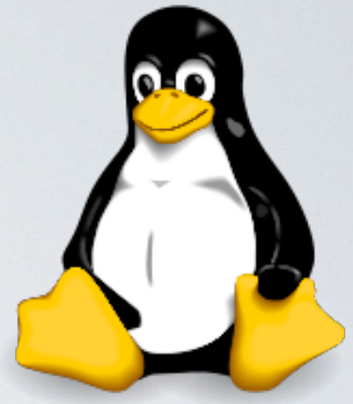
MODULE 1: INTRODUCTION

Linux & bash shell

I. Linux

What is Linux?

- Linux is an operating system (OS) developed by Linus Torvalds in 1991

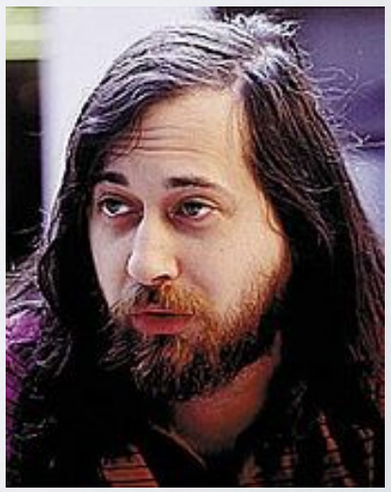


Tux

- Based on UNIX - developed in response to closing legal loophole that made UNIX free

Linus Torvalds

- Many "distributions" - Fedora, RedHat, CentOS, Debian, Ubuntu
- Typically free and open source GNU licensing



Richard Stallman

- Command line interface (CLI) and graphical desktop environments (GDE)

Why Linux?

- Developed by Bell Labs in 1969, and initially free, UNIX was quickly adopted as *de facto* scientific computing OS

- Powerful CLI enables direct low level access
  GDE provides simplicity and usability

- Free and open source makes code development easy

- Linux is everywhere
  - 90% of supercomputers run Linux (incl. Blue Waters)
  - Android OS is based on a Linux kernel
  - Ubuntu distro is the most popular OS in the world

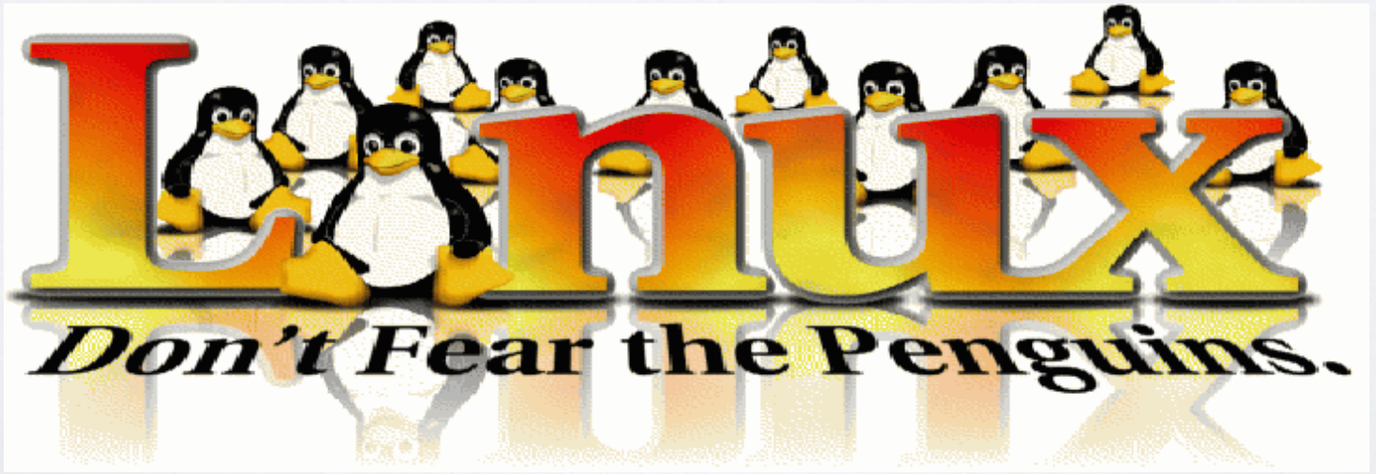
Can't I use Windows / Mac OS X?

- **Maybe.**

- Some software have Mac OS X / Windows / Windows + Cygwin versions to install on your local machine

