The shell
The shell: Why we (I?) teach it

Many of the students in my classes know the computer as a Desktop and a handful of applications (a browser, Word, Excel, maybe R); to introduce the shell means having a discussion about the structure of the computer, about operating systems, about filesystems, about history

The shell offers programmatic access to a computer’s constituent parts, allow students to “do” data analysis on directories, on processes, on their network (and I realize that the “data analysis” metaphor is getting stretched a bit thin)

Given that there are so many flavors of shell, it is also the first time we can talk about choosing tools (that the choice is theirs to make!), about evaluating which shell is best for them, and about the shifting terrain of software development (maybe a point best left for the graduate students)
The shell: Why we teach it

As a practical matter, shell tools are an indispensable part of my own practice, for data “cleaning,” for preliminary data processing, for exploratory analysis; by design, they let you deal with data on a scale that can be difficult from within R

The shell also becomes important as they start to make use of shared course resources (data, software, hardware)
The shell: How I teach it

First, all the students in the class need access to a shell; as we mentioned yesterday, this means having to download something like Cygwin for students with Windows machines (next page)

In my case, lectures can take place in a lab with uniform hardware and software (iMacs); when I’m not in the lab, easily 50% of the students bring their laptops to class and (after preparing them in discussion or lab session), we can have a “find a buddy” interactive session

The tools are taught in the context of some data task (although by this point in our workshop the first couple of data sets that I use have been discussed numerous times and the sense of “discovery” is gone)
What Is Cygwin?

Cygwin is a Linux-like environment for Windows. It consists of two parts:
- A DLL (cygwin1.dll) which acts as a Linux API emulation layer providing substantial Linux API functionality.
- A collection of tools which provide Linux look and feel.

The Cygwin DLL currently works with all recent, commercially released x86 32 bit and 64 bit versions of Windows, with the exception of Windows CE.

Note that the official support for Windows 95, Windows 98, and Windows Me will be discontinued with the next major version (1.7.0) of Cygwin.

What Isn't Cygwin?

- Cygwin is not a way to run native Linux apps on Windows. You have to rebuild your application from source if you want it to run on Windows.
- Cygwin is not a way to magically make native Windows apps aware of UNIX ® functionality, like signals, pts, etc. Again, you need to build your apps from source if you want to take advantage of Cygwin functionality.

Help, contact, web page, other info...

Install or update now! (using setup.exe)
or get help on using setup.exe or find where a package or file lives in the Cygwin release.

Latest Cygwin DLL release version is 1.5.25-14
The shell: What I teach

The next few slides are samples of what I teach; I hadn’t intended to teach the material here, but instead review the kinds of things I talk about.

In a couple places, we’ll have an opportunity to look at some data and I’ll point you to that; we’ll slow down for those less standard parts of what I cover.

They are a work in progress and each year I encounter things that would make them smoother…

Oh and this GAP icon will indicate where my lecture notes have been diced up to focus on just the shell…
For the rest of the session

We will look at the Unix operating system, the philosophy underlying its design and some basic tools

We will use as our case study the last week of traffic across the department’s website www.stat.ucla.edu

You will have a chance to kick the tires on these tools and address some simple web site usage statistics
Operating systems

Most devices that contain a computer of some kind will have an OS; they tend to emerge when the appliance will have to deal with new applications, complex user-input and possibly changing requirements on its function.

Your Tivo, Treo and (soon) Peugeot will all have operating systems.
Operating systems

An operating system is a piece of software (code) that organizes and controls hardware and other software so your computer behaves in a flexible but predictable way.

For home computers, Windows, MacOS and Linux are among the most commonly used operating systems.
Some history

In 1964, Bell Labs partnered with MIT and GE to create Multics (for Multiplexed Information and Computing Service)

“Such systems must run continuously and reliably 7 days a week, 24 hours a day in a way similar to telephone or power systems, and must be capable of meeting wide service demands: from multiple man-machine interaction to the sequential processing of absentee-user jobs; from the use of the system with dedicated languages and subsystems to the programming of the system itself”
Some history

Bell Labs pulled out of the Multics project in 1969, a group of researchers at Bell Labs started work on Unics (Uniplexed information and computing system) because initially it could only support one user; as the system matured, it was renamed Unix, which isn’t an acronym for anything

Richie simply says that Unix is a “somewhat treacherous pun on Multics”
The Unix filesystem

In Multics, we find the first notion of a hierarchical *file system*; files were arranged in a tree structure allowing users to have control of their own areas.

Unix began (more or less) as a file system and then an interactive *shell* emerged to let you examine its contents and perform basic operations.
The kernel and the shell

The Unix *kernel* is the part of the operating system that carries out basic functions like accessing files, handling communication, and other functions will discuss shortly.

The Unix *shell* is a user interface to the kernel (keep in mind that Unix was designed by computer scientists for computer scientists and the interface is not optimized for novices).
Unix shells

A shell is a type of program called an interpreter; think of it as a text-based interface to the kernel.

It operates in a simple loop: It accepts a command, interprets it, executes the command and waits for another.

The shell displays a prompt to tell you that it is ready to accept a command.
* A boring slide, but full of potential!
Unix shells

The shell is itself a program that the Unix operating system runs for you (a program is referred to as a *process* when it is running).

The kernel manages many processes at once, many of which are the result of user commands (others provide services that keep the computer running).

Some commands are built into the shell, others have been added by users.

Either way, the shell waits until the command is executed.
The result of typing in the command `top`; a printout of all the processes running on your computer.

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Name of the command</th>
<th>How hard the computer is thinking about it</th>
<th>How much memory is being used</th>
</tr>
</thead>
<tbody>
<tr>
<td>6420</td>
<td>top</td>
<td>21.8%</td>
<td>456K+</td>
</tr>
<tr>
<td>6334</td>
<td>R.bin</td>
<td>0.0%</td>
<td>14.4M</td>
</tr>
<tr>
<td>6302</td>
<td>tcsh</td>
<td>0.0%</td>
<td>576K</td>
</tr>
<tr>
<td>6301</td>
<td>xterm</td>
<td>0.0%</td>
<td>556K</td>
</tr>
<tr>
<td>6295</td>
<td>tcsh</td>
<td>0.0%</td>
<td>248K</td>
</tr>
<tr>
<td>6288</td>
<td>quartz-wm</td>
<td>0.0%</td>
<td>432K</td>
</tr>
<tr>
<td>6286</td>
<td>Xquartz</td>
<td>2.3%</td>
<td>3.07M</td>
</tr>
<tr>
<td>6285</td>
<td>X11</td>
<td>0.0%</td>
<td>232K</td>
</tr>
<tr>
<td>6260</td>
<td>Keynote</td>
<td>0.0%</td>
<td>186M</td>
</tr>
<tr>
<td>6236</td>
<td>perl</td>
<td>0.0%</td>
<td>3.11M</td>
</tr>
<tr>
<td>5026</td>
<td>Grab</td>
<td>1.5%</td>
<td>4.55M</td>
</tr>
<tr>
<td>5005</td>
<td>Preview</td>
<td>0.0%</td>
<td>4.50M</td>
</tr>
<tr>
<td>5001</td>
<td>Microsoft</td>
<td>0.7%</td>
<td>9.81M</td>
</tr>
<tr>
<td>4987</td>
<td>tcsh</td>
<td>0.0%</td>
<td>564K</td>
</tr>
<tr>
<td>4976</td>
<td>login</td>
<td>0.0%</td>
<td>124K</td>
</tr>
<tr>
<td>4978</td>
<td>tcsh</td>
<td>0.0%</td>
<td>584K</td>
</tr>
</tbody>
</table>
Operating systems

Processor management

Schedules *jobs* (formally referred to as *processes*) to be executed by the computer

Memory and storage management

Allocate space required for each running process in main memory (RAM) or in some other temporary location if space is tight; and supervise the storage of data onto disk
Operating systems

Device management

A program called a *driver* translates data (files from the filesystem) into signals that devices like printers can understand; an operating system manages the communication between devices and the CPU, for example.

