```
CS 97 — MIDTERM REVIEW
======================


WEEK 0
------

UCLA CS 97 lecture - 2020-10-01

looks like people can't hear me; will look into this
switching to phone audio

switching to my laptop

Please use chat for questions

Goals of the course "Software Construction"
  Subset of software engineering (CS 130)

  programming  )   CS 31, 32
  data design  )     basic algorithms, data structures, C++
  integration (gluing together software components)
  configuration (how to start up a program; look at gcc's options)
  testing
  versioning (software mutates - how do you control that?) (a DAG)
  forensics (being a detective when things go wrong)

  Aside: Why is C++ a terrible language for this course?
    we want to be out of our comfort zones,
      to be good at learning new software technologies efficiently
    it's good at low-level stuff (pointers, basic classes)
    it's not so good at integration
    changing one module requires rebuilding "everything"
    Plus, you get all the problems of C:
      core dumps, crashes, unreliability, flaky
    We'll use C for low level stuff,
      something better for higher-level stuff

organization of course

experimental substitute for 35L
Make sure you take it for 4 units.
We tried it in spring quarter.
  Good parts: group-project-based approach won plaudits.
  Bad parts: not enough prep for CS 111 etc., lectures were too late for hw.

  So, we'll spend some more time on the basics, do it earlier
          (or make hw later).
      We'll spend a bit less time on the project infrastructure.

Introduce self.
  Paul Eggert
    background: 1/2 academia, 1/2 industry
      3 year asst prof UCSB
      3 year startup (15 people) software contruction (Prolog + DBs + NLP)
```

```
       3 years #2 computer company in world (at time) Unisys
     ~12 years startup (5 - 40 people) software network consulting (C++ + etc.)
          lucky enough to be colleague of Jun Hamano, Git chief
      18 years UCLA (working on various GNU apps: grep (just this week!))
           (hobby time zones coordinator: IANA time zone database)

   Three TAs.

Q. For reports, can you use markdown?
A. If it looks good, it's OK. Output is browser/viewer dependent,
   so do a portability check.

Q. Advice for success in this class? How to keep up to speed?
A. Come to lectures and discussions* - at least view the recordings.
     *Discussion real-time is more important than lectures.
   We're looking for a steep learning curve.
   Try stuff out - don't just do homework, try other (small) things
     to see how they work.  Try to break things.

   E.g., in Emacs, 'C-u X' means do X 4 times.
    YYYY
          'C-u N X' (where N is a number) means do X N times.
    ZZZZZZZZZZZZZZZZZZZZZZZ
      I did it 50,000 times here.
    AAAAAA...
   How to repeat a digit? C-u 50 C-u 0
   00000000000000000000000000000000000000000000000000


When is midterm?
  Tentatively, Thursday of 5th week, November 5. (will double-check)
  It's done during lecture.
  It's a bit less than 2 hours.


Why Emacs and not Vim?
  They're both great (others, too, Sublime, etc.)
  I was in the Vim camp (Vi, actually).
  Emacs was more programmable (a programmable development tool).

I want exams to be learning experience
  38 hours of lectures + 5 hours of exams
  answers are not obvious
  I don't always know the answer.
  We will give out samples (not exact matches for material;
                              schedule this later).
Is there a curve?
  not in traditional sense
  just add up raw scores for each thing
  division between grades done by intuition
  P/NP?

Group project will be 3-10 weeks.
It's a web app (full-stack - client-server).

I'm doing this lecture on Ubuntu 20.04.
```

Ubuntu is a commercial (free $, free license) distribution of GNU/Linux
  Debian distribution
Debian distributes a set of thousands of software packages based
on this:
  l3: applications - some GNU, many not
  l2: GNU C library
root: Linux kernel
under: x86-64 hardware (or ARM, etc.)

Let's recap.

Connect to Seasnet via SSH.

Start up Emacs by typing 'emacs' to the shell.

If you're SSH-ing in normally, Emacs will take over your text window;
it won't create a window of its own. This is often better for dicey
network connections.

But, you can invoke 'ssh -X penguin', then Emacs will start a new GUI
window of its own.