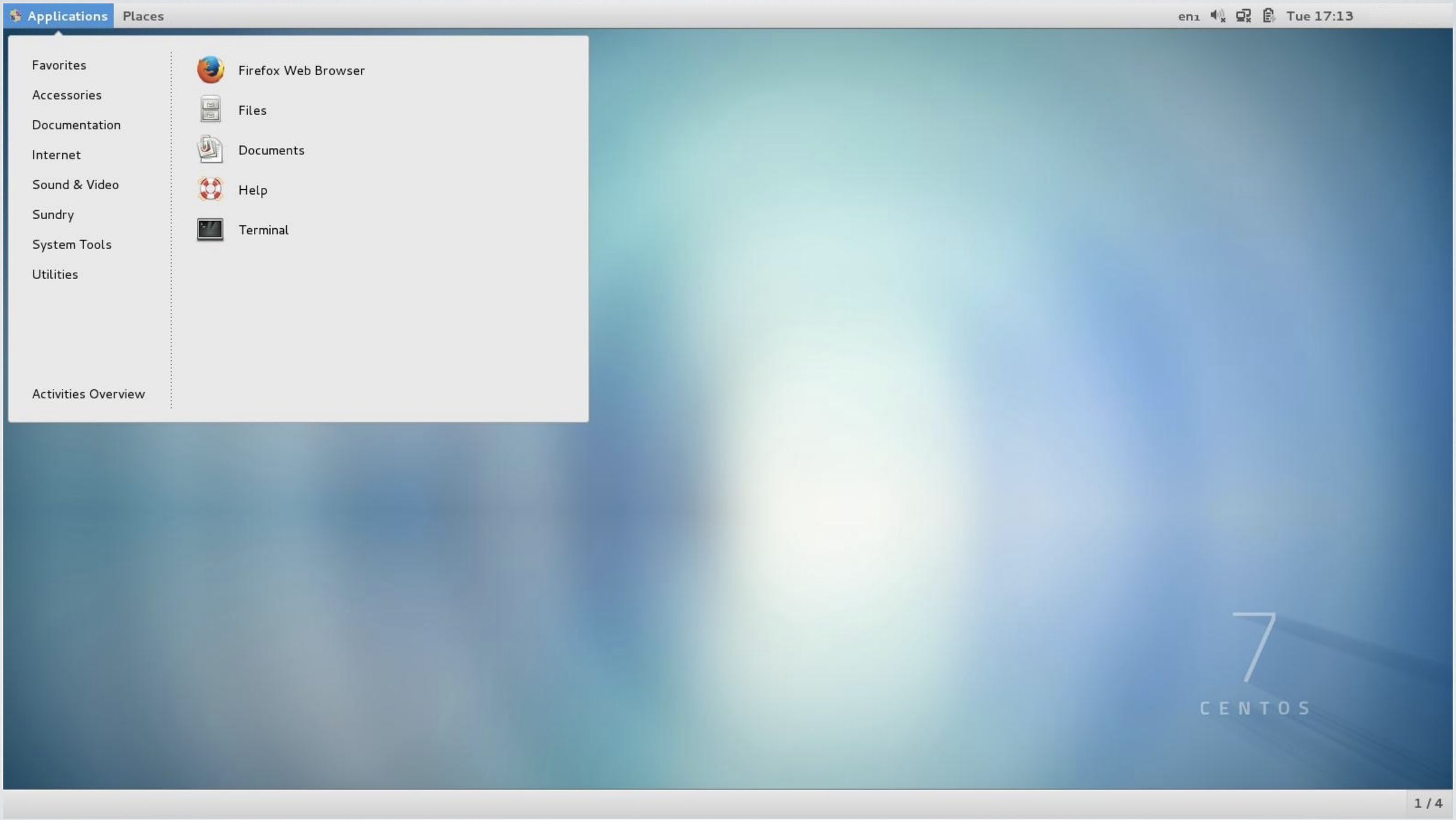
- Remote login via Mac OS X terminal / [Windows + Cygwin / Putty] to SSH into EWS Linux

- A key learning objectives of this course is to develop familiarity and competence using Linux - ***Bon Courage!***



What distro are we using?

- EWS Linux machines run Redhat linux



II. bash shell

The command line

- CLI and GDE offer alternatives to interact with a machine
- Switching to a CLI can be very intimidating for new users!
- CLI interaction is powerful, concise, and efficient
- CLI scripting enables task automation

  e.g. Download of 1500 daily NASDAQ stock prices

  GDE: Point and click file download extremely tedious!
  CLI: Trivially automated using CLI wget loop

bash shell

- "Command line interpreters" or "shells" convert text inputs into OS commands
- Many flavors: sh, bash, ksh, csh, zsh, tcsh
- The bash shell ("Bourne-again shell") is one of the most popular, and the default on many Linux distros

```
mars@marsmain ~ $ pwd
/home/mars
mars@marsmain ~ $ cd /usr/portage/app-shells/bash
mars@marsmain /usr/portage/app-shells/bash $ ls -al
total 130
drwxr-xr-x  3 portage portage  1024 Jul 25 10:06 .
drwxr-xr-x 33 portage portage  1024 Aug  7 22:39 ..
-rw-r--r--  1 root    root    35808 Jul 25 10:06 ChangeLog
-rw-r--r--  1 root    root    27002 Jul 25 10:06 Manifest
-rw-r--r--  1 portage portage  4645 Mar 23 21:37 bash-3.1_p17.ebuild
-rw-r--r--  1 portage portage  5977 Mar 23 21:37 bash-3.2_p39.ebuild
-rw-r--r--  1 portage portage  6151 Apr  5 14:37 bash-3.2_p48-r1.ebuild
-rw-r--r--  1 portage portage  5988 Mar 23 21:37 bash-3.2_p48.ebuild
-rw-r--r--  1 portage portage  5643 Apr  5 14:37 bash-4.0_p10-r1.ebuild
-rw-r--r--  1 portage portage  6230 Apr  5 14:37 bash-4.0_p10.ebuild
-rw-r--r--  1 portage portage  5648 Apr 14 05:52 bash-4.0_p17-r1.ebuild
-rw-r--r--  1 portage portage  5532 Apr  8 10:21 bash-4.0_p17.ebuild
-rw-r--r--  1 portage portage  5660 May 30 03:35 bash-4.0_p24.ebuild
-rw-r--r--  1 root    root     5660 Jul 25 09:43 bash-4.0_p28.ebuild
drwxr-xr-x  2 portage portage  2048 May 30 03:35 files
-rw-r--r--  1 portage portage   468 Feb  9 04:35 metadata.xml
mars@marsmain /usr/portage/app-shells/bash $ cat metadata.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pkgmetadata SYSTEM "http://www.gentoo.org/dtd/metadata.dtd">
<pkgmetadata>
<herd>base-system</herd>
<use>
  <flag name='bashlogger'>Log ALL commands typed into bash; should ONLY be
    used in restricted environments such as honeypots</flag>
  <flag name='net'>Enable /dev/tcp/host/port redirection</flag>
  <flag name='plugins'>Add support for loading builtins at runtime via
    'enable'</flag>
</use>
</pkgmetadata>
mars@marsmain /usr/portage/app-shells/bash $ sudo /etc/init.d/bluetooth status
Password:
* status: started
mars@marsmain /usr/portage/app-shells/bash $ ping -q -c1 en.wikipedia.org
PING rr.esams.wikimedia.org (91.198.174.2) 56(84) bytes of data.

--- rr.esams.wikimedia.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 2ms
rtt min/avg/max/mdev = 49.820/49.820/49.820/0.000 ms
mars@marsmain /usr/portage/app-shells/bash $ grep -i /dev/sda /etc/fstab | cut --fields=-3
/dev/sda1               /boot
/dev/sda2               none
/dev/sda3               /
mars@marsmain /usr/portage/app-shells/bash $ date
Sat Aug  8 02:42:24 MSD 2009
mars@marsmain /usr/portage/app-shells/bash $ lsmod
Module                  Size  Used by
rndis_wlan             23424  0
rndis_host              8696  1 rndis_wlan
cdc_ether               5672  1 rndis_host
usbnet                 18688  3 rndis_wlan,rndis_host,cdc_ether
parport_pc             38424  0
fglrx                2388128  20
parport                39648  1 parport_pc
iTCO_wdt               12272  0
i2c_i801                9380  0
mars@marsmain /usr/portage/app-shells/bash $
```

