Intro to BASH

Group 2

First Year Bootcamp, 2016

Table of Contents

What are we bashing and why?

We assume that you have come in to this group with some knowledge of basic shell use (*ls, mv, ...*). We'll discuss:

What are we bashing and why?

We assume that you have come in to this group with some knowledge of basic shell use (*ls*, *mv*, *...*). We'll discuss:

- A bit about Stanford's computing cluster resources, remind you how to access them with *ssh* & *scp*, and show you how simple hosting websites on them is.

What are we bashing and why?

We assume that you have come in to this group with some knowledge of basic shell use (*ls*, *mv*, *...*). We'll discuss:

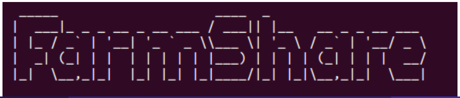
- A bit about Stanford's computing cluster resources, remind you how to access them with *ssh* & *scp*, and show you how simple hosting websites on them is.
- Shell scripting (in a necessarily shallow way, since we only have part of an hour and long textbooks can be written on the subject).

Table of Contents

There are computers other than mine? (Stanford computing clusters)

Stanford has a number of computing clusters that you can log into remotely, including:

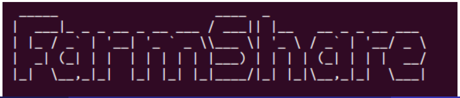
- Class or research (or web hosting etc.) machines:
  - **corn:** The workhorse systems, general purpose servers for running small jobs, accessing your shared file space, hosting your website, etc.



There are computers other than mine? (Stanford computing clusters)

Stanford has a number of computing clusters that you can log into remotely, including:

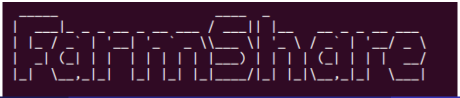
- Class or research (or web hosting etc.) machines:
    - **corn:** The workhorse systems, general purpose servers for running small jobs, accessing your shared file space, hosting your website, etc.
    - **rye:** (somewhat old) GPU machines, still fairly powerful but may not be compatible with newer software.



There are computers other than mine? (Stanford computing clusters)

Stanford has a number of computing clusters that you can log into remotely, including:

- Class or research (or web hosting etc.) machines:
  - **corn:** The workhorse systems, general purpose servers for running small jobs, accessing your shared file space, hosting your website, etc.
  - **rye:** (somewhat old) GPU machines, still fairly powerful but may not be compatible with newer software.
  - **barley:** machines with a job submission system for high memory/high cpu tasks.



There are computers other than mine? (Stanford computing clusters)

Stanford has a number of computing clusters that you can log into remotely, including:

- Class or research (or web hosting etc.) machines:
  - **corn:** The workhorse systems, general purpose servers for running small jobs, accessing your shared file space, hosting your website, etc.
  - **rye:** (somewhat old) GPU machines, still fairly powerful but may not be compatible with newer software.
  - **barley:** machines with a job submission system for high memory/high cpu tasks.
- Research only clusters:
  - **sherlock:** 130 computing nodes, both general purpose and specialized nodes (including GPU nodes with 8 Tesla K20Xm cards and 256 GB RAM, and “big data” nodes with 1.5 TB RAM). PI must request access for you before you can use sherlock.

Connecting to corn

- We'll show you how to connect to one of Stanford's **corn** servers, which are available for general use. Run *ssh your-SUNet-ID@corn.stanford.edu*.
- When prompted for your password, type the password that corresponds to your sunet id. It won't show any characters being typed, just type the password and hit enter. Note: you will probably need to use two-factor authentication, and the timeout is relatively short, have your phone ready.

```
http://farmshare.stanford.edu
```

Connecting to corn

- We'll show you how to connect to one of Stanford's **corn** servers, which are available for general use. Run *ssh your-SUNet-ID@corn.stanford.edu*.
- When prompted for your password, type the password that corresponds to your sunet id. It won't show any characters being typed, just type the password and hit enter. Note: you will probably need to use two-factor authentication, and the timeout is relatively short, have your phone ready.
- You should see a welcome screen. *ls* and look around, anything in the *~/WWW/* folder will become a part of your website at *web.stanford.edu/~your-SUNet-ID/*.

```
http://farmshare.stanford.edu
```

A toy website

- Let's try creating a (very) simple stanford website for you, and remind you how to use *scp* to move things between your computer and servers along the way.

