Introduction to **Shell Scripting with Bash**

Charles Jahnke
Research Computing Services
Information Services & Technology

BOSTON UNIVERSITY

Topics for Today

- Introductions
- Basic Terminology
- How to get help
- Command-line vs. Scripting
- Variables
- Handling Arguments
- Standard I/O, Pipes, and Redirection
- Control Structures (loops and If statements)
- SCC Job Submission Example

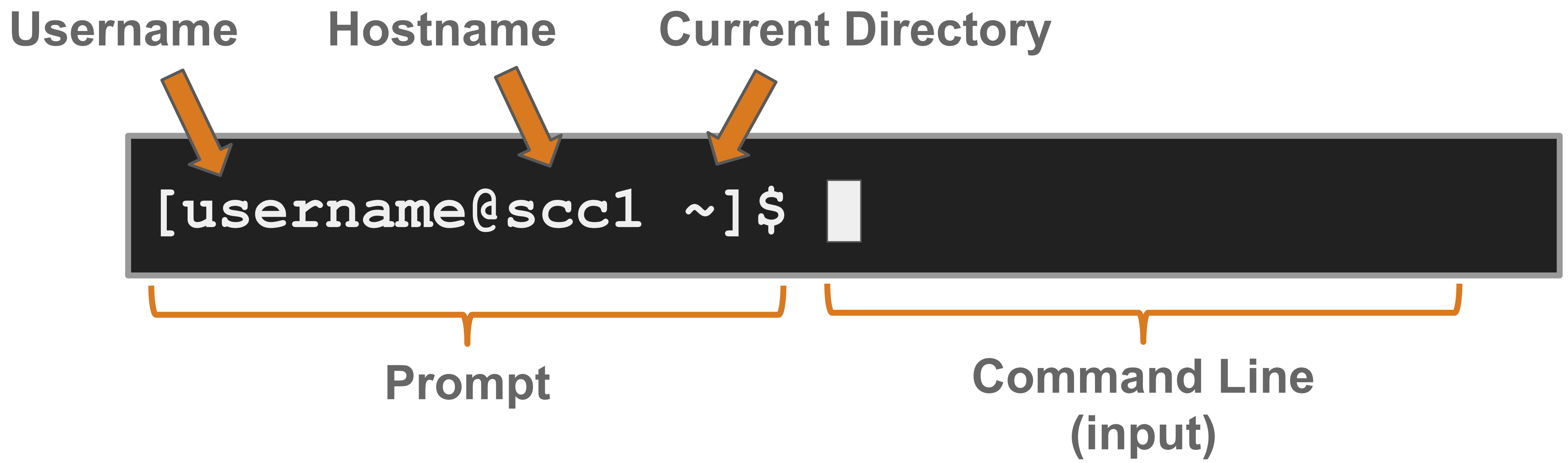
Me

- Research Facilitator and Administrator
- Background in biomedical engineering, bioinformatics, and IT systems
- Offices on both CRC and BUMC
    - Most of our staff on the Charles River Campus, some dedicated to BUMC
- Contact: help@scc.bu.edu

Basic Terminology

The Command-line

The line on which commands are typed and passed to the shell.

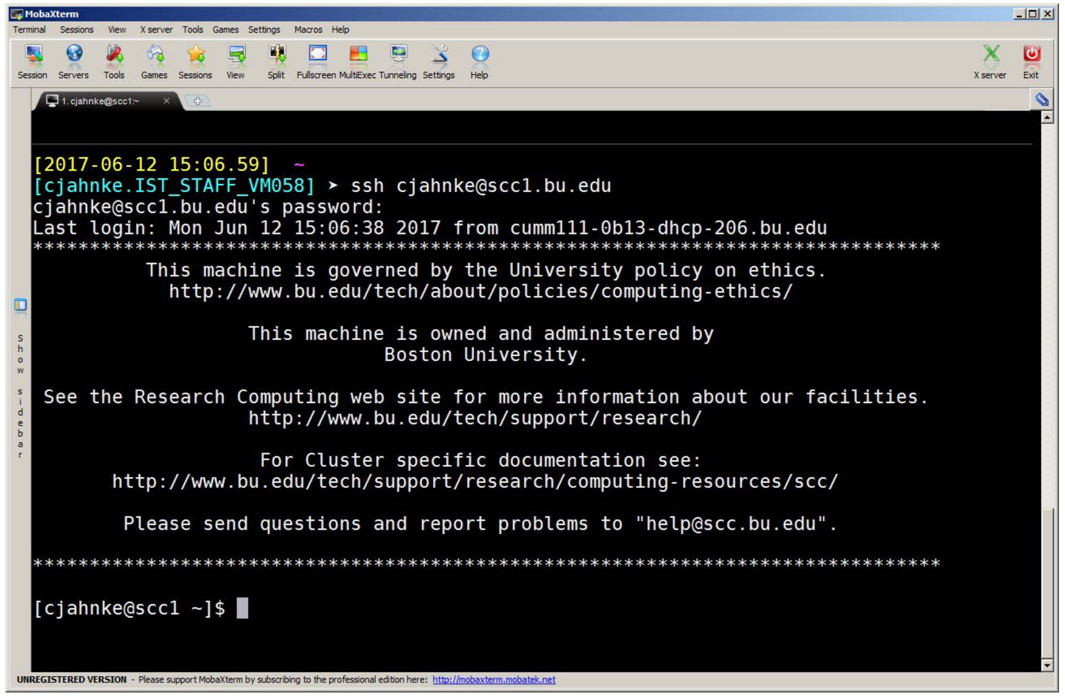


```
[username@scc1 ~]$ 
```

The Shell

- The interface between the user and the operating system
- Program that interprets and executes input

- Provides:
  - Built-in commands
  - Programming control structures
  - Environment variables


MobaXterm
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split Fullscreen MultiExec Tunneling Settings Help X server Exit
1. cjahnke@scc1:~
Show sidebar

```
[2017-06-12 15:06.59]  ~
[cjahnke.IST_STAFF_VM058] ➤ ssh cjahnke@scc1.bu.edu
cjahnke@scc1.bu.edu's password:
Last login: Mon Jun 12 15:06:38 2017 from cumm111-0b13-dhcp-206.bu.edu
********************************************************************************
          This machine is governed by the University policy on ethics.
            http://www.bu.edu/tech/about/policies/computing-ethics/

                   This machine is owned and administered by
                               Boston University.

 See the Research Computing web site for more information about our facilities.
                  http://www.bu.edu/tech/support/research/

                     For Cluster specific documentation see:
     http://www.bu.edu/tech/support/research/computing-resources/scc/

        Please send questions and report problems to "help@scc.bu.edu".

********************************************************************************

[cjahnke@scc1 ~]$ 
```

UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: http://mobaxterm.mobatek.net

Script

- A text file containing a series of commands that an interpreter (like shell) can read and run.