III. bash basics

bash: basics

- Pop a bash terminal by clicking on 
  or navigating Applications ➔ System Tools ➔ Terminal

- `pwd` - show path to present working directory

- `ls` - list contents of current directory

  `ls -alh` - list all contents of cdir in long form with human readable file sizes

  `ls /sw/q` - list contents of directory /sw/q

- `cd <path>` - change directory into <path>

  `cd ..` - change directory up one level

  `cd ../..` - change directory up two levels

bash: basics

- `touch <file>` - make new file <file> or update last access of existing file
- `mkdir <dir>` - make directory

- `chmod 755 <file>`- change file permissions to r+w+x (user), r+x (group, world)

`chmod 644 <file>`- change file permissions to r+w (user), r (group, world)

[N.B. r=4, w=2, x=1]

- `var=ferrari42` - assign ferrari42 to var
- `echo $var` - print $var

bash: basics

- `./<execFile>` - execute execFile in cdir
  `<path>/<execFile>` - execute execFile in path

- `which <cmd>` - location of command
  cmd

- `clear` - clear terminal

- `wget -O <file> <url>` - download url data into file

  e.g. `wget -O myProf.png http://bit.ly/2jt9NAl`

bash: basics

- `cp <source> <target>` - copy file source to target

  e.g. `cp myFile /apps/doc/`

- `cp -r <source> <target>` - copy recursively (copy source *directory* and everything in it)

  e.g. `cp -r myDir ./dir1/dir2/`

- `mv <source> <target>` - move source to target (same for files and directories)

- `rm <file>` - remove file

bash: safety!

- `cp / rm / mv` - These do **exactly what you ask**
  They **do not ask for permission**
- Furthermore, **there is no Trash/Recycling**
- Once you remove / overwrite a file, it's gone.
- Standard "safety" choices: use `alias` in your `.bashrc`

```
alias cp='cp -i'
alias rm='rm -i'
alias mv='mv -i'
setopt noclobber
```

- You don't *have* to do this, but you may breathe a little easier with some safety.

bash: basics

- `whoami` - show your login username
  `who` - show everyone currently logged in

- `cat <file>` - show file contents

- `less <file>` - show file contents (`spacebar` ↓, `b` ↑)

- `head <file>` - show head of file
  `tail <file>` - show tail of file
  `tail -n <nLines> <file>` - show tail nLines of file

bash: basics

- `zip <archive> <file1 file2 ...>`

  - create zip file archive.zip containing file1, file2, ...

`unzip <archive>`

  - unzip zip file archive.zip

- `tar cvzf <archive.tgz> <file1 file2 ...>`

  - create gzip compressed tape archive archive.tgz containing file1, file2, ...

`tar xvzf <archive.tgz>`

  - uncompress end extracted compressed tape

bash: basics

- `top` - show active processes

  `top -o cpu` - show active processes ordered by cpu %

  `top -U <usr>` - show active processes owned by usr

- `grep <str> <file>` - return lines in file containing string str

- `find <path> -name <*str*> -print`

  - print all files in path containing str in their name

bash: special symbols

- ~ - your home directory
- . - current directory
- .. - directory one level up
- * - wildcard character
- \ - escape succeeding character

e.g. `mkdir My\ Directory`

- | - pipe

bash: special symbols

- `>` - redirect standard output and overwrite
  `>>` - redirect standard output and append

  e.g. `echo “Today was great!” >> myDiary.txt`

- `$var` - dereference variable var

- `“ ”` - enclose text string but expand $
  `‘ ’` - enclose text string but do not expand $

  e.g.
```
myVar=“My String With Spaces”
echo “This is $myVar”
```

IV. bash utilities

bash: integer arithmetic

- `expr` - integer arithmetic engine

e.g.

```
$ echo `expr 1 + 1`
2
$ var1=`expr 10 \* 2`
$ var2=`expr 21 / 7`
$ echo $var1 $var2 `expr $var1 /
$var2`
20 3 6
```

bash: quick calculator

- `bc -l` - arbitrary precision calculator (w/ math lib)

```
$ bc -l
2/3
.66666666666666666666
2^3
8
e(1)
2.71828182845904523536
pi=a(1)*4
pi
3.14159265358979323844
s(pi/6)
.49999999999999999999
```

bash: ssh & scp

- **SSH** CLI remote login is supported by ssh (secure shell)

  `ssh <user>@<hostname>` - login to host

  `ssh -Y <user>@<hostname>` - login to host w/ secure X forwarding (use this to get graphics via SSH!)

