Today

- Shell/Bash scripting

1

Review: Unix Commands

- What goes into a bash script?
- How do you write conditionals?
- How can you make a command execute only if another command succeeds? Only if another command fails?
- How do you write comments in bash?
- How do you set and use variables?
  - How do you make a variable an environment variable?
- How do we use parameters in a script? To a function?
- How do we substitute in a command?

2

Using commands in commands

- Examples from my scripts

```
java -cp mail.jar:email.jar grading.Email
  $password $email "$subject" "`cat $filename`"
```

```
jarfiles=`ls $TURNINDIR/$STUDENT/$LAB/*.jar`
for jarfile in $jarfiles
do
    echo "Jar file: $jarfile"
    numJavaFiles=`jar tf $jarfile | grep -c ".java"`
    if [ $numJavaFiles = 0 ]; then
       echo "No Java Files submitted by $STUDENT"
    fi
done
```

Jan 26, 2022 Sprenkle - CSCI397 3

3

Positional Parameters

- The arguments to a shell script
  - `$0, $1, $2, $3 …`
  - Parameter 0 is the name of the shell or the shell script
- The arguments to a shell *function*
- Arguments to the `set` built-in command
  - `set this is a test`
    - `$1=this, $2=is, $3=a, $4=test`
- Manipulated with `shift`
  - `shift 2`
    - `$1=a, $2=test`

Jan 26, 2022 Sprenkle - CSCI397 6

6

Example with Parameters

Script

```
#!/bin/sh

# Parameter 1: file
# Parameter 2: how_many_lines
head -$2 $1
```

Invocation:

```
$ bash toplines /usr/share/dict/words 3
A
A's
AMD
```

Special Parameters

Parameter	Meaning
`$#`	Number of positional parameters
`$-`	Options currently in effect
`$?`	Exit value of last executed command
`$$`	Process number of current process
`$!`	Process number of background process
`$*`	All arguments on command line from 1 on
"$@"	All arguments on command line Individually quoted "$1" "$2" …; good if parameters contain spaces

`params.sh`

MORE FILE COMMANDS

Other File-Related Commands

Command	Purpose
`file`	Determine file type
`basename`	Strip directory and suffix from file names
`dirname`	Strip non-directory suffix from file name
`wc`	Print number of newlines, words, and bytes in files `-l` : lines `-m` : chars `-w` : words

Try Out These Examples

- `echo $HISTFILE`
- `file $HISTFILE`
- `dirname $HISTFILE`
- `basename $HISTFILE`
- `wc $HISTFILE`
- `wc –l $HISTFILE`

Managing Disk Space

Command	Purpose	Options
du	estimate file space usage	-h human readable -s summarize
df	report filesystem disk space usage	-h human readable

Many more options…
See man page

Managing Disk Space

- du Estimate file space usage (disk usage)
  - -h human readable format (e.g., MB, GB rather than KB)
  - -s summarize results for a directory

```
sprenkles@lcomp-fs1:cs397$ du -s handouts/
32888   handouts/
sprenkles@lcomp-fs1:cs397$ du -sh handouts/
33M     handouts/
```

Managing Disk Space

- df File system disk usage
  - -h human readable format (e.g., MB, GB rather than KB)

```
sprenkles@43350-CSCI-ILAB:course397$ df -h
Filesystem                           Size  Used Avail Use% Mounted on
udev                                 7.7G     0  7.7G   0% /dev
/dev/nvme0n1p2                        96G   46G   46G  51% /
tmpfs                                1.6G  2.8M  1.6G   1% /run
…
lcomp-fs1:/csci                      2.0T   86G  1.8T   5% /csci
lcomp-fs1:/users/tkhan@ad.wlu.edu    2.0T   86G  1.8T   5% /home/tkhan@ad.wlu.edu
lcomp-fs1:/users/sprenkles@ad.wlu.edu  2.0T   86G  1.8T   5%
/home/sprenkles@ad.wlu.edu
```

BACK TO BASH

15

What does this script do?

```
ARGS=1
E_BADARGS=65

test $# -lt $ARGS && echo "Usage: `basename $0` <arg1>" && \
exit $E_BADARGS

echo "You are in `pwd`”
```

```
$ bash example.sh
Usage: example.sh <arg1>
$ echo $?
65
$ bash example.sh test
You are in
/csci/courses/cs397/handouts/bash
$ echo $?
0
```