A toy website

- Let's try creating a (very) simple stanford website for you, and remind you how to use *scp* to move things between your computer and servers along the way.
- Create a text file called simplewebsite.txt somewhere on your computer, and put some text in it (like "Hello World!").

A toy website

- Let's try creating a (very) simple stanford website for you, and remind you how to use *scp* to move things between your computer and servers along the way.
- Create a text file called simplewebsite.txt somewhere on your computer, and put some text in it (like "Hello World!").
- Now, open a new terminal on your own computer (leave the other terminal with the ssh connection open, we'll go back to it in a bit). In the new terminal, *cd* to the directory where you saved simplewebsite.txt.

A toy website (cont.)

- To copy *simplewebsite.txt* to the ∼/*WWW* folder on the server, run *scp simplewebsite.txt your-SUNetID@corn.stanford.edu:∼/WWW/*

A toy website (cont.)

- To copy *simplewebsite.txt* to the ∼/*WWW* folder on the server, run *scp simplewebsite.txt your-SUNetID@corn.stanford.edu:∼/WWW/*
- If all went well, you should see the name of the file followed by 100% (since the file is so small, the transfer will complete very rapidly).
- If so, try opening your web browser and going to *web.stanford.edu/∼your-SUNetID/simplewebsite.txt*

A toy website (cont.)

- To copy *simplewebsite.txt* to the *∼/WWW* folder on the server, run *scp simplewebsite.txt your-SUNetID@corn.stanford.edu:∼/WWW/*
- If all went well, you should see the name of the file followed by 100% (since the file is so small, the transfer will complete very rapidly).
- If so, try opening your web browser and going to *web.stanford.edu/∼your-SUNetID/simplewebsite.txt*
- Do you see your file? Congratulations! You've got a very basic website now. You can use the farmshare system to host experiments that you run online (on websites like mTurk, which are becoming a very popular way to run fast cheap experiments), to create a website for yourself so that people can look you up, etc.

Cleaning up the website

- Why did we leave the ssh connection open in the other terminal? Because (hopefully) it allowed you to run *scp* without having to log in to the server again (*scp* is built on *ssh* and uses the same connection). Also, we might want to do something with the file on the server once we have uploaded it.

Cleaning up the website

- Why did we leave the ssh connection open in the other terminal? Because (hopefully) it allowed you to run *scp* without having to log in to the server again (*scp* is built on *ssh* and uses the same connection). Also, we might want to do something with the file on the server once we have uploaded it.
- For instance, you might not want the world to be able to see this file forever, so change to the *WWW* directory and remove the file (with *rm*).

Cleaning up the website

- Why did we leave the ssh connection open in the other terminal? Because (hopefully) it allowed you to run *scp* without having to log in to the server again (*scp* is built on *ssh* and uses the same connection). Also, we might want to do something with the file on the server once we have uploaded it.
- For instance, you might not want the world to be able to see this file forever, so change to the *WWW* directory and remove the file (with *rm*).
- Finally, close your connection to the server by typing *exit*.

Sadness

Unfortunately Stanford's servers do not allow public key authentication for login. Instead, you must use Kerberos if you want to have easier login (and it's required for some clusters, such as sherlock). To find out more, check out `https://web.stanford.edu/group/farmshare/cgi-bin/wiki/index.php/Advanced_Connection_Options`

Table of Contents

Being lazy (a hands-on intro to shell scripting)

- One of the main reasons bash scripting is useful is that you can arrange commands into scripts, which can be used repeatedly (if you have a frequent task you do like converting data files, etc.).