  N.B. For EWS, hostname=linux.ews.illinois.edu

- **SCP** CLI file transfers supported by scp (secure copy)

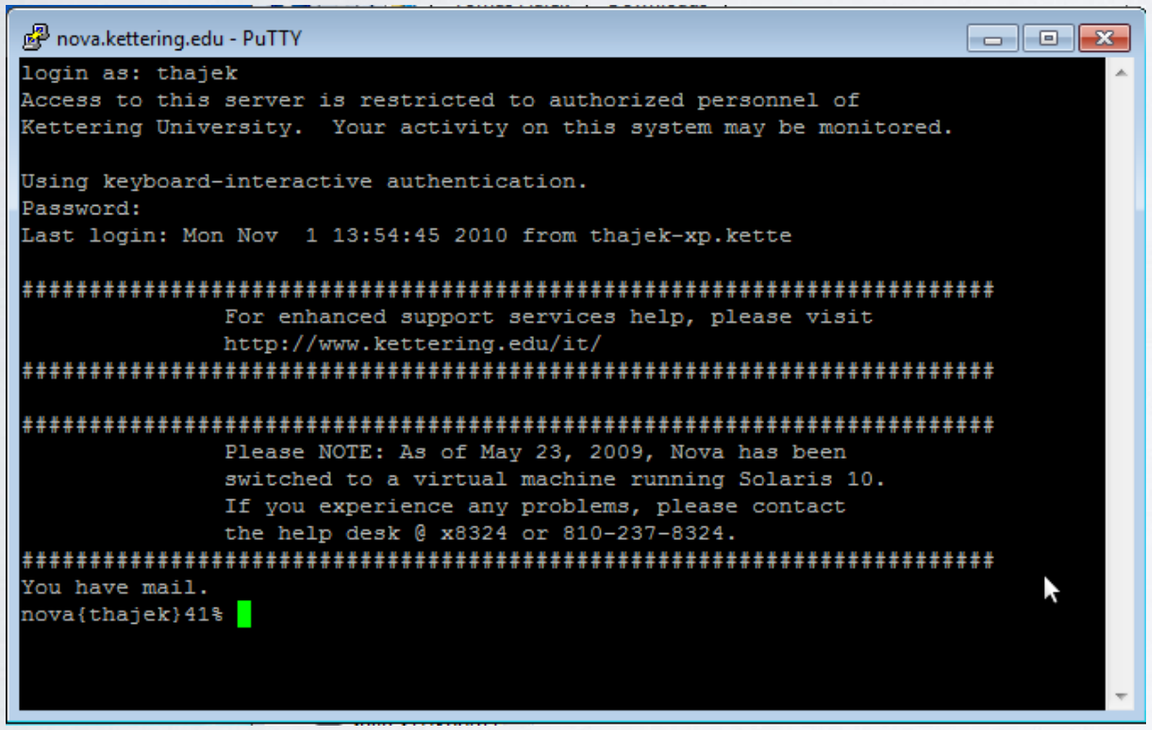
  `scp <src> <user>@<hostname>:<target>` - upload

  `scp <user>@<hostname>:<src> <target>` - download

bash: ssh & scp

- ssh and scp are prepackaged with Linux / Mac OS X and are accessible directly from the bash terminal

- On Windows, you need to download a **third party ssh client** in order to make a ssh connection with EWS



www.putty.org

https://answers.uillinois.edu/illinois.engineering/page.php?id=81727

bash: sftp

- **SFTP** more sophisticated alternative to scp
  (secure file transfer protocol)

```
sftp <user>@<hostname>      - login to host
ls                          - remote ls
lls                         - local ls
pwd                         - remote pwd
lpwd                          - local pwd
cd                          - remote cd
lcd                         - local cd
get <file>                    - download file
put <file>                    - upload file
```

bash: vi/vim

- Two built-in CLI text editors: vi/vim & emacs
  Seem slow and painful, but invaluable for on-the-fly edits

- Use whichever you prefer, I use both.
  (It is very fashionable to argue over which is better...)
- vi/vim is fast for text manipulation, uses two modes
- emacs is has lots of built-in modules, more "Word"-like

- Two-modes: navigation for moving
  insertion for editing
- Nav mode is the default mode, and can be accessed by hitting **Esc**

