# ECS 98F - Shell Scripting

*Stephen Ott*

# Agenda

## Today's lecture

- Why we script and what we can script
- Bash syntax and semantics
- Writing maintainable scripts

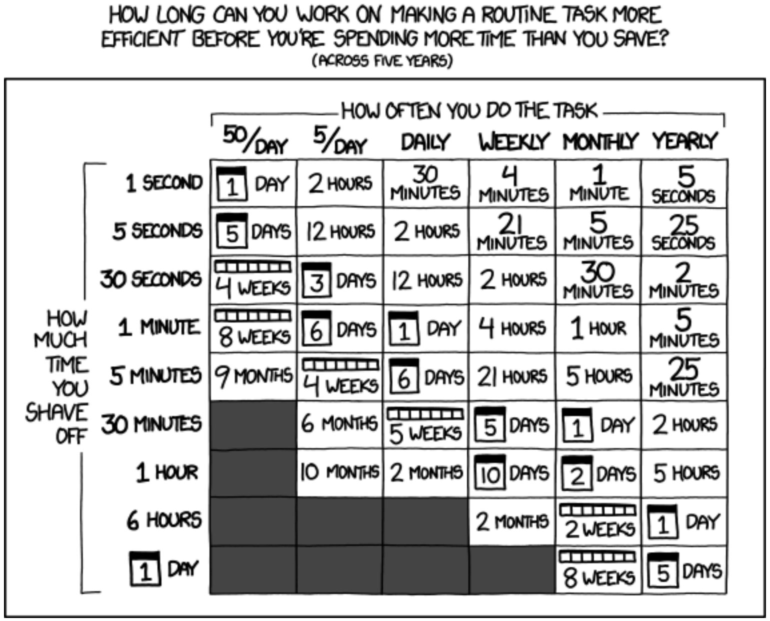
# Philosophy of Scripting

## Why we script

- Save time
- Abstraction
- Adoration from our peers

## Two types of scripts

- The "throwaway" scripts
- The "throwaway" scripts that are now indispensible



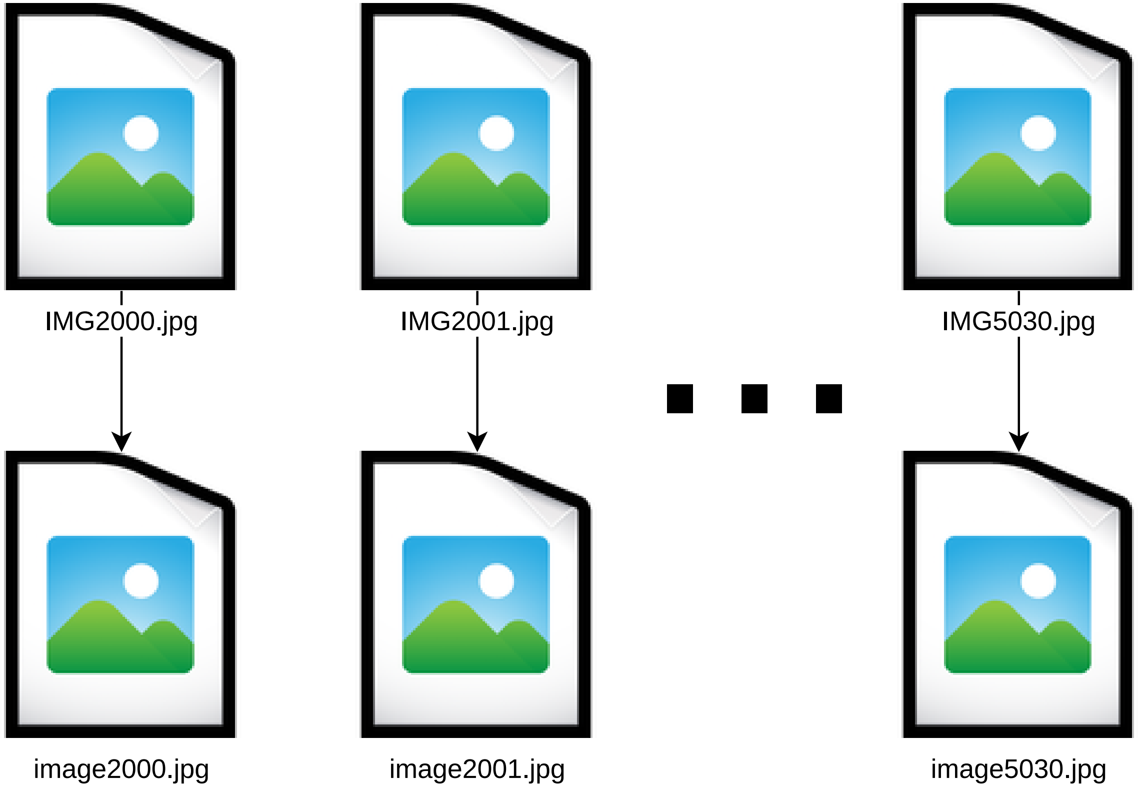
HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