Being lazy (a hands-on intro to shell scripting)

- One of the main reasons bash scripting is useful is that you can arrange commands into scripts, which can be used repeatedly (if you have a frequent task you do like converting data files, etc.).
- This allows you to be lazy (you don't have to do things manually).

Being lazy (a hands-on intro to shell scripting)

- One of the main reasons bash scripting is useful is that you can arrange commands into scripts, which can be used repeatedly (if you have a frequent task you do like converting data files, etc.).
- This allows you to be lazy (you don't have to do things manually).
- Automation with a script that can be used again is also a better research practice (if you're trying to do the same operation on a bunch of files, you might miss a file or apply the operation twice, and when you look back while writing the paper, you won't remember what you did).

Being lazy (a hands-on intro to shell scripting)

- One of the main reasons bash scripting is useful is that you can arrange commands into scripts, which can be used repeatedly (if you have a frequent task you do like converting data files, etc.).
- This allows you to be lazy (you don't have to do things manually).
- Automation with a script that can be used again is also a better research practice (if you're trying to do the same operation on a bunch of files, you might miss a file or apply the operation twice, and when you look back while writing the paper, you won't remember what you did).
- To download the scripts we're discussing, run the following in your terminal:

```
curl -L http://bit.ly/2cFq403 | tar xz
```

Anonymizing participants (files, for loops, and variables)

Here's a simple script that creates copies of some data files with the participants identifier replaced with an integer.

anonymize.sh

```
#!/bin/bash
mkdir ../anonymized_data/
i=0
for f in data_subject_*.json
do
  cp $f ../anonymized_data/data_subject_${i}.json
  i=$((i+1))
done
```

To run this script, you would save it as a .sh file, and then run it by calling it by name (e.g. *./anonymize.sh*). Let's go through the script piece by piece and see how it works.

Anonymizing participants (files, for loops, and variables)

-
  ```
  #!/bin/bash
  ```

  This line tells the shell that you want to run this script with bash. If you wanted to make a directly executable script in another language (like python) you could just replace the */bin/bash* part with the path to the interpreter for that language (which you can often find with the command *which*, e.g. *which python*).

Anonymizing participants (files, for loops, and variables)

-
  ```
  #!/bin/bash
  ```

  This line tells the shell that you want to run this script with bash. If you wanted to make a directly executable script in another language (like python) you could just replace the */bin/bash* part with the path to the interpreter for that language (which you can often find with the command *which*, e.g. *which python*).

-
  ```
  mkdir ../anonymized_data/
  ```

  This line creates a directory called *anonymized_data* in the parent of the current working directory.

Anonymizing participants (files, for loops, and variables)

-
  ```
  i=0
  ```

  This line creates a variable called *i* and sets it to 0. Note the lack of spaces! Variable assignment statements in bash cannot have spaces before or after the equals sign. (Add in some spaces and see if you can figure out what goes wrong.)

Anonymizing participants (files, for loops, and variables)

-
  ```
  i=0
  ```

  This line creates a variable called *i* and sets it to 0. Note the lack of spaces! Variable assignment statements in bash cannot have spaces before or after the equals sign. (Add in some spaces and see if you can figure out what goes wrong.)

-
  ```
  for f in data_subject_*.json
  do
  ...
  done
  ```

  This loop finds all files in the current directory which match the pattern, and assigns one to the variable *f*, runs the body of the loop with that assignment, and then assigns the next file to *f* and repeats.

Anonymizing participants (files, for loops, and variables)

-
  ```
  cp $f ../anonymized_data/data_subject_${i}.
      ↪ json
  ```

  This line copies the files *$f* to the new directory we created, and renames it *data_subject_${i}.json*. Notice that when referencing a variable (but not when creating it or assigning to it!), you put a $ in front of its name. The brackets around the *i* delimit the variable name. They aren't strictly necessary here, but can you think of a place they would be?