Application interface

An API (application programming interface) let programmers use functions of the computer and the operating system without having to know *how* something is done.

User interface

Finally, the operating system turns and looks at you; the UI is a program that defines how users interact with the computer -- some are graphical (Windows is a GUI) and some are text-based (your Unix shell).
Unix shell(s)

There are, in fact, many different kinds of Unix shells

The table on the right lists a few of the most popular; your default shell is tcsh

<table>
<thead>
<tr>
<th>Shell Name</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bourne shell</td>
<td>/bin/sh</td>
<td>The oldest and most standardized shell. Widely used for system startup files (scripts run during system startup). Installed in Mac OS X.</td>
</tr>
<tr>
<td>Bash (Bourne Again SHell)</td>
<td>/bin/bash</td>
<td>Bash is an improved version of sh. Combines features from csh, sh, and ksh. Very widely used, especially on Linux systems. Installed in Mac OS X. <a href="http://www.gnu.org/manual/bash/">http://www.gnu.org/manual/bash/</a></td>
</tr>
<tr>
<td>C shell</td>
<td>/bin/csh</td>
<td>Provides scripting features that have a syntax similar to that of the C programming language (originally written by Bill Joy). Installed in Mac OS X.</td>
</tr>
<tr>
<td>Korn shell</td>
<td>/bin/ksh</td>
<td>Developed at AT&amp;T by David Korn in the early 1980s. Ksh is widely used for programming. It is now open-source software, although you must agree to AT&amp;T's license to install it. <a href="http://www.kornshell.com">http://www.kornshell.com</a></td>
</tr>
<tr>
<td>TC Shell</td>
<td>/bin/tcsh</td>
<td>An improved version of csh. The t in tcsh comes from the TENEX and TOPS-20 operating systems, which provided a command-completion feature that the creator (Ken Greer) of tcsh included in his new shell. Wilfredo Sanchez, formerly lead engineer on Mac OS X for Apple, worked on tcsh in the early 1990s at the Massachusetts Institute of Technology.</td>
</tr>
<tr>
<td>Z shell</td>
<td>/bin/zsh</td>
<td>Created in 1990, zsh combines features from tcsh, bash, and ksh, and adds many of its own. Installed in Mac OS X. <a href="http://zsh.sourceforge.net">http://zsh.sourceforge.net</a></td>
</tr>
</tbody>
</table>
The original UNIX shell was a simple program written in the Fortran programming language. While it was primarily intended to be a command interpreter for the UNIX operating system, it was also used as a programming language. Its primary contribution was that it allowed users to script shell commands, which could be used to automate repetitive tasks.

At the same time, at the University of California at Berkeley, Bill Joy put together a new shell called the C shell. Like the Thompson shell, it was implemented as a command interpreter, but it was not a programming language. While the C shell contained language facilities such as flow control, variables, and arithmetic, it was not intended to be used as a programming language. Instead, it was designed to be a better command interface. It introduced the idea of a history list and an editing facility, so that users didn't have to retype commands that they had entered incorrectly.

In 1982, the UNIX System V shell was converted to K&R C. By this time, the shell had evolved significantly. It had become a full-fledged programming language, and it was used to write modular code that could be reused. The shell had also become more powerful, with features such as history, aliases, and job control.

As use of ksh grew, the need for more functionality became apparent. Like the original shell, ksh was first used primarily for setting up processes and controlling shell scripts. However, it also had the ability to control the execution of commands and the flow of program execution.

When the shell returned from invoking /bin/goto, it read the script. One of the often overlooked contributions of the System V shell and the 1988 version of ksh was the use of shell scripts as part of a pipeline. Shell scripts were added to make it easier to write modular code, since shell scripts could be executed in parallel.

The application was coded as shell scripts. We added a built-in command to output shell variables through a built-in command, and added echo, pwd, and test built-in commands for improved performance. Finally, we added a built-in command to run a command as a process.

At the time of the so-called "shell wars," I worked on a project to develop a new shell that would be more powerful and flexible. We decided to build a new shell, called the "Bourne shell," that would be a superset of the System V shell and a subset of ksh. The new shell was written in an Algol-like variant of C, and it was converted to a more standard K&R version of C. We also added a built-in command to output shell variables through a form mask. We also added a built-in command named let to do arithmetic using a small subset of the C language expression specification. The Bourne shell was written in an Algol-like variant of C, and it was converted to a more standard K&R version of C. We also added a built-in command to output shell variables through a form mask. We also added a built-in command named let to do arithmetic using a small subset of the C language expression specification.
Why the choices?

A shell program was originally meant to take commands, interpret them and then execute some operation

Inevitably, one wants to collect a number of these operations into programs that execute compound tasks; at the same time you want to make interaction on the command line as easy as possible (a history mechanism, editing capabilities and so on)

The original Bourne shell is ideal for programming; the C-shell and its variants are good for interactive use; the Korn shell is a combination of both
And while we are at it...

Unix itself comes in different flavors; the 1980s saw an incredible proliferation of Unix versions, somewhere around 100 (System V, AIX, Berkeley BSD, SunOS, Linux, ...)

Vendors provided (diverging) version of Unix, optimized for their own computer architectures and supporting different features

Despite the diversity, it was still easier to “port” applications between versions of Unix than it was between different proprietary OS

In the 90s, some consolidation took place; today Linux dominates the low-end market, while Solaris, AIX and HP-UX are leaders in the mid-to-high end
A few common commands

First, commands to explore your file system; walk through directories and list files

pwd, ls, cd

mkdir, rmdir

cp, mv, rm
[fad-gadget ""] pwd
/Users/cocteau

[fad-gadget ""] cd stat202
[fad-gadget "/stat202"] pwd
/Users/cocteau/stat202

[fad-gadget "/stat202"] ls
access_2005-12-23_log  lecture1.key  lecture3.key
lec6.key  lecture2  lecture4.key
lecture1  lecture2.key  mtd.key
[fad-gadget "/stat202"]
[fad-gadget "/stat202"] ls -l

```
-rw-r--r--  1  cocteau  staff  2660935974  4  Oct  08:46  access_2005-12-23_log
-rw-r--r--  1  cocteau  staff      1122  2  Oct 15:15   lec6.key
-rw-r--r--  1  cocteau  staff       374  2  Oct 15:49   lecture1
-rw-r--r--  1  cocteau  staff      2924  2  Oct 15:57  lecture1.key
-rw-r--r--  1  cocteau  staff       170  4  Oct 10:33   lecture2
-rw-r--r--  1  cocteau  staff      1462  4  Oct 10:43  lecture2.key
-rw-r--r--  1  cocteau  staff      1130  4  Oct 08:44   lecture3.key
-rw-r--r--  1  cocteau  staff      1020  4  Oct 10:10  lecture4.key
-rw-r--r--  1  cocteau  staff      5168  2  Oct 15:08   mtd.key
```