- Ins mode is accessed by hitting **i**

bash: vi/vim

- **Nav mode**

↑↓←→ - single char / single line movement

gg - go to top of file

^ - go to beginning of line

$ - go to end of line

<n>G - go to line n

w - skip forward one word

b - skip backward one word

yy or y$ - copy current line

y<n>↓ - copy next n lines

bash: vi/vim

- **Nav mode**

`x` - delete character

`o` - create new line below and enter insert mode

`i` - enter insert mode to left of current character

`I` - enter insert mode at beginning of line

`a` - enter insert mode to right of current character

`A` - enter insert mode at end of line

bash: vi/vim

- **Nav mode**

`dd` or `d$` - delete current line

`d<n>w` - delete next n words

`d<n>`↓ - delete next n lines

`u` - undo

`Ctrl+r` - redo

bash: vi/vim

- **Nav mode**

`/<str><Enter>` - search forward for str

`?<str><Enter>` - search backward for str

`n` - go to next match

`<N>n` - go to Nth match

bash: vi/vim

- **Nav mode**

`:w` - writes file

`:w!` - writes file even if read only

`:q` - quit

`:q!` - quit and don't question me
(good way to mess things up)

`:wq` - write quit

`:wq!` - write quit and don't question me
(very good way to mess things up)

bash: vi/vim

- **Ins mode**

Type normally - what you enter appears on screen

↑↓←→ work as in nav mode

Hit **Esc** to get back to nav mode

bash: .bash_profile & .bashrc

- Hidden files start with `.`
- `~/.bashrc` is executed for every new terminal
- `~/.bash_profile` is executed when you login (`~/.bash_profile` calls `~/.bashrc`)
- These files are useful to store aliases and modify **PATH**
- **N.B.** On some systems `~/.bash_profile` is

bash: .bash_profile & .bashrc

- *(i) Use vi to add* `lls` *as alias for* `ls -al` *to .bashrc*

```
$ vi ~/.bashrc          edit .bashrc
$ G                     go to end of file
$ o                     edit line below
$ alias lls="ls -al"    add alias
$ Esc                   escape to navigate mode
$ :wq                   write and quit
```

bash: .bash_profile & .bashrc

alf@linux7:~ — ssh — 115×32

```
# .bashrc
# EWS .bashrc Template
#
# Process the systems global .bashrc (DO NOT REMOVE)
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# If you want to modify an environment variable, for example, the search
# path, take into account the value that the variable already has.  For example
# if you wish to add the directory /local/apps1/cl to your search path,
# you would use
#
# export PATH=$PATH:/local/apps1/cl
#
# This merely appends the directory /local/apps1/cl to the already-existing
# search path.
#
# You can also create aliases for often-used commands.  For example, if you
# would prefer to type 'dir' instead of 'ls -FC', you could use
#
# alias dir="ls -FC"
#
# so that each time you typed 'dir', it would be as though you had typed
# 'ls -FC'
#

# User specific aliases and functions go below
alias lls="ls -l"
~
                                                              1,1           All
```

bash: .bash_profile & .bashrc

- *(ii) Use vi to add* `~/local/bin` *to your* `PATH` *in .bashrc*

```
$ vi ~/.bashrc                    edit .bashrc
$ G                               go to end of file
$ o                               edit line below
$ export PATH=$PATH:~/local/bin   add to PATH
$ Esc                             escape to navigate mode
$ :wq                             write and quit
```

bash: .bash_profile & .bashrc

alf@linux7:~ — ssh — 115×32

```
# .bashrc
# EWS .bashrc Template
#
# Process the systems global .bashrc (DO NOT REMOVE)
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# If you want to modify an environment variable, for example, the search
# path, take into account the value that the variable already has.  For example
# if you wish to add the directory /local/apps1/cl to your search path,
# you would use
#
# export PATH=$PATH:/local/apps1/cl
#
# This merely appends the directory /local/apps1/cl to the already-existing
# search path.
#
# You can also create aliases for often-used commands.  For example, if you
# would prefer to type 'dir' instead of 'ls -FC', you could use
#
# alias dir="ls -FC"
#
# so that each time you typed 'dir', it would be as though you had typed
# 'ls -FC'
#

# User specific aliases and functions go below
alias lls="ls -l"
export PATH=$PATH:~/local/bin
                                                              1,1           All
```

bash: installing software

- Typical anatomy of an installation from source:

```
$ wget <app_url>          download
$ tar xvzf <app.tgz>      uncompress
$ cd ./app
$ ./configure --prefix=<location>
                          configure and specify location
$ make                    compile
$ make install            install
```

V. bash scripting

What is bash scripting?

- A bash script is nothing more than a list of bash commands in an executable text file

- Exactly the same behavior could be achieved by copying and pasting the script into the bash shell

- Extremely powerful way to automate system tasks

e.g. file downloads
system backups
job submission
file processing

Anatomy of a script

- A script is nothing more than a text file
  - write using vi, emacs, Notepad, or favorite text editor

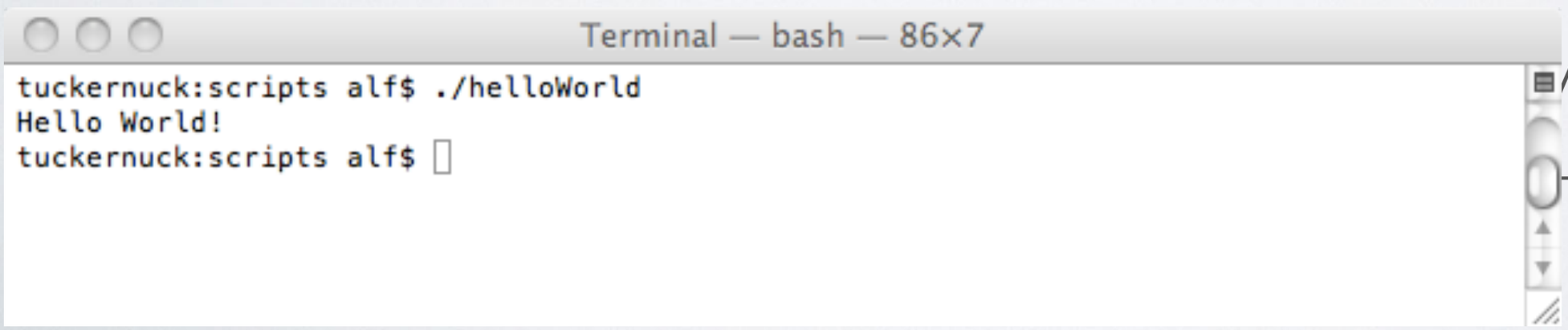
the "sha-bang" line
comments (start with #)

