Minimally competent bash scripting

> "We build our computer (systems) the way we build our cities: over time, without a plan, on top of ruins." - Ellen Ullman

> "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements." - Brian W. Kernighan, Unix for Beginners (1979)

Bash is the worst shell scripting language except for all the others. For many of the analyses you'll write, all you will need is a simple bash script, so let's figure out how to write a decent one. I'll share with you what I've found to be the minimal amount of bash I use.

Statements

All programming language have a grammar where "statements" (like "sentences") are built up from other terms. Some languages like Python and Haskell use whitespace to figure out the end of a "statement," which is usually just the right side of the window. C-like languages such as bash and Perl define the end of a statement with a colon `;`. Bash is interesting because it uses both. If you hit <Enter> or type a newline in your code, Bash will execute that statement. If you want to put several commands on one line, you can separate each with a semicolon. If you want to stretch a command over more than one line, you can use a backslash `\` to continue the line:

```
$ echo Hi
Hi
$ echo Hello
Hello
$ echo Hi; echo Hello
Hi
Hello
$ echo \
> Hi
Hi
```

Comments

Every language has a way to indicate text in the source code that should not be executed by the program. Many Unix/c-style languages use the `#` (hash) sign to indicate that any text to the right should be ignored by the language, but some languages use other characters or character combinations like `//` in Javascript, Java, and Rust. Programmers may use comments to explain what some particularly bit of code is doing, or they may use the characters to temporarily disable some section of code. Here is an example of what you might see:

```
# cf. https://en.wikipedia.org/wiki/Factorial
sub fac(n) {
  # first check terminal condition
  if (n <= 1) {
    return 1
  }
  # no? let's recurse!
  else {
    n * fac(n - 1) # the number times one less the number
  }
}
```

It's worth investing time in an editor that can easily comment/uncomment whole sections of code. For instance, in vim, I have a function that will add or removed the appropriate comment character(s) (depending on the filetype) from the beginning of the selected section. If your editor can't do that (e.g., nano), then I suggest you find something more powerful.

Shebang

Scripting languages (sh, bash, Perl, Python, Ruby, etc.) are generally distinguished by the fact that the "program" is a regular file containing plain text that is interpreted into machine code at the time you run it. Other languages (c, C++, Java, Haskell, Rust) have a separate compilation step to turn their regular text source files into a binary executable. If you view a compiled file with an editor/pager, you'll see a mess that might even lock up your window. (If that happens, refer back to "Make it stop!" to kill it or just close the window and start over.)

So, basically a "script" is a plain text file that is often executable by virtue of having the executable bit(s) turned on (cf. "Permissions"). It does not have to be executable, however. It's acceptable to put some commands in a file and simply tell the appropriate program to interpret the file:

```
$ echo "echo Hello, World" > hello.sh
$ sh hello.sh
Hello, World
```

But it looks cooler to do this:

```
$ chmod +x hello.sh
$ ./hello.sh
Hello, World
```

But what's going on here?

```
$ echo 'print("Hello, World")' > hello.py
$ chmod +x hello.py
$ ./hello.py
./hello.py: line 1: syntax error near unexpected token `"Hello, World"'
./hello.py: line 1: `print("Hello, World")'
```

We put some Python code in a file and then asked our shell (which is bash) to interpret it. That didn't work. If we ask Python to run it, everything is fine:

```
$ python3 hello.py
Hello, World
```

So we just need to let the shell know that this is Python 3 code, and that is what the "shebang" (see "Pronunciations") line is for. It looks like a comment, but it's special line that the shell uses to interpret the script. I'll use an editor to add a shebang to the "hello.py" script, then I'll `cat` the file so you can see what it looks like.

```
$ cat hello.py
#!/usr/bin/env python3
print("Hello, World")
$ ./hello.py
Hello, World
```

Often the shebang line will indicate the absolute path to a program like "/bin/bash" or "/usr/local/bin/gawk," but here I used an absolute path not to Python but to the "env" program which I then passed "python3" as the argument. Why did I do that? To make this script "portable" (for certain values of "portable," cf. "It's easier to port a shell than a shell script." – Larry Wall), I prefer to use the "python3" that is found by the environment as I will usually put my preferred Python first in my `$PATH`.

Let's Make A Script!

Let's make our script say "Hello" to some people:

```
$ cat -n hello2.sh
     1	#!/usr/bin/env bash
     2	
     3	NAME="Newman"
     4	echo "Hello," $NAME
     5	NAME="Jerry"
     6	echo "Hello, $NAME"
$ ./hello2.sh
Hello, Newman
```

```
Hello, Jerry
```

I've created a variable called `NAME` to hold the string "Newman" and print it. Notice there is no `$` when assigning to the variable, only when you use it. The value of `NAME` can be changed at any time. You can print it out like on line 4 as it's own argument to `echo` or inside of a string like on line 6. Notice that the version on line 4 puts a space between the arguments to `echo`.

Because all the variables from the environment (see `env`) are uppercase (e.g., `$HOME` and `$USER`), I tend to use all-caps myself, but this did lead to a problem once when I named a variable `PATH` and then overwrote the actual `PATH` and then my program stopped working entirely as it could no longer find any of the programs it needed. Just remember that everything in Unix is case-sensitive, so `$Name` is an entirely different variable from `$name`.