16

for loops

```
for var in list
do
        command
done
```

- Examples:

```
sum=0
for var in "$@"
do
    sum=`expr $sum + $var`
done
echo "The sum is $sum"
```

sum_params.sh

```
for file in *.sh
do
        echo "We have $file"
done
```

for_file.sh
for_params.sh

Functions

- Functions are similar to scripts and other commands except:
  - They can produce side effects in the caller's script
  - Variables are shared between caller and callee
    - Everything is global
  - The positional parameters are saved and restored when invoking a function.

Function Syntax

```
function name {
     commands
}
```

or

```
name () {
        commands
}
```

- Local variables: positional parameters
  - $0 is the function's name

Function Example

- What is the expected output?

```
function function_B {
    echo Function B.
}

function function_A {
    echo $0: $1
    function_C "$1"
}

function function_D {
    echo Function D.
}
```

functions.sh
functions2.sh

```
function function_C () {
    echo "----------------"
    echo Function C: $1
    echo GLOBAL = $GLOBAL
    let GLOBAL=$GLOBAL+1
    echo "----------------"
}

GLOBAL=1

# FUNCTION CALLS
# Pass parameter to function A
function_A "Function A."
function_B
function_C "Function C."
function_D
```

Command Search Rules

- When bash encounters some command (without a specified path), it needs to figure out what to execute
- In order, bash looks for
  - Functions
  - Built-ins
  - PATH search

UNIX SECURITY

Fundamentals of Security

- UNIX systems have one or more users, identified with a number and name
- A set of users can form a ***group***. A user can be a member of multiple groups
  - A special user (id 0, name **`root`**) has complete control
  - Each user has a primary (default) group

See what groups you belong to…

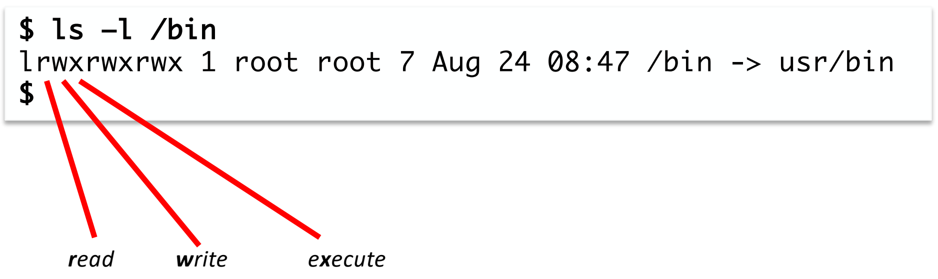
How are Users and Groups Used?

- Used to determine if file or process operations can be performed:
  - Can a given file be read? written to?
  - Can this program be run?
  - Can I use this piece of hardware?
  - Can I stop a particular process that's running?

File Permissions

- UNIX provides a way to protect files based on users and groups
- Three **types** of permissions:
  - **Read**: process may read contents of file
  - **Write**: process may write contents of file
  - **Execute**: process may execute file
- Three **sets** of permissions:
  - Permissions for **owner**
  - Permissions for **group** (1 group per file)
  - Permissions for **other**

A simple example



```
$ ls -l /bin
lrwxrwxrwx 1 root root 7 Aug 24 08:47 /bin -> usr/bin
$
```

Directory permissions

- Same types and sets of permissions as for files:
  - **read**: process may read the directory *contents* (i.e., list files)
  - **write**: process may add/remove files in the directory
  - **execute**: process may open files in directory or subdirectories