Interpreter

- A program that runs commands without compiling (directly from text)

Bash

The name of the most common shell interpreter, it’s language, and syntax.

The default shell on SCC and

What we are going to use today

Teach a Programmer to Fish

How to Get Help

Manuals (“**man**”) and Info (“**info**”)

```
scc1 $ man bash
```

```
BASH(1)              General Commands Manual             BASH(1)

NAME
       bash - GNU Bourne-Again SHell

SYNOPSIS
       bash [options] [file]

COPYRIGHT
       Bash  is  Copyright  (C)  1989-2011  by the Free Software
       Foundation, Inc.

DESCRIPTION
       Bash is an  sh-compatible  command  language  interpreter
       that  executes  commands  read from the standard input or
       from a file.  Bash also incorporates useful features from
       the Korn and C shells (ksh and csh).

       Bash is intended to be a conformant implementation of the
       Shell and Utilities portion of the IEEE POSIX  specifica-
       tion  (IEEE  Standard 1003.1).  Bash can be configured to
       be POSIX-conformant by default.
```

```
scc1 $ info bash
```

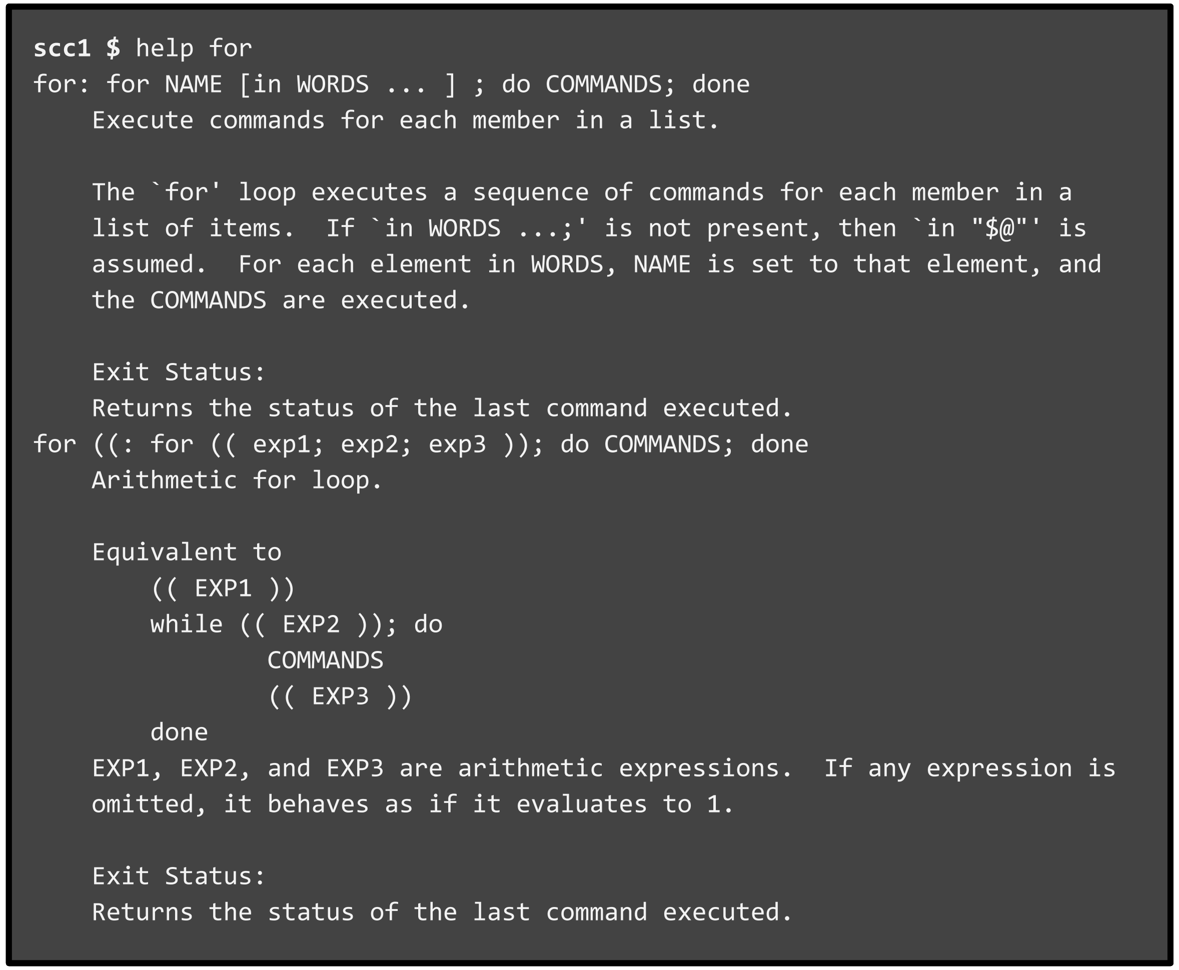
```
File: bash.info,  Node: Top,  Next: Introduction,  Prev: (dir),  Up: dir

Bash Features
*************

This text is a brief description of the features that are present in the
Bash shell (version 4.2, 28 December 2010).

   This is Edition 4.2, last updated 28 December 2010, of 'The GNU Bash
Reference Manual', for 'Bash', Version 4.2.

   Bash contains features that appear in other popular shells, and some
features that only appear in Bash.  Some of the shells that Bash has
borrowed concepts from are the Bourne Shell ('sh'), the Korn Shell
('ksh'), and the C-shell ('csh' and its successor, 'tcsh').  The
following menu breaks the features up into categories based upon which
one of these other shells inspired the feature.

   This manual is meant as a brief introduction to features found in
Bash.  The Bash manual page should be used as the definitive reference
on shell behavior.

* Menu:
```

