ECE560
Computer and Information Security
Fall 2022
Shell Proficiency and Data Manipulation

Tyler Bletsch
Duke University
Motivation

• Everyone needs to manipulate data!

• Attackers need to:
  ▪ Scan target environment for assets
  ▪ Catalog and search target assets for possible vulnerabilities
  ▪ Inspect binaries for specific instruction patterns
  ▪ Extract specific data for processing by other tools (e.g. extracting password hashes from a user database)

• Defenders need to:
  ▪ Scan own environment for assets and malicious entities
  ▪ Catalog own inventory and compare against known vulnerabilities
  ▪ Inspect traffic and data for known attack signatures
  ▪ Extract specific data for processing by other tools (e.g. summarizing login failures to update a firewall blacklist)
Fundamental approach: UNIX Philosophy

• Combine simple tools to get complex effects
• Each tool does one thing and does it well
• Basic format of information is *always* a byte stream and *usually* text
• Core ingredients:
  ▪ Shell (e.g. bash)
  ▪ Pipes and IO redirection
  ▪ A selection of standard tools
• Bonus ingredients:
  ▪ SSH trickery
  ▪ Regular expressions (HUGE!)
  ▪ Terminal magic (color and cursor control)
  ▪ Spreadsheet integration
  ▪ More...
The bash shell and common Unix tools
The bash shell

- Shell: Most modern Linux systems use **bash** by default, others exist
  - We’ll use bash in our examples

- Side-note: You can get a proper UNIX shell on Windows using Windows Subsystem for Linux.
  - PowerShell is Microsoft’s answer to bash...it’s fine.
Shell basics review

- **Standard IO**: stdin, stdout, stderr
- **Pipes**: direct stdout of one to stdin of another
  
  ```
  ls | sort -r  # sort files reverse order
  ```
- **File redirection**: direct any stream to/from a file
  
  ```
  ls > file_list.txt  # save ls to a file (note: no columns!)
  gzip -dc < archive.gz | wc -c  # how big is this file uncompressed?
  find -iname dog.* 2> /dev/null  # suppress stderr
  ```
- **Tab completion**: ALWAYS BE MASHING TAB!!!!!!!!!!!
  - Once = complete, twice = list.
  - Not just to save time – it INFORMS you
- **Semicolon** for multiple commands on one line
  
  ```
  make ; ./myapp
  ```
- Can use && and || for short-circuit logic
  
  ```
  make && ./myapp
  ```
  (Based on return value of program, where 0 is success and nonzero is error)
Stuff from Homework 0 that I assume you know

- `echo`
- `cat`
- `head`
- `tail`
- `less`
- `grep`
- `diff`
- `wc`
- `sort`
- `find`

Note: The guy who did the Lynda video, [Scott Simpson, has more videos](https://www.lynda.com/). See [Learning Bash Scripting](https://www.lynda.com/) for examples of some of the stuff in this lecture.
Bash syntax

• Expansions:
  ▪ **Tilde (~)** is replaced by your home directory (try “echo ~”).
    ~frank expands to frank’s home directory.
  ▪ **Braces** expand a set of options: \{a,b\}{01 .. 03} expands into 6 arguments:
    a01 a02 a03 b01 b02 b03
  ▪ **Wildcards**: ? matches one char in a filename, * matches many chars,
    [qwe0–3] matches just the chars q, w, e, 0, 1, 2, or 3.
    • Non-trivial uses! Find all Makefiles two dirs lower: */*/Makefile
  ▪ **Variables** are set with NAME=VALUE. Values are retrieved with $NAME.
    Names usually uppercase. Fancy expansions exist, e.g. ${FILENAME%.*} will get
    filename extension; see here for info. Variables can be made into environment
    variables with export, e.g. export NAME=VALUE.

• Quotes:
  ▪ By default, each space-delimited token is a separate argument
    (different argv[] elements)
  ▪ To include whitespace in a single argument, quote it.
    • Use **single quotes** to disable ALL expansion listed above: ' |{ool}'
    • Use **double quotes** to allow variable expansion only: "$NAME is |{ool}"
    • Or **backslash** to escape a single character: \$1.21
Bash syntax (2)

- Control and subshells
  
  ```bash
  for NAME in WORDS... ; do COMMANDS; done
  ```
  
  • Execute commands for each member in a list.
  
  ```bash
  while COMMANDS; do COMMANDS; done
  ```
  
  • Execute commands as long as a test succeeds.

  ```bash
  if COMMANDS; then COMMANDS;
  [ elif COMMANDS; then COMMANDS; ]...
  [ else COMMANDS; ] fi
  ```
  
  • Execute commands based on conditional.

- Evaluate to the stdout of COMMAND, e.g.:
  ```bash
  `COMMAND` or $(COMMAND)
  ```
  
  • Evaluate to the stdout of COMMAND, e.g.:
    `USERNAME=`whoami`
Control flow examples

- Keep pinging a server called ‘peridot’ and echo a message if it fails to ping.

  ```bash
  while ping -c 1 peridot > /dev/null ; do sleep 1 ; done ; echo "Server is down!"
  ```

  (Can invert by prepending ‘!’ to ping – waits for server to come up instead)

- Check to see if our servers have been assigned IPs in DNS:

  ```bash
  for A in esa{00..06}.egr.duke.edu ; do host $A ; done
  ```

  - esa00.egr.duke.edu has address 10.148.54.3
  - esa01.egr.duke.edu has address 10.148.54.20
  - esa02.egr.duke.edu has address 10.148.54.27
  - esa03.egr.duke.edu has address 10.148.54.28
  - esa04.egr.duke.edu has address 10.148.54.29
  - esa05.egr.duke.edu has address 10.236.67.31
  - esa05.egr.duke.edu has address 10.148.54.30
  - esa06.egr.duke.edu has address 10.148.54.31

This stuff isn’t just for scripts – you can do it straight on the command line!
Conditionals: [ ], [[ ]], ( ( ) ), ( )

- **Conditionals**
  - **Commands**: Every command is a conditional based on its *exit status*
  - **Test conditionals**: Boolean syntax enclosed in *spaced-out* braces
    - `[ STR1 == STR2 ]`  
      String compare (may need to quote)
    - `[ -e FILE ]`  
      File exists
    - `[ -d FILE ]`  
      File exists and is a directory
    - `[ -x FILE ]`  
      File exists and is executable
    - `[ ! EXPR ]`  
      Negate condition described in EXPR
    - `[ EX1 -a EX2 ]`  
      AND the two expressions
    - `[ EX1 -o EX2 ]`  
      OR the two expressions
    - [See here for full list](#)
    - Double brackets get you newer bash-only tests like regular expressions:  
      ```bash
      [[ $VAR =~ ^https?:// ]]
      ```
      VAR starts off like an HTTP/HTTPS URL
    - Double parens get you arithmetic:  
      ```bash
      (( $VAR < 50 ))
      ```
      VAR is less than 50
    - Single parens get you a subshell (various sometimes-useful side effects)
What is a script?

