Objectives

- Regular Expressions
- Combining Commands
    - backtics

Review

- How do you create an archive file?
    - How do you extract an archive file?
- How do you create a “shortcut” for a command?
    - Where can you create the shortcuts?
- What do we use to send the output from one command to the input of the other command?
- What command do we use to select different columns from a file?
- How can we merge several files in parallel?

Problems with `source` and `cd`

- Alias for `cd` is `ChangeDir`

From `/etc/bashrc`

```
function ChangeDir {
        if [ "$1" = "" ] ; then
          cd
        else
          cd "$1"
        fi
        pwd
}
```

- When execute `source`, seems to get a recursive definition
- Simplify: `alias cd=“cd; pwd”`

Follow Up

- Example of executing more than one command on the command-line:

```
sleep 5m; mplayer foo.mp3
```

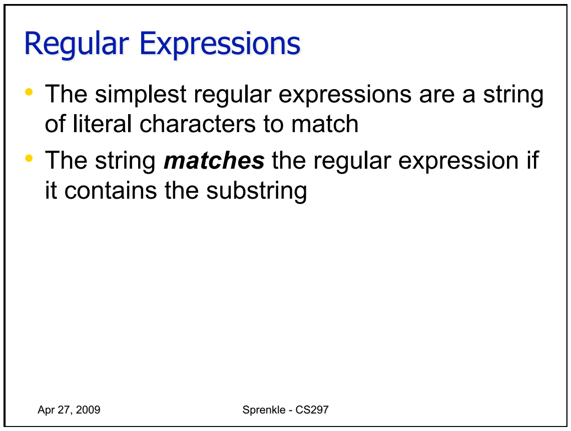
- In my research:

```
start_server
sleep 2m; execute_testcases
```

REGULAR EXPRESSIONS

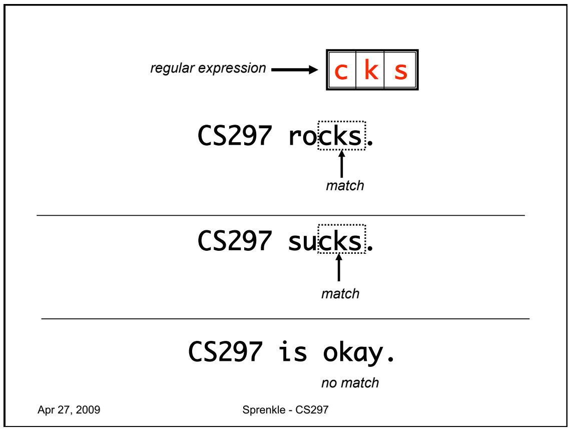
What Is a Regular Expression?

- 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



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



regular expression → c k s

CS297 rocks.

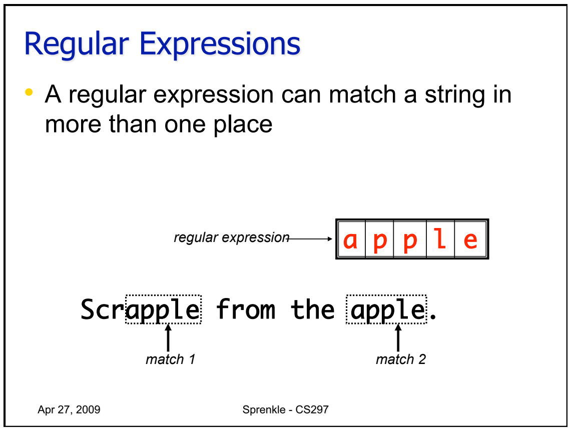
match

CS297 sucks.

match

CS297 is okay.