Bash "**help**"

- Bash comes with built in help functionality
  - Just type "**`help`**"

- Read deeper into help chapters by searching specific keywords
  - "**`help [keyword]`**"

- "Help help"
- "Help for"

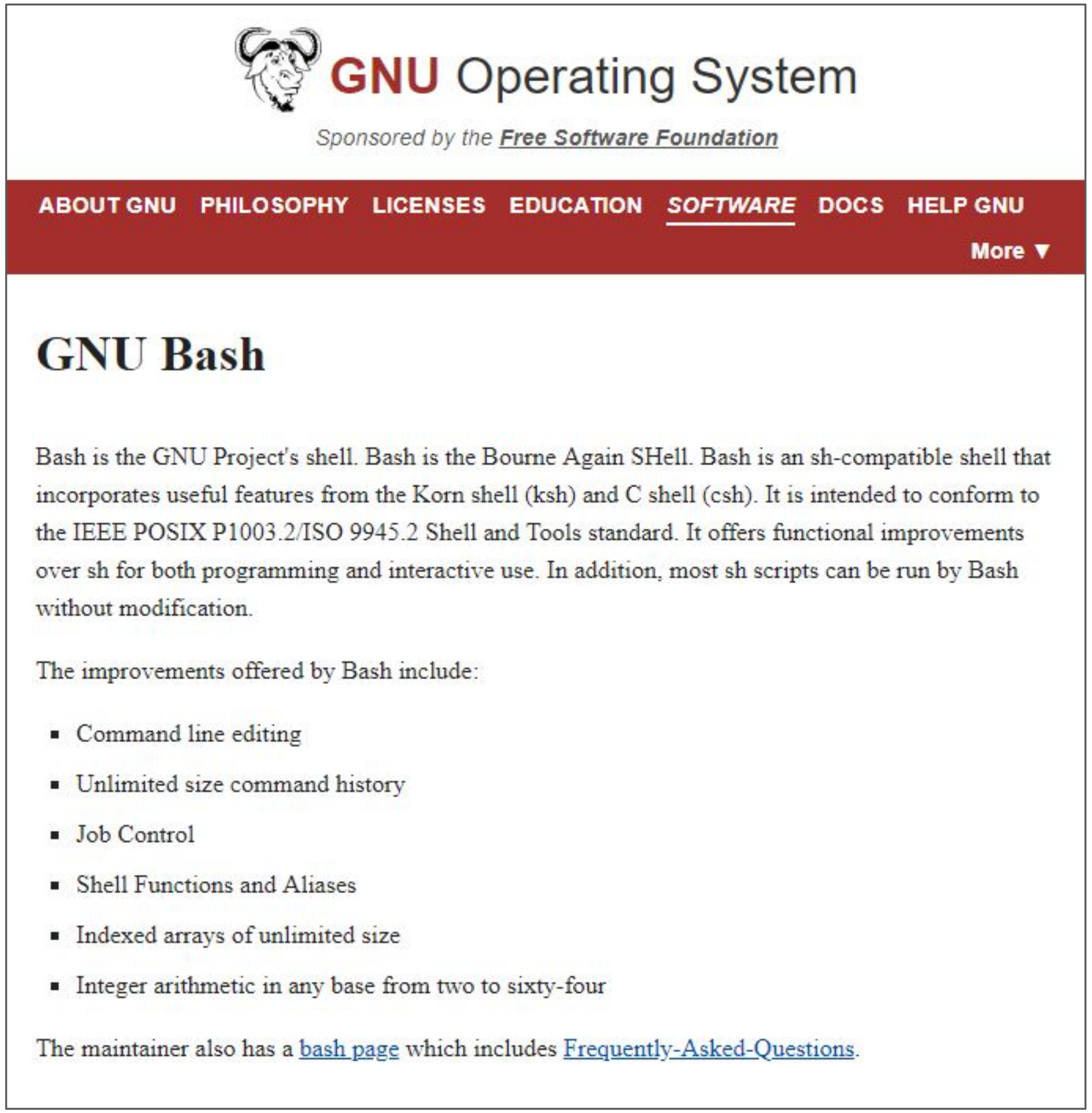


```
scc1 $ help for
for: for NAME [in WORDS ... ] ; do COMMANDS; done
    Execute commands for each member in a list.

    The `for' loop executes a sequence of commands for each member in a
    list of items.  If `in WORDS ...;' is not present, then `in "$@"' is
    assumed.  For each element in WORDS, NAME is set to that element, and
    the COMMANDS are executed.

    Exit Status:
    Returns the status of the last command executed.
for ((: for (( exp1; exp2; exp3 )); do COMMANDS; done
    Arithmetic for loop.

    Equivalent to
        (( EXP1 ))
        while (( EXP2 )); do
                COMMANDS
                (( EXP3 ))
        done
    EXP1, EXP2, and EXP3 are arithmetic expressions.  If any expression is
    omitted, it behaves as if it evaluates to 1.

    Exit Status:
    Returns the status of the last command executed.
```

Documentation

The official documentation is very good!

So good, you might even see some examples copied directly into this tutorial.



GNU Bash

Bash is the GNU Project's shell. Bash is the Bourne Again SHell. Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh). It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. It offers functional improvements over sh for both programming and interactive use. In addition, most sh scripts can be run by Bash without modification.