list of bash commands

Script 1: hello world!

```
$ touch helloWorld          new script file
$ chmod 755 helloWorld      making executable

$ vi helloWorld                  edit line below
$ i                          enter insert mode
$ #!/bin/bash <Enter>
$ # this is my hello world script
<Enter>
$ echo “Hello World!”
```



vigate mode

t

Script 2: backup

- Passing variables $1, $2, $3, ...

backup

```
#!/bin/bash

# compress and store current directory

tar czvf bkp.tgz ./*
mv bkp.tgz bkp_$1.tgz
```

Placing all files in current directory into a compressed tape archive bkp.tgz

Renaming bkp.tgz bkp_<arg>.tgz where arg is the first argument in the call to the executable

Terminal — bash — 63×24

```
tuckernuck:scripts alf$ pwd
/Users/alf/Box Documents/MSE498/1_intro/2_bash/scripts
tuckernuck:scripts alf$ ls
backup          helloWorld
tuckernuck:scripts alf$ ./backup 29thJuly2013
./backup
./helloWorld
tuckernuck:scripts alf$ ls
backup                  helloWorld
bkp_29thJuly2013.tgz
tuckernuck:scripts alf$
```

Script 3: summer

- `while` loop
- arithmetic comparisons

summer

```
#!/bin/bash

# sum all numbers passed to script

sum=0
while [ $# -gt 0 ] ; do
      echo "newNum=$1"
      sum=`expr $sum + $1`
      echo "sum=$sum"
      echo ""
      shift
done
```

Terminal — bash — 46×19

```
tuckernuck:scripts alf$ ./summer
tuckernuck:scripts alf$ ./summer 10
newNum=10
sum=10

tuckernuck:scripts alf$ ./summer 10 -4 9 17
newNum=10
sum=10

newNum=-4
sum=6

newNum=9
sum=15

newNum=17
sum=32

tuckernuck:scripts alf$
```

Initializing sum to 0

while loop - run loop while the variable $# is greater than 0

- $# = number of parameters in exec call

- shift = kick out $1 and shift rest down (i.e. $1 ← $2, $2 ← $3, $3 ← $4, ...)

- arithmetic comparisons:

-lt	<
-gt	>
-le	<=
-ge	>=
-eq	==
-ne	!=

Script 4: oracle

- `if/else` statement
- nesting

oracle

```
#!/bin/bash

# oracle guessing game

magicNumber=15

if [ $1 -eq $magicNumber ] ; then
      echo "You're correct!"
else
      if [ $1 -gt $magicNumber ] ; then
            echo "Too high!"
      else
            echo "Too low!"
      fi
fi
```

if loop

nested if loop

Terminal — bash — 53×19

```
tuckernuck:scripts alf$ ./oracle
./oracle: line 7: [: -eq: unary operator expected
./oracle: line 10: [: -gt: unary operator expected
Too low!
tuckernuck:scripts alf$ ./oracle 26
Too high!
tuckernuck:scripts alf$ ./oracle -12
Too low!
tuckernuck:scripts alf$ ./oracle 15
You're correct!
tuckernuck:scripts alf$
```

- can also use the construct:

```
if [ ] ; then
elif [ ] ; then
elif [] ; then
else
fi
```

Script 5: calculator

- `case conditional`
- `exit`

calculator

```
#!/bin/bash

# calculator for integer pairs

if [ $# -ne 3 ] ; then
      echo "ERROR - unexpected number of parameters"
      echo "Usage: ./calculator <int1> <int2> <operation>"
      exit
fi

case "$3" in
      "add" | "sum")
            z=`expr $1 + $2`
            ;;
      "subtract")
            z=`expr $1 - $2`
            ;;
      "multiply")
            z=`expr $1 \* $2`
            ;;
      "divide")
            z=`expr $1 / $2`
            ;;
      *)
            écho "4 supported operations: (add|subtract|multiply|divide)"
            echo exit
esac

echo $z
```

safeguard on usage

- `exit` terminates script

case conditional

- starts `case`, ends `esac`
- `)` terminates pattern match
- `;;` terminates each case
- `|` is the "or" character
- `*` is the wildcard "catch all"

Script 5: calculator

Terminal — bash — 56×15

```
tuckernuck:scripts alf$ ./calculator
ERROR - unexpected number of parameters
Usage: ./calculator <int1> <int2> <operation>
tuckernuck:scripts alf$ ./calculator 7 3 add
10
tuckernuck:scripts alf$ ./calculator 7 3 sum
10
tuckernuck:scripts alf$ ./calculator 7 3 subtract
4
tuckernuck:scripts alf$ ./calculator 7 3 multiply
21
tuckernuck:scripts alf$ ./calculator 7 3 divide
2
tuckernuck:scripts alf$
```