Unix Permissions

- Categories: owner, group, others
- Permissions: read, write, execute

```
sprenkle@fred:cs397$ ls -lrth
total 12K
drwxr-sr-x 20 sprenkles domain users 4.0K Jan 17 16:25 turnin
drwxrwsr-x  3 sprenkles domain users 4.0K Jan 26 11:02 shared
drwxr-sr-x  6 sprenkles domain users 4.0K Jan 26 11:32 handouts
```

permissions owner group size date modified file name

Unix Permissions

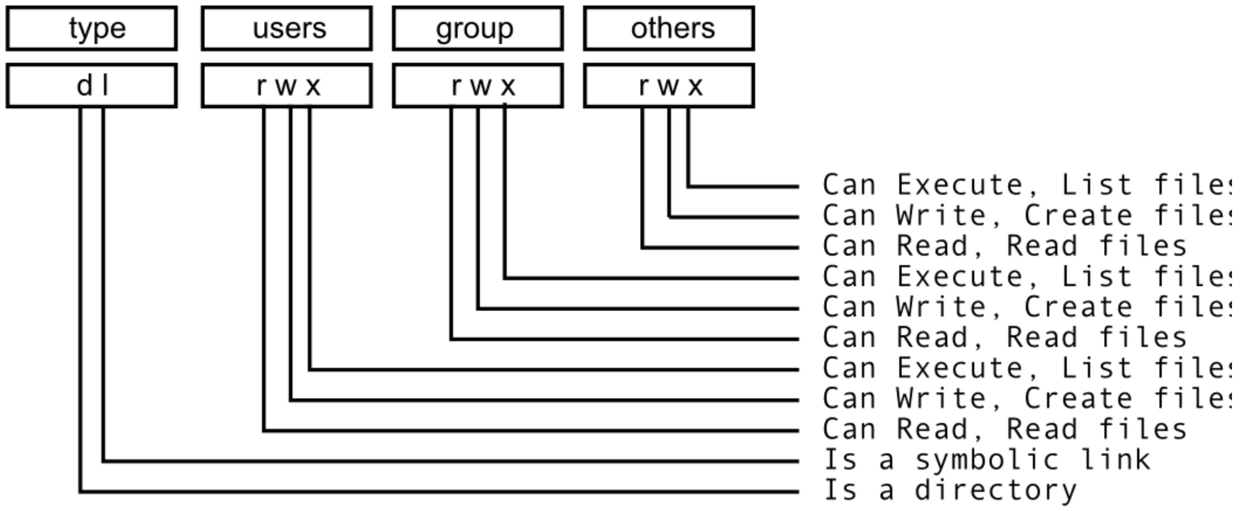
- Categories: owner, group, others
- Permissions: read, write, execute

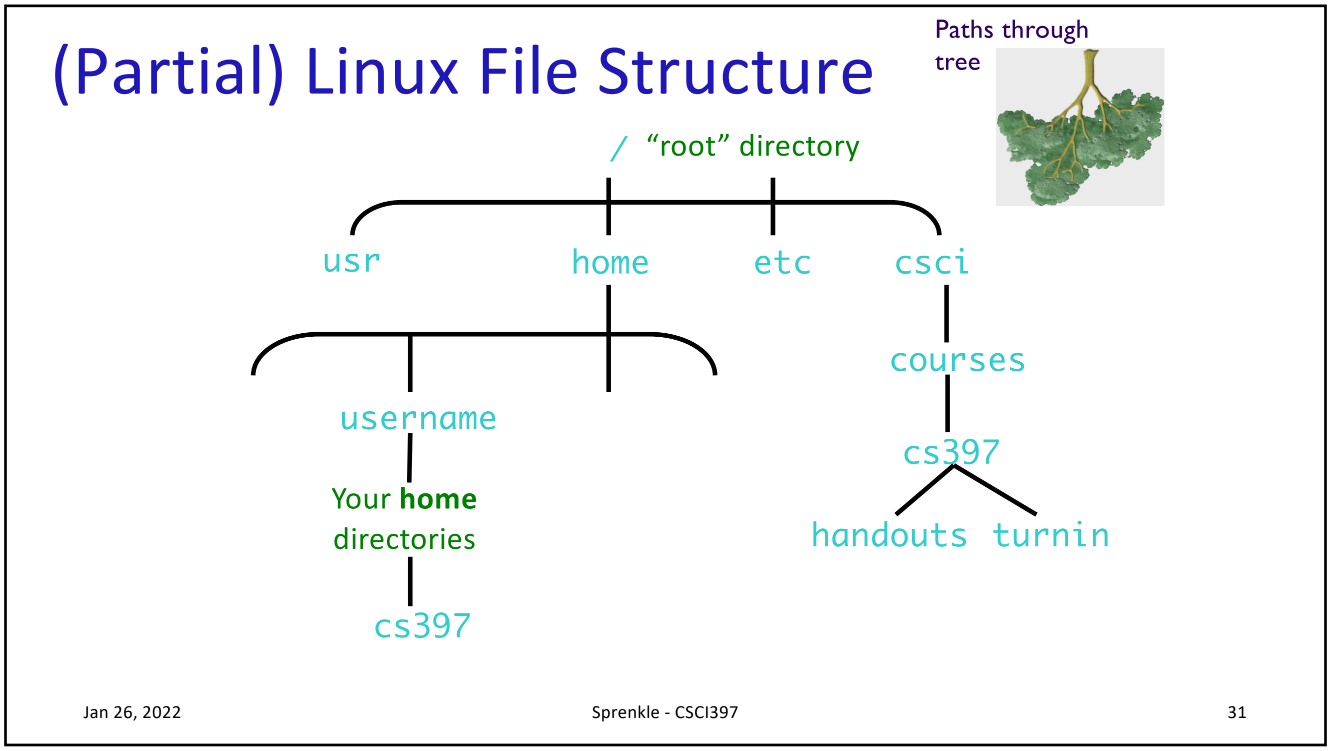
```
sprenkle@fred:cs397$ ls -lrth
total 12K
drwxr-sr-x 20 sprenkles domain users 4.0K Jan 17 16:25 turnin
drwxrwsr-x  3 sprenkles domain users 4.0K Jan 26 11:02 shared
drwxr-sr-x  6 sprenkles domain users 4.0K Jan 26 11:32 handouts
```