no match



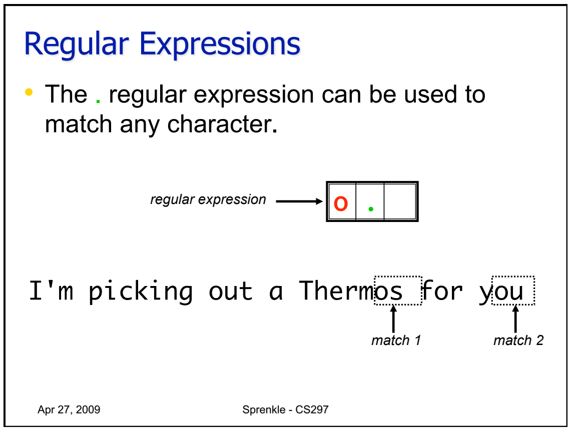
Regular Expressions

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

regular expression → a p p l e

Scrapple from the apple.

match 1 *match 2*



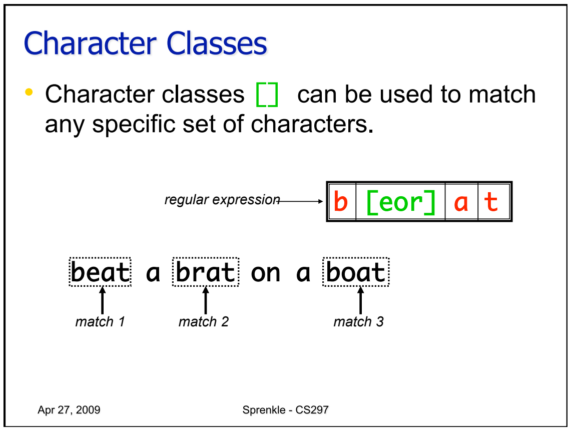
Regular Expressions

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

regular expression → o .

I'm picking out a Thermos for you

match 1 *match 2*



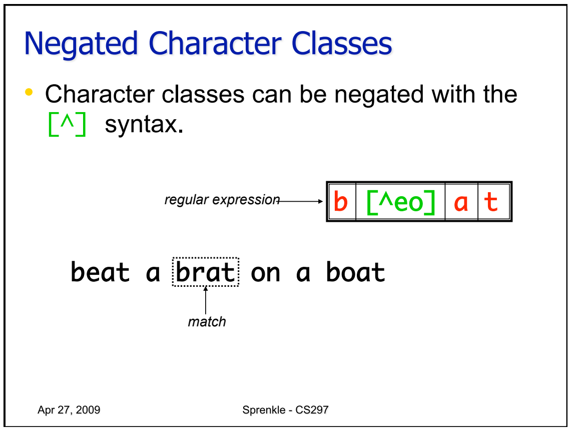
Character Classes

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

regular expression → b [eor] a t

beat a brat on a boat

match 1 *match 2* *match 3*



Negated Character Classes

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

regular expression → b [^eo] a t

beat a brat on a boat

match

More About Character Classes

  - `[aeiou]` will match any of the characters `a`, `e`, `i`, `o`, or `u`
  - `[kK]orn` will match `korn` or `Korn`
- 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

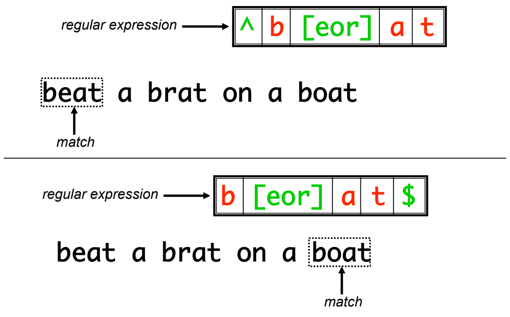
- Commonly used character classes can be referred to by name (*alpha*, *lower*, *upper*, *alnum*, *digit*, *punct*, *cntrl*)
- Syntax `[:`***`name`***`:]`
  - `[a-zA-Z]` → `[[:alpha:]]`
  - `[a-zA-Z0-9]` → `[[:alnum:]]`
  - `[45a-z]` → `[45[:lower:]]`
- Important for portability across languages

