Disclaimer:

Every example shown here is redundant and overly complex. You can find plenty of examples of simple things on the interwebs, but not so many of complicated stuff. The best thing to do is to write something that works, any way you can, then go back and make it pretty later when you have more time. The point of this is to SAVE time, not waste it writing an overly complex program on top of your other work!!

Structure of a script:

1. Very first thing should be **#!/bin/sh** or whatever shell you prefer, otherwise things probably won't work as expected! Many small and annoying differences that will pop up between Bash (*sh*) and C-shell (*csh*), especially syntax and variables.

2. Pick a shell and learn it well. *csh* or *sh* or any other of the 1/0 derivatives, it doesn't really matter if you have the patience for it. That being said, *sh* is far more friendly for complex scripting tasks, especially when redirecting outputs (*stdout* and *stderr*) or dealing with loops and things. For relatively simple things there is probably little difference, but all of my following examples are in ***sh*** and may or may not work without tweaking if run in raw *csh*. But by explicitly putting **#!/bin/sh** as the first line, it tells computer to run using the *sh* shell no matter what the hell you are running as your default login shell. Vice versa too, if your a masochist and write scripts in *csh.*

3. Comment describing what the script does, or what it takes as input, or what it spits out as output. Sounds silly, but also put a date in there and list your revisions underneath the description. You'll look at this thing in a year and a half and have no idea WTF it does otherwise, or what you had to change before.

4. If you depend on specific things that can vary between computers, define them as variables and then use them that way. It helps 10^9 with portability between your office machine and your home machine, or different universities when you get out of here.

Things you can do with shell scripting:

1. Anything possible interactively on the command line you can do in a script.

2. Make life easier! Why type *ps -ef*, then look through the list for the thing you want to kill, try to remember the PID, then *kill -9* it? Consider the following '*die*' program...

3. Make stupid programs easier to deal with, especially ones not written by you or ones that you don't know/want to know the source code for.

4. Customize Customize Customize. There are already 1/0 ways to do things in Linux, so why not make 1/0 + 1?

Resources:

1. The Google

2. man is your friend and godsend most of the time. Note that you can also usually type 'man whatever' into Google and get the same thing, though probably from the MacOS X man page and may be slightly different (but not much for the core things such as sed and awk and grep). It'll say near the top of the page, or should somewhere prominent.

3. Kernighan & Pike 'The UNIX Programming Environment' is pretty much the standard for the basics of the shell, wildcards, *sed* and *awk*, etc. There are probably > 10 copies in the department between everybody.

Fun Tricks:

1. *$$* returns the PID of the current process, i.e. your script. Handy if you need to create a temp file to stash stuff that is (almost) guaranteed to be unique, and you can *rm* it at the end of your script. See *junker* example. Just remember to *trap* or *rm* them to keep your directory clean!

2. The backbone of a complex script are:
   metacharacters, so keep them in mind as you write
   ( ) \ / [ ] * . ? # ! =
   grep, sed, awk, echo, cat, trap, tr, head, tail (and maybe *perl* or *python*)
   Arguably *sed, grep,* and *awk* are the most important and boku time could be devoted just to these three and regular expressions alone.

3. *sed* is fantastic for processing text files, changing commas to blank spaces, and almost any other mundane task that would otherwise have you changing many lines in the same way or manner.

4. *awk* is best for column processing (photometry output, anyone?), since it splits up the given line into a series of variables, separated by the field separator of your choice. Can also do basic math on said data, making sums, averages, etc. a snap so long as your data are split nicely. Note how *sed* and *awk* can play well together...

```
#!/bin/sh
# die - a script to kill most unruly programs most of the times.  See disclaimers
# RTH 07/29/04 - First try. Works 90% of the time
# RTH 09/28/07 - Fixed many annoyances.  Coding while lightly buzzed, caveat emptor
#  Disclaimers:
#   - Anything other than a y/n or Y/N will exit without error.  Need another case statement
#   - It doesn't deal with multiple instances of a given program, and ignores defunct/zombie
# General notes:
#   grep -v = find all things NOT patching the "pattern"
#   `execute everything in this quotes first`
#   Pipes are your friends!
#   exit 1 bails, sort of like a return -1 in C
# Error checking, bail if no arguments given (otherwise would kill our terminal!)
if test $# = 0
then
  echo "You didn't tell me what to kill!"
  exit 1
fi
# This next bit removes all references of $1 in our script - awk and grep and even /bin/sh will show up
# in the list without filtering them out like this.  You could also do this in sed, or just about any other
# UNIX filter old skool style.  This is all for safety sake, strictly speaking the first instance showing up
# should be the one with the lowest PID and the one started earliest on the system.  But this uses more
# functions and stuff :)
# Nifty trick - note how you can jump in and out of the awk environment to get at my input variable $1
pidder=`ps -ef | grep $USER | grep $1 | awk '{if ($8~'$1') {print $0} }' | \\
       grep -v "/bin/sh" | grep -v "awk" | grep -v "grep" | grep -v "defunct"`
if `echo $pidder | grep -qv $1`
then
  echo "Sorry, nothing called \"$1\" was found running.  Perhaps another name?";
  exit 1
fi
echo "Killing:  $pidder"
read -p "Really do it? (y/n) " answer
ans=`echo $answer | tr [A-Z] [a-z]`
case $ans in
  \y) kill -9 `echo $pidder | awk '{print $2}'` ;;
  \n) exit 1 ;;
esac
```

```
#!/bin/sh
# junker - an example showing stuff that wasn't shown in 'die'

# Exit/bailout conditions to clean up afterwards.  Needs some work
# because it doesn't trap everything perfectly, but I'm too lazy to
# fix it.  This will trap interrupts, and maybe something else too.
trap 'rm $$*.tmp; exit 1' 1 2

# Find all things that begin with the letter A. Remember that case matters!
# Then store the results in $$.tmp
ls A* > $$.tmp
# Print contents of each file given in $$.tmp to stdout
for i in `cat $$.tmp`
do
  echo "From file \"$i\":"
  cat $i
  echo ""
done
rm $$.tmp

# Don't like what something says?  Change it with sed!
echo "Original file:"
cat Knights
echo ""
echo "Changed file, after we run it through sed:"
cat Knights | sed 's/Nih/Ecky-ecky-ecky-zoobang-neeeeeewonggrumblegrumble...neeeeeewong/'
#######
#Afile -
#This is a file.  Hope this is helpful to everybody that I do all this stuff, because shell scripting isn't
#as scary as it really seems.
#######
#Anotherfile -
#This is another file.  You can really do pretty much anything in a loop like this, handy for looping
#through a lot of tedious model parameters or fits or something.
#######
#Knights
#We are the Knights who say Nih!
#Knights changes into this, output to stdout:
#We are the Knights who say Ecky-ecky-ecky-zoobang-neeeeeewonggrumblegrumble...neeeeeewong!
```