When assigning a variable, you can have NO SPACES around the `=` sign:

```
$ NAME1="Doge"
$ echo "Such $NAME1"
Such Doge
$ NAME2 = "Doge"
-bash: NAME2: command not found
$ echo "Such $NAME2"
Such
```

Sidebar: Catching Common Errors (set -u)

Bash is an easy language to write incorrectly. One step you can take to ensure you don't misspell variables is to add `set -u` at the top of your script. E.g., if you type `echo $HOEM` on the command line, you'll get no output or warning that you misspelled the `$HOME` variable unless you `set -u`:

```
$ echo $HOEM

$ set -u
$ echo $HOEM
-bash: HOEM: unbound variable
```

This command tells bash to complain when you use a variable that was never initialized to some value. This is like putting on your helmet. It's not a requirement (depending on which state you live in), but you absolutely should do this because there might come a day when you misspell a variable. Note that this will not save you from as error like this:

```
$ cat -n set-u-bug1.sh
     1	#!/bin/bash
     2
     3	set -u
     4
     5    if [[ $# -gt 0 ]]; then
     6      echo $THIS_IS_A_BUG; # never initialized
     7    fi
     8
     9    echo "OK";
$ ./set-u-bug1.sh
OK
$ ./set-u-bug1.sh foo
./set-u-bug1.sh: line 6: THIS_IS_A_BUG: unbound variable
```

You can see that the first execution of the script ran just fine. There is a bug on line 6, but bash didn't catch it because that line did not execute. On the second run, the error occurred, and the script blew up. (FWIW, this is a problem in Python, too.)

Here's another pernicious error:

```
$ cat -n set-u-bug2.sh
     1    #!/bin/bash
     2
     3    set -u
     4
     5    GREETING="Hi"
     6    if [[ $# -gt 0 ]]; then
     7      GRETING=$1 # misspelled
     8    fi
     9
    10    echo $GREETING
$ ./set-u-bug2.sh
Hi
$ ./set-u-bug2.sh Hello
Hi
```

We were foolishly hoping that `set -u` would prevent us from misspelling the `$GREETING`, but at line 7 we simple created a new variable called `$GRETING`. Perhaps you were hoping for more help from your language? This is why we try to limit how much bash we write.

NB: I highly recommend you use the program `shellcheck` https://www.shellcheck.net/to find errors in your bash code.

Common Patterns

This is a cut-and-paste section for you. The idea is that I will describe many common patterns that you can use directly.

Test if a variable is a file or directory

Use the `-f` or `-d` functions to test if a variable identifies a “file” or a “directory,” respectively

```
if [[ -f "$ARG" ]]; then
    echo "$ARG is a file"
fi

if [[ -d "$ARG" ]]; then
    echo "$ARG is a directory"
fi
```

Use `!` to negate this:

```
if [[ ! -f "$ARG" ]]; then
    echo "$ARG is NOT a file"
fi

if [[ ! -d "$ARG" ]]; then
    echo "$ARG is a directory"
fi
```

There are many other test you can use. See `man test` for a complete list. The `-s` is handy to see if a file is empty. You can use more than one test at a time with the `&&` (“and”) or `||` (“or”) operator.

```
if [[ -f "$ARG" ]] && [[ -s "$ARG" ]]; then
    echo "$ARG is a file and is not empty"
fi

if [[ ! -f "$ARG" ]] || [[ ! -s "$ARG" ]]; then
    echo "$ARG is a NOT file or is empty"
fi
```

Exit your script

The `exit` function will cease all operations and immediately exit. With no argument, it will use “0” which means “zero errors”; any other value is considered a error code, so `exit 1` is commonly used indicate some unspecified error.

```
if [[ -f "$ARG" ]]; then
    wc -l "$ARG"
    exit
else
    echo "$ARG must be a file"
    exit 1
fi
```

Check the number of arguments to your program

The first argument to your script is in `$1`, the second in `$2`, and so on. The number of arguments is in `$#`, so you can check the number like this:

```
if [[ $# -eq 0 ]]; then
    echo "Usage: foo.sh ARG"
    exit 1
fi
```

The `-eq` means “equal”. You can also use `-gt` or `-gte` for “greater than (or equal)” and `-lt` or `-lte` for “less than or equal”.

Put the arguments into named variables

You should assign `$1` and `$2` to names that have some meaning in your program.

```
INPUT_FILE=$1
NUM_ITERATIONS=$2
```

Set default values for optional arguments

If an argument is not needed, you can assign a default value. Here we can set `NUM_ITERATIONS` to have a default value of “10”:

```
INPUT_FILE=$1
NUM_ITERATIONS=${2:-10}
```

Read a file

It’s common to use a `while` loop to `read` a file, line-by-line, into some `VARIABLE`. Don’t use a `$` on the `while` line (assigning), do use it when you want to interpolate it:

```
while read -r LINE; do
    echo "$LINE"
done < "$FILE"
```

Use a counter variable

It’s common to use the variable `i` (for “integer” maybe?) as a temporary counter, e.g., iterating over lines in a file. The syntax to increment is clunky. This will print a line number and a line of text from a file:

```
i=0
while read -r LINE; do
    i=$((i+1))
    echo $i "$LINE"
done < "$FILE"
```

Loop operations

Use `continue` to skip to the next iteration of a loop. This will print only the even lines of a file:

```
i=0
while read -r LINE; do
    i=$((i+1))
    if [[ $(expr $i % 2) -eq 0 ]]; then
        continue
    else:
        echo "$i $LINE"
    fi
done < "$FILE"
```

Use `break` to leave a loop. This will print the first 10 lines of a file:

```
i=0
while read -r LINE; do
    echo "$LINE"
    i=$((i+1))
    if [[ $i -eq 10 ]]; then
        break
    fi
done < "$FILE"
```