Script 6: stringer

- arrays
- $@

stringer

```
#!/bin/bash

# appends ".txt" to all strings other than "virus"

strArray=("$@")
echo "strArray = ${strArray[@]}"

fileArray=()
for str in ${strArray[@]} ; do
    if [ $str != "virus" ] ; then
        sz=${#fileArray[@]}
        fileArray[`expr $sz + 1`]=$str"".txt
    fi
done
echo "fileArray = ${fileArray[@]}"
```

Terminal — bash — 65×11

```
tuckernuck:scripts alf$ ./stringer good bad ugly
strArray = good bad ugly
fileArray = good.txt bad.txt ugly.txt
tuckernuck:scripts alf$ ./stringer clean1  virus clean2 clean3
strArray = clean1 virus clean2 clean3
fileArray = clean1.txt clean2.txt clean3.txt
tuckernuck:scripts alf$
```

Create an array strArray from parameters

- `$@` = all parameters passed to bash call
- `${ARRAY[@]}` = array contents

Create empty array fileArray

For all strings except "virus" append txt and store in fileArray

- `${#ARRAY[@]}` = array size
- "" terminates $ dereference string
- str comparisons:

=	equal
!=	not equal
>	greater than
<	less than
-n <str>	not empty

Script 7: filer

- infinite loop
- `Ctrl + C`
- read user input

filer

```
#!/bin/bash

# tests whether user supplied strings are files in pwd

while [ 1 -ne 0 ] ; do
      echo "Please enter a file name to test (Ctrl + C to exit):"
      read str
      if [ -f $str ] ; then
            echo "$str is a regular file in pwd"
      else
            echo "$str is not a regular file in pwd"
      fi
      echo ""
done
```

Terminal — bash — 67×19

```
tuckernuck:scripts alf$ ls
backup          filer           myDirectory     stringer
calculator      helloWorld      oracle          summer
tuckernuck:scripts alf$ ./filer
Please enter a file name to test (Ctrl + C to exit):
summer
summer is a regular file in pwd

Please enter a file name to test (Ctrl + C to exit):
winter
winter is not a regular file in pwd

Please enter a file name to test (Ctrl + C to exit):
myDirectory
myDirectory is not a regular file in pwd

Please enter a file name to test (Ctrl + C to exit):
^C
tuckernuck:scripts alf$
```

Infinite loop (`Ctrl + C` to break)

Read user input into str

Test if str is a regular file in the present working directory

- file comparison operators
  - -e file exists (may be directory)
  - -f file exists (not directory)
  - -d directory exists
  - -r file readable
  - -w file writable
  - -x file executable

Script 8: squarer

- iterating
- functions
- `sleep`

squarer

```
#!/bin/bash

# function declarations

function square() {
        z=`expr $1 \* $1`
        echo $z
}

# prints square of every eighth number between 56 and 107

start=56
stop=107
step=8

i=$start
while [ $i -le $stop ] ; do
        i2=`square $i`
        echo "$i x $i = $i2"
        sleep 0.5
        i=`expr $i + $step`
done
```

Declaring a function at top of script

As for main function `$1`,`$2`,... are passed variables

Setting up the iterative loop

Performing square using our function

`sleep 0.5` = 0.5 s pause between prints

incrementing loop variable

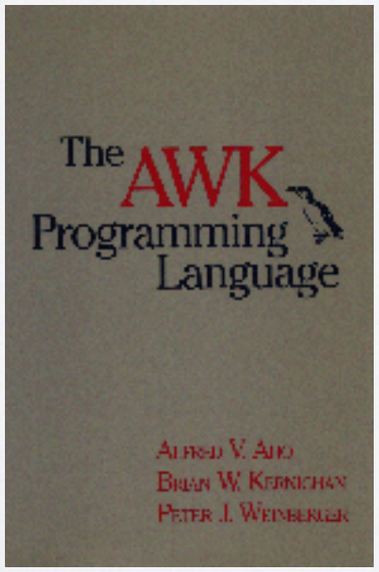
Terminal — bash — 47×10

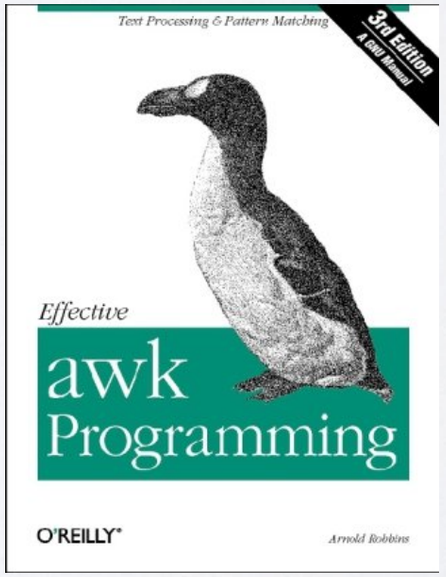
```
tuckernuck:scripts alf$ ./squarer
56 x 56 = 3136
64 x 64 = 4096
72 x 72 = 5184
80 x 80 = 6400
88 x 88 = 7744
96 x 96 = 9216
104 x 104 = 10816
tuckernuck:scripts alf$
```

VI. awk

awk

- awk is a programming language in its own right
- Developed at Bell Labs in 70's by Aho, Weinberger, & Kernighan
- Powerful, simple, fast and flexible language
- Standard part of most Linux distributions, used primarily for rapid and efficient line-by-line text processing





Why awk?