The improvements offered by Bash include:

- Command line editing
- Unlimited size command history
- Job Control
- Shell Functions and Aliases
- Indexed arrays of unlimited size
- Integer arithmetic in any base from two to sixty-four

The maintainer also has a bash page which includes Frequently-Asked-Questions.

https://www.gnu.org/software/bash

Command-line vs. Scripting

Recap of Command Line vs Script Definitions

Command-line

- Has a prompt
- Not saved
- One line at a time
- The text based way to interact with a computer

Script

- No prompt
- Is a file
- Still runs one line at a time
- Runs all the lines in file without interaction

Example CLI Task: Organize some downloaded data

```
[username@scc1 ~]$ cd /projectnb/scv/jpessin/introToBashScripting_sampleScripts/cli_script
[username@scc1 cli_script]$ ls data
LICENSE      sample1.chr1.bam  sample1.chr4.bam  sample2.chr1.bam  sample2.chr4.bam  sample3.chr1.bam  sample3.chr4.bam
README       sample1.chr2.bam  sample1.chr5.bam  sample2.chr2.bam  sample2.chr5.bam  sample3.chr2.bam  sample3.chr5.bam
report.html  sample1.chr3.bam  sample1.log       sample2.chr3.bam  sample2.log       sample3.chr3.bam  sample3.log
[username@scc1 cli_script]$ cd data
[username@scc1 data]$ mkdir sample1
[username@scc1 data]$ mv sample1.chr*.bam > sample1
-bash: sample1: Is a directory
[username@scc1 data]$ mv sample1.chr*.bam  sample1/
[username@scc1 data]$ cd sample1/
[username@scc1 sample1]$ ls sample1.* > sample1.fileset.txt
[username@scc1 sample1]$ less sample1.fileset.txt
[username@scc1 sample1]$ mv sample1.fileset.txt ../
[username@scc1 sample1]$ cd ..
[username@scc1 data]$ ls
LICENSE      sample1              sample2.chr1.bam  sample2.chr4.bam  sample3.chr1.bam  sample3.chr4.bam
README       sample1.fileset.txt  sample2.chr2.bam  sample2.chr5.bam  sample3.chr2.bam  sample3.chr5.bam
report.html  sample1.log          sample2.chr3.bam  sample2.log       sample3.chr3.bam  sample3.log
```

Example CLI Task (cont.)

```
[username@scc1 data]$ ls
LICENSE      sample1              sample2.chr1.bam  sample2.chr4.bam  sample3.chr1.bam  sample3.chr4.bam
README       sample1.fileset.txt  sample2.chr2.bam  sample2.chr5.bam  sample3.chr2.bam  sample3.chr5.bam
report.html  sample1.log          sample2.chr3.bam  sample2.log       sample3.chr3.bam  sample3.log
[username@scc1 data]$ mkdir sample2
[username@scc1 data]$ mv sample2.chr*.bam sample2
[username@scc1 data]$ mkdir sample3
[username@scc1 data]$ mv sample3.chr*.bam sample3
[username@scc1 data]$ ls
LICENSE  report.html  sample1.fileset.txt  sample2              sample2.log  sample3.fileset.txt  sample4              sample4.log
README   sample1      sample1.log          sample2.fileset.txt  sample3      sample3.log          sample4.fileset.txt
[username@scc1 data]$ mkdir logs
[username@scc1 data]$ mv sample*.log logs/
[username@scc1 data]$ rm LICENSE
rm: remove regular empty file 'LICENSE'? y
[username@scc1 data]$ rm README
rm: remove regular empty file 'README'? y
[username@scc1 data]$ ls
logs         sample1              sample2              sample3              sample4
report.html  sample1.fileset.txt  sample2.fileset.txt  sample3.fileset.txt  sample4.fileset.txt
```

Command-line Interface

- Difficult to read
- One-directional / Non-reproducible
  - What did I do last time?
  - What should someone do next time?
- Manual
- Potentially error-prone
- Wasn’t really that fast

Write a Script Instead

reorgData.sh

```
#!/bin/bash

# Take datadir from input
datadir=$1

cd $datadir

# Detect number of samples
numSamples=$(ls sample*.bam | cut -d. -f1 | uniq | wc -l)

# Reorg sample files into sample dirs
for sampleNum in $(seq 1 $numSamples); do
     mkdir sample$sampleNum
     mv sample$sampleNum*.chr*.bam sample$sampleNum/
     ls sample$sampleNum > sample$sampleNum.filelist.txt
done

# Organize Logs
mkdir logs
mv sample*.log logs/

# Remove extra files
rm -f LICENSE
rm -f README
```

```
scc1 $ ls data
LICENSE           sample1.chr5.bam  sample2.log
README            sample1.log       sample3.chr1.bam
report.html       sample2.chr1.bam  sample3.chr2.bam
sample1.chr1.bam  sample2.chr2.bam  sample3.chr3.bam
sample1.chr2.bam  sample2.chr3.bam  sample3.chr4.bam
sample1.chr3.bam  sample2.chr4.bam  sample3.chr5.bam
sample1.chr4.bam  sample2.chr5.bam  sample3.log

scc1 $ bash reorgData.sh data/

scc1 $ ls data
logs          sample1        sample2        sample3
report.html   sample1.files  sample2.files  sample3.files
```

Running Scripts: Interpreter

- Simply call the “bash” interpreter and provide the script.
- It will read line by line as if on the command line

This is what we did previously.

```
scc1 $ ls data
LICENSE          sample1.chr5.bam  sample2.log
README           sample1.log       sample3.chr1.bam
report.html      sample2.chr1.bam  sample3.chr2.bam
sample1.chr1.bam sample2.chr2.bam  sample3.chr3.bam
sample1.chr2.bam sample2.chr3.bam  sample3.chr4.bam
sample1.chr3.bam sample2.chr4.bam  sample3.chr5.bam
sample1.chr4.bam sample2.chr5.bam  sample3.log

scc1 $ bash reorgData.sh data/

scc1 $ ls data
logs        sample1       sample2       sample3
report.html sample1.files sample2.files sample3.files
```

Running Scripts: Executable

Files can be made “executable” on their own.