Capture the output of a command

Historically `bash` used backticks (the same key as the tilde on a US QWERTY keyboard) to execute a command and put the results into a variable:

```
DIR=`ls`
```

Most people now use `$()` as it stands out much better:

```
DIR=$(ls)
LINES=$(grep foo bar.txt)
```

Count the number of lines in a file

```
NUM_LINES=$(wc -l "$FILE" | awk '{print $1}')

if [[ $NUM_LINES -lt 1 ]]; then
    echo "There is noting in $FILE"
    exit 1
fi
```

Get a temporary file or directory

Sometimes you need a temporary file to store something. If the name and location of the file is unimportant, use `mktemp` to get a temporary file or `mktemp -d` to get a temporary directory.

```
TMP_FILE=$(mktemp)
cat "foo\nbar\n" > "$TMP_FILE"

TMP_DIR=$(mktemp -d)
cd "$TMP_DIR"
```

Get the last part of a file or directory name

If you have "/path/to/my/file.txt" and you just want to print "file.txt", use `basename`:

```
FILE="/path/to/my/file.txt"
basename "$FILE"
```

Or put that into a variable name to use:

```
FILE="/path/to/my/file.txt"
BASENAME=$(basename "$FILE")
echo "Basename is $BASENAME"
```

Similary `dirname` is use to get "/path/to/my" from the above:

```
FILE="/path/to/my/file.txt"
DIRNAME=$(dirname "$FILE")
echo "Dirname is $DIRNAME"
```

Print with echo and printf

The `echo` command will print messages to the screen (standard out):

```
USER="Dave"
echo "I'm sorry, $USER, I can't do that."
```

The `printf` command is useful for formatting the output. The command expects a "template" first and then all the arguments for each formatting code in the template. The percent sign `%` is used in the template to indicate the type and options, e.g., an integer right-justified and three digits wide is `%3d`. Use `man printf` to learn more. Here is an example to print the line numbers in a file more prettier:

```
i=0
while read -r LINE; do
    i=$((i+1))
    printf "%3d: %s\n" $i $LINE
done < "$FILE"
```

Capture many items into a file for looping

Bash doesn't do lists (many items in a series) very well, so I usually put lists into files; e.g., I want to find how many files are in a directory and iterate over them:

```
FILES=$(mktemp)
find "$DIR" -type f -name \*.f[aq] > "$FILES"
NUM_FILES=$(wc -l "$FILES" | awk '{print $1}')

if [[ $NUM_FILES -lt 1 ]]; then
    echo "No usable files in $DIR"
    exit 1
fi

echo "Found $NUM_FILES in $DIR"

i=0
while read -r FILENAME; do
    i=$((i+1))
    BASENAME=$(basename "$FILENAME")
    printf "%3d: %s\n" $i "$FILENAME"
done < "$FILES"
```

For Loops

Often we want to do some set of actions for all the files in a directory or all the identifiers in a file. You can use a `for` loop to iterate over the values in some command that returns a list of results:

```
$ for FILE in *.sh; do echo "FILE = $FILE"; done
FILE = args.sh
FILE = args2.sh
FILE = args3.sh
FILE = basic.sh
FILE = hello.sh
FILE = hello2.sh
FILE = hello3.sh
FILE = hello4.sh
FILE = hello5.sh
FILE = hello6.sh
FILE = named.sh
FILE = positional.sh
FILE = positional2.sh
FILE = positional3.sh
FILE = set-u-bug1.sh
FILE = set-u-bug2.sh
```

Here it is in a script:

```
$ cat -n for.sh
     1	#!/bin/bash
     2	
     3	set -u
     4	
     5	DIR=${1:-$PWD}
     6	
     7	if [[ ! -d "$DIR" ]]; then
     8	    echo "$DIR is not a directory"
     9	    exit 1
    10	fi
    11	
    12	i=0
    13	for FILE in $DIR/*; do
    14	    let i++
    15	    printf "%3d: %s\n" $i "$FILE"
    16	done
```

On line 5, I default `DIR` to the current working directory which I can find with the environmental variable `$PWD` (print working directory). I check on line 7 that the argument is actually a directory with the `-d` test (`man test`). The rest should look familiar. Here it is in action:

```
$ ./for.sh | head
  1: /Users/kyclark/work/metagenomics-book/bash/args.sh
  2: /Users/kyclark/work/metagenomics-book/bash/args2.sh
  3: /Users/kyclark/work/metagenomics-book/bash/args3.sh
  4: /Users/kyclark/work/metagenomics-book/bash/basic.sh
  5: /Users/kyclark/work/metagenomics-book/bash/config1.sh
  6: /Users/kyclark/work/metagenomics-book/bash/config2.sh
  7: /Users/kyclark/work/metagenomics-book/bash/count-fa.sh
  8: /Users/kyclark/work/metagenomics-book/bash/for-read-file.sh
  9: /Users/kyclark/work/metagenomics-book/bash/for.sh
 10: /Users/kyclark/work/metagenomics-book/bash/functions.sh
$ ./for.sh ../problems | head
  1: ../problems/cat-n
  2: ../problems/common-words
  3: ../problems/dna
  4: ../problems/gapminder
  5: ../problems/gc
  6: ../problems/greeting
  7: ../problems/hamming
  8: ../problems/hello
  9: ../problems/proteins
 10: ../problems/tac
```

You will see many examples of using `for` to read from a file like so:

```
$ cat -n for-read-file.sh
     1	#!/usr/bin/env bash
     2	
     3	FILE=${1:-'srr.txt'}
     4	for LINE in $(cat "$FILE"); do
     5	    echo "LINE \"$LINE\""
     6	done
$ cat srr.txt
SRR3115965
SRR516222
SRR919365
$ ./for-read-file.sh srr.txt
LINE "SRR3115965"
LINE "SRR516222"
LINE "SRR919365"
```