• **Normal executable**: binary file in an OS-defined format (e.g. Linux “ELF” format) appropriate for loading machine code, marked with +x permission.

• **Script**: Specially formatted text file marked with +x permission. Starts with a “hashbang” or “shebang”, then the name of binary that can interpret it, e.g.:

  ```bash
  #!/bin/bash
  ```
  
  or
  
  ```bash
  #!/usr/bin/python
  ```

  ▪ On execution, OS runs given binary with script as an argument, then any given command-line arguments. No shebang? Defaults to running with bash.

  ▪ Example: “./myscript -a 5” is run as “bash ./myscript -a 5”.

  ▪ Can also just run a script with bash manually (e.g. “bash myscript”)

• **When should you write a bash script?**

  ▪ When the thing you’re doing is >80% shell commands with a bit of logic

  ▪ Need lots of logic, math, arrays, etc.? Python or similar is usually better.
Examples (1)

• Making an assignment kit for another of my classes:

$ echo `ls` > buildkit
$ cat buildkit

Autograder_rubric.docx Autograder_rubric.pdf byseven.s grading_tests homework2-grading.tgz
HoopStats.s HoopStats.s-cai hw2grade.py
HW2_GRADING_VERSION Makefile recurse.s

$ nano buildkit
$ cat buildkit

tar czf kit.tgz Autograder_rubric.pdf byseven.s grading_tests hw2grade.py HW2_GRADING_VERSION Makefile recurse.s

$ chmod +x buildkit
$ ./buildkit

$ ls -l kit.tgz
-rw-r--r-- 1 tkb13 tkb13 771264 Sep 14 18:14 kit.tgz
Examples (2)

- Script to run the ECE650 “hot potato” project for grading:

```bash
#!/bin/bash
./ringmaster 51015 40 100 |& tee out-14-rm.log &
./player `hostname` 51015 |& tee out-14-p00.log &
./player `hostname` 51015 |& tee out-14-p01.log &
./player `hostname` 51015 |& tee out-14-p02.log &
...
./player `hostname` 51015 |& tee out-14-p07.log &
./player `hostname` 51015 |& tee out-14-p08.log &
./player `hostname` 51015 |& tee out-14-p09.log &
wait
```

- Backticks to get external hostname
- Backgrounded
- Shorthand for “stdout and stderr together”
- Pause until all child processes have exited.
More common commands (1)

- **diff**: Compare two files
  - Example use: How does this config file differ from the known-good backup?
    
    ```
    $ diff config config-backup
    2d1
    < evil=true
    
    Second line, first column
    Left file ('<') has this extra line
    ```

- **md5sum/sha*sum**: Hash files
  - Example use: Hash all static files, compare hashes later (e.g. using diff)
    
    ```
    $ find /path -exec sha256sum '{}' ';' > SHA256SUM.orig
    ... (time passes) ...
    $ find /path -exec sha256sum '{}' ';' > SHA256SUM.now
    $ diff SHA256SUM.orig SHA256SUM.now
    ```

- **dd**: Do block IO with fine degree of control
  - Example use: Overwrite the first 1MB of a hard drive (destroys filesystem, but data is still intact – insecure but fast drive erasure)
    
    ```
    $ dd if=/dev/zero of=/dev/sda bs=1k count=1k
    ```
More common commands (2)

- **hd/hexdump/od**: Hex dump (comes in a few variants)
  - Example use: Examine a config file for non-printable or Unicode characters that may be triggering a parser bug.
    
    ```
    $ hd config1
    00000000  73 65 74 74 69 6e 67 31  3d 79 65 73  
    |setting1=yes..se|
    00000010  74 74 69 6e 67 32 3d 6f  0a                    |tting2=ok.|
    0000001a
    ```

- **strings**: Scan an otherwise binary file for printable strings
  - Example use: Quickly assess an unknown binary file for clues as to its nature
    
    ```
    $ strings setup.exe | less
    ```
    
    (scroll through lots of content quickly)

    ```
    <assemblyIdentity version="1.0.0.0" processorArchitecture="X86"
    name="DS.SolidWorks.setup" type="win32"></assemblyIdentity><description>This file
    will allow SolidWorks to take advantage of xp themes.</description>
    ```

    Conclusion: this is an installer for SolidWorks.
More common commands (3)

- **file**: Identify what kind of file you have by its format
  - Example use: Attacker pulled down an opaque file, what is it?
    
    ```
    $ file hax.dat
    hax.dat: gzip compressed data, last modified: Thu Aug 9 16:50:37 2018, from Unix
    $ gzip -cd hax.dat | file -
    /dev/stdin: PE32+ executable (console) x86-64, for MS Windows
    ```

    Conclusion: It’s a gzip’d Windows executable

- **wget/curl**: Fetch internet stuff via HTTP (and other protocols)
  - **wget** downloads to file by default, **curl** writes to stdout by default (but either can do the other with options)
  - Example use 1: Download a file
    
    ```
    $ wget http://150.2.3.5/attacker-kit.tgz
    ```

  - Example use 2: Hit a web API (the URL below tells you your external IP)
    
    ```
    $ curl http://dsss.be/ip/
    152.3.64.179
    vcm-292.vm.duke.edu
    ```
Examples (1)

- Quick SSH banner recon:

```bash
$ for H in `cat hostlist` ; do printf "%-30s" "$H" ; echo hi | nc $H 22 | head -n1 ; done
```

remote.eos.ncsu.edu   SSH-2.0-OpenSSH_7.4
x.dssss.be            SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.4
dssss.be              SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2.10
reliant.colab.duke.edu SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.4
davros.egr.duke.edu   SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.4
esa00.egr.duke.edu    SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.4
esa01.egr.duke.edu    SSH-2.0-OpenSSH_7.6p1 Ubuntu-4
storemaster.egr.duke.edu SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.4
```

It’s like echo, but it’s printf.
Examples (2)

• Download all the course notes (well, all linked PDFs):