[fad-gadget "/stat202"]
Another view of the filesystem; here, your Mac will display directories as folders and you navigate by clicking rather than typing commands.
An example

```
[fad-gadget ~/stat202] ls -l

  total 5197144
  drwxr-xr-x   11 cocteau  staff         374  4 Oct 10:10  .
  drwx------  315 cocteau  staff       10710  4 Oct 08:59  ..
  -rw-r--r--    1 cocteau  staff  2660935974  4 Oct 08:46  access__2005...
  drwxr-xr-x   33 cocteau  staff         1122  2 Oct 15:15  lec6.key
  drwxr-xr-x   11 cocteau  staff         374  2 Oct 15:49  lecture1
  drwxr-xr-x   86 cocteau  staff         2924  2 Oct 15:57  lecture1.key
  drwxr-xr-x    4 cocteau  staff         136  4 Oct 09:41  lecture2
  drwxr-xr-x   32 cocteau  staff         1088  4 Oct 10:10  lecture2.key
  drwxr-xr-x   35 cocteau  staff         1190  4 Oct 08:44  lecture3.key
  drwxr-xr-x  152 cocteau  staff         5168  2 Oct 15:08  mtd.key

Your user name
  Shorthand for your present working directory (where you’re at)
  Shorthand for the directory one level above

The group you belong to that owns the file
  The file’s size in bytes
  The file’s creation date
  The file’s name
Kinds of files

What you’ll notice right away is that there are different types of files having different permissions

Unix filesystem conventions places (shared, commonly used) executable files in places like /usr/bin or /usr/local/bin

Different files are opened by different kinds of programs; in OSX, there is a beautiful command called open that decides which program to use
Kinds of files

Filenames which contain special characters like * and ~ are candidates for *filename substitution*

~ refers to your home directory and * is a wildcard for any number of characters

Other special characters like {, [ and ? can also be expanded, but we’ll get to them when we learn a bit more about regular expressions
Unix shells

There are many flavors of Unix Shells; that is, there are many kinds of programs that operate as shells

sh, csh, tcsh, bash, ksh

They are all programs and can be found on the file system

which sh
An example: HTTP access logs

www.stat.ucla.edu

The department runs an Apache Web server running on taia.stat.ucla.edu

Each request, each click by a user out on the Internet browsing our site, is logged

There are standards for these files, but in general, they can be a bit hairy to “parse”
Data

The students (if I’m in the lab) will have a data set (access_log.txt) pre-loaded for them; otherwise I point them to the location of the data and we (inevitably) have to talk about downloading, etc.

http://www.stat.ucla.edu/~cocteau/stat202a/data
HTTP access logs

A bit of digging...

Commands

- `pwd`, `ls`, `cd`
- `more/less`, `tail`, `wc`
- `cut`, `sort`, `uniq`
% head access_log.txt
134.226.32.57 - - [20/Sep/2007:07:54:29 -0700] "GET /~sczhu/icons/daught.gif HTTP/1.0" 200 1898 "http://www.stat.ucla.edu/~sczhu/" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7"
134.226.32.57 - - [20/Sep/2007:07:54:29 -0700] "GET /~sczhu/icons/bio.gif HTTP/1.0" 200 1681 "http://www.stat.ucla.edu/~sczhu/" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7"
134.226.32.57 - - [20/Sep/2007:07:54:30 -0700] "GET /~sczhu/Zhu_LA_sm.gif HTTP/1.0" 200 39313 "http://www.stat.ucla.edu/~sczhu/" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7"
134.226.32.57 - - [20/Sep/2007:07:54:30 -0700] "GET /favicon.ico HTTP/1.0" 200 318 ""Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7"
164.67.132.219 - - [20/Sep/2007:07:54:55 -0700] "GET /robots.txt HTTP/1.0" 200 559 ""gsa-crawler%20%28gsa1%2C%20contact%3A%20jhuang%40ais.ucla.edu%29%20(Enterprise;%20S5-J4JEBZS9PUJJA; jhuang@ais.ucla.edu)"
164.67.132.219 - - [20/Sep/2007:07:54:55 -0700] "GET /rss/feed.php?unit=uclastat HTTP/1.0" 200 1739 ""gsa-crawler%20%28gsa1%2C%20contact%3A%20jhuang%40ais.ucla.edu%29%20(Enterprise;%20S5-J4JEBZS9PUJJA; jhuang@ais.ucla.edu)"
134.226.32.57 - - [20/Sep/2007:07:55:03 -0700] "GET /%7Esczhu/talks.html HTTP/1.0" 200 9489 "http://www.stat.ucla.edu/~sczhu/" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7"
134.226.32.57 - - [20/Sep/2007:07:55:03 -0700] "GET /%7Esczhu/icons/back2.gif HTTP/1.0" 200 17061 "http://www.stat.ucla.edu/%7Esczhu/talks.html" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7"
134.226.32.57 - - [20/Sep/2007:07:55:03 -0700] "GET /%7Esczhu/icons/bio.gif HTTP/1.0" 200 1681 "http://www.stat.ucla.edu/%7Esczhu/talks.html" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.7) Gecko/20070914 Firefox/2.0.0.7"
% wc access_log.txt
200000 3890201 46321543 access.txt
% tail access_log.txt
76.169.68.146 - - [26/Sep/2007:19:31:57 -0700] "GET /graphics/rss20.gif HTTP/1.1" 200 219 "http://www.stat.ucla.edu/" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)"
76.168.75.194 - - [26/Sep/2007:19:31:58 -0700] "GET /favicon.ico HTTP/1.1" 304 - ""Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.0.12) Gecko/20070731 Ubuntu/dapper-security Firefox/1.5.0.12"
68.180.251.16 - - [26/Sep/2007:19:32:12 -0700] "GET /index.php HTTP/1.0" 200 17447 ""Wget/1.10.2 (Red Hat modified)"
68.180.251.16 - - [26/Sep/2007:19:32:12 -0700] "GET /index.css HTTP/1.1" 200 5869 "http://www.stat.ucla.edu/" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; InfoPath.1; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30)"
68.180.251.16 - - [26/Sep/2007:19:32:12 -0700] "GET /css/uclastat/site.css HTTP/1.1" 200 4822 "http://www.stat.ucla.edu/" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; InfoPath.1; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30)"
Combined log format

IP address
Identity
Userid
date
Request
Status
Bytes
Referrer
Agent
Unix pipes

Programs usually take some kind of input and generate some kind of output.

Unix tools often take input from the user and print the output to the screen.

“Redirection” of data to and from files or programs is controlled by pipes.
Redirecting output with "|"

Takes output from one command and submits it as input to the next command

Examples

```
cut -d" " -f1,10 access_log.txt

cut -d" " -f9 access_log.txt
```
```
% cut -d" " -f9 access_log.txt | head

200
200
200
200
200
302
200
304
200
200
200
200
200
200
200
200
200
200
200
302
200
200
200
200
200
200
...  
200
```
In general...

```
cut -d" " -f1,5,9  select the first, fifth and ninth
cut -d" " -f1-5    select the first through the fifth
cut -d" " -f1       select just the first
```
Sending output to a file with “>”

With this form of redirection, we take a stream of processed data and store it in a file

Example

```bash
cut -d" " -f1 access_log.txt > ips.txt
```
Taking input from a file with "<"

With this form of redirection, we create an input stream from a file

Example