- *"Forget awk, I'll just use vi / emacs / Notepad!"*
- *OK, good luck...*
  - extract the third column of this 50,000 line file
  - divide the second field of each line by the seventh, and save results in csv format
  - extract every 15th line of this file and invert the field ordering to run from last to first

- awk can do these things (and many others!) extremely efficiently and quickly using "one liner" commands
  - integrates seamlessly into bash shell `cat <file> | awk ...`
  - integrates seamlessly into bash scripts
  - great power using only a handful of commands

awk basics

- Rudimentary awk, comprehensive beginner's tutorial at: http://www.grymoire.com/Unix/Awk.html

- Anatomy of an awk program

```
awk 'BEGIN { ... }
{ ... }
END { ... }'
inFile > outFile
```

Do stuff before starting [optional]

Line-by-line processing

Do stuff after end-of-file [optional]

Read from inFile, write to outFile

- Can place within a script, or enter directly into terminal

- White space doesn't matter

awk basics

- Alternatively, can pipe input from terminal

```
cat inFile | awk 'BEGIN { ... }
                  { ... }
                  END { ... }'
                  > outFile
```

- Omit "> outFile" to output directly to terminal

- Use ">>" instead of ">" to append rather than overwrite

```
cat inFile > awk 'BEGIN { ... }
                  { ... }
                  END { ... }'
                  >> outFile
```

What goes in the { }?

- Commands perform line-by-line text processing
- Assignment of internal awk variables
- Flow control and loops
- Pulling in of bash variables from surrounding script
- Printing to terminal or file
- Basic arithmetic

[bash + awk] script example

- Extract the x,y,z coordinates of peptide atoms from pdb formatted files peptide[1-3].pdb into coords[1-3].txt
- Concatenate coords[1-3].txt into coords_concat.txt
- Use bash to iterate over files
- Use awk to perform text processing

peptide1.pdb

```
REMARK    GENERATED BY TRJCONV
TITLE     Protein in water t=   0.00000
REMARK    THIS IS A SIMULATION BOX
CRYST1   30.000   30.000   30.000  90.00  90.00  90.00 P 1           1
MODEL        1
ATOM      1  CH3 ACE     1      12.710  13.550  14.060  1.00  0.00
ATOM      2  C   ACE     1      13.940  13.780  13.200  1.00  0.00
ATOM      3  O   ACE     1      13.810  13.690  11.980  1.00  0.00
ATOM      4  N   ALA     2      15.140  14.090  13.710  1.00  0.00
ATOM      5  CA  ALA     2      15.470  14.590  15.030  1.00  0.00
ATOM      6  CB  ALA     2      15.820  13.370  15.880  1.00  0.00
ATOM      7  C   ALA     2      16.680  15.490  14.870  1.00  0.00
ATOM      8  O   ALA     2      17.650  15.130  14.200  1.00  0.00
ATOM      9  N   NAC     3      16.760  16.620  15.570  1.00  0.00
ATOM     10  CH3 NAC     3      15.700  17.290  16.300  1.00  0.00
TER
ENDMDL
```

coords1.txt

```
 12.710  13.550  14.060
 13.940  13.780  13.200
 13.810  13.690  11.980
 15.140  14.090  13.710
 15.470  14.590  15.030
 15.820  13.370  15.880
 16.680  15.490  14.870
 17.650  15.130  14.200
 16.760  16.620  15.570
 15.700  17.290  16.300
```

[bash + awk] script example

awker

```
#!/bin/bash

# extract x,y,z coords from pdb files

# setting up file iteration

inFile_name=peptide
inFile_suff=pdb

outFile_name=coords
outFile_suff=txt

start=1
stop=3
step=1

if [ -e $outFile_name""_concat.$outFile_suff ] ; then
      rm $outFile_name""_concat.$outFile_suff
fi
touch $outFile_name""_concat.$outFile_suff

# iterating over files

i=$start
while [ $i -le $stop ] ; do

      awk '{
            if (NR > 5 && NR < 16) {
                  printf("%8.3f%8.3f%8.3f\n",$6,$7,$8)
            }
      }' "$inFile_name$i.$inFile_suff" > "$outFile_name$i.$outFile_suff"

      cat $outFile_name$i.$outFile_suff >> $outFile_name""_concat.$outFile_suff
      #rm $outFile_name$i.$outFile_suff

      i=`expr $i + $step`
done
```

setting up in and out files

setting up iteration

initializing concat file

awking each file in turn
- printf is formatted print
- formatting like Matlab
- $n are field codes
- NR is a special variable for number of records = number of lines

cat each file into concat file

rm each coordsX.txt file?

increment iterator

[bash + awk] script example

coords_concat.txt

```
12.710  13.550  14.060
13.940  13.780  13.200
13.810  13.690  11.980
15.140  14.090  13.710
15.470  14.590  15.030
15.820  13.370  15.880
16.680  15.490  14.870
17.650  15.130  14.200
16.760  16.620  15.570
15.700  17.290  16.300
12.718  13.940  14.128
13.912  13.876  13.177
13.779  13.427  12.038
15.126  14.114  13.654
15.559  14.482  14.993
15.903  13.162  15.687
16.707  15.479  14.919
17.719  15.207  14.290
16.661  16.648  15.569
15.603  17.318  16.306
12.710  13.615  13.995
13.968  13.676  13.137
13.908  13.358  11.952
15.136  14.106  13.627
15.468  14.683  14.911
15.388  13.763  16.136
16.738  15.517  14.857
17.472  15.395  13.877
16.923  16.519  15.721
15.969  17.005  16.693
```

- Doing this [by hand / in Excel / in Matlab] at any significant scale would be extremely tedious and error prone!