But that can break badly when the file contains more than one "word" per line (as defined by the `$IFS` [input field separator]):

```
$ column -t pov-meta.tab
name               lat_lon.ll
GD.Spr.C.8m.fa     -17.92522,146.14295
GF.Spr.C.9m.fa     -16.9207,145.9965833
L.Spr.C.1000m.fa   48.6495,-126.66434
L.Spr.C.10m.fa     48.6495,-126.66434
L.Spr.C.1300m.fa   48.6495,-126.66434
L.Spr.C.500m.fa    48.6495,-126.66434
L.Spr.I.1000m.fa   48.96917,-130.67033
```

```
L.Spr.I.10m.fa    48.96917,-130.67033
L.Spr.I.2000m.fa  48.96917,-130.67033
$ ./for-read-file.sh pov-meta.tab
LINE "name"
LINE "lat_lon.ll"
LINE "GD.Spr.C.8m.fa"
LINE "-17.92522,146.14295"
LINE "GF.Spr.C.9m.fa"
LINE "-16.9207,145.9965833"
LINE "L.Spr.C.1000m.fa"
LINE "48.6495,-126.66434"
LINE "L.Spr.C.10m.fa"
LINE "48.6495,-126.66434"
LINE "L.Spr.C.1300m.fa"
LINE "48.6495,-126.66434"
LINE "L.Spr.C.500m.fa"
LINE "48.6495,-126.66434"
LINE "L.Spr.I.1000m.fa"
LINE "48.96917,-130.67033"
LINE "L.Spr.I.10m.fa"
LINE "48.96917,-130.67033"
LINE "L.Spr.I.2000m.fa"
LINE "48.96917,-130.67033"
```

While Loops

The proper way to read a file line-by-line is with `while`:

```
$ cat -n while.sh
     1	#!/usr/bin/env bash
     2
     3	FILE=${1:-'srr.txt'}
     4	while read -r LINE; do
     5	    echo "LINE \"$LINE\""
     6	done < "$FILE"
$ ./while.sh srr.txt
LINE "SRR3115965"
LINE "SRR516222"
LINE "SRR919365"
$ ./while.sh meta.tab
LINE "GD.Spr.C.8m.fa    -17.92522,146.14295"
LINE "GF.Spr.C.9m.fa    -16.9207,145.9965833"
LINE "L.Spr.C.1000m.fa    48.6495,-126.66434"
LINE "L.Spr.C.10m.fa    48.6495,-126.66434"
LINE "L.Spr.C.1300m.fa    48.6495,-126.66434"
LINE "L.Spr.C.500m.fa    48.6495,-126.66434"
LINE "L.Spr.I.1000m.fa    48.96917,-130.67033"
LINE "L.Spr.I.10m.fa    48.96917,-130.67033"
LINE "L.Spr.I.2000m.fa    48.96917,-130.67033"
```

Another advantage is that `while` can break the line into fields:

```
$ cat -n while2.sh
     1	#!/usr/bin/env bash
     2	
     3	FILE='meta.tab'
     4	while read -r SITE LOC; do
     5	    echo "$SITE is located at \"$LOC\""
     6	done < "$FILE"
$ ./while2.sh
GD.Spr.C.8m.fa is located at "-17.92522,146.14295"
GF.Spr.C.9m.fa is located at "-16.9207,145.9965833"
L.Spr.C.1000m.fa is located at "48.6495,-126.66434"
L.Spr.C.10m.fa is located at "48.6495,-126.66434"
L.Spr.C.1300m.fa is located at "48.6495,-126.66434"
L.Spr.C.500m.fa is located at "48.6495,-126.66434"
L.Spr.I.1000m.fa is located at "48.96917,-130.67033"
L.Spr.I.10m.fa is located at "48.96917,-130.67033"
L.Spr.I.2000m.fa is located at "48.96917,-130.67033"
```

Sidebar: Saving Function Results in Files

Often I want to iterate over the results of some calculation. Here is an example of saving the results of an operation (`find`) into a temporary file:

```
$ cat -n count-fa.sh
     1	#!/usr/bin/env bash
     2	
     3	set -u
     4	
     5	if [[ $# -ne 1 ]]; then
     6	    printf "Usage: %s DIR\n" "$(basename "$0")"
     7	    exit 1
     8	fi
     9	
    10	DIR=$1
    11	TMP=$(mktemp)
    12	find "$DIR" -type f -name \*.fa > "$TMP"
    13	NUM_FILES=$(wc -l "$TMP" | awk '{print $1}')
    14	
```

```
     15    if [[ $NUM_FILES -lt 1 ]]; then
     16        echo "Found no .fa files in $DIR"
     17        exit 1
     18    fi
     19
     20    NUM_SEQS=0
     21    while read -r FILE; do
     22        NUM_SEQ=$(grep -c '^>' "$FILE")
     23        NUM_SEQS=$((NUM_SEQS + NUM_SEQ))
     24        printf "%10d %s\n" "$NUM_SEQ" "$(basename "$FILE")"
     25    done < "$TMP"
     26
     27    rm "$TMP"
     28
     29    echo "Done, found $NUM_SEQS sequences in $NUM_FILES files."
$ ./count-fa.sh ../problems
        23 anthrax.fa
         9 burk.fa
Done, found 32 sequences in 2 files.
```

Line 11 uses the `mktemp` function to give us the name of a temporary file, then I `find` all the files ending in ".fa" or ".fasta" and put that into the temporary file. I could them to make sure I found something. Then I read from the tempfile and use the `FILE` name to count the number of times I see a greater-than sign at the beginning of a line.