Regular Expressions

- Most of what we went through can be used in commands, like `rm` (be careful!), `mv`, `cp`, …
  - 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

Anchors

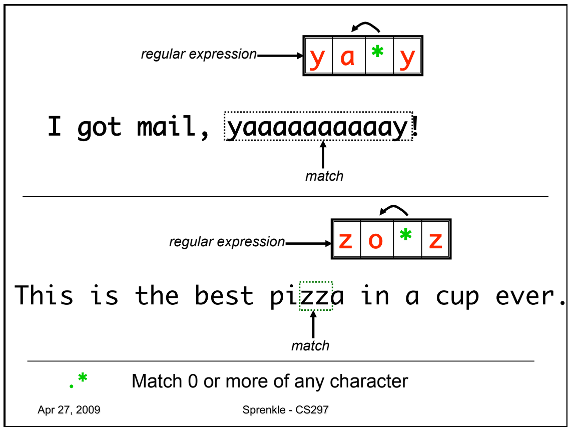
- Anchors are used to match at the beginning or end of a line (or both)
- `^` means beginning of the line
- `$` means end of the line



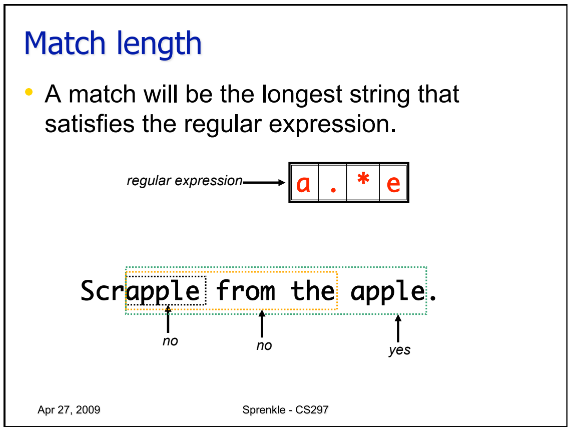
`^word$` `^$`

Repetition

- The `*` is used to define **zero or more** occurrences of the *single* regular expression preceding it.

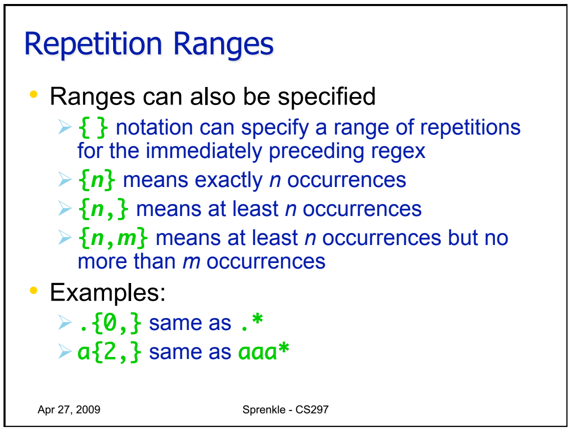

regular expression
y a * y
I got mail, yaaaaaaaaaay!
match
regular expression
z o * z
This is the best pizza in a cup ever.
match

`.*` Match 0 or more of any character



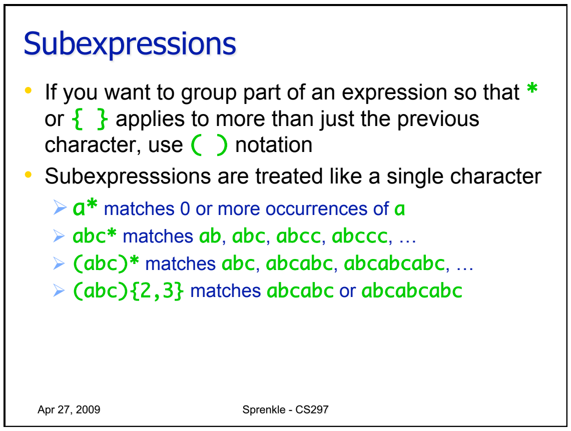
Match length

- A match will be the longest string that satisfies the regular expression.