To do this, we need to:

- Provide interpreter information in script
- Set executable permission
- Run the script directly ./script

```
scc1 $  head -n 1 reorgData.sh
#!/bin/bash

scc1 $ ls -l
drwxr-sr-x 6 cjahnke scv 32768 Jun 1 2:36 data
-rw-r--r-- 1 cjahnke scv   453 Jun 1 2:37 reorgData.sh

scc1 $ chmod +x reorgData.sh

scc1 $ ls -l
drwxr-sr-x 6 cjahnke scv 32768 Jun 1 2:36 data
-rwxr-xr-x 1 cjahnke scv   453 Jun 1 2:37 reorgData.sh

scc1 $ ./reorgData.sh

scc1 $
```

Variables

Environment Variables

- Contain environment configuration
  - Typically for the shell, but other programs can set their own.
- Created automatically when logged in.
- Scope is global
  - Other programs can read/use them to know how to behave.
- Type “env” to see the full list.

```
scc1 $ echo $USER
cjahnke

scc1 $ echo $PWD
/usr3/bustaff/cjahnke

scc1 $ echo $HOSTNAME
scc1

scc1 $ env
MODULE_VERSION_STACK=3.2.10
XDG_SESSION_ID=c8601
HOSTNAME=scc1
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
TMPDIR=/scratch
SSH_CLIENT=128.197.161.56 55982 22
...
```

Shell Variables

- A character string to which a user assigns a value.
- Not real data, but could point to data (lists, file, device, etc)
- Shell variables have limited scope
  - only current shell
- Can create, assign, and delete.

```
scc1 $ myvar=foo
scc1 $ echo $myvar
foo

scc1 $ myvar=bar
scc1 $ echo $myvar
bar

scc1 $ unset myvar
scc1 $ echo $myvar

scc1 $

scc1 $ myvar=foo
scc1 $ bash
scc1 $ echo $myvar

scc1 $ exit
exit
scc1 $ echo $myvar
foo
```

Choosing a Variable Name and Style

Variable names cannot have spaces. Pick and try to stick to a style.

- CAPITALS
  - Environment variables and OS shell variables are usually capitalized.
- lowercase
  - Effective for simple scripts, hard to read if names are complicated (e.g. **`$mynewvar`**).
- Under_scores
  - Common alternative to spaces (e.g. **`$my_new_var`**). Bash does not accept hyphens.
- camelCase
  - Capitalization patterns are concise and easy enough to read (e.g **`$myNewVar`**).

Using variables: The dollar sign and quotes

- No quote
  - Simple. Bash shell interprets variable
- Escape Special Character (“\”)
  - The “$” is special and indicates a variable in Bash. The “\” escapes special behavior and instructs bash to treat it as a character.
- Single Quote
  - Literal. Exactly the contents.
- Double Quote
  - Interpreted. Allows variable expansion.

```
scc1 $ hi=Hello

scc1 $ echo $hi
Hello

scc1 $ echo \$hi
$hi

scc1 $ echo '$hi'
$hi

scc1 $ echo "$hi"
Hello
```

Using Variables: Strings, spaces, and quotes

Spaces are special too

- We can escape (“\”) the special behavior
- Or we can quote the string.
    - Single or double quotes are effectively the same if there is nothing to be interpreted.

```
scc1 $ hello0=Hello World
-bash: World: command not found
scc1 $ echo $hello0
Hello

scc1 $ hello1=Hello\ World
scc1 $ echo $hello1
Hello World

scc1 $ hello2='Hello World'
scc1 $ echo $hello2
Hello World

scc1 $ hello3="Hello World"
scc1 $ echo $hello3
Hello World
```

Build up simple script

myscript.sh

```
echo Hello World

myScriptVar=bar
echo “My working directory \$PWD
prints $PWD”

echo $myScriptVar
```

```
scc1 $ bash myscript.sh
Hello World
My working directory $PWD prints
/usr3/bustaff/cjahnke/bash
bar

scc1 $ echo $myScriptVar

scc1 $
```

Handling Arguments

Command-line Arguments in Bash

The command used to start a bash script passes the command information to the script as variables when it runs. This information is accessed through numbered variables where the “#” is the index of the information.

- $0 → The script name
- $1 → The first argument following the script name
- $2 → The second argument following the script name
- ...

Note: only 9 arguments are captured; after that, you need to be creative.

Simple Command Line Argument Example

cli_arg.sh

```bash
#!/bin/bash

# $0 is the script itself
echo '$0' is "$0"
# $1 is the first argument
echo '$1' is "$1"
# $2 is the second argument
echo '$2' is "$2"
```

Terminal

```
scc1 $

scc1 $ ./cli_arg.sh arg1 “2 items” 3rd
$0 is ./cli_arg.sh
$1 is arg1
$2 is 2 items
```

