Intro to BASH

https://bit.ly/2lYz34C

First Year Bootcamp, 2019

Table of Contents

What are we bashing and why?

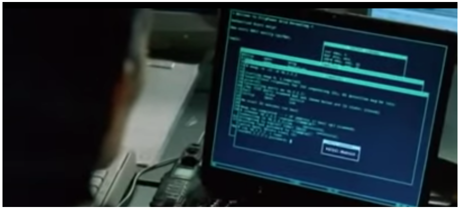
- No violence involved, bash is a program!

What are we bashing and why?

- No violence involved, bash is a program!
- bash (the name is an acronym for Bourne-Again SHell, don't ask) is a powerful command line interpreter that is the default on most Linux distros and OS X.

What are we bashing and why?

- No violence involved, bash is a program!
- bash (the name is an acronym for Bourne-Again SHell, don't ask) is a powerful command line interpreter that is the default on most Linux distros and OS X.
- In other words, we're going back to the way people used to use computers in the old days, or the way that "hackers" use them on TV.



What are we bashing and why?

- No violence involved, bash is a program!
- bash (the name is an acronym for Bourne-Again SHell, don't ask) is a powerful command line interpreter that is the default on most Linux distros and OS X.
- In other words, we're going back to the way people used to use computers in the old days, or the way that "hackers" use them on TV.
- Why? Bash allows you to:
  - Simplify tasks that you could possibly do in other ways. Want to rename 1000 data files? Bash makes it (relatively) easy!

What are we bashing and why?

- No violence involved, bash is a program!
- bash (the name is an acronym for Bourne-Again SHell, don't ask) is a powerful command line interpreter that is the default on most Linux distros and OS X.
- In other words, we're going back to the way people used to use computers in the old days, or the way that "hackers" use them on TV.
- Why? Bash allows you to:
  - Simplify tasks that you could possibly do in other ways. Want to rename 1000 data files? Bash makes it (relatively) easy!
  - Access powerful tools like ssh and git that you otherwise couldn't.

Opening bash

- **OS X/Linux:** Open terminal, bash is default shell
- **Windows:** Go to Start $\rightarrow$ Git $\rightarrow$ Git Bash

Opening bash

- **OS X/Linux:** Open terminal, bash is default shell
- **Windows:** Go to Start $\rightarrow$ Git $\rightarrow$ Git Bash
- You can get a list of commands by typing *help* in the prompt and hitting enter.

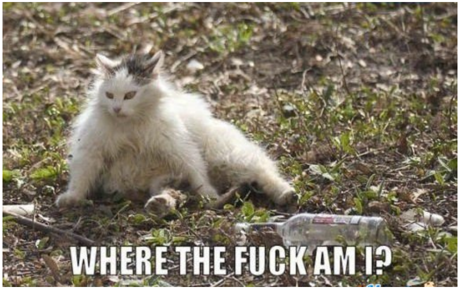
Opening bash

- **OS X/Linux:** Open terminal, bash is default shell
- **Windows:** Go to Start → Git → Git Bash
- You can get a list of commands by typing *help* in the prompt and hitting enter.
- You can learn more about a command or program by typing *help command* or *man command* (Note: *man* doesn't work on windows, but you can always use google!)

Table of Contents

Where am I? (Directories)

- Just like when you use explorer or finder to navigate on your computer, bash sees your files as organized into directories (folders). Whenever you use bash, you're always in a directory (your working directory). To find out what directory you're in right now, try typing *pwd* (print working directory) in the prompt, and then hitting enter.