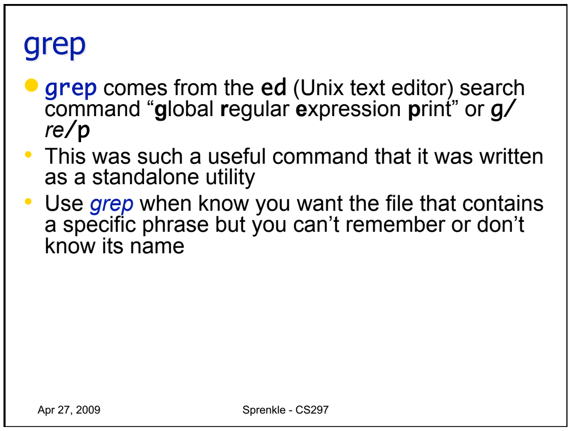
Repetition Ranges

- Ranges can also be specified
  - `{ }` notation can specify a range of repetitions for the immediately preceding regex
  - `{n}` means exactly *n* occurrences
  - `{n,}` means at least *n* occurrences
  - `{n,m}` means at least *n* occurrences but no more than *m* occurrences
- Examples:
  - `.{0,}` same as `.*`
  - `a{2,}` same as `aaa*`



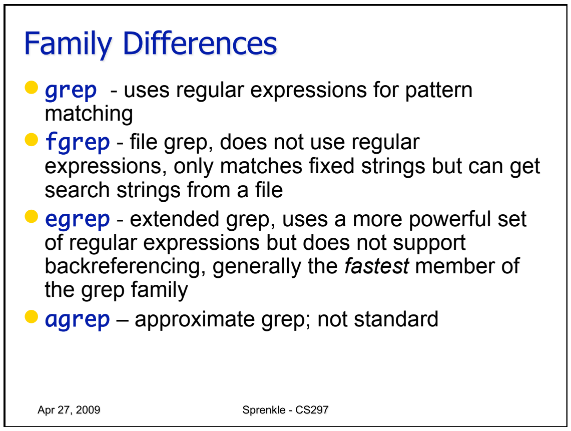
Subexpressions

- If you want to group part of an expression so that `*` or `{ }` applies to more than just the previous character, use `( )` notation
- Subexpresssions are treated like a single character
  - `a*` matches 0 or more occurrences of `a`
  - `abc*` matches `ab`, `abc`, `abcc`, `abccc`, …
  - `(abc)*` matches `abc`, `abcabc`, `abcabcabc`, …
  - `(abc){2,3}` matches `abcabc` or `abcabcabc`



grep

- **grep** comes from the **ed** (Unix text editor) search command "**g**lobal **r**egular **e**xpression **p**rint" or ***g/re/p***
- This was such a useful command that it was written as a standalone utility
- Use *grep* when know you want the file that contains a specific phrase but you can't remember or don't know its name



Family Differences

- **grep** - uses regular expressions for pattern matching
- **fgrep** - file grep, does not use regular expressions, only matches fixed strings but can get search strings from a file
- **egrep** - extended grep, uses a more powerful set of regular expressions but does not support backreferencing, generally the *fastest* member of the grep family
- ***agrep*** – approximate grep; not standard

Syntax

- Regular expression concepts we have seen so far are common to grep and egrep
- `grep` and `egrep` have slightly different syntax
  - `grep`: BREs
  - `egrep`: EREs (enhanced features we will discuss)
- Major syntax differences:
  - `grep`: \(and \), \{ and \}
  - `egrep`: (and), { and }

Protecting Regex Metacharacters

- Many special characters used in regexs also have special meaning to the shell

Single quote your regexs

- Protects special characters from being operated on by the shell
- If you habitually do it, you won't have to worry about when it is necessary