| | | HOW OFTEN YOU DO THE TASK | | | | | |
|---|---|---|---|---|---|---|---|
| | | 50/DAY | 5/DAY | DAILY | WEEKLY | MONTHLY | YEARLY |
| HOW MUCH TIME YOU SHAVE OFF | 1 SECOND | 1 DAY | 2 HOURS | 30 MINUTES | 4 MINUTES | 1 MINUTE | 5 SECONDS |
| | 5 SECONDS | 5 DAYS | 12 HOURS | 2 HOURS | 21 MINUTES | 5 MINUTES | 25 SECONDS |
| | 30 SECONDS | 4 WEEKS | 3 DAYS | 12 HOURS | 2 HOURS | 30 MINUTES | 2 MINUTES |
| | 1 MINUTE | 8 WEEKS | 6 DAYS | 1 DAY | 4 HOURS | 1 HOUR | 5 MINUTES |
| | 5 MINUTES | 9 MONTHS | 4 WEEKS | 6 DAYS | 21 HOURS | 5 HOURS | 25 MINUTES |
| | 30 MINUTES | ■ | 6 MONTHS | 5 WEEKS | 5 DAYS | 1 DAY | 2 HOURS |
| | 1 HOUR | ■ | 10 MONTHS | 2 MONTHS | 10 DAYS | 2 DAYS | 5 HOURS |
| | 6 HOURS | ■ | ■ | ■ | 2 MONTHS | 2 WEEKS | 1 DAY |
| | 1 DAY | ■ | ■ | ■ | ■ | 8 WEEKS | 5 DAYS |

# Problem Scenario

## Importing photos

You have just imported some pictures from your phone onto your computer. All of the files are named something like `IMG2002.jpg`. The rest of your pictures are named something like `image2002.jpg`. How can we rename all of the `IMG____.jpg` files to `image____.jpg`?



# Problem Scenario

## What we want

- We want to be able to run command from our shell that will perform the rename for every jpeg named this way
- It would be nice if we could solve this, not only for this problem, but for the general problem of renaming multiple files with similiar names
- What information would we need?

```
$ ./batch_rename.sh IMG image <all_jpeg_files_in_a_directory>
```

# Problem Scenario

## First attempt

```
$ mv IMG2000.jpg image2000.jpg
$ mv IMG2001.jpg image2001.jpg
$ mv IMG2002.jpg image2002.jpg
...
```

### Pros

- This technically works
- The command is really easy to remember
- Builds muscle memory

### Cons

- What if there's 1000 files?
- What if it's some other file format?

# The Bash Language

## Thinking of bash as a language

- Bash is a programming language like C
- You already have written some Bash!
- Unlike C, Bash is interpreted and dynamically-typed

# The Bash Language

## Our first script

- Copy and paste what we were typing in the terminal into a file

```
$ mv IMG2000.jpg image2000.jpg
$ mv IMG2001.jpg image2001.jpg
$ mv IMG2002.jpg image2002.jpg
...
$ mv IMG2020.jpg image2020.jpg
```

```
mv IMG2000.jpg image2000.jpg
mv IMG2001.jpg image2001.jpg
mv IMG2002.jpg image2002.jpg
...
mv IMG2020.jpg image2020.jpg
```

attempt1.sh

Then run

```
$ bash attempt1.sh
```

# The Bash Language

## Variables

### Assigning variables

- Bash is dynamically typed
- Spaces matter!

```bash
# This works
var="foo"

# This errors
var = "foo"

# This stores stdout from ls
files=$(ls)
```

### Accessing variables

```bash
echo $var

# This works as well
echo "$var"
```

# The Bash Language

## String interpolation

- Bash strings can use either single or double-quoted strings
- Single-quoted will perform no interpolation

```
var='cool'

# Prints Stephen is pretty cool
echo "Stephen is pretty $var"
# This works as well and is sometimes more readable
echo "Stephen is pretty ${var}"
# Prints Stephen is not $var
echo 'Stephen is not so $var'
```

## String substitution

- Bash allows you to easily substitute strings

```
message="Stephen is not cool"
correct_message={message/'cool'/'uncool'}
# Prints Stephen is not uncool
echo $correct_message
```