Where am I? (Directories)

- Just like when you use explorer or finder to navigate on your computer, bash sees your files as organized into directories (folders). Whenever you use bash, you're always in a directory (your working directory). To find out what directory you're in right now, try typing *pwd* (print working directory) in the prompt, and then hitting enter.
- e.g. */Users/my-user-name/* (This is my "home" directory)

Where am I? (Directories)

- Just like when you use explorer or finder to navigate on your computer, bash sees your files as organized into directories (folders). Whenever you use bash, you're always in a directory (your working directory). To find out what directory you're in right now, try typing *pwd* (print working directory) in the prompt, and then hitting enter.
- e.g. */Users/my-user-name/* (This is my "home" directory)
- Did it print a directory name? Good! Now try typing *ls* (list) into the prompt. This should list the directories and files that are within this directory.

Where am I? (Directories)

- Just like when you use explorer or finder to navigate on your computer, bash sees your files as organized into directories (folders). Whenever you use bash, you're always in a directory (your working directory). To find out what directory you're in right now, try typing *pwd* (print working directory) in the prompt, and then hitting enter.
- e.g. */Users/my-user-name/* (This is my "home" directory)
- Did it print a directory name? Good! Now try typing *ls* (list) into the prompt. This should list the directories and files that are within this directory.
- You can change to a new directory by typing *cd* (change directory) followed by the directory name. For example, try *cd* $\sim$. ($\sim$ is a special character that refers to your home directory, usually */home/your-user-name/*.)

Where am I? (Directories)

- Just like when you use explorer or finder to navigate on your computer, bash sees your files as organized into directories (folders). Whenever you use bash, you're always in a directory (your working directory). To find out what directory you're in right now, try typing *pwd* (print working directory) in the prompt, and then hitting enter.
- e.g. */Users/my-user-name/* (This is my "home" directory)
- Did it print a directory name? Good! Now try typing *ls* (list) into the prompt. This should list the directories and files that are within this directory.
- You can change to a new directory by typing *cd* (change directory) followed by the directory name. For example, try *cd* $\sim$. ($\sim$ is a special character that refers to your home directory, usually */home/your-user-name/*.)
- Now try using *ls* again, and then using *cd* to enter one of the directories you see (perhaps Documents, Desktop, Downloads or similar, depending how your OS is set up).

Arguments and flags for *ls*

- What if you want to list the contents of a directory other than your current one? In that case you can just give that directory's path as an argument to *ls*. For example: *ls /home* will tell you what's in the directory */home*, no matter where you are.

Arguments and flags for *ls*

- What if you want to list the contents of a directory other than your current one? In that case you can just give that directory's path as an argument to *ls*. For example: *ls /home* will tell you what's in the directory */home*, no matter where you are.
- There are various flags (usually a - followed by a single character, or a – followed by a word) you can pass to *ls*. For example, try *ls -l* (more information every item on its own line), *ls -a* (lists hidden files as well)

Arguments and flags for *ls*

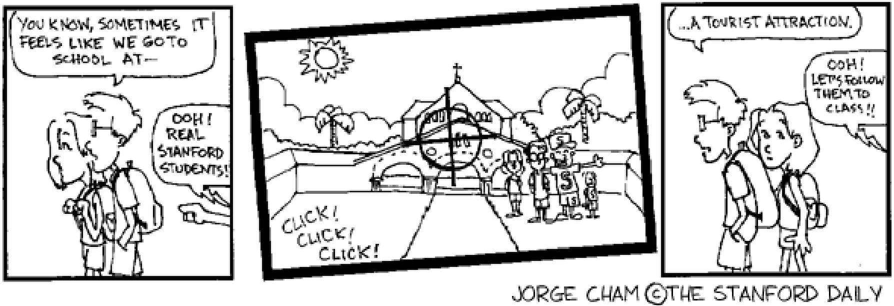
- What if you want to list the contents of a directory other than your current one? In that case you can just give that directory's path as an argument to *ls*. For example: *ls /home* will tell you what's in the directory */home*, no matter where you are.
- There are various flags (usually a - followed by a single character, or a – followed by a word) you can pass to *ls*. For example, try *ls -l* (more information every item on its own line), *ls -a* (lists hidden files as well)

Arguments & flags

Most commands we will talk about can take many different kinds of arguments and flags to alter their function, don't forget to use *man* and *help* to find out more about them! Try it now with *man ls*.

