code

0 = SUCCESS

non-zero = FAILURE

Note: functions ARE NOT sub-processes. More later...

Only exported variables from parent processes are known/visible.
Simplifications
my_boolean=$1
if [ "$my_boolean" == "true" ]; then
  echo "your boolean is true"
fi
echo "Resulting status code: $?"
# always 0

my_boolean=$1
if $my_boolean; then
  echo "your boolean is true."
fi
# status is still always 0

$my_boolean && \ 
  echo "Your boolean is true"
$my_boolean || \ 
  echo "Your boolean is false"
# status depends on my_boolean

female=${1:-false}
name='Michael'; $female && name="Michelle"

female=${1:-false}
amae='Michael'; $female && name="Michelle"

# status depends on my_boolean
Simplification: Booleans

• “false” is a program that always returns the status code 1 (failure).
• “true” is a program that always returns the status code 0 (success).
• ”:” is a no-operation program, but since it always returns 0, it’s effectively the same as “true”.
Simplification: Here Documents

```
echo "1 a"
echo "2 b"
echo "3 c"
```

```
cat << EOF
1 a
2 b
3 c
EOF
```

```
local y=3
cat << EOF
a=1
x=$y
EOF
# a=1
# x=3
```

```
local y=3
cat << "EOF"
a=1
x=$y
EOF
# a=1
# x=$y
```
Simplification: Here Strings

x="some text"

```
echo "$x" | wc -c
```

x="some text"

```
wc --c <<< "$x"
```

Advice: avoid using `wc`, especially within if conditions.
Variable Scope
Understanding Variable Scope: SubProcesses

• Current Process:
  • `my_function` # invoking
  • `source my_script.sh`
  • `. my_script.sh` # same as source
  • `{ ... }` # command grouping
  • `my_var=( x y z )` # array definition

• Child Process:
  • `program` ... # invoking
  • `bash -c 'command ...'`
  • `my_script.sh`
  • `command | my_function` ... # commands after a pipe
  • `var=${(my_function ...)}` # command substitution
  • `var=\`my_function ...` \` # back-ticks, same as `${( ... )}
  • `( my_function ... )` # commands in parentheses
Process Variable Visibility

Process
- export alpha=set
- beta=set
- local charlie=set

Sub-process
- export delta=set
- epsilon=set
- local gamma=set

invokes
Function Variables Visibility

**Note:** this assumes you are not wrapping the function with $(), back-ticks, or (), and that it’s not run after a | (pipe).
Order of Operations
Command Parsing

- Expansion happens before arguments are chosen.
- Most of the time, variables should be quoted.

```bash
text="could not find"
cat my.log | grep $text
# should have quoted the variable.

cat my.log | grep could not find
# grep: not: No such file or directory
# grep: find: No such file or directory

ddebug_arg="
false && debug_arg='--debug'
my_sub_command "$debug_arg"
# should NOT have quoted the variable.
# parameter "" is not supported.
```
Command Parsing

- Special operators are processed before parameter expansion
- Operators (examples):
  - | # pipe
  - || or &&
  - $( ... ), ` ...`
  - >, <, 1>&2, 2>&1
- Expansions:
  - variable: $1, $var, ${var}
  - glob: *.txt, temp-*, etc.

- Recommended reading: [http://mywiki.wooledge.org/BashParser](http://mywiki.wooledge.org/BashParser)
Binary Operator Precedence

- && and || have equal precedence.
- Commands are grouped from left to right.

```bash
succeeded=true
$succeeded || echo "failed" && return 1
```

```bash
{ true || echo "failed"; } && return 1
```

```bash
true && return 1
```

```bash
return 1 # you failed
```

```bash
succeeded=true
$succeeded || \{ echo "failed" && return 1; \}
```

```bash
true
```

```bash
# status code is 0, you succeeded
```
Returning Values
Capturing Values

```bash
square() {
    x="$1"
    echo -n "${(x*x)}"
}
run() {
    three_squared=${square 3}
    echo "3 squared is: $three_squared"
}
```

How bash does simple integer math.

- Don’t return values with “return”.
- Return is for **status code (success/failure)**.
- Return them on the standard output stream.
- Capture the standard output stream.
Capturing Values: Alternative Method

```
square() {
    x=$1
    squared_value="$(x * x)"
}
run() {
    square 3
    echo "3 squared is $squared_value"
}
```

- **Opinion**: I don’t prefer this method
  - a variable name change breaks the code.
  - Can **clobber** other variables
  - not minimizing the scope of variables
Writing Effective Functions
DRY Principle: Don’t Repeat Yourself!