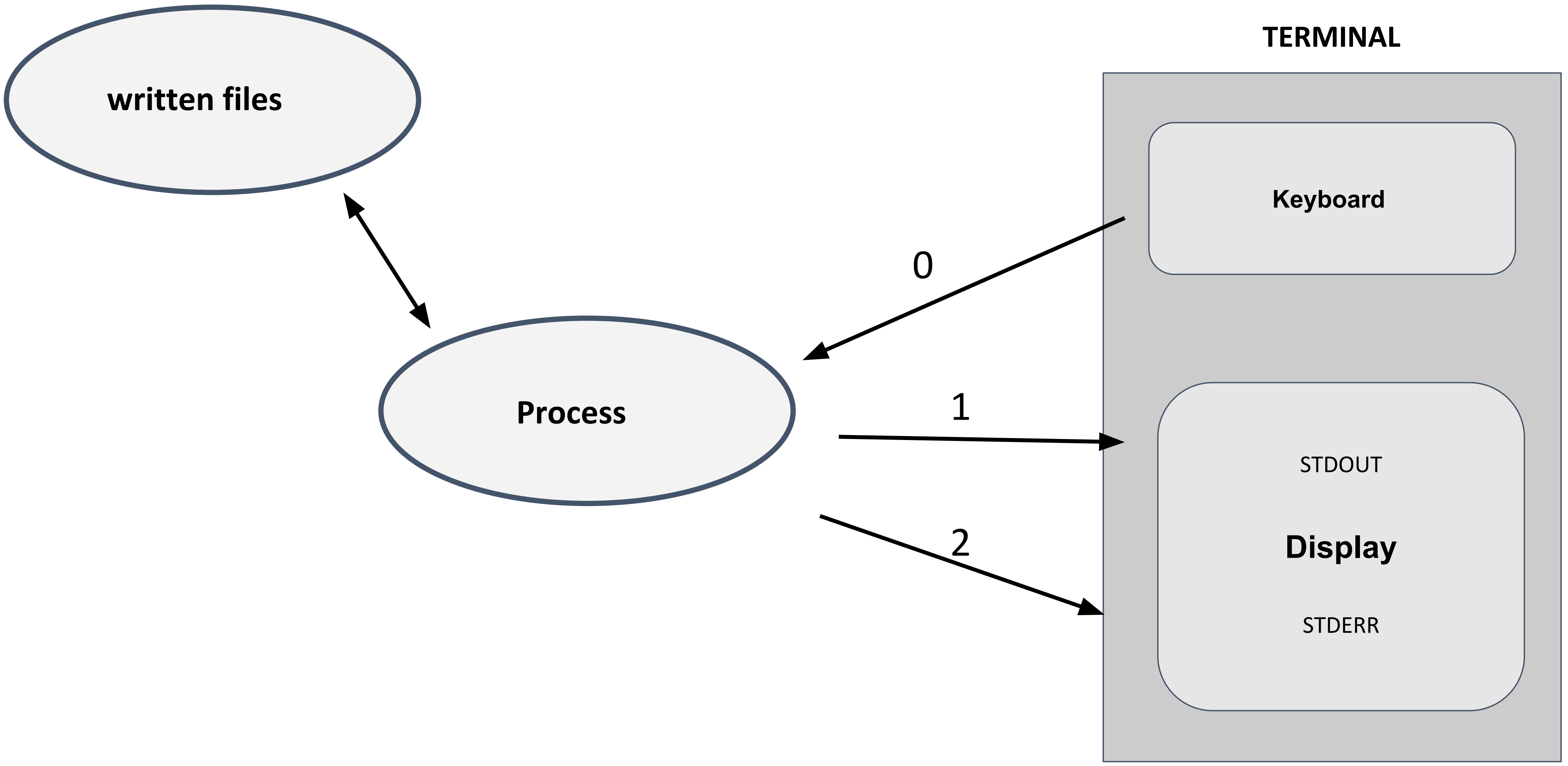
Standard I/O, Pipes, and Redirection

Jumping into Standard I/O

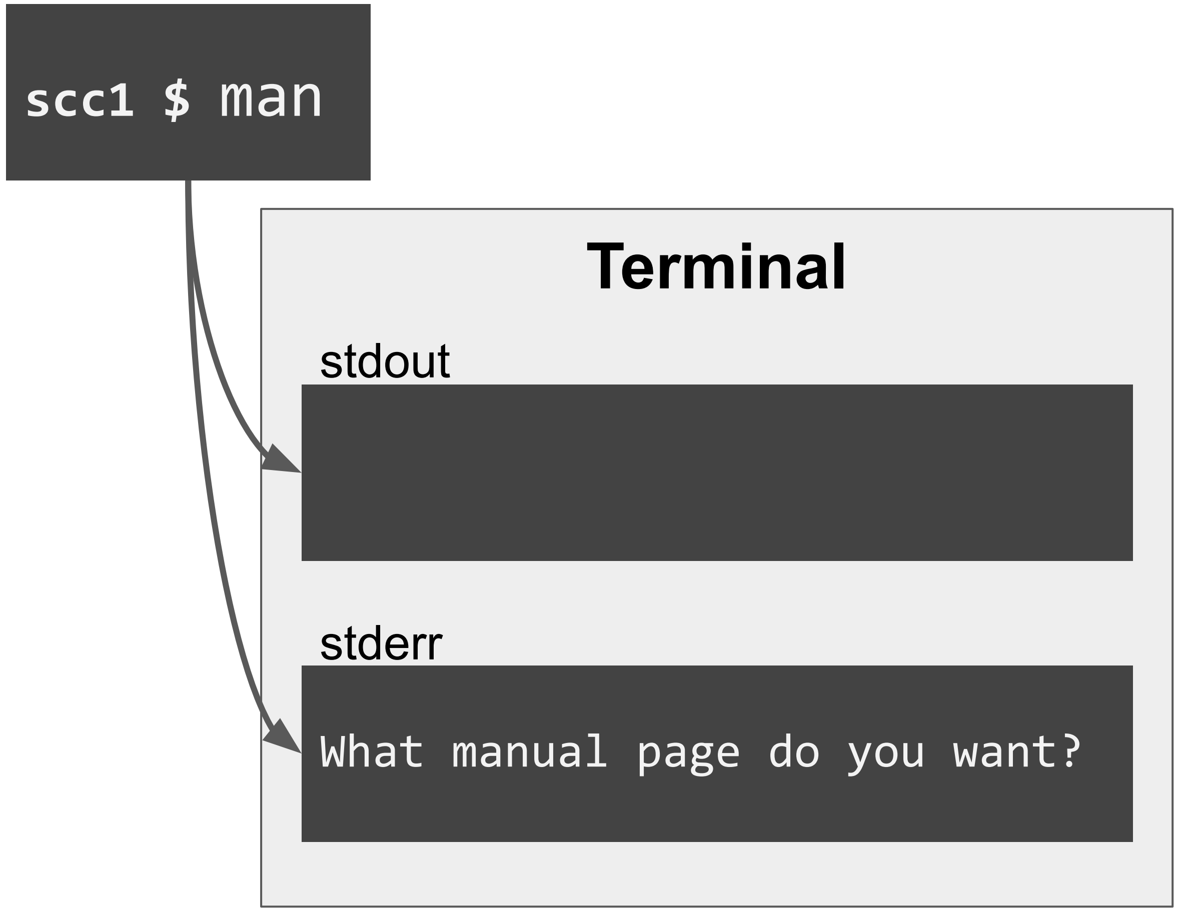
There are 3 standard methods of communicating with a program

Name	Shorthand	Purpose *	Stream ID
Standard In	Stdin	Command line inputs	0
Standard Out	Stdout	Normal output	1
Standard Error	Stderr	Error or other information	2

* What they are actually used for is entirely dependent on the program


written files
Process
TERMINAL
Keyboard
0
1
2
STDOUT
Display
STDERR

Standard Out & Standard Error



```
scc1 $ man
What manual page do you want?

scc1 $ man 1> man.stdout 2> man.stderr

scc1 $ cat man.stdout

scc1 $ cat man.stderr
What manual page do you want?
```

Pipes

- Pipes (“ | ”) redirect the standard output of a command to the standard input of another command.