Creating, moving, and destroying directories

- Maybe you want to create a new directory to house all your exciting new grad stuff, like the pictures that random tourists take of you. To do this, type *mkdir* (make directory) followed by the name for the directory, e.g. *mkdir exciting-grad-school-stuff*


YOU KNOW, SOMETIMES IT FEELS LIKE WE GO TO SCHOOL AT—
OOH! REAL STANFORD STUDENTS!
CLICK! CLICK! CLICK!
...A TOURIST ATTRACTION.
OOH! LET'S FOLLOW THEM TO CLASS!!
JORGE CHAM ©THE STANFORD DAILY

Creating, moving, and destroying directories

- Maybe you want to create a new directory to house all your exciting new grad stuff, like the pictures that random tourists take of you. To do this, type *mkdir* (make directory) followed by the name for the directory, e.g. *mkdir exciting-grad-school-stuff*
- Maybe you changed your mind about what to call it though, so you want to change it to *boring-grad-school-stuff*, and also move it somewhere else. Not to worry! You can use *mv* (move) to rename the directory, or to move it to a new place. For example, you could use *mv exciting-grad-school-stuff ~/boring-grad-school-stuff* to move it to your home directory and rename it. Give it a try!

Creating, moving, and destroying directories

- Maybe you want to create a new directory to house all your exciting new grad stuff, like the pictures that random tourists take of you. To do this, type *mkdir* (make directory) followed by the name for the directory, e.g. *mkdir exciting-grad-school-stuff*
- Maybe you changed your mind about what to call it though, so you want to change it to *boring-grad-school-stuff*, and also move it somewhere else. Not to worry! You can use *mv* (move) to rename the directory, or to move it to a new place. For example, you could use *mv exciting-grad-school-stuff ~/boring-grad-school-stuff* to move it to your home directory and rename it. Give it a try!
- There is also a command called *cp* (copy) that works like *mv* except it copies the file.

Creating, moving, and destroying directories

- Maybe you want to create a new directory to house all your exciting new grad stuff, like the pictures that random tourists take of you. To do this, type *mkdir* (make directory) followed by the name for the directory, e.g. *mkdir exciting-grad-school-stuff*
- Maybe you changed your mind about what to call it though, so you want to change it to *boring-grad-school-stuff*, and also move it somewhere else. Not to worry! You can use *mv* (move) to rename the directory, or to move it to a new place. For example, you could use *mv exciting-grad-school-stuff ∼/boring-grad-school-stuff* to move it to your home directory and rename it. Give it a try!
- There is also a command called *cp* (copy) that works like *mv* except it copies the file.
- Finally, maybe you think this is a silly directory and want to get rid of it. You can do that by using the command *rmdir* (remove directory), e.g. *rmdir ∼/boring-grad-school-stuff*.

Special directories

- There are two special directories you'll see listed if you type *ls -a*, *.* and *..*, which are used to refer to the current directory and its parent, respectively.

Special directories

- There are two special directories you'll see listed if you type *ls -a*, . and .., which are used to refer to the current directory and its parent, respectively.
- For example, if you are in $\sim$*/Documents/grad/* and type *cd ..* your working directory will change to the parent of your current directory, i.e. $\sim$*/Documents*.

Table of Contents

What's all this stuff? (Files)

- Now let's look at the files within a directory. Many of the commands you've already learned still apply, *ls* will list them, and *mv* will move them.

What's all this stuff? (Files)

- Now let's look at the files within a directory. Many of the commands you've already learned still apply, *ls* will list them, and *mv* will move them.
- Let's create an empty file to play around with. You can do this by typing *touch emptyfile.txt* (in real life you'll usually be working with files you create in other programs, this way of creating them is just an example).