```bash
$ wget -r -l1 -A pdf http://people.duke.edu/~tkb13/courses/ece560/
$ find
```

Default behavior prints everything below here in the directory tree – a quick way to check what we got.
Search a big directory tree for a file in old dBase format

• Using find’s -exec option:

```bash
$ find -exec file '{}' ';' | grep -i dbase
```

```
./server01-back/dat/cust20150501/dbase_03.dbf: FoxBase+/dBase III DBF, 14 records * 590,
update-date 05-7-13, at offset 1025 1st record "0507121   CMP   circular     12"
```

-exec will run a command for each file found, with {} as the filename, terminating the command with ;.

• Using xargs for efficiency (run fewer discrete processes):

```bash
$ find | xargs file | grep -i dbase
```

```
./server01-back/dat/cust20150501/dbase_03.dbf: FoxBase+/dBase III DBF, 14 records * 590,
update-date 05-7-13, at offset 1025 1st record "0507121   CMP   circular     12"
```

xargs takes files in stdin and runs the given command on many of them at a time

• Using xargs with null delimiters to deal with filenames with spaces:

```bash
$ find -print0 | xargs -0 file | grep -i dbase
```

```
./server01-back/dat/cust20150501/spacey filename.dbf: FoxBase+/dBase III DBF, 14 records * 590,
update-date 05-7-13, at offset 1025 1st record "0507121   CMP   circular     12"
```

Both find’s output and xargs’s input are set to null-delimited instead of whitespace delimited.
Advanced uses of SSH
Advanced SSH: Tunnels

- Secure Shell (SSH): We know it logs you into stuff and is encrypted. *It does WAY MORE.*

- **SSH tunnels:** Direct TCP traffic through the SSH connection
  - `ssh -L <bindport>:<farhost>:<farport> <server>`
    - **Local forward:** Opens port `bindport` on the local machine; any connection to it will *tunnel* through the SSH connection and cause `server` to connect to `farhost` on `farport`.
  - `ssh -R <bindport>:<nearhost>:<nearport> <server>`
    - **Remote forward:** Opens port `bindport` on `server`; any connection to it will *tunnel* back through the SSH connection and the local machine will connect to `nearhost` on `nearport`.
  - `ssh -D <bindport> <server>`
    - **Dynamic proxy:** Opens port `bindport` on the local machine. This port acts as a SOCKS proxy (a protocol allowing clients to open TCP connections to arbitrary hosts/ports); the proxy will exit on the server side. Browsers and other apps support SOCKS proxy protocol.
    - Easy way to punch into or out of a restricted network environment.
Advanced SSH: Tunnel examples

- Example **local forward**:
  - You want to connect to an app over the network, but it doesn’t support encryption and/or you don’t trust its security.
  - Solution:
    - Set app daemon to only listen on loopback connections (127.0.0.1) port 8888
    - SSH to server with local forward enabled:
      
```bash
ssh -L 8888:localhost:8888 myserver.com
```
    - Connect your client to localhost:8888 instead of myserver.com:8888. All traffic is tunneled through encryption; access requires SSH creds.

- Example **remote forward**:
  - You’re an attacker with SSH credentials to a machine behind a NAT. You have an exploit that lets you run a command on another machine behind the NAT.
  - Solution: SSH to a server you control with a reverse SSH forwarder:
    
```bash
ssh -R 2222:victim:22 hackerserver.com
```
    - Can then connect to hackerserver.com’s loopback port 2222 to get to victim.

- Example **dynamic proxy**: Turn it on. Set browser to use it. Surf via server.
  - Bypass censorship, do web-admin on a restricted network, tunnel through a NAT, etc.
Advanced SSH: Keys

• You’re used to using passwords to login. That’s...decent.

• Alternative: SSH supports public/private key pairs!
  ▪ Pro: Allows passwordless login (or you can protect the key with a passphrase)
  ▪ Pro: Key file is random and way longer than password (kills dictionary attack)
  ▪ Pro: Can distribute your public key to any server you want easy access to
  ▪ Con: Private key must be kept secure! It allows login!!
Advanced SSH: Key generation

• Create key pair:

$ ssh-keygen  
(can provide various options, but default are ok)

Generating public/private rsa key pair.
Enter file in which to save the key (/home/tkbletsc/.ssh/id_rsa): mykey
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in mykey.
Your public key has been saved in mykey.pub.
The key fingerprint is:
SHA256:kywUn3ny1+LH0nsOYND5+FY7qIaTS+Ta0bXVjGTGY3Y tkbletsc@FREEMAN
The key's randomart image is:
+---[RSA 2048]----+
 | . . . . .   | 
 | . . o + = E | 
 | . o . B . o o | 
 | . + + O    | 
 | . + = S =   | 
 | o o = O + . | 
 | +o. B =    | 
 | ++..o.+..  | 
 | . o+. o= . | 
+-----[SHA256]-----+
Advanced SSH: Key files

- **Examining the keys:**

  ```
  $ cat mykey
  -----BEGIN RSA PRIVATE KEY-----
  MIIEpAIBAAKCAQEAg6vZKqVSLfZoiXd6yEgu3ZdLO/gv8mBaepWvJbISE5YKQw63
  dBqnLAZc0rJcoqzHgwBjddWUyzDh7g7+MZYgf+n+xE+3QDchqdrktPxj96TMfWUZ
  tHltpY1UNdbIStAhMbGr/L6aKFs/Ouk5RhWw+GPA7N1diATD0SYibTqdG5+JQqGn
  ... /
  /4zTb3GDixFIY9+raaFZ1XLJKBzfhi3ED4ga3nqmeKK60CDTvx8QbA==
  -----END RSA PRIVATE KEY-----
  