permissions owner group size date modified file name

- What are the permissions on files within `handouts`?
- In the permissions, how can we distinguish between an executable file and directory?
- What does it mean for a file to be executable?

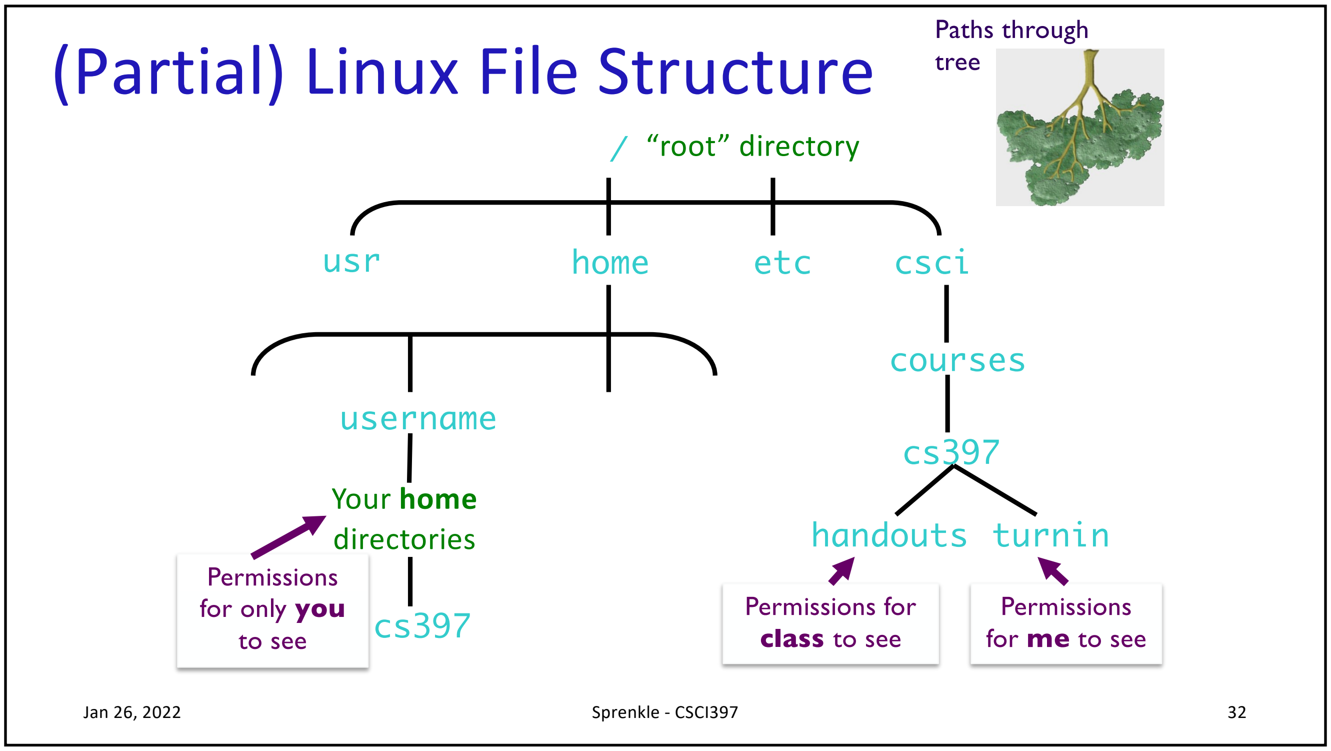
Permissions





(Partial) Linux File Structure

31



(Partial) Linux File Structure

Paths through tree
/ "root" directory
usr
home
etc
csci
username
Your **home** directories
cs397
Permissions for only **you** to see
courses
cs397
handouts
turnin
Permissions for **class** to see
Permissions for **me** to see

32

Utilities for Manipulating File Attributes

chmod	change file permissions
chown	change file owner
chgrp	change file group
umask	user file creation mode mask

- Only owner or super-user can change file attributes
- Upon creation, default permissions given to file modified by process's **umask** value

33