What's all this stuff? (Files)

- Now let's look at the files within a directory. Many of the commands you've already learned still apply, *ls* will list them, and *mv* will move them.
- Let's create an empty file to play around with. You can do this by typing *touch emptyfile.txt* (in real life you'll usually be working with files you create in other programs, this way of creating them is just an example).
- Now let's try to put some text in the file and save it, just for fun. You can do this from the command line, but how exactly you do it will depend on your OS.
    - **OS X:** try *open emptyfile.txt*.
    - **GNOME-based linux distros:** try *gedit emptyfile.txt*.
    - **Windows:** try *notepad emptyfile.txt*

Manipulating files, listing selectively

- Now that we put some text in the file, maybe we ought to rename it to *nonemptyfile.txt*. How do you think we do that?

Manipulating files, listing selectively

- Now that we put some text in the file, maybe we ought to rename it to *nonemptyfile.txt*. How do you think we do that?
- Yup, *mv emptyfile.txt nonemptyfile.txt*.

Manipulating files, listing selectively

- Now that we put some text in the file, maybe we ought to rename it to *nonemptyfile.txt*. How do you think we do that?
- Yup, *mv emptyfile.txt nonemptyfile.txt*.

Warning!

mv overwrites any files with the same name(s) in its destination, so be careful when using it! If there was another file in this directory called nonemptyfile.txt, we would have overwritten it.

Selectivity and globs

- Let's suppose you come back tomorrow and can't remember what you called this file. You can type *ls* to list everything in the directory and look through for it, but there might be a lot of other stuff. You can make commands be more selective by giving them some hints. For example, type *ls *.txt* to list all files in the current directory ending with a *.txt* extension. Type *ls *empty** to list files with *empty* in their name.

Selectivity and globs

- Let's suppose you come back tomorrow and can't remember what you called this file. You can type *ls* to list everything in the directory and look through for it, but there might be a lot of other stuff. You can make commands be more selective by giving them some hints. For example, type *ls *.txt* to list all files in the current directory ending with a *.txt* extension. Type *ls *empty** to list files with *empty* in their name.

Globs

The character * is called a glob, because it sticks together all the files that complete the rest of the pattern I guess, I don't know. Globs are often useful, e.g. you could move all the csv files in the current directory to new directory by typing *mv *.csv /Documents/my-new-data-directory*

Removing files

- Just like you can remove directories, you can remove files by using the command *rm* (remove). Try it now by typing *rm nonemptyfile.txt*.

Warning!

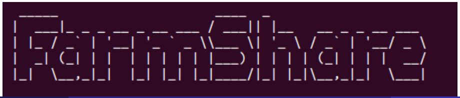
rm is VERY DANGEROUS, especially when used with globs and/or with certain flags (use *man rm* to find out more). It does not simply move a file to the trash, it deletes it completely. Double check what you type before you run it, and don't use *rm* if you're not sure what you're doing.

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:

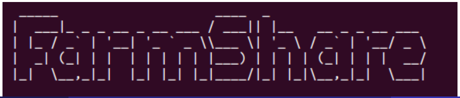
- Class or research (or web hosting etc.) machines:
  - **corn:** The workhorse systems, general purpose servers for running small jobs, accessing your shared file space, hosting your website, etc.



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

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

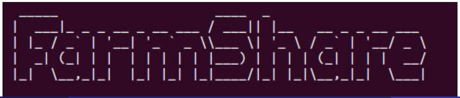
- Class or research (or web hosting etc.) machines:
  - **corn:** The workhorse systems, general purpose servers for running small jobs, accessing your shared file space, hosting your website, etc.
  - **rye:** (somewhat old) GPU machines, still fairly powerful but may not be compatible with newer software.