Getting Data Into Your Program: Arguments

We would like to get the NAME from the user rather than having it hardcoded in the script. I'll show you three ways our script can take in data from outside:

1. Command-line arguments, both positional (i.e., the first one, the second one, etc.) or named (e.g., `-n NAME`)
2. The environment
3. Reading a configuration file

First we'll cover the command-line arguments which are available through a few variables:

- `$#`: The number (think "#" == number) of arguments
- `$@`: All the arguments in a single string
- `$0`: The name of the script
- `$1, $2`: The first argument, the second argument, etc.

A la:

```
$ cat -n args.sh
```

```
     1	#!/usr/bin/env bash
     2
     3	echo "Num of args    : \"$#\""
     4	echo "String of args : \"$@\""
     5	echo "Name of program: \"$0\""
     6	echo "First arg      : \"$1\""
     7	echo "Second arg     : \"$2\""
$ ./args.sh
Num of args    : "0"
String of args : ""
Name of program: "./args.sh"
First arg      : ""
Second arg     : ""
$ ./args.sh foo
Num of args    : "1"
String of args : "foo"
Name of program: "./args.sh"
First arg      : "foo"
Second arg     : ""
$ ./args.sh foo bar
Num of args    : "2"
String of args : "foo bar"
Name of program: "./args.sh"
First arg      : "foo"
Second arg     : "bar"
```

If you would like to iterate over all the arguments, you can use `$@` like so:

```
$ cat -n args2.sh
     1	#!/usr/bin/env bash
     2
     3	if [[ $# -lt 1 ]]; then
     4	    echo "There are no arguments"
     5	else
     6	    i=0
     7	    for ARG in "$@"; do
     8	        let i++
     9	        echo "$i: $ARG"
    10	    done
    11	fi
$ ./args2.sh
There are no arguments
$ ./args2.sh foo
1: foo
$ ./args2.sh foo bar "baz quux"
1: foo
2: bar
```

```
3: baz quux
```

Here I'm throwing in a conditional at line 3 to check if the script has any arguments. If the number of arguments (`$#`) is less than (`-lt`) 1, then let the user know there is nothing to show; otherwise (`else`) do the next block of code. The `for` loop on line 7 works by splitting the argument string (`$@`) on spaces just like the command line does. Both `for` and `while` loops require the `do/done` pair to delineate the block of code (some languages use `{}`, Haskell and Python use only indentation). Along those lines, line 11 is the close of the `if` – "if" spell backwards; the close of a `case` statement in bash is `esac`.

The other bit of magic I threw in was a counter variable (which I always use lowercase `i` ["integer"], `j` if I needed an inner-counter and so on) which is initialized to "0" on line 6. I increment it, I could have written `$i=$(($i + 1))`, but it's easier to use the `let i++` shorthand. Lastly, notice that "baz quux" seen as a single argument because it was placed in quotes; otherwise arguments are separated by spaces.

Sidebar: Make It Pretty (or else)

Note that indentation doesn't matter as the program below works, but, honestly, which one is easier for you to read?

```
$ cat -n args3.sh
     1	#!/usr/bin/env bash
     2
     3	if [[ $# -lt 1 ]]; then
     4	echo "There are no arguments"
     5	else
     6	i=0
     7	for ARG in "$@"; do
     8	let i++
     9	echo "$i: $ARG"
    10	done
    11	fi
$ ./args3.sh foo bar
1: foo
2: bar
```

Our First Argument

AT LAST, let's return to our "hello" script!

```
$ cat -n hello3.sh
     1	#!/usr/bin/env bash
     2
     3	echo "Hello, $1!"
$ ./hello3.sh Captain
Hello, Captain!
```

This should make perfect sense now. We are simply saying “hello” to the first argument, but what happens if we provide no arguments?

```
$ ./hello3.sh
Hello, !
```

Checking the Number of Arguments

Well, that looks bad. We should check that the script has the proper number of arguments which is 1:

```
$ cat -n hello4.sh
     1	#!/usr/bin/env bash
     2
     3	if [[ $# -ne 1 ]]; then
     4	    printf "Usage: %s NAME\n" "$(basename "$0")"
     5	    exit 1
     6	fi
     7
     8	echo "Hello, $1!"
$ ./hello4.sh
Usage: hello4.sh NAME
$ ./hello4.sh Captain
Hello, Captain!
$ ./hello4.sh Captain Picard
Usage: hello4.sh NAME
```

Line 3 checks if the number of arguments is not equal (`-ne`) to 1 and prints a help message to indicate proper “usage.” Importantly, it also will `exit` the program with a value which is not zero to indicate that there was an error. (NB: An exit value of “0” indicates 0 errors.) Line 4 uses `printf` rather than `echo` so I can do some fancy substitution so that the results of calling the `basename` function on the `$0` (name of the program) is inserted at the location of the `%s` (a string value, cf. man pages for “printf” and “basename”).

Here is an alternate way to write this script:

```
$ cat -n hello5.sh
     1	#!/usr/bin/env bash
     2
     3	if [[ $# -eq 1 ]]; then
     4	    NAME=$1
     5	    echo "Hello, $NAME!"
     6	else
     7          printf "Usage: %s NAME\n" "$(basename "$0")"
     8          exit 1
     9      fi
```

Here I check on line 3 if there is just one argument, and the `else` is devoted to handling the error; however, I prefer to check for all possible errors at the beginning and `exit` the program quickly. This also has the effect of keeping my code as far left on the page as possible.