# The Bash Language

## Wildcard

- `*` is a meta-character that represents zero or more of any characters
- Also called glob star

```
$ ls
main.c   helper.h   helper.c   run.log   run2.log   run3.log
$ rm *.log
$ ls
main.c   helper.h   helper.c
```

# The Bash Language

## Arrays

- Same idea as arrays in C
- Index from zero
- Use `+=` to append elements

```bash
pies=('apple' 'cherry' 'pecan')
echo ${pies[1]}
pies+=('blueberry')
echo ${pies[3]}
```

array_example.sh

```
$ bash array_example.sh
cherry
blueberry
```

- Index `*` to get all elements in the array
- Use `unset` to remove elements of an array at an index
- Prefix the name of the array with `#` and index `@` to get the length of the array

```bash
pies=('apple' 'cherry' 'pecan')
echo ${pies[*]}
unset pies[2]
echo ${pies[*]}
echo ${#pies[@]}
```

array_example2.sh

```
$ bash array_example2.sh
apple cherry pecan
apple cherry
2
```

# The Bash Language

## For loops

- Bash's for loops are iterator based

```
pies=('apple' 'cherry' 'pecan' 'blueberry')

# Prints all the pies in the array
for i in ${pies[@]}
do
    echo $i
done
```

for_loop_test.sh

```
$  bash for_loop_test.sh
apple
cherry
pecan
blueberry
```

# The Bash Language

## A second attempt

```
for i in IMG*.jpg
do
    new_file_name=${i/IMG/image}
    echo "Renaming $i to $new_file_name"
    mv $i $new_file_name
done
```

attempt2.sh

```
$ bash attempt2.sh
Renaming IMG1.jpg to image1.jpg
Renaming IMG2.jpg to image2.jpg
Renaming IMG3.jpg to image3.jpg
```

### Pros

- Works with 2, 50, or 1000 files
- Frees up time for you to get more coffee and do homework

### Cons

- Someone else wouldn't be able to use it
- Only works for this scenario

# Making It Scalable

## Comments

- Single line comments in Bash start with `#`

```
# Using comments keeps your code readable
```

## She-bang lines

- This is a single line that appears on the first line of the file to provide a path to the interpreter

```
#!/bin/bash

for i in IMG*.jpg
do
    new_file_name=${i/IMG/image}
    echo "Renaming $i to $new_file_name"
    mv $i $new_file_name
done
```

attempt2.sh

# Making It Scalable

## Command line arguments

- In bash, arguments are also indexed by position
  - `$0` is the name of the script
  - `$1, $2, ...` refer to the positional arguments in their respective order

```bash
#!/bin/bash

echo $3
echo $2
echo $1
```

arg_test.sh

```
$ chmod +x arg_test.sh
$ ./arg_test.sh Stephen Grant Noah
Noah
Grant
Stephen
```

## Variable number of arguments

- Scripts can have a variable number of arguments
- These arguments are stored in the special variable `$@`
  - `shift` is a reserved keyword to shift which section of the sequence of arguments we are looking at

```bash
#!/bin/bash

echo $@
shift 2
echo $@
```

arg_test2.sh

```
$ chmod +x arg_test2.sh
$ ./arg_test2.sh a b c d e f
a b c d e f
c d e f
```

# Making It Scalable

## Conditionals

### If-statements

- Enclose statement in double square brackets

```
if [[ 2 != 3 ]]
then
    echo "2 is not 3"

else
    echo "If 2 is 3, then pi is a rational number"
fi
```

### Case statements

- Nice way to replace `if...else if...else if...else` blocks

```
case $some_string in
   ni)
       echo "We are the knight who say ni"
       ;;
   it)
       echo "AHHHHH! Don't say that word!"
       ;;
   *)
       echo "ni"
esac
```

# Making It Scalable

## Exit codes

### Checking exit codes

- Recall that a program has failed if it has a nonzero exit code
- The exit code of the last command is stored in the variable `$?`
- This also works with the `&&` and `||` operators to allow you to succinctly express this logic

```
#!/bin/bash

grep 'Gandalf' list_of_dwarves.txt

if [[ $? != 0 ]]
then
  echo "Gandalf is not a dwarf"
  exit 1
else
  echo "Gandalf is a dwarf"
  exit 0
fi
```

dwarf_search.sh

### Providing exit codes

- Use the `exit` command followed by an integer
- Numbering these in a meaningful way makes it easy to identify problems in your code
- Defaults to 0

```
$ chmod +x dwarf_search.sh
$ ./dwarf_search.sh
Gandalf was not in the list of dwarves
$ echo $?
1
```