Escaping Special Characters

- To get literal characters, escape the character with a \ (backslash)
- Suppose we want to search for the character sequence `a*b*`
  - `a*b*` will match zero or more 'a's followed by zero or more 'b's (not what we want)
  - Use `a\*b\*`
    - Asterisks are now treated as regular characters

Egrep: Alternation

- Regex also provides an alternation character | for matching one or another subexpression
  - `(T|Fl)an` will match 'Tan' or 'Flan'
  - `^(From|Subject):` will match the From and Subject lines of a typical email message
    - It matches a beginning of line followed by either the characters 'From' or 'Subject' followed by a ':'
- Subexpressions are used to limit the scope of the alternation
  - `At(ten|nine)tion` then matches "Attention" or "Atninetion"
  - `Atten|ninetion` would match "Atten" or "ninetion"

Egrep: Repetition Shorthands

- `*` (star) specifies zero or more occurrences of the immediately preceding character
- `+` (plus) means "one or more"
  - `abc+d` will match 'abcd', 'abccd', or 'abccccccd' but will not match 'abd'
  - Equivalent to `{1,}`
- `?` (question mark) specifies an *optional* character
  - Single character that immediately precedes it
  - `July?` will match 'Jul' or 'July'
  - Equivalent to `{0,1}` and `(Jul|July)`

Egrep: Repetition Shorthands

- *, ?, and + are known as ***quantifiers*** because they specify the *quantity* of a match
- Quantifiers can also be used with subexpressions
  - `(a*c)+`

Egrep: Repetition Shorthands

- *, ?, and + are known as ***quantifiers*** because they specify the *quantity* of a match
- Quantifiers can also be used with subexpressions
  - `(a*c)+` matches ‘c’, ‘ac’, ‘aac’ or ‘aacaacac’ but will not match ‘a’ or a blank line

Practical Regex Examples

- Variable names in C/Python
- Dollar amount with optional cents
- Time of day
- HTML headers <h1> <H1> <h2> …

Practical Regex Examples

- Variable names in C/Python
  - `[a-zA-Z_][a-zA-Z_0-9]*`
- Dollar amount with optional cents
  - `\$[0-9]+(\.[0-9][0-9])?`
- Time of day
  - `(1[012]|[1-9]):[0-5][0-9] (am|pm)`
- HTML headers <h1> <H1> <h2> …
  - `<[hH][1-4]>`
  - New standard is lower case h

Grep: Backreferences

- ***Backreferences*** allow us to refer to a match that was made earlier in a regex
  - `\n` is the backreference specifier, where n is a number
  - Looks for nth subexpression
- Example: HTML Tags
  - `<h[1-6]>.*</h[1-6]>` is not good enough to match html headers, since it matches `<h1>Hello world</h3>`
  - `<h\([1-6]\).*</h\1>` matches what we were trying to match before.

Grep: Backreference Examples

- To find if the first word of a line is the same as the last:
  - `^\([[:alpha:]]\{1,\}\) .* \1$`
  - `\([[:alpha:]]\{1,\}\)` matches 1 or more letters
- Another example:
  - `"Mr \(dog\|cat\) came home to Mrs \1 and they went to visit Mr \(dog\|cat\) and Mrs \2 to discuss the meaning of life”`

What text should this match?

grep Family Syntax

grep [-hilnv] [-e expression] [filename]

egrep [-hilnv] [-e expression] [-f filename] [expression] [filename]

fgrep [-hilnxv] [-e string] [-f filename] [string] [filename]

Option	Meaning
`-h`	Do not display filenames
`-i`	Ignore case
`-l`	List only filenames containing matching lines
`-n`	Precede matching line with its line number
`-v`	Select non-matching lines
`-x`	Match whole line only
`-e expression`	Specify expression as option
`-f filename`	Take regular expression (egrep) or a list of strings (fgrep) from filename

grep Examples

- `grep 'men' GrepMe`
- `grep 'fo*' GrepMe`
- `egrep 'fo+' GrepMe`
- `egrep -n '[Tt]he' GrepMe`
- `fgrep 'The' GrepMe`
- `egrep 'NC+[0-9]*A?' GrepMe`
- `fgrep -f expfile GrepMe`