  $ cat mykey.pub
  ssh-rsa
  AAAAB3NzaC1yc2EAAAADAQABAAABAQCrq9kqpVI79miJd3rISC7dl0s7+C/yYFp6la8l
  shJ71gpDDrd0GqcsBlzSllyirMeDAGN11ZTLMOHuDv4xliB/6fuJK0D4BCFbhD8Y2eGh
  TZ/l/g9uIwIv7merL+UQduCSKvqLo1X4JYsI5VSknKCjcLo71JoCOUazqmttkX2EBSGd
  3VYp97Eu3XC3rqDAa/FnUe3E4w8nHLk9mB6/qbyr tkbletsc@FREEMAN
  ```

Informational comment, defaults to `username@hostname`, could be anything.
Advanced SSH: Key usage

• Authorizing a key:
  ▪ Copy **only** mykey.pub to the remote machine you want to establish access to
  ▪ On remote machine, add it to ~/.ssh/authorized_keys:
    
    ```
    $ cat mykey.pub >> ~/.ssh/authorized_keys
    ```

• Using a key to login:
  ▪ Provide the identity file (private key) with -i:
    
    ```
    $ ssh -i mykey remotephost.com
    ```
  ▪ SSH will use ~/.ssh/id_rsa by default – can use this “default key” without extra options.

• Remember: *keep your private key safe!!!*
Advanced SSH: Commands

- Can give a command with `ssh` to only do that command (no interactive session). Stdin/stdout/stderr are tunneled appropriately!
  - Really works great with passwordless keys!

- Find out uptime of server quickly:
  ```
  $ ssh myserver uptime
  ```

- Reboot nodes in a cluster:
  ```
  $ for A in node{0..7} ; do ssh root@$A reboot ; done
  ```

- Back up remote physical disk image:
  ```
  $ ssh root@server bash -c "gzip -c < /dev/sda" > server.img.gz
Advanced SSH: SCP, SFTP, and Rsync

• Almost any SSH server is also a file server using the SFTP protocol. The `scp` command is one way to use this.
  - Copy a file to remote home directory:
    ```bash
    $ scp file1.txt username@myserver:
    ```
  - Copy a directory to remote server’s web root:
    ```bash
    $ scp -r dir1/ webadmin@myserver:/var/www/
    ```

• Can also use a tool called `rsync` to copy only changes to files
  - Here’s the script I use to update the course site:
    ```bash
    echo COLLECTING COURSE SITES
    rsync -a --delete-delay ./ECE560/website/ ./www/courses/ece560/
    rsync -a --delete-delay ./ECE250/website/ ./www/courses/ece250/
    rsync -va --progress --delete-delay --no-perms www/* tkb13@login.oit.duke.edu:public_html/
    ```
Understanding and controlling the terminal
Brief terminal history

- Original terminal: **the teletype machine**
  - Based on typewriter technology
  - It’s why we say “carriage return” and “line feed”
- Then came: **the serial terminal**
  - CRT display with basic logic to speak serial protocol
  - Many hooked up to one mainframe
  - Needed new *codes* to do new things like “clear screen” and “underline” without breaking compatibility
- Now we have: **terminal emulators** like **xterm**
  - Even more *codes* to do color, cursor movement, non-scrolling regions, etc.

- In Linux, the physical display is a “TTY” (teletype), e.g. `/dev/tty1`.
- Logical terminals like SSH sessions are “pseudoterminals”, e.g. `/dev/pts/0`
Terminal control sequences: Basic idea

• Some ASCII values are special:
  ▪ 0x0A = Linefeed (move cursor to next line), written as `\n`
  ▪ 0x0D = Carriage return (move cursor to left), written as `\r`
  ▪ 0x07 = Bell (beep), written as `\a`
  ▪ 0x1B = Escape – indicates a special multi-byte sequence, written as `\e`
    • MANY sequences exist. [Full documentation here](#).

• Example: Show a progress line without scrolling:
  ```bash
  $ for A in {00..99} ; do echo -ne "\r$A% done" ; sleep 0.1 ; done
  ```
  `-n means "no newline", -e means "allow escape characters".`

• Example: How does the ‘clear’ command work?
  ```bash
  $ clear | hd
  00000000 1b 5b 48 1b 5b 32 4a
  00000007 7 bytes
  $ 
  ```
Terminal control sequences: Color!

Terminal Escape Code Table 1.6 by Tyler Bletsch

Color control codes are of the form:
\e[<Code>m OR \e[<Code>;<Code>;m

where:
* \e is the escape character (ASCII 27, \033, \x1B)
* <Code> is a style or color code below, or none to indicate a reset.

For example:
\e[36;44mCyan on blue. \e[96mHi-cyan. \e[1mBold, \e[4munderline. \e[m Reset.
Yields:

Style codes: 0=Reset, 1=Bold, 2=Faint, 4=Underline, 7=Reverse, 9=Strikeout

Colors:

<table>
<thead>
<tr>
<th></th>
<th>40</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
<th>45</th>
<th>46</th>
<th>47</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>;1</td>
<td>30</td>
<td>90</td>
<td>;1</td>
<td>30</td>
<td>90</td>
<td>;1</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>91</td>
<td>;1</td>
<td>31</td>
<td>91</td>
<td>;1</td>
<td>31</td>
<td>91</td>
<td>;1</td>
</tr>
<tr>
<td>32</td>
<td>92</td>
<td>;1</td>
<td>32</td>
<td>92</td>
<td>;1</td>
<td>32</td>
<td>92</td>
<td>;1</td>
</tr>
<tr>
<td>33</td>
<td>93</td>
<td>;1</td>
<td>33</td>
<td>93</td>
<td>;1</td>
<td>33</td>
<td>93</td>
<td>;1</td>
</tr>
<tr>
<td>34</td>
<td>94</td>
<td>;1</td>
<td>34</td>
<td>94</td>
<td>;1</td>
<td>34</td>
<td>94</td>
<td>;1</td>
</tr>
<tr>
<td>35</td>
<td>95</td>
<td>;1</td>
<td>35</td>
<td>95</td>
<td>;1</td>
<td>35</td>
<td>95</td>
<td>;1</td>
</tr>
<tr>
<td>36</td>
<td>96</td>
<td>;1</td>
<td>36</td>
<td>96</td>
<td>;1</td>
<td>36</td>
<td>96</td>
<td>;1</td>
</tr>
<tr>
<td>37</td>
<td>97</td>
<td>;1</td>
<td>37</td>
<td>97</td>
<td>;1</td>
<td>37</td>
<td>97</td>
<td>;1</td>
</tr>
</tbody>
</table>

The xterm 256-color gamut:
\e[38;5;Nm=Foreground to code N. \e[48;5;Nm=Background to code N.

N in 0..15 Standard colors above:
N in 232..255 Grayscale:
N in 16..231 (N-16) is the RGB in base 6, 36r+6g+b+16 for r,g,b in 0..5:

\^ Re-run with the -e option for extended color list.
Why bother?