# Making It Scalable

## Functions

### Differences from C

- It is then helpful to think of functions as mini-scripts
  - This includes passing and accessing arguments to the function

### Example

```bash
#!/bin/bash

lines_in_file()
{
   # Here $1 is the name of a file
   wc -l $1
}
lines_in_file list_of_dwarves.txt
```

function_example.sh

```
$ chmod +x function_example.sh
$ ./function_example.sh
13
```

# A Third Attempt

## The script

- Let's look at a solution with our new knowledge of Bash

```
#!/bin/bash

mode=$1
find=$2
replace=$3
# Shift 3, so $@ will refer to the target files
shift 3
files=$@

print_help()
{
...
}

do_rename()
{
...
}
```

batch_rename.sh

```
case $mode in
   '-h')
      print_help
      ;;
   '-d')
      echo 'Performing dry run'
      do_rename true
      ;;
   '-f')
      echo 'Performing rename in place'
      do_rename false
      ;;
   *)
      echo 'Invalid arguments. Try again'
      print_help
      exit 1
esac
```

batch_rename.sh

# A Third Attempt

## The print_help function

```
print_help()
{
   # Prints a help message to the user
   echo "Usage: ./batch_rename.sh [-h|-d|-f] FIND REPLACE FILES"
   echo ""
   echo "FIND: The substring that constitutes a file name we want to change"
   echo "REPLACE: What we will replace the FIND string with"
   echo "FILES: The target files to be renamed"
   echo "   -h: Help. Prints this help message and exits"
   echo "   -d: Dry run. Prints what the changes would be but does not execute the changes"
   echo "   -f: Force. Changes the file names in place."
}
```

batch_rename.sh

# A Third Attempt

## The do_rename function

```
do_rename()
{
   dry_run=$1
   # For each file
   for i in $files
   do
      # If $find is a substring of $i
      if [[ $i == *"${find}"* ]]
      then
         # perform the substitution
         new_name=${i/$find/$replace}

         # if the user indicated they don't want to do a dry run
         if [[ $dry_run = false ]]
         then
            # perform the rename
            mv $i $new_name && echo "Successfully renamed $i to $new_name"
         else
            # else just print what the result would be
            echo "Would rename $i to $new_name"
         fi
      fi
```

batch_rename.sh

# A Third Attempt

## Let's run it!

```
$ ls
IMG1.jpg  IMG2.jpg  IMG3.jpg
$ ./batch_rename.sh -f IMG image *.jpg
Performing rename in place
Successfully renamed test_files/IMG1.jpg to test_files/image1.jpg
Successfully renamed test_files/IMG2.jpg to test_files/image2.jpg
Successfully renamed test_files/IMG3.jpg to test_files/image3.jpg
$ ls
image1.jpg  image2.jpg  image3.jpg
```

## This fixes all the cons of the previous iteration

- Will work with any file naming scheme
- Can be run from other directory locations
- Friendly for other users

# Shellcheck

- Utility to lint your bash scripts
- Catches syntactic errors and code that may be a problem under certain cases

```
$ shellcheck batch_rename.sh

In batch_rename.sh line 8:
files=$@
      ^-- SC2124: Assigning an array to a string! Assign as array, or use * instead of @ to co
ncatenate.


In batch_rename.sh line 40:
            mv $i $new_name
               ^-- SC2086: Double quote to prevent globbing and word splitting.
                  ^-------^ SC2086: Double quote to prevent globbing and word splitting.

Did you mean:
            mv "$i" "$new_name"

For more information:
  https://www.shellcheck.net/wiki/SC2124 -- Assigning an array to a string! A...
  https://www.shellcheck.net/wiki/SC2086 -- Double quote to prevent globbing ...
```

# Conclusion

- Bash is a programming language
- Shell scripting is a powerful tool to automate repetitive tasks
- However, it is *not* a development language[1]
- Shell scripting should *not* be used when[1]
  - Performance matters
  - The task has non-straightforward logic
  - Your script has to perform massive amounts of data manipulation
- Other languages can be used for scripting

[1]: Source: Google's Style Guide

# References & Further Reading

- Shell Scripting Tutorial
    - This serves as a good refresher on bigger concepts and constructs of the language
- Google Shell Script Style Guide
    - While the style guide informs style, it also gives a great philosophy of what you should script
- What is Bash?
    - GNU's introduction to what Bash is. Gives a good overview of language semantics
- Bash cheat sheet
    - Concise reminder on features of the language. Sometimes helpful to have open in another window while writing a script.