Sidebar: Saving Function Results

In the previous script, you may have noticed `$(basename "$0")`. I was passing the script name (`$0`) to the function `basename` and then passing that to the `printf` function. To call a function in bash and save the results into a variable or use the results as an argument, we can use either backticks (“) (under the ~ on a US keyboard) or `$()`. I find backticks to be too similar to single quotes, so I prefer the latter. To demonstrate:

```
$ ls | head
args.sh*
args2.sh*
args3.sh*
basic.sh*
hello.sh*
hello2.sh*
hello3.sh*
hello4.sh*
hello5.sh*
hello6.sh*
$ FILES=`ls | head`
$ echo $FILES
args.sh args2.sh args3.sh basic.sh hello.sh hello2.sh hello3.sh hello4.sh hello5.sh hello6.s
```

Here is a script that shows:

1. Calling `basename` and having the result print out (line 5)
2. Using `$()` to capture the results of `basename` into a variable (line 8)
3. Using `$()` to call `basename` as the second argument to `echo`
4. Showing that `$()` can be interpolated **inside a string**
5. Using `$()` to call `basename` as an argument to `printf`

```
$ cat -n functions.sh
     1      #!/usr/bin/env bash
     2
     3      # call function
     4      echo -n "1: BASENAME: "
     5      basename "$0"
     6
     7	# put function results into variable
     8	BASENAME=$(basename "$0")
     9	echo "2: BASENAME: $BASENAME"
    10
    11	# use results of function as argument to another function
    12	echo "3: BASENAME:" "$(basename "$0")"
    13	echo "4: BASENAME: $(basename "$0")"
    14	printf "5: BASENAME: %s\n" "$(basename "$0")"
$ ./functions.sh
1: BASENAME: functions.sh
2: BASENAME: functions.sh
3: BASENAME: functions.sh
4: BASENAME: functions.sh
5: BASENAME: functions.sh
```

Providing Default Argument Values

Here is how you can provide a default value for an argument with `:-`:

```
$ cat -n hello6.sh
     1	#!/usr/bin/env bash
     2
     3	echo "Hello, ${1:-Stranger}!"
$ ./hello6.sh
Hello, Stranger!
$ ./hello6.sh Govnuh
Hello, Govnuh!
```

Arguments From The Environment

You can also use look in the environment for argument values. For instance, we could accept the `NAME` as either the first argument to the script (`$1`) or the `$USER` from the environment:

```
$ cat -n hello7.sh
     1	#!/usr/bin/env bash
     2
     3	NAME=${1:-$USER}
     4	[[ -z "$NAME" ]] && NAME='Stranger'
     5	echo "Hello, $NAME
$ ./hello7.sh
Hello, kyclark
$ ./hello7.sh Barbara
Hello, Barbara
```

What's interesting is that you can temporarily over-ride an environmental variable like so:

```
$ USER=Bart ./hello7.sh
Hello, Bart
$ ./hello7.sh
Hello, kyclark
```

Exporting Values to the Environment

Notice that I can set `USER` for the first run to "Bart," but the value returns to "kyclark" on the next run. I can permanently set a value in the environment by using the `export` command. Here is a version of the script that looks for an environmental variable called `WHOM` (please do override your `$USER` name in the environment as things will break):

```
$ cat -n hello8.sh
     1	#!/usr/bin/env bash
     2
     3	echo "Hello, ${WHOM:-Marie}"
$ ./hello8.sh
Hello, Marie
```

As before I can set it temporarily:

```
$ WHOM=Doris ./hello8.sh
Hello, Doris
$ ./hello8.sh
Hello, Marie
```

Now I will `export WHOM` so that it persists:

```
$ WHOM=Doris
$ export WHOM
$ ./hello8.sh
Hello, Doris
$ ./hello8.sh
Hello, Doris
```

To remove `WHOM` from the environment, use `unset`:

```
$ unset WHOM
$ ./hello8.sh
Hello, Marie
```

Some programs rely heavily on environmental variables (e.g., Centrifuge, TACC LAUNCHER) for arguments. Here is a short script to illustrate how you would use such a program:

```
$ cat -n hello9.sh
     1	#!/usr/bin/env bash
     2
     3	WHOM="Who's on first" ./hello8.sh
     4	WHOM="What's on second"
     5	export WHOM
     6	./hello8.sh
     7	WHOM="I don't know's on third" ./hello8.sh
$ ./hello9.sh
Hello, Who's on first
Hello, What's on second
Hello, I don't know's on third
```

Required and Optional Arguments

Now we're going to accept two arguments, "GREETING" and "NAME" while providing defaults for both:

```
$ cat -n positional.sh
     1	#!/usr/bin/env bash
     2
     3	set -u
     4
     5	GREETING=${1:-Hello}
     6	NAME=${2:-Stranger}
     7
     8	echo "$GREETING, $NAME"
$ ./positional.sh
Hello, Stranger
$ ./positional.sh Howdy
Howdy, Stranger
$ ./positional.sh Howdy Padnuh
Howdy, Padnuh
$ ./positional.sh "" Pahnuh
Hello, Pahnuh
```

You notice that if I want to use the default argument for the greeting, I have to pass an empty string `""`.