```bash
wc < access_log.txt
```
The pipeline

As the name might imply, you can connect pipes and have data stream from process to process

Example

cut -d" " -f1 access_log.txt | sort | uniq -c | sort -rn

cut -d" " -f9 access_log.txt | sort | uniq -c

cut -d" " -f1 access_log.txt | sort | uniq | wc
% cut -d " " -f1 access_log.txt | sort | uniq | wc

17128  17128  238213

% cut -d"" -f9 access_log.txt | sort | uniq -c | sort -rn

158760 200
16161  304
9690   206
6794   404
6043   301
1652   403
836    302
 25    401
 19    405
 12    500
   4    400
   3    501
   1    416
What are these numbers?

A fast Google search gives us a list of possible errors

Note that Error 200 actually means a success

Error 206 means that only part of the file was delivered; the user cancelled the request before it could be delivered

Error 304 is “not modified”; sometimes clients perform conditional GET requests

HTTP Error 101
Switching Protocols. Again, not really an “error”, this HTTP Status Code means everything is working fine.

HTTP Error 200
Success. This HTTP Status Code means everything is working fine. However, if you receive this message on screen, obviously something is not right... Please contact the server’s administrator if this problem persists. Typically, this status code (as well as most other 200 Range codes) will only be written to your server logs.

HTTP Error 201
Created. A new resource has been created successfully on the server.

HTTP Error 202
Accepted. Request accepted but not completed yet, it will continue asynchronously.

HTTP Error 203
Non-Authoritative Information. Request probably completed successfully but can't tell from original server.

HTTP Error 204
No Content. The requested completed successfully but the resource requested is empty (has zero length).

HTTP Error 205
Reset Content. The requested completed successfully but the client should clear down any cached information as it may now be invalid.

HTTP Error 206
Partial Content. The request was canceled before it could be fulfilled. Typically the user gave up waiting for data and went to another page. Some download accelerator programs produce this error as they submit multiple requests to download a file at the same time.

HTTP Error 300
Multiple Choices. The request is ambiguous and needs clarification as to which resource was requested.

HTTP Error 301
Moved Permanently. The resource has permanently moved elsewhere, the response indicates where it has gone to.

HTTP Error 302
Moved Temporarily. The resource has temporarily moved elsewhere, the response indicates where it is at present.

HTTP Error 303
See Other/Redirect. A preferred alternative source should be used at present.
% cut -d" " -fl access_log.txt | sort | uniq -c | sort -rn | more

13050 70.184.223.117
8086 164.67.132.219
4227 164.67.132.220
2304 128.97.86.248
1661 128.97.55.194
1360 66.249.73.99
1161 128.97.55.208
1081 208.68.136.250
1064 207.46.98.57
956 76.167.214.187
808 207.46.98.56
763 87.237.114.11
757 207.46.98.58
720 63.241.61.68
668 61.149.63.50
569 164.67.134.26
548 69.12.181.75
518 196.1.114.240
513 65.55.209.79
505 76.167.183.169
503 65.55.209.83
497 217.212.224.159
496 76.168.72.146
487 65.55.209.82
478 65.55.209.78
473 65.55.209.80
The pipeline

In 1972, pipes appear in Unix, and with them a philosophy, albeit after some struggle for the syntax; should it be

\[
\text{more(sort(cut)))}
\]

[Remember this; S/R has this kind of functional syntax]

The development of pipes led to the concept of tools -- software programs that would be in a tool box, available when you need them

“And that’s, I think, when we started to think consciously about tools, because then you could compose things together... compose them at the keyboard and get them right every time.”

from an interview with Doug McIlroy
Read the man pages!

If the command `uniq` is unfamiliar, you can look up its usage

Example

```
man uniq
man host
```
**NAME**
uniq - report or filter out repeated lines in a file

**SYNOPSIS**
uniq [-c | -d | -u] [-f fields] [-s chars] [input_file [output_file]]

**DESCRIPTION**
The `uniq` utility reads the standard input comparing adjacent lines, and writes a copy of each unique input line to the standard output. The second and succeeding copies of identical adjacent input lines are not written. Repeated lines in the input will not be detected if they are not adjacent, so it may be necessary to sort the files first.

The following options are available:

- `-c` Precede each output line with the count of the number of times the line occurred in the input, followed by a single space.
- `-d` Don't output lines that are not repeated in the input.
- `-f fields`
% host 70.184.223.117
117.223.184.70.in-addr.arpa domain name pointer
wsip-70-184-223-117.om.om.cox.net.

% host 164.67.132.219
219.132.67.164.in-addr.arpa domain name pointer
gsa1.ais.ucla.edu.

% host 66.249.73.99
99.73.249.66.in-addr.arpa domain name pointer
crawl-66-249-73-99.googlebot.com..
whois 70.184.223.117
Cox Communications Inc. NETBLK-COX-ATLANTA-10 (NET-70-160-0-0-1)
70.160.0.0 - 70.191.255.255
Cox Communications NETBLK-OM-CBS-70-184-208-0 (NET-70-184-208-0-1)
70.184.208.0 - 70.184.223.255

# ARIN WHOIS database, last updated 2007-09-26 19:10
# Enter ? for additional hints on searching ARIN's WHOIS database.
blowtorch:- cocteau$ whois 66.249.73.99

OrgName:    Google Inc.
OrgID:      GOGL
Address:    1600 Amphitheatre Parkway
City:       Mountain View
StateProv:  CA
PostalCode: 94043
Country:    US

NetRange:   66.249.64.0 - 66.249.95.255
CIDR:       66.249.64.0/19
NetName:    GOOGLE
NetHandle:  NET-66-249-64-0-1
Parent:     NET-66-0-0-0-0
NetType:    Direct Allocation
NameServer: NS1.GOOGLE.COM
NameServer: NS2.GOOGLE.COM
NameServer: NS3.GOOGLE.COM
NameServer: NS4.GOOGLE.COM
Comment:    
RegDate:    2004-03-05
Updated:    2007-04-10

OrgTechHandle: ZG39-ARIN
OrgTechName:   Google Inc.
OrgTechPhone:  +1-650-318-0200
OrgTechEmail:  arin-contact@google.com

# ARIN WHOIS database, last updated 2007-09-26 19:10
# Enter ? for additional hints on searching ARIN's WHOIS database.
But what are we really after?

Rather than splitting up the data in access_log.txt by day, we might consider dividing it by IP.

Once we have such a thing, we can use the command `wc` to tell us about the number of accesses from each user.

We can also start to “fit” user-level models that can be used to predict navigation.
Rudimentary pattern matching

`grep` can be used to skim lines from a file that have (or don’t have) a particular pattern

Patterns are specified via regular expressions, something we will learn more about later

The name comes from an editing operation on Unix: `g/re/p`

Example