Changing Permissions

- **chmod** command
  - Syntax: `chmod [options] <mode> <file(s)>`
- Examples:

```
chmod u+x script.sh
chmod a-w readDir
chmod -R ug+r myDir
```

Recursive

Shorthand	Meaning
u	User/owner
g	Group
o	Others
a	All
r	Read permission
w	Write permission
x	eXecutable permission

34

`chmod` command

- Symbolic access modes {u,g,o} / {r,w,x}
  - example: `chmod +r` *`file`*
- Octal access modes
  - What's the pattern?

octal	read	write	execute
0	No	No	No
1	No	No	Yes
2	No	Yes	No
3	No	Yes	Yes
4	Yes	No	No
5	Yes	No	Yes
6	Yes	Yes	No
7	Yes	Yes	Yes

Changing Ownership, Group

- To change the owner of a file:
  - `chown <owner> <file(s)>`
  - `chown <owner:group> <file(s)>`
  - `-R` recursive option available
- To change the group of a file
  - `chgrp <group> <file(s)>`
  - `-R` recursive option available

REGULAR EXPRESSIONS

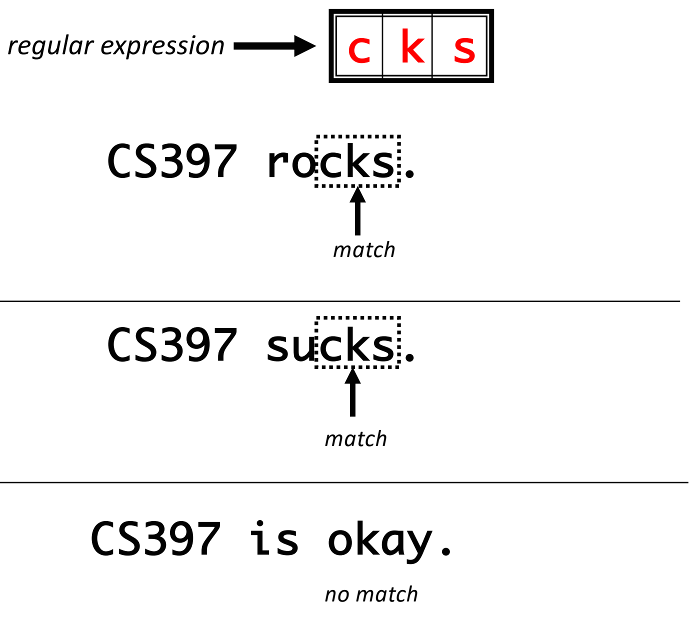
What Is a Regular Expression?