- Avoid having code outside of functions.
  - Isolation: Minimize variable scope
  - Easier to refactor
  - Easier to read and understand.

```bash
#!/usr/bin/env bash
TODAY=Friday
echo "hello world it's $TODAY"
echo "hello mercury it's $TODAY"
echo "hello venus it's $TODAY"
echo "hello mars it's $TODAY"
```

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

function say_hello_and_the_day {
  local subject=$1
  local today=$2
  echo "hello ${subject} it's ${today}"
}

function main {
  local today=Friday
  for s in world mercury venus mars; do
    say_hello_and_the_day "$s" "$today"
  done
}

main $@
```
Minimize the Scope of Variables

- Prefer to mark variables “local” in functions.
- Avoid using “export”.
- Prefer to pass variables between functions using named parameters.
- Avoid passing variables around scripts as environment variables. You get very brittle code that’s difficult to troubleshoot.
- declare -r
Avoid Using Positional Parameters

```bash
#!/usr/bin/env bash
function say_hi {
  echo "Hi $1"
}
function main {
  say_hi joe
}
main $@
```

```bash
#!/usr/bin/env bash
function say_hi {
  local name=""
  if [[ "$1" =~ -n|--name ]]; then
    shift
    name="$1"
  fi
  echo "Hi $name"
}
function main {
  say_hi --name joe
}
main $@
```
Avoid Using Positional Parameters

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

function print_favorites {
    echo "$1's Favorite food: $2"
    echo "$1's Favorite color: $3"
    echo "$1's Favorite season: $4"
    echo "$1's Favorite pet: $5"
}

function main {
    print_favorites joe pizza blue summer dog
    print_favorites pizza blue summer dog
}

main $@
```

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

function print_favorites {
    local name=joe; local food=pizza; local color=blue
    local season=summer; local pet=dog
    while [ "$1" ]; do
        case "$1" in
            -n|--name)
                shift; name="$1";;
            -f|--food)
                shift; food="$1";;
            -c|--color)
                shift; color="$1";;
            -s|--season)
                shift; season="$1";;
            -p|--pet)
                shift; pet="$1";;
        esac
        shift
    done
    echo "${name}'s Favorite food: ${food}"
    echo "${name}'s Favorite color: ${color}"
    echo "${name}'s Favorite season: ${season}"
    echo "${name}'s Favorite pet: ${pet}"
}

function main {
    print_favorites --name joe --food pizza --color blue --season summer --pet dog
    print_favorites --food pizza --color blue --season summer --pet dog
}

main $@
```
Avoid Using Positional Parameters

```bash
echo_parameters() {
    source parse-quick-parameters "my_file=-f|--file,bool:run=-r|--run,name=-n|--name" $@
    || return 0
    echo "my file is: $my_file"
    echo "my name is: $name"
    echo "run is: $run"
}
```

Custom bash script, see git repo: http://github.com/msknapp/maintainable-bash
while [ "$1" ]; do
  case "$1" in
    -h|--help|help) cat << EOF
      
greet: Prints a greeting.

Parameters:
  -n|--name|--names <arg> ... - Sets the names
  -g|--greeting <arg> - Sets the greeting, the default is 'hello'
  -c|--caps - sets the output to write in all caps.
  -h|--help|help - prints this help information.

Assumptions:
  - awk is installed.

Examples:
  greet -n joe shmoe --greeting "Good morning" --caps
  # output: GOOD MORNING JOE SHMOE

EOF
  return 0 ;;
  esac
  shift
done

Function Help

• Build a help parameter into every program and function.
• State assumptions
• Declare parameters
• Provide examples
Troubleshooting

function greet {
    debug=false; dry_run=false; verbosity=0
    local name=joe;
    while [ "$1" ]; do
case "$1" in
    -n|--name)shift; name="$1";;
    --debug)debug=true;;
    --dry_run)dry_run=true;;
    -v|--verbose)verbosity=$((verbosity+1));;
    esac
    shift
done
...

$debug && echo "Name: $name" 1>&2
$debug && [ "$verbosity" -gt 1 ] && \
    echo "$(date)" 1>&2
$debug && [ "$verbosity" -gt 2 ] && \
    echo "OMG this is soooo verbose!" 1>&2
if $dry_run; then
    echo "Would run:" 1>&2
    cat 1>&2 << EOF
    echo "Hello $name"
EOF
else
    echo "Hello $name"
fi
...
Check Your Assumptions

# assumes an argument was set.
user="$1"

# assumes an env variable is set.
pass=$THE_PASSWORD

# assumes an operation was successful.
curl -d "$data ""…" > /tmp/out.json

# assumes a file exists.
# assumes jq is installed.
jq -r '.output' < /tmp/out.json
Other Advice for Function Initialization

- Assign smart default values to variables.
- Prefer to lookup values that were omitted.
  - Check a file
  - Run a CLI command
  - Invoke an API
- See bonus material.
Idempotence

- `echo "testing" > $HOME/my-test`
- `apt-get install docker`
- `docker pull my-image:1`
- `curl -X POST "..." -d '{"id":"1"}'`
- `aws s3 cp myfile s3://mybucket/myfile`
- `kubectl apply -f my-deployment.yaml`

Prefer to make your code idempotent.
Log Files