- Example:

```
[cjahnke@scc1 ~]$ cat sample.vcf  |  cut -f1,2,7  |  sort -k3
```

```
     cat sample.vcf               |      cut -f1,2,7     |     sort -k3

#CHROM  POS      ID         REF  ...  | #CHROM POS     FILTER  | #CHROM POS     FILTER
3       14370    rs6054257  G    ...  | 3      14370   PASS    | 1      1110696 PASS
2       17330    .          T    ...  | 2      17330   q10     | 3      1230237 PASS
1       1110696  rs6040355  A    ...  | 1      1110696 PASS    | 3      14370   PASS
3       1230237  .          T    ...  | 3      1230237 PASS    | 6      1234567 PASS
6       1234567  microsat1  GTCT ...  | 6      1234567 PASS    | 2      17330   q10
```

Redirection

- The "**>**" symbol redirects the standard output (default) of a command to a file.

Redirection	Description
`COMMAND <   filename`	Input - Directs a file ☆
`COMMAND <<  stream`	Input - Directs a stream literal
`COMMAND <<< string`	Input - Directs a string
`COMMAND >   filename`	Output - Writes output to file (will "clobber") ☆
`COMMAND >>  filename`	Output - Appends output to file ☆

- Example:

```
[cjahnke@scc1 ~]$ cat sample.vcf | cut -f1,2,7 | sort -k3 > sorted.txt
```

Many characters use or modify this behavior

● A < file	Use the contents of file as input for A
● B > file	Create a new file and write the standard out of B there (overwrites)
● C >> file	If file exists append standard out of C to file, if file does not exist create it
● D 2> file	Create a new file and write the standard err of D there
● E &> file	Combined the standard error and standard out and write to file
● F | G	Use the standard out of F as the standard in of G
● H |& K	Combine the standard out and err of H and use as the standard in of K
● M | tee file	Write the standard out of M to both the terminal and to file

```
scc1 $ module -t avail |& tee allmodules | grep python
```

Control Structures

Loops, Conditionals, and Tests

Loops

★ - **for**
    - Expand expr and execute commands once for each member in the resultant list, with name bound to the current member.

```
for (( expr )) ; do
    commands
done
```

- **while**
    - Execute consequent-commands as long as test-commands has an exit status of zero.

```
while test-commands; do
    consequent-commands
done
```

- **until**
    - Execute consequent-commands as long as test-commands has an exit status which is not zero.

```
until test-commands; do
    consequent-commands
done
```

For Loop (Simple)

- A simple countdown

- Components:
  - The “i” becomes our iterating variable “**`$i`**”
  - List expansion of `{5..1}` is `5 4 3 2 1`
  - “**`echo`**” command prints line
  - “**`sleep`**” command waits for 1 second

- Take each item, one at a time, perform operation in loop. Advance until end of list

```
scc1 $ \
for i in {5..1}; do
     echo "$i seconds left"
     sleep 1s
done

5 seconds left
4 seconds left
3 seconds left
2 seconds left
1 seconds left

scc1 $
```

For Loop (In Practice)

Let's iterate on something more interesting

- Input Items can be called with $@

```
#!/bin/bash

# This loop iterates over input items

for input in "$@"; do
   echo "$input"
done
```

```
scc1 $ bash forloop1.sh a b c
a
b
c

scc1 $ bash forloop1.sh a "b c" d
a
b c
d
```

For Loop (In Practice)

```bash
#!/bin/bash

# This script takes one argument, a
# directory, and prints the basename of
# contents.

echo $0
echo ""
echo $1

for doc in "$1"/*; do
    shortname=$(basename $doc)
    # now that we have the name, we
    # could do something interesting
    echo "  $shortname"
done
```

```
scc1 $ bash forloop2.sh ~/bash
forloop2.sh

/usr3/bustaff/cjahnke/bash
  forloop1.sh
  forloop2.sh
  myscript.sh
```

Syntax - Best Practice

```
for content in *; do
   echo “$content”
done
```

```
for content in *
   do
   echo “$content”
done
```

```
For content in *
   do echo “$content”
done
```

```
For content in *; do echo “$content” ; done
```

https://google.github.io/styleguide/shell.xml#Loops

Conditional Constructs

- test “`[[ .. ]]`”
  - Evaluates expression inside brackets and returns 0 (TRUE) or 1 (FALSE)
- `if`
  - Executes commands following conditional logic.
- `case`
  - Selectively execute commands corresponding to pattern matching.
  - Like if/then statements, but usually used for parsing inputs and determining flow.
- `select`
  - Used for creating user input/selectable menus, executes commands on selection.
- Arithmetic “`(( .. ))`”
  - Will perform arithmetic. Use caution, precision can be tricky.

Tests “[[..]]”

Double square brackets return an exit status of 0 (true) or 1* (false) depending on the evaluation of the conditional expression inside.

- Standard Test
  - `[[ expression ]]`
- Negative Test
  - `[[ ! expression ]]`
- AND Test
  - `[[ expression1 && expression2 ]]`
- OR Test
  - `[[ expression1 || expression2 ]]`

```
scc1 $ [[ 1 == 1 ]] ; echo $?
0

scc1 $ [[ 1 == 2 ]] ; echo $?
1

scc1 $ [[ ! cow == dog ]]; echo $?
0

scc1 $ [[ 1 == 2 && cow == cow ]]; echo $?
1

scc1 $ [[ 1 == 1 || cow == dog ]]; echo $?
0
```

* Anything >=1 is considered false. Programs may have many possible exit codes. 0 is success, everything else is a descriptive error.