- Making output visually distinctive can *greatly* accelerate a task!
- Tester for ECE650 root kit: which would you rather use?
for testnum in {0..15} ; do
  if ./dotest $testnum ; then
    echo "test $testnum: ok"
  else
    echo -e \e[41mtest $testnum: FAIL!\e[m"
  fi
done
Also you can do cool crap
Scripting languages and regular expressions

Regular expression material is adapted from “Regular Expressions” in “Python for Informatics: Exploring Information” by Charles Severance at Univ. Michigan and “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Higher-level scripting languages

• Key languages categories commonly used:
  ▪ **Application**: Java, C#, maybe C++
  ▪ **Systems programming**: C, maybe Rust
  ▪ **Shell**: bash (or ksh, tcsh, etc.)
  ▪ **Scripting**: Python, Perl, or Ruby

• You *can* do everything in bash, but it gets ugly. Things bash is awkward at:
  ▪ Math
  ▪ Arrays
  ▪ Hash/dictionary data structures
  ▪ Really any data structures...

• Turn to *scripting languages*: dynamic, interpreted, compact
Scripting language key insight:
three fundamental types

- Most data manipulation tasks can be phrased as simple algorithms against these three types:
  - **Scalar**: simple value, numeric or string
  - **Array**: list of values (can nest)
  - **Hash/dictionary/map**: relationship between keys and values (can nest)

```perl
my %pairs = ( "hello" => 13,
              "world" => 31,
              "!"    => 71 );
foreach my $key ( keys %pairs ) {
    print "key = $key, value = $pairs{$key}\n";
}
```

```python
myDict = { "hello": 13,
           "world": 31,
           "!"    : 71 }
for key, value in myDict.items():
    print ("key = %s, value = %s" % (key, value))
```

```ruby
my_dict = { "hello" => 13,
            "world" => 31,
            "!"    => 71 }
my_dict.each { |key, value| puts "key = #{key}, value = #{value}"}
```
One-liners

- Scripting languages support one-liners (typed from shell as a single command).
- Perl is the king of one-liners.
  - `-e` to provide code
  - `-n` automatically wraps code in “for each line of input from stdin or files”
  - `-i` replaces the content of given files with stdout of program (can provide filename extension to back up original data to)
- Quick Perl intro
  - Scalar variables start with a dollar sign, e.g. `$var`
  - Most functions, if you don’t specify, affect a variable called `$_`
  - Reference an array element value with `$array[$i]`, whole array is `@array`
  - Reference a hash element value with `$hash{$k}`, whole hash is `%hash`
  - Variables you make reference to are automatically created if they don’t exist (including arrays and hashes)
  - One-line comments with `#`
Perl one-liner example

• Remove duplicate lines from a file while preserving original order

Long-winded Perl:

```perl
while (<>) {  # for each line
    if (!$hash{$_}) {
        print;
    }
    $hash{$_}=1;
}
```

Run it:

```bash
$ perl dedupe.pl in.txt
```

One-liner:

```bash
$ perl -ne 'if (!$h{$_}){print} $h{$_}=1;' in.txt
```

Crazy dense one-liner:

```bash
$ perl -ne '$h{$_}++||print;' in.txt
```
Manipulating text

• Task: extract the hostname part of a URL, e.g. http://google.com/images

• Thought process:
  ▪ Idea: Start at character 7, capture until you find a slash
  ▪ Problem: what about https?

  ▪ Idea: Go until you see two slashes in a row, then capture until you find a slash
  ▪ Problem: can have more than two slashes at start

  ▪ Idea: Go until you see two or more slashes in a row, then capture until you find a slash
  ▪ Problem: What about username specifier (user@) and port number (:80)?

  ▪ ugh nevermind just give up 😞

• Solution: We need a language to describe string processing!
THE LANGUAGE OF STRING PROCESSING

REGULAR EXPRESSIONS
Regular Expressions

- Regular expressions are expressive rules for walking a string
  - May capture parts of the string (parsing) or modify it (substitution)
  - Like a fancy find-and-replace
Understanding Regular Expressions

- Very powerful, cryptic, and fun
- Regular expressions are a language:
  - Based on "marker characters" - programming with characters

- The "gold standard" variant is from Perl:
  *Perl-Compatible Regular Expressions (PCRE)*

- Many languages support Perl-Compatible Regular Expressions:
  *Perl, grep* (with -P), *sed* (mostly), *Python, Ruby, Java*,
  *most text editors*, basically any language/tool worth using.

- Common among languages: Actual syntax inside regex
- Differs between languages: Syntax to call a regex, get feedback from it, provide options, etc.

- We’ll use both Perl and Python examples – easy to port to others

Adapted from “Regular Expressions” in “Python for Informatics: Exploring Information” by Charles Severance at Univ. Michigan
Introduction to Regular Expressions

• Basic syntax
  ▪ In Perl and sed, RegEx statements begin and end with `/`
    (This is language syntax, not the case for Python and others)
    • `/something/
  ▪ Escaping reserved characters is crucial
    • `/\(i.e. /` is invalid because `( must be closed
    • However, `/\(i\.e\). /` is valid for finding ‘(i.e.’
    • Reserved characters include:
      . * ? + ( ) [ ] { } / \ | 
      ▪ Also some characters have special meanings based on their position
        in the statement

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Regular Expression Matching

• Text Matching
  • A RegEx can match plain text
    • ex. if ($name =~ /Dan/) { print "match"; }  
  • But this will match Dan, Danny, McDaniel, etc...

• Full Text Matching with Anchors
  • Might want to match a whole line (or string)
    • ex. if ($name =~ /^Dan$/) { print "match"; }  
    • This will only match Dan  
    • ^ anchors to the front of the line  
    • $ anchors to the end of the line

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
In Python: The Regular Expression Module

- Before you can use regular expressions in your program, you must import the library using "import re"
- Use `re.search()` to see if a string matches a regex
- Use `re.findall()` to extract parts of a string that match your regex
- Use `re.sub()` to replace a regex match with another string
- Use `re.split()` to separate a string by a regex separator

- Example:
  ```python
  if re.search(r'Dan', name): print "match"
  ```

In Python, r-quotes mean “raw string”, i.e. “don’t interpret escapes in this string”, which makes it convenient to write Regexes which use all sorts of weird punctuation

Adapted from “Regular Expressions” in “Python for Informatics: Exploring Information” by Charles Severance at Univ. Michigan
General operation

• Engine searches string from the beginning
  ▪ Plain text is treated literally
  ▪ Special characters allow more flexible matching

• A regex is just a way to write a finite state machine (FSM)
  ▪ FSM proceeds through states as matching characters are encountered; if a full regex is walked, that’s a match.