```bash
function get_name {
    local log_file=/var/log/action.log

    # use params to set log file.
    echo "calling whoami" >> $log_file
    name=$(whoami)
    echo "Name: $name" >> $log_file
    echo -n "$name"
}

name=$(get_name --log-file /home/bob/get-name.log)
echo "Good morning $name"

# ~ or ~
name=$(get_name 2> /dev/null)
echo "Good morning $name"
# You have more control over logging.
```

```bash
function get_name {
    echo "Going to call whoami" 1>&2
    name=$(whoami)
    echo "Found the name is: $name" 1>&2
    echo -n "$name"
}

log_file="/home/bob/get-name.log"

name=$(get_name 2> "$log_file")
echo "Good morning $name"
```

You have more control over logging.
Keep rows containing

```bash
keep_rows_containing() { grep "$1"; }
keep_column_number() { awk '{print $'$1'}'; } 
print_text() { cat <<< "$1"; }
```

```bash
people_status=$(cat << EOF
mike true
henry false
susan true
rachel false
EOF
)
```

```
list_people_going() {
  print_text "$people_status" | keep_rows_containing "true" | keep_column_number 1
}
```

It's nice to make code easy to read and understood for co-workers.
#!/usr/bin/env bash

function is_docker_installed {
  if [ "$(which docker 2> /dev/null | wc -l)" -lt 1 ]
    then
      echo "yes"
    else
      echo "no"
  fi
}

if [ "no" == "$(is_docker_installed)" ];
  then
    apt-get install docker
fi

# this is equivalent:

# can do “&> :” instead of “&> /dev/null”
Use Return

• Use return to deliver status of your function.
• Prefer to check the return status of commands you call.
• Red Flag: Excessively using `wc -l` in if statements.
• Red Flag: Using `$?`, it’s *usually* not necessary.
Returning Multiple Values

```bash
function create_name {  
  local name="$1"
  local first="${name:0:1}"
  local rem="${name:1:}
  local piglatin="${rem}${first}ay"
  local len="${#piglatin}"
  local length_squared=$(($len*$len))
  cat << EOF
    {
      "original_name": "$name",
      "piglatin": "$piglatin",
      "length": "$piglatin_length",
      "length_squared": "$length_squared"
    }
  EOF
}

n="susan"
result=$(create_name "$n")
name=$(jq -r '.original_name' <<< "$result")
my_piglatin=$(jq -r '.piglatin' <<< "$result")
length=$(jq -r '.length' <<< "$result")
length_squared=$(jq -r '.length_squared' <<< "$result")
sh

echo "$n in piglatin is $my_piglatin"
```
General Advice
Should I Put Code In .bashrc? Or On the PATH?

• .bashrc
  • Constantly needing to source it
  • Not easily used by peers on the system
  • Aliases cannot be used in other scripts/functions
  • Can change environment variables of the shell.
  • Sourcing scripts produces brittle code.

• Putting Scripts on the PATH
  • Updates are effective immediately
  • Code is easily re-used in other scripts (no sourcing necessary)
  • Cannot change environment variables of the shell.

• Advice: only use .bashrc or .bash_profile if you must alter a global environment variable.
General Advice

• Keep learning bash commands, learn their parameters.
• Very useful commands to learn: awk, sed, jq, tr, cut
• Think of APIs as opportunities to automate things:
  • Datadog, Artifactory, AWS, Kubernetes, Helm/tiller, Vault, Jenkins, Github, Dockyard
  • Many sites you use every day also have public APIs.
• For very complicated tasks, switch to a more robust language.
• The only thing worse than "code without documentation" is "documentation without code". Don’t write another confluence page that instructs people to run a long sequence of bash commands, script it instead.
• Study my examples:
  • https://github.com/msknapp/maintainable-bash
Best Advice I Can Give You

AUTOMATE EVERYTHING!
Questions?
Bonus Material
But… Why Bash?

- How quick and easy is it to get something running on Kubernetes?

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

# Every two seconds this prints the next fibonacci number, it's running on K8s.

kubectl run fibonacci --image ubuntu -l app=fibonacci --command --

`a=1;b=0;
while :: do
    echo $a;
    t=$a; a=$((a+b)); b=$t;
sleep 2;
done`

kubectl logs -f -l app=fibonacci
```
By default, any variable that is set is visible throughout the current process, but is not visible to sub or parent processes.

export makes variables visible to the current shell/program and all sub-processes/shells.

local means a variable will not be visible outside the current function.
Functions vs. Sub-processes