If Statement (Simple)

- An “**`if`**“ statement executes commands based on conditional tests.
- The “**`then`**” keyword begins commands to execute if conditional is true.
- An “**`elif`**” keyword can extend an if statement for multiple conditions.
  - The tests are performed in order.
    Only the first true test is run.
- A catch-all “**`else`**” keyword is used to execute commands if no conditions are met.
- The “**`fi`**” keyword closes the statement

```
if test-commands; then
   consequent-commands;
elif more-test-commands; then
   more-consequents;
else
   alternate-consequents;
fi
```

If-Then in Practice

Let’s say we are in a directory with the following objects:

- TheJungleBook.txt
- d
- newfile.sh
- test.qsub

I can iterate through all the files.

If it is a file, echo that it is a file

If it is a directory, echo that it is a directory

```
scc1 $ ls
TheJungleBook.txt  d  newfile.sh  test.qsub

scc1 $ \
for contents in *; do
    if [[ -f "$contents" ]] ; then
        echo "$contents" is a file
    elif [[ -d "$contents" ]]; then
        echo $contents is a dir
    else
        echo "not identified"
    fi
done

TheJungleBook.txt is a file
d is a dir
newfile.sh is a file
test.qsub is a file
```

practice some loops

First get the sample files

```
$ cp /projectnb/scv/bash_examples.tar .

$ tar xf bash_examples.tar

$ cd bash_examples

$ ls

    answer_scripts    numbers    rebuildSentence
```

Each file has a word from a sentence, try to reconstruct the sentence

Each file has a word from a sentence, try to reconstruct the sentence

```
for task in {0..13}; do

    cat "$task".txt >> file

done

tr '\n' ' ' < file
```

Each file has a word from a sentence, try to reconstruct the sentence

```
for task in {0..13}; do
    cat “$task”.txt >> file
done
tr '\n' ' ' < file
```

returns:

```
    Scripting in bash makes many many things much easier, like putting this
    sentence together.
```

SCC Job Submission Example

using a loop to submit jobs on SCC with names.

step 1 create a file with the names

```
$ for file in *_1.txt; do echo "$file" >>
filenames.txt; done
$ cat filenames.txt
AG_1.txt
aA_1.txt
ab_1.txt
ac_1.txt
ad_1.txt
af_1.txt
ag_1.txt
ah_1.txt
ai_1.txt
aj_1.txt
order_1.txt
outof_1.txt
```

using a loop to submit jobs on SCC with names.

step 1 create a file with the names

step 2 get the number of filenames

```
$ for file in *_1.txt; do echo "$file" >>
filenames.txt; done
$ cat filenames.txt
AG_1.txt
aA_1.txt
ab_1.txt
ac_1.txt
ad_1.txt
af_1.txt
ag_1.txt
ah_1.txt
ai_1.txt
aj_1.txt
order_1.txt
outof_1.txt

$ wc -l filenames.txt
12 filenames.txt
```

using a loop to submit jobs on SCC with names.

step 1 create a file with the names

step 2 get the number of filenames

step 3 create a submission script that accepts inputs (remember to chmod +x)

```
#!/bin/bash -l

#$ -P tutorial

value1=$(cat "$1")
value2=$(cat "$2")

valueNew=$(( $value1 + $value2 ))

echo "$1" Has a value of $value1
echo "$2" Has a value of $value2
echo These sum to $valueNew
```

using a loop to submit jobs on SCC with names.

step 1 create a file with the names

step 2 get the number of filenames

step 3 create a submission script that
accepts inputs (remember to chmod +x)

step 3a (if practical) test it locally

step 3b test a single qsub

```
$ ./fileadder.qsub aA_1.txt aA_2.num
aA_1.txt Has a value of 30565
aA_2.num Has a value of 16775
These sum to 47340

$ qsub ./fileadder.qsub aA_1.txt aA_2.num
Your job 6853253 ("fileadder.qsub") has been
submitted
```

using a loop to submit jobs on SCC with names.

step 1 create a file with the names

step 2 get the number of filenames

step 3 create a submission script that accepts inputs (remember to chmod +x)

step 3a (if practical) test it locally

step 3b test a single qsub

step 4 Create a file to loop the submission

step 4a set up for a test the loop

```
#!/bin/bash -l

for i in {1..12}; do
    name=$(sed -n -e "$i p" filenames.txt)
    base=$(basename "$name" _1.txt)

   #qsub fileadder.qsub "$base"_1.txt "$base"_2.num
    fileadder.qsub "$base"_1.txt "$base"_2.num
    echo $i "$base"

done
```

using a loop to submit jobs on SCC with names.

step 1 create a file with the names

step 2 get the number of filenames

step 3 create a submission script that accepts inputs (remember to chmod +x)

step 3a (if practical) test it locally

step 3b test a single qsub

step 4 Create a file to loop the submission

step 4a set for a test loop

step 4b reset for submissions

```
#!/bin/bash -l

for i in {1..12}; do
    name=$(sed -n -e "$i p" filenames.txt)
    base=$(basename "$name" _1.txt)

    qsub fileadder.qsub "$base"_1.txt "$base"_2.num
    # fileadder.qsub "$base"_1.txt "$base"_2.num
    # echo $i "$base"

done
```

using a loop to submit jobs on SCC with names.

step 1 create a file with the names

step 2 get the number of filenames

step 3 create a submission script that accepts inputs (remember to chmod +x)

step 3a (if practical) test it locally

step 3b test a single qsub

step 4 Create a file to loop the submission

step 4a set for a test loop

step 4b reset for submissions

step 5 submit

```
$ ./submit_fileadder
Your job 6853078 ("fileadder.qsub") has been submitted
Your job 6853079 ("fileadder.qsub") has been submitted
Your job 6853080 ("fileadder.qsub") has been submitted
Your job 6853081 ("fileadder.qsub") has been submitted
Your job 6853082 ("fileadder.qsub") has been submitted
Your job 6853083 ("fileadder.qsub") has been submitted
Your job 6853084 ("fileadder.qsub") has been submitted
Your job 6853085 ("fileadder.qsub") has been submitted
Your job 6853086 ("fileadder.qsub") has been submitted
Your job 6853087 ("fileadder.qsub") has been submitted
Your job 6853088 ("fileadder.qsub") has been submitted
Your job 6853089 ("fileadder.qsub") has been submitted
```