• Every character matters!
  ▪ / s/ is not the same as / s/
Regular Expression Char Classes

- Allows specification of only certain allowable chars
  - \[dofZ\] matches only the letters d, o, f, and Z
    - If you have a string ‘dog’ then /\[dofZ\]/ would match ‘d’ only even though ‘o’ is also in the class
    - So this expression can be stated “match one of either d, o, f, or Z.”
  - \[A–Za–z\] matches any letter
  - \[a–fA–F0–9\] matches any hexadecimal character
  - \[^*$/\] matches anything BUT *, $, /, or \n    - The ^ in the front of the char class specifies ‘not’
    - In a char class, you only need to escape: \( \) – ^

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Regular Expression Char Classes

• Special character classes match specific characters
  
  • \d matches a single digit
  • \w matches a word character: [A-Za-z0-9_]
  • \b matches a word boundary,
    e.g. /\bword\b/ catches “my word!” but not “mr. wordly”
  • \s matches a whitespace character (space, tab, newline)
  • . wildcard matches everything but newlines (can make it include newlines)
    • Use very carefully, you could get anything!

• To match “anything but...” capitalize the char class
  • i.e. \D matches anything that isn’t a digit

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Character Class Examples

- `/e\w\w/`
  - Matches ear, eye, etc

- `$thing = '1, 2, 3 strikes!'; $thing =~ /\s\d/;`
  - Matches ‘2’

- `$thing = '1, 2, 3 strikes!'; $thing =~ /[\s\d]/;`
  - Matches ‘1’

- Not always useful to match single characters
  - `$phone =~ /\d\d\d-\d\d\d-\d\d\d\d/;`
  - There’s got to be a better way...

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Regular Expression Repetition

• Repetition allows for flexibility
  • Range of occurrences
    • $weight =~ /\d{2,3}/;
      • Matches any two- or three-digit integer
    • $name =~ /\w{5,}/;
      • Matches any name longer than 5 letters
    • if ($SSN =~ /\d{9}/) { print “valid SSN!”; }  
      • Matches exactly 9 digits

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Regular Expression Repetition

• General Quantifiers

  • Some more special characters

  * matches *zero or more* of the preceding, so `/\d*/` matches any size number *or no number at all*

  + matches one or more of the preceding, so `/\w+/` matches one or more characters

  ? causes the preceding to be optional (match 0 or 1 times), so `/\d?/` means “an optional digit”
Regular Expression Repetition

• Greedy vs Non-greedy matching
  • Greedy matching gets the longest results possible
  • Non-greedy matching gets the shortest possible
  • Let’s say $robot = ‘The12thRobotIs2ndInLine’
    • $robot =~ /\w*\d+/; (greedy)
      • Matches The12thRobotIs2
      • Maximizes the length of \w
    • $robot =~ /\w*?\d+/; (non-greedy)
      • Add a ‘?’ to a repetition to make it non-greedy!
      • Matches The12
      • Minimizes the length of \w

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
• Greedy vs Non-greedy matching

  • Suppose \( \$\text{txt} = 'something is so cool' \)
    
    • \( \$\text{txt} =~ /something/; \)
      
      • Matches ‘something’
    
    • \( \$\text{txt} =~ /so(mething)?/; \)
      
      • Matches ‘something’ and the second ‘so’
      
      • Parenthesis can be used for \textbf{grouping} (e.g. being modified by ‘?’) and \textbf{capture} (covered later)
• Using what you’ve learned so far, you can...

• Validate an email address (note: regex below is a little oversimplified)
  • $email =~ /^\[w\.\-]+@((\w+)\.)\*(\w+)$/  

• Determine if log entry includes an IPv4 address
  • /\d\{1,3\}\./\d\{1,3\}\./\d\{1,3\}\./\d\{1,3\}/

• Regular expressions can be hard to write and even harder to read

• Two techniques can help:
  ▪ Languages have various ‘verbose’ or ‘extended’ modes so that a regex can be multiple lines, include comments, etc.
  ▪ Can use an interactive regex editor such as http://regex101.com/

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
• The IP address example

**Regular Expression**

/\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/ig

**Explanation**

- `/\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/ig`
- `/\d{1,3}/` matches a digit (equal to `[0-9]`) 1 to 3 times, as many times as possible, giving back as needed (greedy)
- `\d{1,3}` matches a digit (equal to `[0-9]`) 1 to 3 times, as many times as possible, giving back as needed (greedy)
- matches the character `.` literally (case insensitive)
- `\d{1,3}` matches a digit (equal to `[0-9]`) 1 to 3 times, as many times as possible, giving back as needed (greedy)
- matches the character `.` literally (case insensitive)
- `\d{1,3}` matches a digit (equal to `[0-9]`) 1 to 3 times, as many times as possible, giving back as needed (greedy)
- matches the character `.` literally (case insensitive)
- `\d{1,3}` matches a digit (equal to `[0-9]`) 1 to 3 times, as many times as possible, giving back as needed (greedy)
- `Quantifier` — Matches between 1 and 3 times, as many times as possible, giving back as needed (greedy)
- **Global pattern flags**
  - `i` modifier: insensitive. Case insensitive match (ignores case of [a-zA-Z])
  - `g` modifier: global. All matches (don't return after first match)
Alternation

- Alternation allows multiple possibilities

  - Let $\text{story} = \text{‘He went to get his mother’}$
    
    $\text{story} =~ /^(He|She)\ b.*?\b(his|her)\ b.*?\ (mother|father|brother|sister|dog)/;$
    
    • Also matches ‘She punched her fat brother’

  - Make sure the grouping is correct!
    