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

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

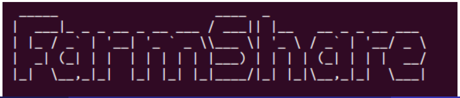
- Class or research (or web hosting etc.) machines:
    - **corn:** The workhorse systems, general purpose servers for running small jobs, accessing your shared file space, hosting your website, etc.
    - **rye:** (somewhat old) GPU machines, still fairly powerful but may not be compatible with newer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



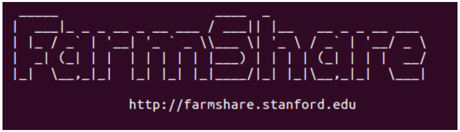
Connecting to corn

- Let's try to connect to one of Stanford's **corn** servers, which are available for general use. Type *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

- Let's try to connect to one of Stanford's **corn** servers, which are available for general use. Type *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. If so, congrats, you're connected to a server!



```
 _____                     ____  _
|  ___|_ _ _ __ _ __ ___  / ___|| |__   __ _ _ __ ___
| |_ / _` | '__| '_ ` _ \ \___ \| '_ \ / _` | '__/ _ \
|  _| (_| | |  | | | | | | ___) | | | | (_| | | |  __/
|_|  \__,_|_|  |_| |_| |_||____/|_| |_|\__,_|_|  \___|

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

Basic SSH

- Luckily these servers are running bash as well, so all the commands you've just learned should work here! Try *ls* and see what's around. This is your space on Stanford's servers, you can store documents here, and if you make a Stanford website it will be hosted in the WWW folder.
- Remember, you're no longer on your own computer, so the directory structure here will be different.
- When you're finished, you can type *exit* to close the connection to the server and return to your computer. (But don't do so yet, we're going to keep using it.)

A toy website

- Let's try creating a (very) simple website for you!

A toy website

- Let's try creating a (very) simple website for you!
- 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 website for you!
- 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). *cd* to the directory where you saved simplewebsite.txt.

A toy website (cont.)

- We need to move *simplewebsite.txt* to the Stanford servers. To do that, we'll use the command *scp* (secure copy), which allows you to copy files from/to your local computer to/from a remote one much like you would copy a file on your local computer using *cp*

A toy website (cont.)

- We need to move *simplewebsite.txt* to the Stanford servers. To do that, we'll use the command *scp* (secure copy), which allows you to copy files from/to your local computer to/from a remote one much like you would copy a file on your local computer using *cp*
- To copy *simplewebsite.txt* to the ∼*/WWW* folder on the server, try *scp simplewebsite.txt your-SUNetID@corn.stanford.edu:∼/WWW/*

A toy website (cont.)

- We need to move *simplewebsite.txt* to the Stanford servers. To do that, we'll use the command *scp* (secure copy), which allows you to copy files from/to your local computer to/from a remote one much like you would copy a file on your local computer using *cp*
- To copy *simplewebsite.txt* to the *∼/WWW* folder on the server, try *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.)

- We need to move *simplewebsite.txt* to the Stanford servers. To do that, we'll use the command *scp* (secure copy), which allows you to copy files from/to your local computer to/from a remote one much like you would copy a file on your local computer using *cp*
- To copy *simplewebsite.txt* to the *∼/WWW* folder on the server, try *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, to create a website for yourself so that people can look you up, etc. The process will be similar to this, except that you'll probably be creating html files instead of text files.

Cleaning up the website

- We left the ssh connection open in the other terminal so we can do something with the file on the server.

Cleaning up the website

- We left the ssh connection open in the other terminal so we can do something with the file on the server.
- Go ahead and run *ls* $\sim$/*WWW* on the server so you can see *simplewebsite.txt* is there.

Cleaning up the website

- We left the ssh connection open in the other terminal so we can do something with the file on the server.
- Go ahead and run *ls ~/WWW* on the server so you can see *simplewebsite.txt* is there.
- You might not want the world to be able to see this file forever, so change to the *WWW* directory and remove the file.

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`

Cleaning up the website

- We left the ssh connection open in the other terminal so we can do something with the file on the server.
- Go ahead and run *ls ~/WWW* on the server so you can see *simplewebsite.txt* is there.
- You might not want the world to be able to see this file forever, so change to the *WWW* directory and remove the file.
- 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

Wrapping up the basics

Bash contains or allows access to many powerful tools, and has many arcane (but useful!) features. Here are a few examples of the things you can do with it that we won't have time to explain to you. If you're interested in more info, talk to us and we'll be happy to provide it!

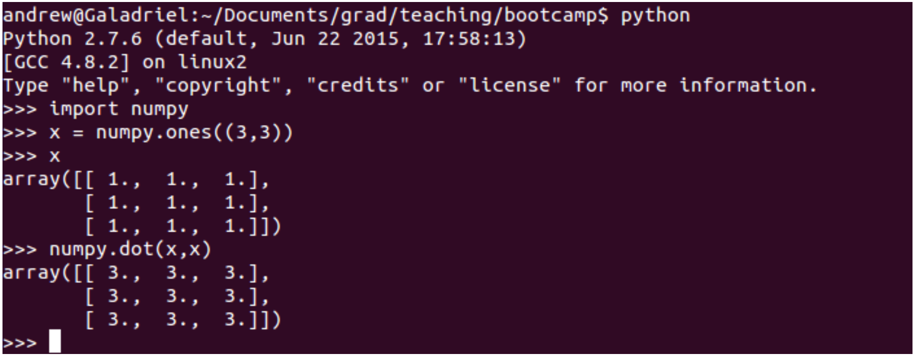
Being lazy (scripts)

One of the main reasons bash is useful is that bash commands can be saved into scripts that can be reused. For example, here's a simple shell script I wrote to create anonymized data files by replacing participant identifiers in filenames with numbers starting from 0:

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

Programming

The terminal gives you access to programming languages like python, both by running programs directly, and by using **interpreters**, programs that run in the terminal and allow you to run commands interactively, much as you would in Matlab or R.