```
grep 85.249.135.15 access_log.txt
grep /~dinov access_log.txt
```
% grep 70.184.223.117 access_log.txt | grep -v library
Quien es mas macho?

In online marketing, hits rule the roost; the more raw traffic you attract, the greater your opportunities for making a sale.

Alright, so it’s not as simple as that, but let’s see what we can learn about centers of activity on our website.
Quien es mas macho?

Compute the number of hits to the portions of the site owned by Song-Chun Zhu, Vivian Lew, Brian Kriegler, Debbie Barrera and Ivo Dinov

Who received the most hits last week?

What can you say about the kinds of files that were downloaded?

What was the most popular portion of each site?
Pull back a little and tell me about the site and the habits of its visitors; specifically, think about:

When is the site active? When is it quiet?

Do the visitors stay for very long? Do they download any of our papers or software? What applications do they run?

On the balance, is our traffic “real” or mostly the result of robots or automated processes?
A second data set

We have assembled a list of all the bylines associated with articles appearing in the New York Times in 1950

Some of this was entered by hand when the archive was scanned into digital form, but that doesn’t mean the data are clean!

We have a simple task: Provide me with a list of journalists and the number of items they wrote in 1950
NYT Data set

We have a simple task: Provide me with a list of journalists and the number of items they wrote in 1950

For this week, simply have a look at the data and anticipate complications that you might encounter when taking on the somewhat simply-stated accounting operation
From machine to machine

Before we get on with today’s lecture, there are a couple dangling topics we should mention

In your other courses (and in the bootcamp a week ago) you will run/ran R on the computer on your desk

The Statistics Department has a number of computers available to you that are more powerful (memory, speed) than those on your desk

You can navigate between machines by invoking `ssh` and move files using `scp` (example shortly)
% ssh otter.berkeley.edu

[fad-gadget ""] ssh otter.berkeley.edu
Password:
Linux Debian/Ubuntu 2.6.12-1-amd64-k8-smp #1 SMP Wed Sep 28 CEST 2005 x86_64

Commands and programs that run under Solaris may not be available
or may behave differently under GNU/Linux.

You have new mail.
Last login: Mon Oct  9 09:54:42 2006 from 169.232.148.1

Please visit http://www.stat.berkeley.edu/trouble when reporting problems.

NOTICE: The SCF modems are temporarily out of service.

>> SYSTEM DOWNTIME: The server will be rebooted at noon on Monday, Oct 9.

otter.berkeley.edu 1> ls *.txt
abstractraphael.txt dj.txt fri.txt infl.txt resp.txt trace.txt
cmu.abs.txt fl.txt ignorable.txt list.txt source.txt tt.txt
tonter.berkeley.edu 2>
Running jobs

Last time we discussed some basic facts about operating systems; a large part of their functioning is devoted to managing jobs (or programs) run by different users.

We used the `top` command to give a dynamic display of what was running, how much of the computer’s resources it was using up, etc.
Running top on lab-compute.stat.ucla.edu

How many processes are running?

How much RAM is available?

<table>
<thead>
<tr>
<th>PID</th>
<th>COMMAND</th>
<th>%CPU</th>
<th>TIME</th>
<th>#TH</th>
<th>#PRTS</th>
<th>MREGS</th>
<th>RPRVT</th>
<th>RSHRD</th>
<th>RSIZE</th>
<th>VSIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>27487</td>
<td>top</td>
<td>9.2%</td>
<td>0:02:50</td>
<td>1</td>
<td>19</td>
<td>22</td>
<td>424K</td>
<td>3.91M</td>
<td>2.29M</td>
<td>26.9M</td>
</tr>
<tr>
<td>27482</td>
<td>tcsh</td>
<td>0.0%</td>
<td>0:00:16</td>
<td>1</td>
<td>15</td>
<td>22</td>
<td>580K</td>
<td>4.16M</td>
<td>1.05M</td>
<td>31.1M</td>
</tr>
<tr>
<td>27481</td>
<td>sshd</td>
<td>0.0%</td>
<td>0:00:01</td>
<td>1</td>
<td>11</td>
<td>40</td>
<td>108K</td>
<td>4.78M</td>
<td>496K</td>
<td>29.9M</td>
</tr>
<tr>
<td>27477</td>
<td>sshd</td>
<td>0.0%</td>
<td>0:00:13</td>
<td>1</td>
<td>18</td>
<td>40</td>
<td>100K</td>
<td>4.78M</td>
<td>1.19M</td>
<td>30.0M</td>
</tr>
<tr>
<td>26416</td>
<td>tcsh</td>
<td>0.0%</td>
<td>0:00:32</td>
<td>1</td>
<td>15</td>
<td>22</td>
<td>580K</td>
<td>4.16M</td>
<td>1.16M</td>
<td>31.1M</td>
</tr>
<tr>
<td>26415</td>
<td>sshd</td>
<td>0.0%</td>
<td>0:00:42</td>
<td>1</td>
<td>11</td>
<td>40</td>
<td>112K</td>
<td>4.78M</td>
<td>496K</td>
<td>29.9M</td>
</tr>
<tr>
<td>26409</td>
<td>sshd</td>
<td>0.0%</td>
<td>0:00:19</td>
<td>1</td>
<td>18</td>
<td>40</td>
<td>100K</td>
<td>4.78M</td>
<td>1.15M</td>
<td>30.0M</td>
</tr>
<tr>
<td>19378</td>
<td>httpd</td>
<td>0.0%</td>
<td>0:00:01</td>
<td>1</td>
<td>12</td>
<td>196</td>
<td>256K</td>
<td>16.3M</td>
<td>1.45M</td>
<td>62.3M</td>
</tr>
<tr>
<td>13328</td>
<td>httpd</td>
<td>0.0%</td>
<td>0:00:01</td>
<td>1</td>
<td>12</td>
<td>196</td>
<td>338K</td>
<td>16.3M</td>
<td>1.62M</td>
<td>62.3M</td>
</tr>
<tr>
<td>18444</td>
<td>check_afp</td>
<td>0.0%</td>
<td>0:00:01</td>
<td>2</td>
<td>24</td>
<td>21</td>
<td>160K</td>
<td>4.21M</td>
<td>1.98M</td>
<td>27.1M</td>
</tr>
<tr>
<td>382</td>
<td>AppleVNCSe</td>
<td>0.0%</td>
<td>3:18:57</td>
<td>7</td>
<td>&gt;&gt;&gt;</td>
<td>49</td>
<td>1.27M</td>
<td>4.32M</td>
<td>2.15M</td>
<td>159M</td>
</tr>
<tr>
<td>381</td>
<td>ARDAgent</td>
<td>0.0%</td>
<td>0:02:73</td>
<td>6</td>
<td>95</td>
<td>78</td>
<td>934K</td>
<td>4.85M</td>
<td>2.16M</td>
<td>192M</td>
</tr>
<tr>
<td>377</td>
<td>SecurityAg</td>
<td>0.0%</td>
<td>0:06:17</td>
<td>1</td>
<td>71</td>
<td>115</td>
<td>3.05M</td>
<td>12.6M</td>
<td>10.1M</td>
<td>216M</td>
</tr>
<tr>
<td>376</td>
<td>authorizat</td>
<td>0.0%</td>
<td>0:00:68</td>
<td>1</td>
<td>22</td>
<td>24</td>
<td>260K</td>
<td>4.35M</td>
<td>368K</td>
<td>27.3M</td>
</tr>
<tr>
<td>375</td>
<td>mcsvd</td>
<td>0.0%</td>
<td>2:24:04</td>
<td>1</td>
<td>43</td>
<td>55</td>
<td>17.0M</td>
<td>5.71M</td>
<td>17.6M</td>
<td>179M</td>
</tr>
<tr>
<td>371</td>
<td>WindowsServ</td>
<td>0.0%</td>
<td>3:32:31</td>
<td>3</td>
<td>109</td>
<td>108</td>
<td>1.35M</td>
<td>11.4M</td>
<td>6.91M</td>
<td>195M</td>
</tr>
</tbody>
</table>
Running `top` on taia.stat.ucla.edu

How many processes are running?
How much RAM is available?
What do you reckon this computer does?
Another way to get at processes

While `top` gives you a dynamic, um constantly updating, view of what the processor is doing, you can use the command `ps` to give you a snapshot.

The command `ps` has lots and lots of options; it lets you look at all users, just a specific user and control the format of the output.

Just typing `ps` will give you the processes that you started (or were started on your behalf); we can also see what others are up to (ah, the joys of a multi-user system).
Running "ps -aux" on login.stat.ucla.edu

What can we see?

Who are the users?

* the option -a gives you information on all users, -u gives you a popular view (fields) of the processes, and -x gives you processes that aren’t necessarily associated with a terminal -- this output has been edited slightly with some "grep -v"s
Job control

Unix allows you to run several processes at once; each process is given a number which you can use to change the status of the process.

Because many jobs are running on the computer, the amount of “attention” they get from the central processing unit is controlled by their priorities (-20 to 20, with the higher the number meaning the lower the priority)

nice and renice lets you lower the priority on a job that you know will run for a long time, freeing system resources for others; kill can be used to end processes (politely or with a greater sense of urgency)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

>
Job control

C-z stops jobs, C-c kills them, and C-d kills your shell

You can also set jobs to run in the *background* (which means your prompt returns)

The command `jobs` lets you see what jobs you have running

If you stop a job, you can restart it or restart it in the background using the commands `bg` and `fg`
In response, Unix stops the job and gives you a list of other jobs you have stopped or are running in the background.
We get to running R on a file of commands, running R in batch mode and so on later in the quarter...
Back to your “assignments” from last time

Who had the most hits and how did you compute it?

What about the open-ended questions... What are the active periods of the day? What did you learn?

Take about a half an hour to discuss what you found
Also...

I might not have emphasized enough that each of the commands we worked with can either take input from a file as in

```
sort access_log.txt
```

or from a pipe (so-called standard input), as in

```
grep dinov access_log.txt | sort
```
New data

We are now going to look at a series of log files containing data from chat sessions recorded last year; in a two-hour recording session last year we captured 46,000 lines from 4,500 people.

The data are stored on lab-compute.stat.ucla.edu and you can bring them to your desktop with the command