Anonymizing participants (files, for loops, and variables)

-
  ```
  cp $f ../anonymized_data/data_subject_${i}.
      ↪ json
  ```

  This line copies the files *$f* to the new directory we created, and renames it *data_subject_${i}.json*. Notice that when referencing a variable (but not when creating it or assigning to it!), you put a $ in front of its name. The brackets around the *i* delimit the variable name. They aren't strictly necessary here, but can you think of a place they would be?

-
  ```
  i=$((i+1))
  ```

  This line increments the variable *i*. The *$((...))* are how you tell bash to do arithmetic. (Note there are many other ways this line could be written, bash does include operators such as $+=$ and $++$ that you may be familiar with from other languages.)

Anonymizing participants (files, for loops, and variables)

anonymize.sh

```
#!/bin/bash
mkdir ../anonymized_data/
i=0
for f in data_subject_*.json
do
  cp $f ../anonymized_data/data_subject_${i}.json
  i=$((i+1))
done
```

Putting it all together, this script makes a new directory, loops through the data files in the current directory and creates a copy of each in the new directory with the subject identifier replaced with an anonymous ID number.

Files, for loops, and variables exercise

anonymize.sh

```
#!/bin/bash
mkdir ../anonymized_data/
i=0
for f in data_subject_*.json
do
  cp $f ../anonymized_data/data_subject_${i}.json
  i=$((i+1))
done
```

Using this code as a guide, try to write (and test!) a script that loops through all text files in the current directory and prints their name and their first line. (Hint: check out the commands *echo* and *head*, you can use *man* to find out more about them, e.g. *man echo*.)

Files, for loops, and variables exercise

Here's one possible answer:

text_preview.sh

```
#!/bin/bash
for f in *.txt
do
    echo $f
    head -n 1 $f
done
```

TSV to CSV (Conditionals, arguments, and streams)

Here's an example of a script that will convert tab-separated value files to comma-separated. It has a few fancier features than the previous script: it takes as an argument the directory to convert files in, and performs some basic error-handling.

tsv_to_csv.sh

```
#!/bin/bash
these_files=(${1}/*.tsv)
if [ ! -e "${these_files[0]}" ]
then
    echo Exiting, no .tsv files found in $1
    exit 1
fi
for f in ${1}/*.tsv
do
    sed s/\\t/,/g $f > ${f%.tsv}.csv
done
```

TSV to CSV (Conditionals, arguments, and streams)

-
  ```
  these_files=(${1}/*.tsv)
  ```

  This creates a variable called these_files, and stores in it the tsv files in the variable `$1`. You'll notice that we haven't defined this variable. (In fact, you can't.) That's because `$1` refers to the first argument passed to this script on the command line when it was run.

TSV to CSV (Conditionals, arguments, and streams)

-
  ```
  these_files=(${1}/*.tsv)
  ```

  This creates a variable called these_files, and stores in it the tsv files in the variable $1. You'll notice that we haven't defined this variable. (In fact, you can't.) That's because $1 refers to the first argument passed to this script on the command line when it was run.

-
  ```
  if [ ! -e "${these_files[0]}" ]
  then
  ...
  fi
  ```

  This construct is a conditional, it only executes the enclosed code under the condition specified in the [...]. What's the condition? First, The *-e* checks if the first of `$these_files` exists (as a file), and the *!*, negates the value of whatever expression comes after it. Thus, the code in this chunk only executes if no TSV files exist in the directory that was passed.

TSV to CSV (Conditionals, arguments, and streams)

-
  ```
  echo Exiting, no .tsv files found in $1
  exit 1
  ```

  The echo line tells the user what went wrong, and the exit line exits the script with exit code 1, which means something went wrong. (Exit codes are useful when one script calls another script or program, they allow graceful error handling.)

TSV to CSV (Conditionals, arguments, and streams)

-
  ```
  echo Exiting, no .tsv files found in $1
  exit 1
  ```

  The echo line tells the user what went wrong, and the exit line exits the script with exit code 1, which means something went wrong. (Exit codes are useful when one script calls another script or program, they allow graceful error handling.)

-
  ```
  for f in ${1}/*.tsv
  do
  ...
  done
  ```

  Much like the *for* loop we used previously, but loops over .tsv files in the directory $1.