```
andrew@Galadriel:~/Documents/grad/teaching/bootcamp$ python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> x = numpy.ones((3,3))
>>> x
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> numpy.dot(x,x)
array([[ 3.,  3.,  3.],
       [ 3.,  3.,  3.],
       [ 3.,  3.,  3.]])
>>> 
```

Etc.

Also:

- Powerful commands for text manipulation (*sed, awk, grep*). Want to find all occurrences of a variable in many different code files and rename it? Want to extract lines from your datafiles that match a certain condition without having to read the files into R? Have a directory full of awful tab-separated datafiles and want to make them comma-separated? These tools can do it.
- Streams and piping: make commands work together, do file I/O, etc.
- Automation: more system specific, but *cron*, rc files, etc. allow for automation of things like backups, mounting filesystems when you start your computer, or sending reminders to subjects on a schedule.
- Text-editors like *vim*.
- Git version control!

Cheatsheet

Command	**Effect**
man command or *help command*	Get manual/help for a command
pwd	Print working directory
cd dir	Change working directory to *dir*
ls	List directory contents
mkdir dir	Make a new directory called *dir*
rmdir dir	Remove the directory *dir* permanently
mv source dest	Move *source* file or folder to *dest*
cp source dest	Copy *source* file to *dest* (use -r for folders)
rm file	Remove *file* permanently
ssh user@server	Connect to *server* as *user*
scp [[user1@]server1:]source [[user2@]server2:]dest	Copy *source* from *server1* to *dest* on *server2* (if copying to/from your local computer, just omit the server and user parts in that file's path)

(For a more comprehensive reference see `https://gist.github.com/LeCoupa/122b12050f5fb267e75f`)

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/2cFq4O3 | 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

```bash
#!/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
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
```

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

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 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 ~/Docume
  ```

  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 ~/Docume
  ```

  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 ~/Docume
  ```

  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 ~/Docume
  ```

  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
```

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
ssh $server '(rm -i $(find ~/Documents/backup_files
```

(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/