```
scp -r lab-compute.stat.ucla.edu:/Data/chat .
```

copy recursively (a directory and all its contents) 

Copy the directory `/Data/chat` from the computer `lab-compute.stat.ucla.edu`

copy it to a directory of the same name (in this case “chat”) on your local machine
```bash
[fad-gadget ""] cd chat
[fad-gadget "/chat"] ls -l
total 11640
-rw-r--r-- 1 cocteau staff 507715 9 Oct 12:40 now.1159914021.txt
-rw-r--r-- 1 cocteau staff 555894 9 Oct 12:40 now.1159917841.txt
-rw-r--r-- 1 cocteau staff 454160 9 Oct 12:40 now.1159920765.txt
-rw-r--r-- 1 cocteau staff 447638 9 Oct 12:40 now.1159924151.txt
-rw-r--r-- 1 cocteau staff 450546 9 Oct 12:40 now.1159927582.txt
-rw-r--r-- 1 cocteau staff 529160 9 Oct 12:40 now.1159992088.txt
-rw-r--r-- 1 cocteau staff 479573 9 Oct 12:40 now.1159995531.txt
-rw-r--r-- 1 cocteau staff 407640 9 Oct 12:40 now.1159998154.txt
-rw-r--r-- 1 cocteau staff 613252 9 Oct 12:40 now.1160410547.txt
-rw-r--r-- 1 cocteau staff 540785 9 Oct 12:40 now.1160413861.txt
-rw-r--r-- 1 cocteau staff 494676 9 Oct 12:40 now.1160416555.txt
-rw-r--r-- 1 cocteau staff 456485 9 Oct 12:40 now.1160419326.txt
[fad-gadget "/chat"] wc *
  10000  92622  507715 now.1159914021.txt
  10000 101173  555894 now.1159917841.txt
  10000  82973  454160 now.1159920765.txt
  10000  81708  447638 now.1159924151.txt
  10000  82399  450546 now.1159927582.txt
  10000  96956  529160 now.1159992088.txt
  10000  88012  479573 now.1159995531.txt
  10000  74815  407640 now.1159998154.txt
  10000 111328  613252 now.1160410547.txt
  10000  98653  540785 now.1160413861.txt
  10000  89663  494676 now.1160416555.txt
  10000  83455  456485 now.1160419326.txt
120000 1083763  5937524 total
[fad-gadget "/chat"]
```
Rudimentary pattern matching

We have already seen some basic pattern matching notions; recall the command “wc *.txt”

In this expression “*” acts as a wildcard and matches anything

The files now.1159914021.txt, now.1159927582.txt, now.1160410547.txt, now.1159917841.txt, now.1159992088.txt, now.1160413861.txt, now.1159920765.txt, now.1159995531.txt, now.1160416555.txt, now.1159924151.txt, and now.1159998154.txt will all be returned by this command
Rudimentary pattern matching

In the expression “*.txt” we can name two kinds of characters

The “.txt” is made up of literal or normal text characters

The “*” is a metacharacter
We get to running R on a file of commands, running R in batch mode and so on later in the quarter...
Before the break...

We discussed your experiments with the Web server data

How would we get a time series of hits per day?
Who had the largest number of hits?
What can you say about the files being accessed?

The command `grep` let us extract lines from a file that contained a string of characters; as we started digging into the data, we wanted a more expressive tool for defining patterns

Before the break, we discussed so-called regular expressions, a language for describing patterns in text data
And in this session...

Sidestepping the issue of a text corpus somewhat, we are now in a good position to start looking at the Enron email data set.

Initially, we have to understand the structure of these data before we dig a bit into the social networking “analysis” that is to come.

It will also give us an opportunity to consider simple shell scripts; a mechanism by which we collect commands into reusable programs.
Enron

Today we are going to start our work on a set of data related to the Enron corporation.

Some relevant links are


http://www.cs.cmu.edu/~enron/

http://www.stat.ucla.edu/~cocteau/klimt-ecml04-1.pdf

http://www.stat.ucla.edu/~cocteau/Enron_Employee_Status.htm
Enron emails

As part of its investigation into Enron, the Federal Energy Regulatory Commission released the emails of about 150 of its top executives.

These data were then cleaned up by groups at MIT and SRI and are now publicly available through the CMU CS Department.

To respect the privacy of the individuals involved, I have replaced the body of each email with x’s; our interest is not in what was said but who sent email to whom.
Organization of the data

The data itself is organized into a series of directories, each named after an executive

Under each directory, you will find possibly more directories, each representing a different mail folder

At the lowest level, you have a series of email messages, one per file; the files in each directory are named 1., 2., 3., etc.

The files we will work with are in /Data/mailfiles on lab-compute
<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
<th>Name</th>
<th>Name</th>
<th>Name</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>allen-p</td>
<td>fischer-m</td>
<td>kitchen-l</td>
<td>phanis-s</td>
<td>smith-m</td>
<td></td>
</tr>
<tr>
<td>arnold-j</td>
<td>forney-j</td>
<td>kuykendall-t</td>
<td>pimenov-v</td>
<td>solberg-g</td>
<td></td>
</tr>
<tr>
<td>arora-h</td>
<td>fossum-d</td>
<td>lavorato-j</td>
<td>platter-p</td>
<td>south-s</td>
<td></td>
</tr>
<tr>
<td>badeer-r</td>
<td>gang-l</td>
<td>lay-k</td>
<td>presto-k</td>
<td>staab-t</td>
<td></td>
</tr>
<tr>
<td>bailey-s</td>
<td>gay-r</td>
<td>lenhart-m</td>
<td>quenet-j</td>
<td>stclair-c</td>
<td></td>
</tr>
<tr>
<td>bass-e</td>
<td>geaccone-t</td>
<td>lewis-a</td>
<td>quigley-d</td>
<td>steffes-j</td>
<td></td>
</tr>
<tr>
<td>baughman-d</td>
<td>germany-c</td>
<td>linder-e</td>
<td>rapp-b</td>
<td>stepenovitch-j</td>
<td></td>
</tr>
<tr>
<td>beck-s</td>
<td>gilbertsmith-d</td>
<td>lokay-m</td>
<td>reitmeyer-j</td>
<td>stokley-c</td>
<td></td>
</tr>
<tr>
<td>benson-r</td>
<td>giron-d</td>
<td>love-p</td>
<td>ring-a</td>
<td>storey-g</td>
<td></td>
</tr>
<tr>
<td>blair-l</td>
<td>griffith-j</td>
<td>lucoy-t</td>
<td>richey-c</td>
<td>sturm-f</td>
<td></td>
</tr>
<tr>
<td>brawner-s</td>
<td>grigsby-m</td>
<td>lucci-p</td>
<td>ring-r</td>
<td>swerzbin-m</td>
<td></td>
</tr>
<tr>
<td>buy-r</td>
<td>guzman-m</td>
<td>maggi-m</td>
<td>rodrigue-r</td>
<td>symes-k</td>
<td></td>
</tr>
<tr>
<td>campbell-l</td>
<td>haedicke-m</td>
<td>mann-k</td>
<td>rogers-b</td>
<td>taylor-m</td>
<td></td>
</tr>
<tr>
<td>carson-m</td>
<td>hain-m</td>
<td>martin-t</td>
<td>ruscitti-k</td>
<td>tholt-j</td>
<td></td>
</tr>
<tr>
<td>cash-m</td>
<td>harris-s</td>
<td>may-l</td>
<td>sager-e</td>
<td>thomas-p</td>
<td></td>
</tr>
<tr>
<td>caushollim-m</td>
<td>haylett-r</td>
<td>mcarthy-d</td>
<td>saibi-e</td>
<td>townsend-j</td>
<td></td>
</tr>
<tr>
<td>corman-s</td>
<td>heard-m</td>
<td>mcconnell-m</td>
<td>salisbury-h</td>
<td>tycholiz-b</td>
<td></td>
</tr>
<tr>
<td>crandell-s</td>
<td>hendrickson-s</td>
<td>mckay-b</td>
<td>sanchez-m</td>
<td>ward-k</td>
<td></td>
</tr>
<tr>
<td>cuilla-m</td>
<td>hernandez-j</td>
<td>mckay-j</td>
<td>sanders-r</td>
<td>watson-k</td>
<td></td>
</tr>
<tr>
<td>dasovich-j</td>
<td>hodge-j</td>
<td>mclaughlin-e</td>
<td>scholes-d</td>
<td>weldon-c</td>
<td></td>
</tr>
<tr>
<td>davis-d</td>
<td>holst-k</td>
<td>merriss-s</td>
<td>schoolcraft-d</td>
<td>whalley-g</td>
<td></td>
</tr>
<tr>
<td>dean-c</td>
<td>horton-s</td>
<td>meyers-a</td>
<td>schwieger-j</td>
<td>whalley-l</td>
<td></td>
</tr>
<tr>
<td>delaine-y</td>
<td>hyatt-k</td>
<td>mims-thurston-p</td>
<td>scott-s</td>
<td>white-s</td>
<td></td>
</tr>
<tr>
<td>derrick-j</td>
<td>hyvl-d</td>
<td>motley-m</td>
<td>semperger-c</td>
<td>whitt-m</td>
<td></td>
</tr>
<tr>
<td>dickson-s</td>
<td>jones-t</td>
<td>neal-s</td>
<td>shackleton-s</td>
<td>williams-j</td>
<td></td>
</tr>
<tr>
<td>donoho-l</td>
<td>kaminski-v</td>
<td>nemec-g</td>
<td>shankman-j</td>
<td>williams-w3</td>
<td></td>
</tr>
<tr>
<td>donohoe-t</td>
<td>keen-s</td>
<td>panus-s</td>
<td>shapiro-r</td>
<td>wolfe-j</td>
<td></td>
</tr>
<tr>
<td>dorland-c</td>
<td>keavey-p</td>
<td>parks-j</td>
<td>shively-h</td>
<td>ybarbo-p</td>
<td></td>
</tr>
<tr>
<td>ermis-f</td>
<td>keiser-k</td>
<td>pereira-s</td>
<td>skillings-j</td>
<td>zipper-a</td>
<td></td>
</tr>
<tr>
<td>farmer-d</td>
<td>king-j</td>
<td>perlingiere-d</td>
<td>slinger-r</td>
<td>zufferli-j</td>
<td></td>
</tr>
</tbody>
</table>
An example

Here we select the ex-Vice President for Regulatory Affairs, Shelley Corman

We see the 11 mail folders; selecting the calendar folder, we exhibit the content of mail 2.

Note again, that all textual content has been replaced by x's; we are only interested in (at best) the pattern of communication
Some questions

What is the distribution of numbers of emails per user?

Are the users organizing their email into folders?

Are certain folders common to all users?

What is the distribution of emails per folder?
Hint: One more helpful command

The Unix command `find` traverses a directory tree and returns the files and directories it finds; you can limit the search with various options.

For example:

Consider only those email messages numbered 404.