TSV to CSV (Conditionals, arguments, and streams)

-
  ```
  sed s/\\t/,/g $f > ${f%.tsv}.csv
  ```
  There's a lot going on in this line.

TSV to CSV (Conditionals, arguments, and streams)

-
  ```
  sed s/\\t/,/g $f > ${f%.tsv}.csv
  ```

  There's a lot going on in this line.

  - `sed s/\\t/,/g $f` invokes the command *sed* (stream editor) on the file *$f* to replace tabs with commas, and then outputs the result.

TSV to CSV (Conditionals, arguments, and streams)

-
  ```
  sed s/\\t/,/g $f > ${f%.tsv}.csv
  ```

  There's a lot going on in this line.

  - `sed s/\\t/,/g $f` invokes the command *sed* (stream editor) on the file *$f* to replace tabs with commas, and then outputs the result.
  - > is an operator which redirects the output stream from sed. We won't have time to talk about streams in detail, but what > basically does is take output from the thing on the left (that would normally be printed) and save it to the file on the right. (Try running `echo Hello world! > this_is_a_test.txt` to see an example.)

TSV to CSV (Conditionals, arguments, and streams)

-
  ```
  sed s/\\t/,/g $f > ${f%.tsv}.csv
  ```

  There's a lot going on in this line.

  - `sed s/\\t/,/g $f` invokes the command *sed* (stream editor) on the file *$f* to replace tabs with commas, and then outputs the result.
  - > is an operator which redirects the output stream from sed. We won't have time to talk about streams in detail, but what > basically does is take output from the thing on the left (that would normally be printed) and save it to the file on the right. (Try running `echo Hello world! > this_is_a_test.txt` to see an example.)
  - `${f%.tsv}.csv` uses some of bash's fancy variable access abilities to get the filename but remove the .tsv extension, and then gives it a .csv extension instead. (% removes the pattern following it from the end of the variable's value.)

Conditionals, arguments, and streams exercise

```
#!/bin/bash
if [ ! -e ${1}/*.tsv ]
then
    echo Exiting, no .tsv files found in $1
    exit 1
fi
for f in ${1}/*.tsv
do
    sed s/\\t/,/g $f > ${f%.tsv}.csv
done
```

You might want to add several features, such as:

- distinguishing between a directory which does not exist and a directory which does exist but doesn't have .tsv files
- having a default directory (such as the current working directory) if no argument is supplied

Try to modify the above code to include one or both of these (hint: check out the variable $#, and read `man test` for the comparison operators).

Conditionals, arguments, and streams exercise

A possible solution for both:

```
#!/bin/bash
if [ $# -gt 0 ]; then
    if [ ! -d "$1" ]; then
        echo Exiting, $1 is not a directory!
        exit 1
    fi
    translation_dir=$1
else
    translation_dir=.
fi
if [ ! -e ${translation_dir}/*.tsv ]; then
    echo Exiting, no .tsv files found in $translation_dir
    exit 1
fi
for f in ${translation_dir}/*.tsv; do
    sed s/\\t/,/g $f > ${f%.tsv}.csv
done
```

Backup script (SSH review, more streams)

Here's a simple script to back up important files to FarmShare (this is just an example, note that GitHub, which you'll learn about later, provides a better place to back up your work in many cases).

backup.sh

```
#!/bin/bash
today=$(date +%F)
tobackup=~/to_backup
server=your-SUNet-ID@corn.stanford.edu
tar czf - $tobackup | ssh $server '( cd ~/
    ↪ Documents/backup_files/ ; cat > '${today}'.
    ↪ tar.gz )'
```

Backup script (SSH review, more streams)

-
  ```
  today=$(date +%F)
  ```

  This creates a variable called today, and stores the date in it (the `+%F` is an argument telling date what format to output). the `$(...)` essentially tells bash to run the commands inside the parentheses and then stick the output into the rest of the line (i.e. save the output to the today variable here).

-
  ```
  tobackup=~/to_backup
  server=your-SUNet-ID@corn.stanford.edu
  ```

  Creates more variables, one for folder to back up, and one for server + user information.