What if I want to require at least one argument?

```
$ cat -n positional2.sh
     1	#!/usr/bin/env bash
     2
     3	set -u
     4
     5	if [[ $# -lt 1 ]]; then
     6          printf "Usage: %s GREETING [NAME]\n" "$(basename "$0")"
     7          exit 1
     8      fi
     9
    10      GREETING=$1
    11      NAME=${2:-Stranger}
    12
    13      echo "$GREETING, $NAME"
$ ./positional2.sh "Good Day"
Good Day, Stranger
$ ./positional2.sh "Good Day" "Kind Sir"
Good Day, Kind Sir
```

It's also important to note the subtle hints given to the user in the "Usage" statement. `[NAME]` has square brackets to indicate that it is an option, but `GREETING` does not to say it is required. As noted before I wanted to use the GREETING "Good Day," so I had to put it in quotes so that the shell would not interpret them as two arguments. Same with the NAME "Kind Sir."

```
$ ./positional2.sh Good Day Kind Sir
Good, Day
```

Not Too Few, Not Too Many (Goldilocks)

Hmm, maybe we should detect that the script had too many arguments?

```
$ cat -n positional3.sh
     1      #!/usr/bin/env bash
     2
     3      set -u
     4
     5      if [[ $# -lt 1 ]] || [[ $# -gt 2 ]]; then
     6          printf "Usage: %s GREETING [NAME]\n" "$(basename "$0")"
     7          exit 1
     8      fi
     9
    10      GREETING=$1
    11      NAME=${2:-Stranger}
    12
    13      printf "%s, %s\n" "$GREETING" "$NAME"
$ ./positional3.sh Good Day Kind Sir
Usage: positional3.sh GREETING [NAME]
$ ./positional3.sh "Good Day" "Kind Sir"
Good Day, Kind Sir
```

To check for too many arguments, I added an "OR" (the double pipes `||`) and another conditional ("AND" is `&&`). I also changed line 13 to use a `printf` command to highlight the importance of quoting the arguments *inside the script* so that bash won't get confused. Try it without those quotes and try to figure out why it's doing what it's doing. I highly recommend using the program "shellcheck" (https://github.com/koalaman/shellcheck) to find mistakes like this. Also, consider using more powerful/helpful/sane languages – but that's for another discussion.

Named Arguments To The Rescue

I hope maybe by this point you're thinking that the script is getting awfully complicated just to allow for a combination of required an optional arguments all given in a particular order. You can manage with 1-3 positional arguments, but, after that, we really need to have named arguments and/or flags to indicate how we want to run the program. A named argument might be `-f mouse.fa` to indicate the value for the `-f` ("file," probably) argument is "mouse.fa," whereas a flag like `-v` might be a yes/no ("Boolean," if you like) indicator that we do or do not want "verbose" mode. You've encountered these with programs like `ls -l` to indicate you want the "long" directory listing or `ps -u $USER` to indicate the value for `-u` is the `$USER`.

The best thing about named arguments is that they can be provided in any order:

```
$ ./named.sh -n Patch -g "Good Boy"
Good Boy, Patch!
```

Some may have values, some may be flags, and you can easily provide good defaults to make it easy for the user to provide the bare minimum information to run your program. Here is a version that has named arguments:

```
$ cat -n named.sh
     1	#!/usr/bin/env ash
     2	
     3	set -u
     4	
     5	GREETING=""
     6	NAME="Stranger"
     7	EXCITED=0
     8	
     9	function USAGE() {
    10	    printf "Usage:\n  %s -g GREETING [-e] [-n NAME]\n\n" $(basename $0)
    11	    echo "Required arguments:"
    12	    echo " -g GREETING"
    13	    echo
    14	    echo "Options:"
    15	    echo " -n NAME ($NAME)"
    16	    echo " -e Print exclamation mark (default yes)"
    echo
    exit ${1:-0}
}

[[ $# -eq 0 ]] && USAGE 1

while getopts :g:n:eh OPT; do
  case $OPT in
    h)
      USAGE
      ;;
    e)
      EXCITED=1
      ;;
    g)
      GREETING="$OPTARG"
      ;;
    n)
      NAME="$OPTARG"
      ;;
    :)
      echo "Error: Option -$OPTARG requires an argument."
      exit 1
      ;;
    \?)
      echo "Error: Invalid option: -${OPTARG:-""}"
      exit 1
  esac
done

[[ -z "$GREETING" ]] && USAGE 1
PUNCTUATION="."
[[ $EXCITED -ne 0 ]] && PUNCTUATION="!"

echo "$GREETING, $NAME$PUNCTUATION"
```

When run without arguments or with the `-h` flag, it produces a help message.

```
$ ./named.sh
Usage:
  named.sh -g GREETING [-e] [-n NAME]

Required arguments:
 -g GREETING

Options:
 -n NAME (Stranger)
 -e Print exclamation mark (default yes)
```

Our script just got much longer but also more flexible. I've written a hundred shell scripts with just this as the template, so you can, too. Go search for how `getopt` works and copy-paste this for your bash scripts, but the important thing to understand about `getopt` is that flags that take arguments have a `:` after them (`g:` == "-g something") and ones that do not, well, do not (`h` == "-h" == "please show me the help page). Both the"h" and "e" arguments are flags:

```
$ ./named.sh -n Patch -g "Good Boy"
Good Boy, Patch.
$ ./named.sh -n Patch -g "Good Boy" -e
Good Boy, Patch!
```

I've introduced a new function called `USAGE` that prints out the "Usage" statement so that it can be called when:

- the script is run with no arguments (line 21)
- the script is run with the "-h" flag (lines 25-26)
- the script is run with bad input (line 47)

I initialized the NAME to "Stranger" (line 6) and then let the user know in the "Usage" what the default value will be. When checking the GREETING in line 44, I'm actually checking that the length of the value is greater than zero because it's possible to run the script like this:

```
$ ./named01.sh -g ""
```

Which would technically pass muster but does not actually meet our requirements.