```
find corman-s -name 404.
```

Consider only those entries that don’t end in a period (.)

```
find corman-s regex ‘[^.]$’
```
[fad-gadget maildir] find corman-s/-name 404.
corman-s//deleted_items/404.
corman-s//inbox/archives/404.
corman-s//sent_items/404.

[fad-gadget maildir]
[fad-gadget maildir] find corman-s -regex '.*[^.]$'
corman-s
corman-s/all_documents
corman-s/calendar
corman-s/communications
corman-s/contacts
corman-s/deleted_items
corman-s/discussion_threads
corman-s/inbox
corman-s/inbox/archives
corman-s/inbox/archives/old_messages
corman-s/inbox/archives___post_revised_order
corman-s/inbox/bankruptcy
corman-s/inbox/budget
corman-s/inbox/kidsave
corman-s/inbox/measurement
corman-s/inbox/naesb
corman-s/inbox/oneok
corman-s/inbox/tw_neg_rates
corman-s/inbox/twdatarequests
corman-s/inbox/vacation_schedules
corman-s/ingaastudy
corman-s/marketingaffiliate
corman-s/osh
corman-s/sent_items

[fad-gadget maildir]
Putting this to work

We can now answer some of the questions about folder usage with calls to `find`, `cut`, and `sort`; first, emails per user and folders per user.

```
[fad-gadget Data] find maildir -type f | cut -d"/" -f2 | uniq -cl sort -rn | head
28465 kaminski-v
28234 dasovich-j
25351 kean-s
23381 mann-k
19950 jones-t
18687 shackleton-s
13875 taylor-m
13032 farmer-d
12436 germany-c
11830 beck-s

[fad-gadget Data] find maildir -type d | cut -d"/" -f2 | uniq -cl sort -rn | head
199 kean-s
138 beck-s
117 shapiro-r
112 shackleton-s
89 taylor-m
82 mcconnell-m
78 griffith-j
72 germany-c
70 watson-k
70 blair-l
```
Counts per user

As was the case for hit counts per IP address, we see a very skewed distribution (what Malcolm Gladwell would call a “hockey stick” distribution)

In the bottom figures we present a histogram and a Q-Q plot for the logarithm of the counts
Identifying common folders

The email from 150 executives are included in this file; some folders have similar structures
Looking inside

In a previous version of these slides, we considered a unique message ID tag; instead, let’s consider a time series of the number of emails by day.

If we look at the structure of the email header we see that a message’s date is kept in a field called Date:
Message-ID: <4493790.1075858840249.JavaMail.evans@thyme>
Date: Mon, 29 Oct 2001 08:33:30 -0800 (PST)
From: forrester@forrester.com
To: weekly_research@frstrelay001.forrester.com
Subject: New Research From Forrester -- 10/29/2001
Mime-Version: 1,0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
X-From: Forrester Research, Inc. <forrester@FORRESTER.COM>
X-To: WEEKLY_RESEARCH@frstrelay001.forrester.com
X-cc:
X-bcc:
X-Folder: \SCORMAN (Non-Privileged)\Inbox
X-Origin: Cormax-S
X-FileName: SCORMAN (Non-Privileged).pst

==================================================================

xxx xxxxxxxx xx xxx,xxxxxxxxxxxxxxx -- xx/xx/xxxx
==================================================================

xxx xx xxx xxxxxxxxxxxxx xxx
xxx xxxxxxx xxxxxxxxxxxxxx xx xx 'xxx xx xxx xxx xx xxxxxx xxx
xxx xxxxxxx xx xxxxxxxxxxxxx, xxxxx xxxxx xxx xxx xxxxxxxx xxxxxx?
xxx xxx xxx xxxxxx? xx xxxxxx xxxxxxxxx xxx xx xxxxx?
xxxxxxxxxx'x xxxxxxxxxxxxxxxxxxxxxxxxxx xxx xxxxxxxxxx xxx xxxxxxxxxx
xxxxxxxxxxxxxxx xxx xxx xxx xxx xxx xxxxxxxxx'x xxxxxxxxxxxxxxxxxxxxxxxxxx.
xxxxxxxx xxxxxxxxxxx xxxxxxxx:

xxxxxx x. xxxxx, xxxxxxxxxx, xxxxxxxxxxx, xxx xxx, xxxxxxxxxxx xxx
Timing is everything

The dates in Corman’s inbox folder can be extracted with a simple call to `grep`
Shell programs

There are over 150 different directories and it will be hard to extract all the information we are after by hand

Technically, we can use the `find` command to execute a program on each file or directory it encounters*; for the moment, we will ignore this and use date extraction as an application of *shell scripting*

You can collect a series of Unix commands into a shell program; this allows you to repeat commands over different inputs

* The command would look something like
  
  `find maildir -type f -exec egrep '^Date:' {} ';'`

Consult the web site below for more information on `find`

Not really a gap, but a good time to split...

From here, I teach a bit about shell programming; I do this because I want the students to see that the commands they’ve been using can be assembled into programs that can repeat their operations

This will be, of course, a theme in the class; moving from exploratory computing to program-writing; it also lets me talk a bit about permission bits and some trailing filesystem facts
A simple shell program

At the right we have a short program contained in a file dates.sh

OK, it isn’t much of a program, but it’s a reasonably good place to start

The $1 here refers to the first argument we use to call the program

egrep '^Date:' $1
Running a shell script

There are two ways to run a shell script; you can either execute it within a new shell (recall that the shell `sh` is just another command)