- Find all lines with signed numbers

```
$ egrep '[-+][0-9]+\.?[0-9]*' *.c
bsearch.c: return -1;
compile.c: strchr("+1-2*3", t-> op)[1] - '0', dst,
convert.c: Print integers in a given base 2-16 (default 10)
convert.c: sscanf( argv[ i+1], "% d", &base);
strcmp.c: return -1;
strcmp.c: return +1;
```

- **egrep** has its limits: For example, it cannot match all lines that contain a number divisible by 7.

Fun with the Dictionary

- `/usr/share/dict/words` contains over 400,000 words
  - `egrep hh /usr/share/dict/words`
    - aarrghh
    - Ahhiyawa
    - archhead
    - archheart

    …
- **egrep** as a simple spelling checker: Specify plausible alternatives you know

```
egrep "n(ie|ei)ther" /usr/share/dict/words
Neither
```

- How many words have 3 a's one letter apart? 3 u's?

Fun with the Dictionary

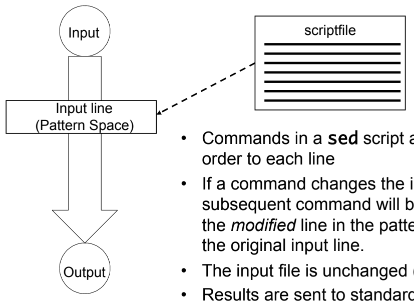
- How many words have 3 a's one letter apart?
  - `egrep a.a.a /usr/dict/words | wc -l`
    - 1632
- How many words have 3 u's one letter apart?
  - `egrep u.u.u /usr/dict/words | wc -l`
    - 84

SED

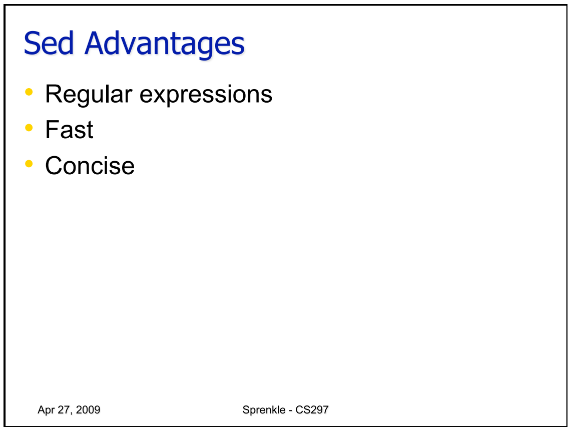
Sed: Stream-oriented, Non-Interactive, Text Editor

- Look for patterns one line at a time, like **grep**
- *Change* lines of the file
- Non-interactive text editor
  - Editing commands come in as *script*
  - There is an interactive editor *ed* which accepts the same commands
- A Unix filter
  - Superset of previously mentioned tools

sed Architecture

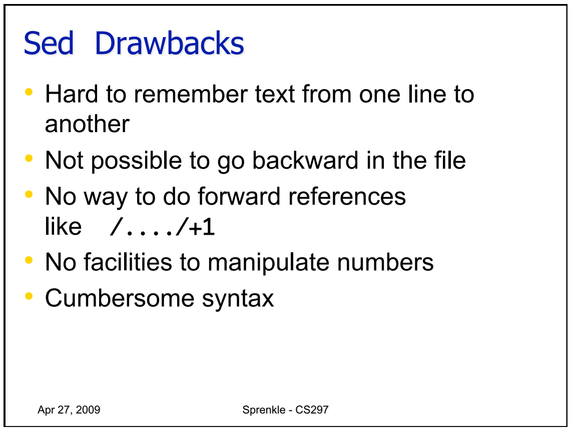


- Commands in a **sed** script are applied in order to each line
- If a command changes the input, subsequent command will be applied to the *modified* line in the pattern space, not the original input line.
- The input file is unchanged (sed is a filter).
- Results are sent to standard output unless redirected.