Backup script (SSH review, more streams)

-
  ```
  tar czf - $tobackup | ssh $server '( cd ~/
      ↪ Documents/backup_files/ ; cat > '${today
      ↪ }'.tar.gz )'
  ```

  This command (note: this is one line broken to fit) compresses the $tobackup folder and saves it to the $server. In detail:

  - `tar czf - $tobackup` compresses the `$tobackup` directory and passes the result to the stdout stream

Backup script (SSH review, more streams)

-
  ```
  tar czf - $tobackup | ssh $server '( cd ~/
     ↪ Documents/backup_files/ ; cat > '${today
     ↪ }'.tar.gz )'
  ```

  This command (note: this is one line broken to fit) compresses the `$tobackup` folder and saves it to the `$server`. In detail:

  - `tar czf - $tobackup` compresses the `$tobackup` directory and passes the result to the stdout stream
  - `|` the `|` is called a pipe. It takes the stdout stream and redirects it as input to the following command.

Backup script (SSH review, more streams)

-
  ```
  tar czf - $tobackup | ssh $server '( cd ~/
      ↪ Documents/backup_files/ ; cat > '${today
      ↪ }'.tar.gz )'
  ```

  This command (note: this is one line broken to fit) compresses the `$tobackup` folder and saves it to the `$server`. In detail:

  - `tar czf - $tobackup` compresses the `$tobackup` directory and passes the result to the stdout stream
  - `|` the `|` is called a pipe. It takes the stdout stream and redirects it as input to the following command.
  - `ssh $server...` logs in to `$server` using ssh, and then runs the commands that follow

Backup script (SSH review, more streams)

-
  ```
  tar czf - $tobackup | ssh $server '( cd ~/
      ↪ Documents/backup_files/ ; cat > '${today
      ↪ }'.tar.gz )'
  ```

  This command (note: this is one line broken to fit) compresses the $tobackup folder and saves it to the $server. In detail:

  - `tar czf - $tobackup` compresses the $tobackup directory and passes the result to the stdout stream
  - `|` the | is called a pipe. It takes the stdout stream and redirects it as input to the following command.
  - `ssh $server...` logs in to $server using ssh, and then runs the commands that follow
  - `'( cd ~/Documents/backup_files/ ; cat > '${today}'.tar.` `↪ gz )'` The quotes around everything except ${today} tell your computer to not run these commands, instead give them to ssh to run on the server. The `cat > '${today}'.tar.gz` part tells the server to take the input it's given (from the pipe above) and save it out to the named file.

Backup script exercise

```
#!/bin/bash
today=$(date +%F)
tobackup=~/to_backup
server=your-SUNet-ID@corn.stanford.edu
tar czf - $tobackup | ssh $server '( cd ~/
    ↪ Documents/backup_files/ ; cat > '${today}'.
    ↪ tar.gz )'
```

You might also want the script to remove the backup files older than some relative date (e.g. a week ago.) Try to add this functionality. (Be very careful testing scripts that contain `rm`. Use `rm -i` to prompt before removing each file.) (Hint: remember you can use `ssh $server '(...)'` to pass commands to the server. You can pass filenames that you get from another command to `rm` using the `$(...)` syntax of above. To find files that are older than some date, check out the `find` command, and its flags like `-mtime`.)

Backup script excercise

One possible answer:

backup.sh

```
#!/bin/bash
today=$(date +%F)
tobackup=~/to_backup
server=your-SUNet-ID@corn.stanford.edu
tar czf - $tobackup | ssh $server '( cd ~/
    ↪ Documents/backup_files/ ; cat > '${today}'.
    ↪ tar.gz )'
ssh $server '(rm -i $(find ~/Documents/
    ↪ backup_files/*.tar.gz -mtime +7) )'
```

(This throws a weird message when there are no old backup files to remove, how could we fix that?)

Further information

- **Detailed cheatsheet:** https://gist.github.com/LeCoupa/122b12050f5fb267e75f
- **Advanced Bash scripting guide:** http://tldp.org/LDP/abs/html/