    $\text{ans} =~ /^(true|false)$/$$
    
    • Matches only ‘true’ or ‘false’
    
    $\text{ans} =~ /^{true|false}$/ (same as /(^true|false$)/)
    
    • Matches ‘true never’ or ‘not really false’

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Grouping for Backreferences

• **Backreferences** (also known as **capture groups**)
  ▪ We want to know what the expression finally ended up matching
    • Parenthesis give you **backreferences** let you see what was matched
    • Can be used *after* the expression has evaluated or even *inside* the expression itself!
  • Handled differently in different languages
  • Numbered from left to right, starting at 1

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Grouping for Backreferences

• Perl backreferences
  • Used inside the expression
    • $txt =~ /\b(\w+)\s+\1\b/
    • Finds any duplicated word, must use \1 here (true in most languages)
  • Used after the expression
    • $class =~ /(\.+?)-(\d+)/
    • The first word between hyphens is stored in the Perl variable $1 (not \1) and the number goes in $2. (This part varies between languages)
    • print “I am in class $1, section $2”;

• Equivalent Python:
  import re
  cls = "ECE560-02"
  m = re.match(r'(\.+?)-(\d+)', cls)
  print "I'm in class "+m.group(1)+", section "+m.group(2)"

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Example: Email Headers

• Here are some email headers.
  
  Date: Sep 15, 2018, 5:15 PM
  X-Sieve: CMU Sieve 2.3
  X-DSPAM-Result: Innocent
  X-DSPAM-Confidence: 0.8475
  X-Content-Type-Message-Body: text/plain

• Let’s write a regex to just match just the X- ones:

```regex
/X- .*: .*/
```
Using Regex101.com to understand this

Adapted from “Regular Expressions” in “Python for Informatics: Exploring Information” by Charles Severance at Univ. Michigan
Example: Email Headers
Capturing name and value

- We still have these email headers
  
  Date: Sep 15, 2018, 5:15 PM
  X-Sieve: CMU Sieve 2.3
  X-DSPAM-Result: Innocent
  X-DSPAM-Confidence: 0.8475
  X-Content-Type-Message-Body: text/plain

- Let’s amend our regex to capture the NAME and VALUE.

```
/(X-.*) : (.*)/
```

Adapted from “Regular Expressions” in “Python for Informatics: Exploring Information” by Charles Severance at Univ. Michigan
What if we want to PARSE those headers?

• Parenthesis used for **capture** of part of a match
Refining a regex (1)

• What if our content includes some confusing non-headers mixed in?

Adapted from “Regular Expressions” in “Python for Informatics: Exploring Information” by Charles Severance at Univ. Michigan
Refining a regex (2)

- Make regex more specific so it *just* matches what we want:

  ^\(X-\S^*)\: (.*\)

  Must be start of line  Non-whitespace characters only

Adapted from “Regular Expressions” in “Python for Informatics: Exploring Information” by Charles Severance at Univ. Michigan
Grouping *without* Backreferences

- Sometimes you just need to make a group
  - If important groups must be backreferenced, disable backreferencing for any unimportant groups
  - `$sentence =~ /^(?:He|She) likes (.+)/;`
    - I don’t care if it’s a he or she
    - All I want to know is what he/she likes
    - Therefore I use `(?::)` to forgo the backreference
    - `$1` will contain that thing that he/she likes

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Matching Modes

- Matching has different functional modes
  - In Perl, these are specified as letters after the regex.
    - $name =~ /[a-z]+/i;
      - i turns off case sensitivity
    - $xml =~ /title="([\w \]*)".*keywords="([\w \]*)"/s;
      - s enables . to match newlines
    - $report =~ /^\s*Name: [\s\S]*?The End.\s*$/m;
      - m allows newlines between ^ and $
  - In Python, you pass an additional optional argument with named constants (either short like the above or with full names), e.g.:
    - re.search(r'[a-z]+', name, re.I) # or re.IGNORECASE

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Regular Expression Substitution

- **Substitutions** simplify complex data modification
  - First part is a regex of what to find, second part is text to replace it
  - Backreferences can be included in replacement
  - For sophisticated work, most languages let you give a callback function so that the replacement can be programmatically generated for each match

- **Perl replacement syntax**
  - `$phone =~ s/\D//;`
    - Removes the first non-digit character in a phone number
      (Leaving the replacement blank means “replace with nothing”, i.e. “delete”)
  - `$html =~ s/^\s*(\s*)/$1\t/;`
    - Adds a tab to a line of HTML using backreference `$1`

- **Python uses** `re.sub()`

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Substitutions Modes

- Substitutions have modes like matches (ignore case, multiline, etc.)
- Important one: Substitutions can be performed singly or **globally**
  - In Perl, use the `g` flag to force the expression to scan the entire string
    - `$phone =~ s/\D//g;`
    - Removes all non-digits in the phone number
  - In Python’s `re.sub()` function, specify a `count` parameter to limit replacements (e.g. `count=1` for traditional “first match only” behavior)

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Combining one-liners and regexes

• Remember this slide when I compared color output to plain?

```perl
perl -pe 's/\e.*?m//g' orig > plain
```

- `-p` means “read one line at a time like -n, but print the line afterwards”
- `\e` is the escape character.
Regular Expression Quick Guide

^ Matches the beginning of a line

$ Matches the end of the line

. Matches any character (except newline, unless you give an option)

\s Matches whitespace

\S Matches any non-whitespace character

\w Matches a “word-like” character (letters/numbers/underscore)

\d Matches a decimal digit (0-9)

\b Matches a word boundary

? Makes a character or group optional (appears zero or one times)

* Repeats a character or group zero or more times

*? Repeats a character zero or more times (non-greedy)

+ Repeats a character one or more times

+? Repeats a character one or more times (non-greedy)

| Alternation – allows either/or. Usually used with parens: (this) | (that)

[aeiou] Matches a single character in the listed set

[^XYZ] Matches a single character not in the listed set

[a–z0–9] The set of characters can include a range

( and ) Indicates a group (used to capture part of a match or group stuff for modifiers)

A more complete quick-ref guide is here and linked on the course site. See also the Python re module docs.

Adapted from “Regular Expressions” in “Python for Informatics: Exploring Information” by Charles Severance at Univ. Michigan
Where can I use regexes?