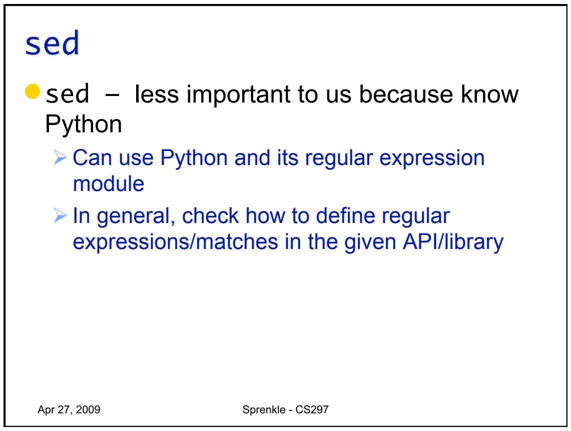
Sed Advantages

- Regular expressions
- Fast
- Concise



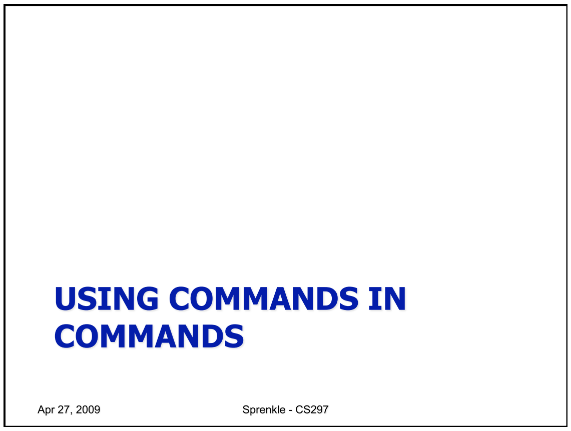
Sed Drawbacks

- Hard to remember text from one line to another
- Not possible to go backward in the file
- No way to do forward references like `/..../+1`
- No facilities to manipulate numbers
- Cumbersome syntax

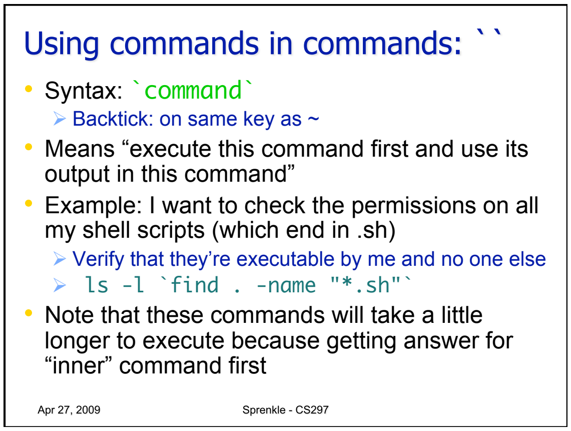


sed

- `sed` – less important to us because know Python
  - Can use Python and its regular expression module
  - In general, check how to define regular expressions/matches in the given API/library