```
% sh dates.sh
```

This should explain the funny suffix we used for our filename; this kind of naming convention will help you (and others) recognize this file as a shell script (program)

The second way to run this script is to make the file `executable`; that is, it becomes just like any command Unix knows about

Let’s see how this is done; it requires looking a little into how the filesystem specifies `permissions`, who can do what to a file
Permission bits

Unix can support many users on a single system and each user can belong to one or more groups

Every file in a Unix filesystem is owned by some user and one of that user’s groups; each file also has a set of permissions specifying which users can

r: read
w: write (modify) or
x: execute

the file; these are specified with three “bits” and we need three sets of bits to define what the user can do, what their group (that owns the file) can do and what others can do
An example

[fad-gadget marketingaffiliate] ls -al
total 72
drwxr-xr-x 14 cocteau  staff  476 11 Oct 13:33  ..
-rw-r--r--  1 cocteau  staff  2191 11 Oct 14:14  1.
-rw-r--r--  1 cocteau  staff  4587 11 Oct 14:14  2.
-rw-r--r--  1 cocteau  staff  1061 11 Oct 14:14  3.
-rw-r--r--  1 cocteau  staff  1845 11 Oct 14:14  4.
-rw-r--r--  1 cocteau  staff  2220 11 Oct 14:14  5.
-rw-r--r--  1 cocteau  staff  4163 11 Oct 14:14  6.
-rw-r--r--  1 cocteau  staff  2101 11 Oct 14:14  7.

Your user name
The group you belong to that owns the file
The file’s name
The file’s size in bytes
The file’s creation date
Shorthand for the directory one level above
Shorthand for your present working directory (where you’re at)
An example

The type of file

- `drwxr-xr-x`: 9 cocteau staff
- `drwxr-xr-x`: 14 cocteau staff
- `rw-r--r--`: 1 cocteau staff
- `rw-r--r--`: 1 cocteau staff
- `rw-r--r--`: 1 cocteau staff
- `rw-r--r--`: 1 cocteau staff

What you can do to the file

What the owning group can do

What others can do
In general...

The command chmod changes the permissions on a file; here are some examples

% chmod g+x dates.sh
% chmod ug-x dates.sh
% chmod a+w dates.sh
% chmod go-w dates.sh

You can also use binary to express the permissions; so if we think of bits ordered as rwx, then

<table>
<thead>
<tr>
<th></th>
<th>(1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>r-x</td>
<td>5</td>
</tr>
<tr>
<td>rwx</td>
<td>7</td>
</tr>
<tr>
<td>r--</td>
<td>4</td>
</tr>
</tbody>
</table>

and we can specify permissions with these values

% chmod 755 dates.sh
A simple shell program

In addition to making our program executable, we need to give the operating some help in figuring out what interpreter to use.

That is, we need to tell Unix that the following lines are to be executed in the shell:

```
#!/bin/sh
egrep '^Date:' $1
```

We start the file with the location of the shell command; if we were working in Python (we’ll see this next time), we’d have `usr/bin/python`
A simple shell program

After that long detour, we could do the following; note that to call the program, we have to tell Unix where to find it
A simple shell program

Arguably, we haven’t done much in terms of easing our workload

Instead, we could consider looping over all the files in a directory; the slight elaboration of our original program is given at the right

Here we see our basic command to find dates, but it’s in the body of a for loop

```bash
#!/bin/sh

for i in `ls`; do
  egrep '^Date:' $i
done
```
For loops

The basic structure of this construction is given at the right

If you have done any programming, this loop will function as you expect; each pass through the loop assigns one value in list to var

This is one of several constructions that control the operation or flow of your running program

In our script, the variable i takes the output from the command ls; note that when we want the value of i we use $i

```bash
#!/bin/sh
for i in `ls`; do
egrep '^Date:' $i
done
```
A simple shell program

In our script, the symbol i is a variable; it takes the output from the command ls

When we want the value of i in our script we refer to $i

Variables are used by the shell to remember information; for example, when you start a shell, a number of variables get set by default

```
#!/bin/sh

for i in `ls`; do
  egrep '^Date:' $i
done
```
Your home directory

The path your shell searches for programs; remember which?
A simple shell program

In this short program, we see two different kinds of quotation marks; there are, in fact, three different such constructions

1. `command`: Backquotes execute the enclosed command and catch the output; here it is assigned in turn to i

2. "string": Double quotes allow us to slip in special characters that are expanded; so “echo $1” would print the first argument

3. 'string': Single quotes aren’t very fancy; everything inside is as it appears

```bash
#!/bin/sh

for i in `ls`; do
    egrep '^Date:' $i
done
```
A simple shell program

Running this program generates a single date line for every file in the directory; it scrolls by rather quickly and then...
A simple shell program

...we find a series of errors; what do these mean?
Conditional execution

The problem is that we cannot call egrep on a directory.

Therefore, we’d like to assess what kind of file we are dealing with, and only execute the command where we should.

Unix provides a conditional evaluation utility called `test`; in addition to performing simple numerical comparisons, it also provides facilities for interrogating files.

Here the flag `-f` returns true if `$i` is a regular file.

```bash
#!/bin/sh
for i in `ls`; do
  if test -f $i; then
    egrep '^Date:' $i
  fi
done
```
Conditional execution

Often, rather than explicitly using the `test` function, programmers will use a shorthand construction:

```bash
if condition ; then
  commands
[elif condition; then
  commands]
[else
  commands]
fi
```

The `[]`'s are an implicit call to `test`; to be precise, there's a command called `/bin/`:

```bash
#!/bin/sh
for i in `ls`; do
  if [ -f $i ]; then
    egrep '^Date:' $i
  fi
done
```
Finally...

So far, all we’ve done is execute a `egrep` command in the directory where we call our program.

The commands at the right recurse through the directory provided as an argument to the program:

```bash
#!/bin/sh
for i in `find $1`; do
  if [ -f $i ]; then
    egrep '^Date:' $i
  fi
done
```
A simple shell program

Now we can call the program from any directory; here we call it from /Data (note the ./ telling Unix where to find the file)
Overview

My goal here is to have you realize that the commands we have been working with can be collected into programs or scripts; along the way, we learned something about file permissions.

The scripting facilities in Unix let you do standard things like loop over values, execute commands conditionally, and so on.

There are other structures like `while loops` and `case statements` that further redirect the control of your program.

```bash
while condition; do
    commands
done

case expression in
    pattern)
    commands
;;
esac
```
Overview

Again, much of the recursion over directories can be done with the *find* command

This presentation was meant to walk you through the structure of a shell script, illustrating what programming tools are at your disposal

This material will come up again as we consider distributing code, say through an R package
Oh, and what were we after?

Here we have a plot of the total number of emails recorded during this period; what patterns do we see?
The gist

I teach the shell for both practical as well as pedagogical reasons; it is structurally similar to other “shells” they will encounter (Python, R), but the vocabulary is fairly limited.

With a few commands they can do some pretty powerful analysis, whether or not you want to call that statistics is another question, and you can quickly motivate the need for making programs...

What do y’all think?