Reading a Configuration File

The last way I'll show you to get data into your program is to read a configuration file. This builds on the earlier example of using `export` to put values into the environment:

```
$ cat -n config1.sh
     1	export NAME="Merry Boy"
     2	export GREETING="Good morning"
$ cat -n read-config.sh
     1	#!/usr/bin/env bash
     2
     3	source config1.sh
     4	echo "$GREETING, $NAME!"
$ ./read-config.sh
Good morning, Merry Boy!
```

To make this more flexible, let's pass the config file as an argument:

```
$ cat -n read-config2.sh
     1	#!/usr/bin/env bash
     2	
     3	CONFIG=${1:-config1.sh}
     4	if [[ ! -f "$CONFIG" ]]; then
     5	    echo "Bad config \"$CONFIG\""
     6	    exit 1
     7	fi
     8	
     9	source $CONFIG
    10	echo "$GREETING, $NAME!"
$ ./read-config2.sh
Good morning, Merry Boy!
$ cat -n config2.sh
     1	export NAME="François"
     2	export GREETING="Salut"
$ ./read-config2.sh config2.sh
Salut, François!
$ ./read-config2.sh foo
Bad config "foo"
```

A Full Bag of Tricks

Lastly I'm going to show you how to create some sane defaults, make missing directories, find user input, transform that input, and report back to the user. Here's a script that takes an IN_DIR, counts the lines of all the files therein, and reports said line counts into an optional OUT_DIR.

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

set -u

IN_DIR=""
OUT_DIR="$PWD/$(basename "$0" '.sh')-out"

function lc() {
    wc -l "$1" | awk '{print $1}'
}

function USAGE() {
    printf "Usage:\n  %s -i IN_DIR -o OUT_DIR\n\n" "$(basename "$0")"

    echo "Required arguments:"
    echo " -i IN_DIR"
    echo "Options:"
    echo " -o OUT_DIR"
    echo
    exit "${1:-0}"
}

[[ $# -eq 0 ]] && USAGE 1

while getopts :i:o:h OPT; do
    case $OPT in
        h)
            USAGE
            ;;
        i)
            IN_DIR="$OPTARG"
            ;;
        o)
            OUT_DIR="$OPTARG"
            ;;
        :)
            echo "Error: Option -$OPTARG requires an argument."
            exit 1
            ;;
        \?)
            echo "Error: Invalid option: -${OPTARG:-""}"
            exit 1
    esac
done

if [[ -z "$IN_DIR" ]]; then
    echo "IN_DIR is required"
    exit 1
fi

if [[ ! -d "$IN_DIR" ]]; then
    echo "IN_DIR \"$IN_DIR\" is not a directory."
    exit 1
fi

echo "Started $(date)"

FILES_LIST=$(mktemp)
find "$IN_DIR" -type f -name \*.sh > "$FILES_LIST"
NUM_FILES=$(lc "$FILES_LIST")

if [[ $NUM_FILES -gt 0 ]]; then
```

```
63          echo "Will process NUM_FILES \"$NUM_FILES\""
64
65          [[ ! -d $OUT_DIR ]] && mkdir -p "$OUT_DIR"
66
67          i=0
68          while read -r FILE; do
69              BASENAME=$(basename "$FILE")
70              let i++
71              printf "%3d: %s\n" $i "$BASENAME"
72              wc -l "$FILE" > "$OUT_DIR/$BASENAME"
73          done < "$FILES_LIST"
74
75          rm "$FILES_LIST"
76          echo "See results in OUT_DIR \"$OUT_DIR\""
77      else
78          echo "No files found in \"$IN_DIR\""
79      fi
80
81      echo "Finished $(date)"
```

The IN_DIR argument is required (lines 46-49), and it must be a directory (lines 51-54). If the user does not supply an OUT_DIR, I will create a reasonable default using the current working directory and the name of the script plus "-out" (line 6). One thing I love about bash is that I can call functions inside of strings, so OUT_DIR is a string (it's in double quotes) of the variable $PWD, the character "/", and the result of the function call to `basename` where I'm giving the optional second argument ".sh" that I want removed from the first argument, and then the string "-out".

At line 58, I create a temporary file to hold the names of the files I need to process. A line 59, I look for the files in IN_DIR that need to be processed. You can read the manpage for `find` and think about what your script might need to find (".fa" files greater than 0 bytes in size last modified since some date, etc.). At line 60, I call my `lc` (line count) function to see how many files I found. If I found more than 0 files (line 62), then I move ahead with processing. I check to see if the OUT_DIR needs to be created (line 65), and then create a counter variable ("i") that I'll use to number the files as I process them. At line 68, I start a `while` loop to iterate over the input from redirecting *in* from the temporary file holding the file names (line 73, `< "$FILES_LIST"`). Then a `printf` to let the user know which file we're processing, then a simple command (`wc`) but where you might choose to BLAST the sequence file to a database of pathogens to determine how deadly the sample is. When I'm done, I clean up the temp file (line 75).

The alternate path when I find no input files (line 77-79) is to report that fact. Bracketing the main processing logic are "Started/Finished" statements so I can see how long my script took. When you start your coding career, you will usually sit and watch your code run before you, but eventually you'll submit the your jobs to an HPC queue where they will be run for you on a separate machine when the resources become available.

The above is, I would say, a minimally competent bash script. If you can understand everything in there, then you know enough to be dangerous and should move on to learning more powerful languages – like Python!