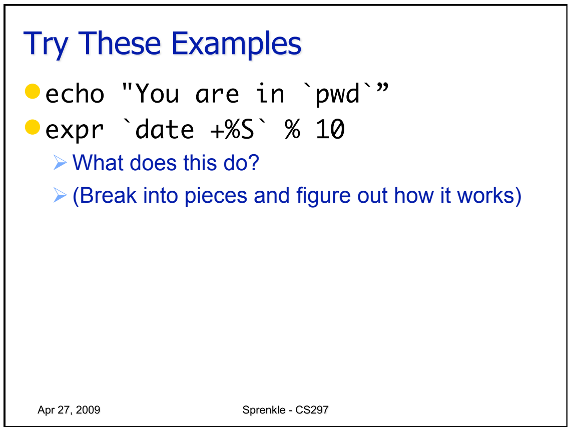


USING COMMANDS IN COMMANDS



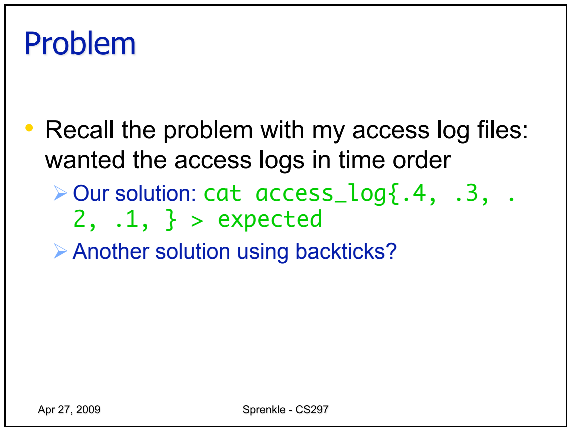
Using commands in commands: ``

- Syntax: `` `command` ``
  - Backtick: on same key as ~
- Means “execute this command first and use its output in this command”
- Example: I want to check the permissions on all my shell scripts (which end in .sh)
  - Verify that they’re executable by me and no one else
  - ``ls -l `find . -name "*.sh"` ``
- Note that these commands will take a little longer to execute because getting answer for “inner” command first



Try These Examples

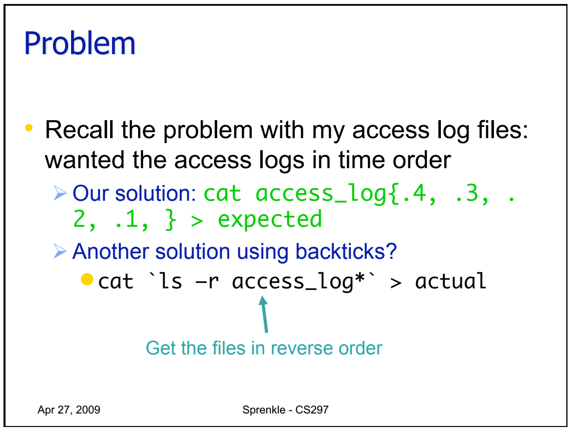
- ``echo "You are in `pwd`”``
- ``expr `date +%S` % 10``
  - What does this do?
  - (Break into pieces and figure out how it works)



Problem

- Recall the problem with my access log files: wanted the access logs in time order
  - Our solution: `cat access_log{.4, .3, .2, .1, } > expected`
  - Another solution using backticks?

Apr 27, 2009 Sprenkle - CS297

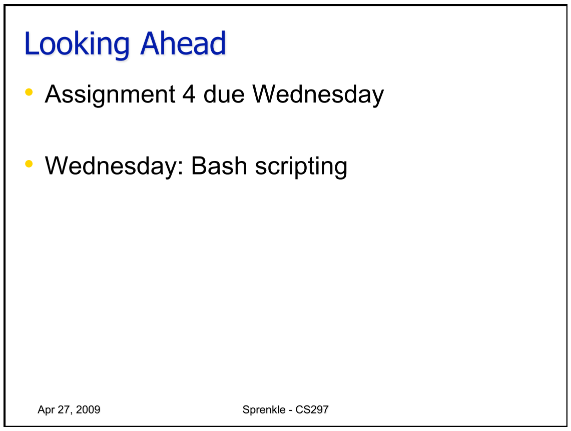


Problem

- Recall the problem with my access log files: wanted the access logs in time order
  - Our solution: `cat access_log{.4, .3, .2, .1, } > expected`
  - Another solution using backticks?
    - `` cat `ls -r access_log*` > actual ``

Get the files in reverse order

Apr 27, 2009 Sprenkle - CS297



Looking Ahead

- Assignment 4 due Wednesday
- Wednesday: Bash scripting

Apr 27, 2009 Sprenkle - CS297