- A ***regular expression*** (***regex***) describes a set of possible input strings
- Regular expressions descend from a fundamental concept in Computer Science called *finite automata theory*
- Regular expressions are endemic to UNIX
  - vi, ed, sed, and emacs
  - awk, tcl, perl and Python
  - grep, egrep, fgrep
  - Compilers
- Search functionality → often can check a box for regular expressions

Regular Expressions

- The simplest regular expressions are a string of literal characters to match
- The string ***matches*** the regular expression if it contains the substring

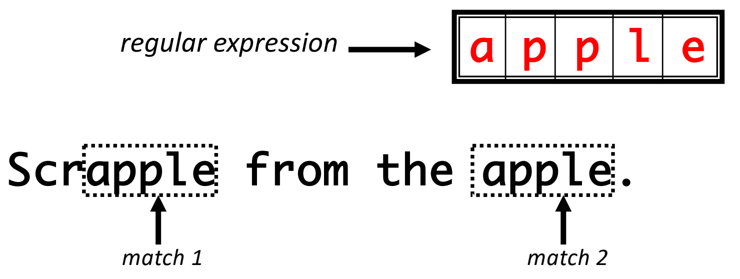
39



40

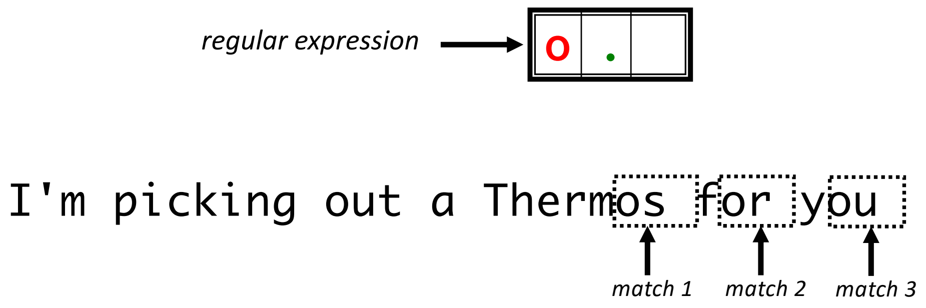
Regular Expressions

- A regular expression can match a string in more than one place



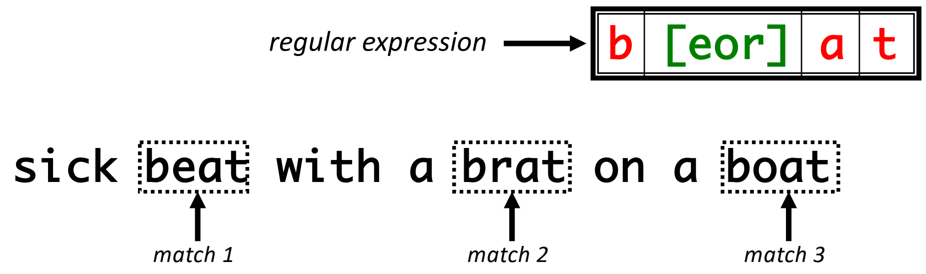
Regular Expressions

- The . regular expression can be used to match any character.



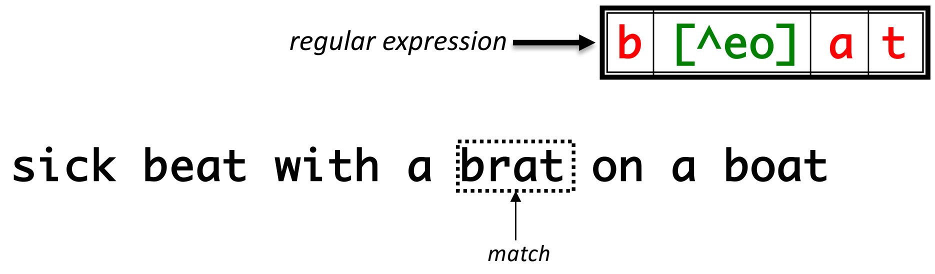
Character Classes

- Character classes [] can be used to match any specific set of characters.



Negated Character Classes

- Character classes can be negated with the [^] syntax.