- Obviously, in Perl and Python
- Also: Javascript, Java, .NET, PHP, R, C/C++, PowerShell, Ruby
- Also: your text editor

Tons of shell tools:

- `grep` -P
- `sed` (just has `em`)
- `awk` (just has `em`)
- `less` (Press `/`)

- Everyone cool is using regexes! Don’t get left behind!!!!
References for learning more about regexs

- Regex editor, code generator, and community database of regexs
  - [regex101.com](http://regex101.com/)

- Tutorials for various programming languages
  - [www.regular-expressions.info](http://www.regular-expressions.info/)

- Python in-depth docs
  - [https://docs.python.org/3/library/re.html](https://docs.python.org/3/library/re.html)

- Perl in-depth docs
  - [https://perldoc.perl.org/perlre.html](https://perldoc.perl.org/perlre.html)

Adapted from “Regular Expressions” by Ian Paterson at Rochester Institute of Technology
Manipulating tabular data
Hey, how about Excel? That thing’s cool, right?

- Terminals are nice, but did you know: GUIs exist?
- Some tasks benefit from non-terminal interface
- Example: tabular data wants to be in a spreadsheet

- Let’s cover some quick tips on (ab)using Excel (or Google Sheets)
**Data in/out**

- File format for getting in/out of Excel: 
  
  *Comma-Separated Values (CSV)*
  
  - Trivial for simple data: bob,2,19
  - If you have commas in data, enclose in quotes: "Jimmy, PhD",4,50
  - If you have quotes, double them up: "This is a ""quote""",7,94
  - Save with “.csv” extension an Excel loads it right up
  - Can generate well enough with simple commands
  - Can use common libraries to do everything “right” (quoting, etc.); e.g. Python has a built-in `csv` module

- For fast stuff, can just use the **clipboard**
  
  - Often quick just to copy/paste instead of making actual files
  - Format for spreadsheet <-> plaintext via clipboard is **tab-separated**
  - For a single column of data, there’s no tabs – it’s just in lines!
Formulas

- Spreadsheet formulas are outside of our scope – if you aren’t familiar, you **need** to learn them

- One thing to add: you can do string manipulation as well as math
  - & is the concatenation operator
  - TEXT() can format numbers in arbitrary formats
Auto-filter

- Take a sheet, make sure it has headers, highlight your data, turn on auto filter → Bam! instant sort/filter controls.
  - Example: Requesting badge access for some students.
Auto-filter

• Take a sheet, make sure it has headers, highlight your data, turn on auto filter → Bam! instant sort/filter controls.
  ▪ Example: Requesting badge access for some students.
Auto-filter

• Take a sheet, make sure it has headers, highlight your data, turn on auto filter → Bam! **instant sort/filter controls.**
  ▪ Example: Requesting badge access for some students.

I paste this NetID list into my email to IT to get students badge access.
Putting it all together
Example data manipulation task:
Planning Homework 1

- **What I have**: The Homework 1 draft writeup
- **My goal**: Plan out point allocation for questions
- **What I want**: Table of question number, topic, and points
Planning Homework 1: Acquire source data

- Select all, copy
- In shell, run "`cat > q`" and paste (middle-click), then `Ctrl+D` for EOF
Planning Homework 1:
Develop regex

```
Planning Homework 1:
Develop regex

Oh, I must need a perl-level regex. Switch to -P

Looks good, let’s switch to perl to capture fields.

Mismatched quotes so shell waited for more input, Ctrl+C.

These last commands didn't match anything – I realize I forgot to turn on ‘read a file line by line’ mode (-n)
```
Planning Homework 1: Debug regex

So it's matching but no output? Oh, I forgot to capture the fields I wanted! Add capture parens.

Working, but I realize I want the topics too. Add those parens.

Forgot to include $3 in output.
Planning Homework 1: Clean output

Why copy from text editor instead of shell? Shell will render those tabs as spaces for clipboard purposes; editor preserves them.
Planning Homework 1: Check results

- Paste into excel
- Resize columns
- Compare a few rows against document to confirm (never forget to check that your work actually got what you think it did!)

- I can immediately see Q13 has an extra space so I fix that in the doc
- Can consider and assign points accordingly.

![Victory Image]
Example task: Organizing PPTs
Gathering info

• Reviewing ECE651 course content, need to organize slides
• Have syllabus and downloaded content, want to put in order
• Can’t fully automate: matching syllabus to filenames is fuzzy
• Excel can help:
  
  $ ls > x.csv

Excel sheet:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture.pptx</td>
<td>05 Architecture.pptx</td>
</tr>
<tr>
<td>Dependable Systems.pptx</td>
<td>13 Dependable Systems.pptx</td>
</tr>
<tr>
<td>ECE651_CloudComputing.pptx</td>
<td>14 ECE651_CloudComputing_Michael.pptx</td>
</tr>
<tr>
<td>Intros and Overview of Software Eng.pptx</td>
<td>01 Intros and Overview of Software Eng.pptx</td>
</tr>
<tr>
<td>jason_s dis. sys..pdf</td>
<td>10 jason_s dis. sys..pdf</td>
</tr>
<tr>
<td>PresentationSkills.pptx</td>
<td>11 PresentationSkills.pptx</td>
</tr>
<tr>
<td>Project Management.pptx</td>
<td>06 Project Management.pptx</td>
</tr>
<tr>
<td>Risk Management.pptx</td>
<td>07 Risk Management.pptx</td>
</tr>
<tr>
<td>Software Design.pptx</td>
<td>08a Software Design.pptx</td>
</tr>
<tr>
<td>Software Evolution.pptx</td>
<td>11 Software Evolution.pptx</td>
</tr>
<tr>
<td>Software Implementation.pptx</td>
<td>12 Software Implementation.pptx</td>
</tr>
<tr>
<td>Software Security.pptx</td>
<td>13 Software Security.pptx</td>
</tr>
</tbody>
</table>

Filenames appear here

Manually give them numbers from syllabus

Formula makes new filenames:

=IF(B1<>"",TEXT(B1,"00")&" ",""")&A1
Example task: Organizing PPTs Generating script and renaming

- We can even have Excel make our renaming shell script:

  Generate rename commands: 
  ```
  ="mv "&A1&" "&C1&"
  ```

  Paste to script in same dir and run; done!
Conclusion

- Time it took to do this: 3 minutes
- Time it would have taken to do this manually: 6 minutes?
  - Odds of a typo or transcription error: way higher than automated way
- Time it would take to do this if I were learning the skills for the first time: way more than 6 minutes.

- So is it worth it to learn to automate?
  - Yes: you learn once, benefit many times. 😊
  - One of the sources of developer or sysadmin productivity!
- Keep doing things the dumb manual way because it’s faster?
  - You never improve! 😞

**Conclusion: PRACTICE THIS STUFF!!**