• Functions are not sub-processes!
• However, very similar.
  • use streams the same way
  • take arguments the same way
  • produce status codes the same way
• Since functions are still run in the same process
  • they share the variable namespace.
  • variables set in a script are seen by functions they call
  • variables set in a function are seen by the calling script.
  • Exception: using “local” prevents this.
Default Values

```
run() {
  local user=""
  local password=""
  local token=""
  parse-quick-parameters \ 
    'user=-u,password=-p,token=-t' @$
  # do something.
}
```

• Assign default values
• Prefer to lookup values.
Check Your Assumptions

- Check your assumptions in the code. Exit early if they are not met.
- Confirm that arguments are all set. Assign defaults where possible.
- Exit early if a value is mandatory and there is no default.
- Confirm that software is installed using `which`.
- Confirm that files exist beforehand.
function list_resource_record_sets {
    local pattern='prod'; local zone='my-hard-coded-zone'
    parse_quick_parameters 'zone=-z|--zone,pattern=-p|--pattern' $@
    aws route53 list-resource-record-sets --hosted-zone-id "${zone}" --output text --query 'ResourceRecordSets[?contains(Name,`$pattern`)].[Name, ResourceRecords[0].Value]'
}

function list_resource_record_sets {
    local pattern='prod'; local region='us-east-1'; local zone="";
    parse_quick_parameters 'zone=-z|--zone,pattern=-p|--pattern,region=-r|--region' $@
    # If the zone is not set, then get it:
    [ "$zone" ] || zone=$(get_hosted_zone -r "$region")
    ...
}
Initializing Functions Summary

- Put all code in functions
- Limit the scope of your variables
- Avoid using positional parameters
- Prefer using named parameters in all functions
- Add a help parameter to each function
- Use default values for parameters
- Check your script assumptions, fail fast if they are not true.
- Lookup values if they were not provided.
- Add troubleshooting parameters to each function (--debug, --dry-run, etc.)
Other Advice for Main Code

• Prefer to make your script easy to read, by creating “alias” functions.
• Avoid having low level functions write to a log file, use standard error instead.
  • Leaves more control with higher level functions.
• See Bonus slides.
Code Body Summary

• Make your script idempotent
• Avoid using log files in low level functions
• Use standard error to deliver logs
• Design your scripts to be easy to read and understand.
Returning Multiple Values

```bash
create_name() {
    original_name="$1"
    local first="${name:0:1}"  # Extract the first character
    local rem="${name:1:}"     # Extract the remaining part
    piglatin="${rem}${first}ay"
    length=${#piglatin}         # Length of piglatin
    length_squared=$(($length*$length))
}
length=50
n="susan"
create_name "susan"

echo "$n in piglatin is $piglatin"    # usansay
echo "length was changed to $length"  # 7

# The problem is those variables may have overridden
# other ones. We are not isolating their scope
# with "local" in this example.
```
Multi-value Return (Abridged)

• Return a JSON document to standard output.
• Calling function extracts values with jq
• See bonus material.
Returning Info Summary

• Use return status for delivering boolean results
• Check the return status of commands you call
• Use standard output to return object/text information.
• Use JSON documents to return multiple values
Portability

• Detect the OS using the command “uname”
  • Darwin for Mac

• Design scripts so they can be run on any system they might be placed on.

• Example differences:
  • CentOS and RedHat use yum, not apt.
  • Mac uses brew, not apt
  • tar extraction produces annoying backup files on a mac.
  • netstat arguments vary between operating systems.
  • On mac, “sed –i” takes an argument, on Ubuntu it does not.
Scope Examples

setXUpdated() { x=updated; }
exportXUpdated() { export x=updated; }
say_x() { echo "${x:-original}"; }
setXLocal() { local x=updated; say_x }
unset x; x=original;
# insert code block….
echo $x

<table>
<thead>
<tr>
<th>x=updated</th>
<th>updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>say_x</td>
<td>updated</td>
</tr>
<tr>
<td>export x=updated ( say_x )</td>
<td>updated</td>
</tr>
<tr>
<td>setXLocal</td>
<td>original</td>
</tr>
<tr>
<td>bash -c 'x=updated'</td>
<td>original</td>
</tr>
<tr>
<td>{ x=updated }</td>
<td>updated</td>
</tr>
<tr>
<td>setXUpdated</td>
<td>updated</td>
</tr>
<tr>
<td>( exportXUpdated )</td>
<td>original</td>
</tr>
</tbody>
</table>
Scope Example

bad() {
    sum=0
    ls -l | grep .txt | awk '{print $5}' | \\
    while read line; do
        sum=$(($sum+line))
    done
    echo "And the sum is: $sum" # 0, WTH? 😞
}

good() {
    sum=0
    while read line; do
        sum=$(($sum+line))
    done < <(ls -l | grep .txt | awk '{print $5}')
    echo "And the sum is: $sum" # correct. 😊
}