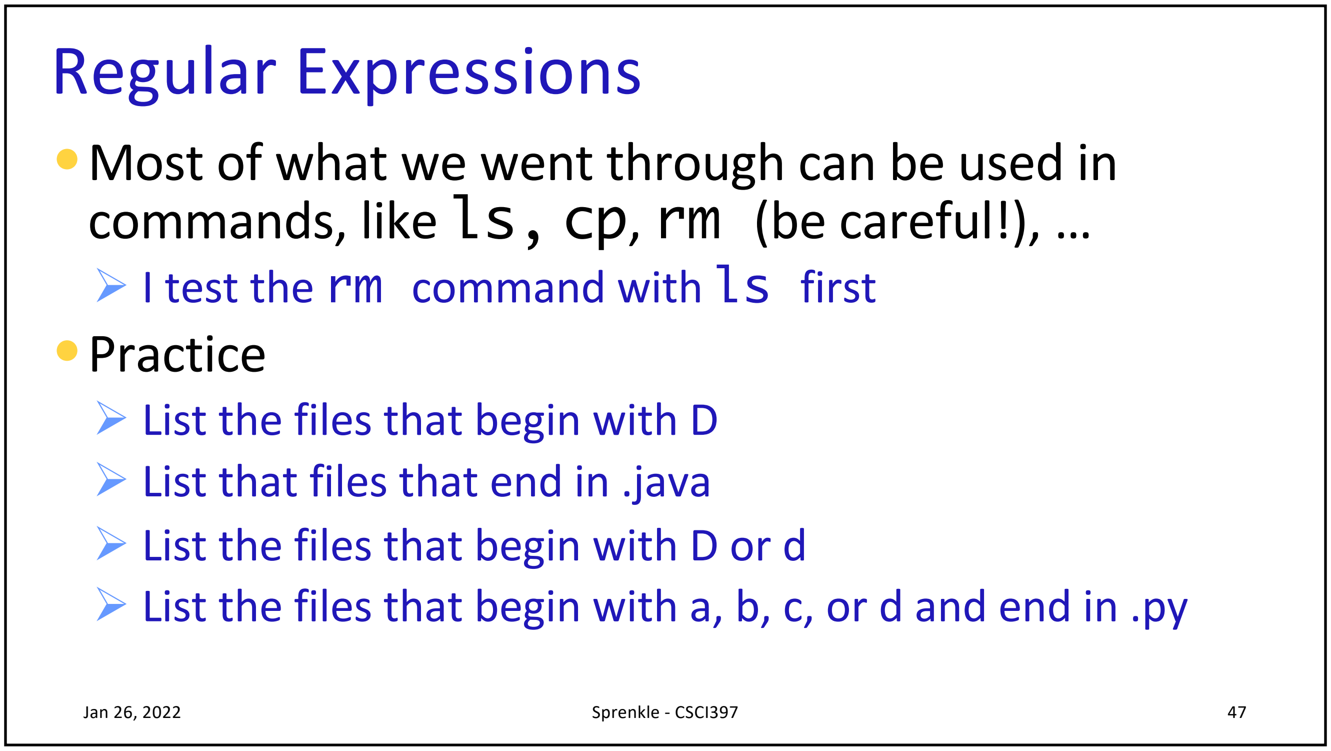


More About Character Classes

- `[aeiou]` will match any of the characters `a`, `e`, `i`, `o`, or `u`
- `[bB]ash` will match `bash` or `Bash`
- Ranges can be specified in character classes
  - `[1-9]` is the same as `[123456789]`
  - `[abcde]` is equivalent to `[a-e]`
  - You can also combine multiple ranges
    - `[abcde123456789]` is equivalent to `[a-e1-9]`
  - Note that the `-` character has a special meaning in a character class ***but only*** if it is used within a range,
    `[-123]` would match the characters `-`, `1`, `2`, or `3`

Named Character Classes

- Commonly used character classes can be referred to by name (*alpha*, *lower, upper, alnum*, *digit*, *punct*, *cntrl*)
- Syntax `[:`***name***`:]`
  - `[a-zA-Z]` → `[[:alpha:]]`
  - `[a-zA-Z0-9]` → `[[:alnum:]]`
  - `[45a-z]` → `[45[:lower:]]`
- Important for portability across languages



Regular Expressions

- Most of what we went through can be used in commands, like `ls`, `cp`, `rm` (be careful!), …
  - I test the `rm` command with `ls` first
- Practice
  - List the files that begin with D
  - List that files that end in .java
  - List the files that begin with D or d
  - List the files that begin with a, b, c, or d and end in .py

Jan 26, 2022 Sprenkle - CSCI397 47

47