C-x C-b
C-x o To switch to other buffer

Emacs can edit your filesystem, via Dired, just as it can edit text in
ordinary buffers.

kernel - See CS 111 - low level of your OS, manages all its resources,
  including files, CPUs, time

a buffer in Emacs is a bunch of text in memory

WEEK 1

UCLA CS 97 lecture - 2020-10-06

For some reason Zoom seems to be putting some people into a waiting
room even though I have disabled the waiting room. I'll check for
this just before class starts and admit any people stuck in the
nonexistent waiting room.

Outline for today:
  * Emacs intro
  * Command-line basics and the shell
  * Unix file system organization
  * Combine all the above.

We need our overall applications to:
  * survive power outages (i.e., be persistent)
  * be fast (we can't make *everything* persistent)
  * be understandable (easy to read, write, maintain)

```
Persistent is like nonvolatile but
  “volatile” has a different meaning in C, C++ than it does when
  you’re talking about nonvolatile storage (we’ll get to that later)

We’ll use Emacs as a window into this situation.  It’s just another
program - *you* could have written it, if you had the time, or you
could write a substitute.

  Aside: why Emacs?
    - I’m biased - I contribute to Emacs.
    - Emacs gives us two important things:
       . scriptability via a clean language
       . introspection - Emacs is “self-aware”
  If I type “C-j”, what will happen - it inserts a newline
   C-X stands for “control X” - This is an actual ASCII character
      ('x' & 0x1f) == control-X
   M-x stands for “meta X” - this is a combination of keystrokes.
     Alt-X on my keyboard - depends on your keyboard.
   RET - RETURN or ENTER key

  You can find out what a key K will do by typing
    C-h k K

  C-x o - switch to other buffer
  C-x 1 - look at just this buffer (it’s fast to type on this!)
  C-x 2 - split buffer in half
  C-x 3 - split buffer in half vertically
  C-x 5 - create a new window (Zoom hates this)
  C-b NAME - switch this frame to buffer NAME
  C-x C-b - put buffer listing into a new buffer, and display as other buffer
  C-x C-s - save the current buffer into the corresponding file
       Emacs saves a backup file FOO~ if you edit a file FOO.
  C-x d RET - create a buffer containing a list of what’s in current directory
      every directory has two entries:
        . - current directory
    .. - parent directory
    In a directory listing, typing ‘g’ means refresh it from the file system.
  M-q - reformat the current paragraph

  buffer - bunch of text in Emacs’s memory (it’s fast, not persistent)

Emacs is a *modeful* editor - the action it takes when type a
character depends on the mode that Emacs is in.  If you type ‘g’ in
directory listing, it means revert-buffer (that is, sync from file
system); if in a text buffer, it means insert the letter “g”.

Some users hate multiple windows, and work just like in this lecture,
one big Emacs controlling the screen.

For now, simple version:
  There are only two types of files (actually there’s more, but let’s pretend).
    1. Regular files are sequences of bytes.
    2. Directories are mappings from file names to files.

A buffer is a sequence of bytes (or characters).
To edit a file, Emacs creates a buffer, copies file’s contents into buffer,
```

```
  and then you go to town, editing the buffer.
When you C-x C-s, Emacs copies buffer contents back into the file!

M-! COMMAND RET - runs the shell command COMMAND
M-x shell RET - starts up a shell inside Emacs

  shell prompt (output by shell) is "$ " traditionally,
  my prompt is command number followed by the host name.
  I'm doing this lecture on my desktop, named 'day'.




Some shell commands -

  - echo FOO : write "FOO" to output
  - ls : output names of files in current directory
    ls -l : likewise, but give metainformation about each file
    ls -a : like ls, but also output names of files beginning with ".";
               normally these files are suppressed

When you create a file, its owner is you (the creator), and its group
  is typically inherited from the group of the parent directory;
  (this isn't always true but is good enough for now)

Sample ls -l output

-rw-rw-r-- 1 eggert eggert 3190 Oct  6 16:45  notes.txt
   -rw-rw-r-- mode (1st character is file type, '-' regular file, 'd' directory)
                  next 3 characters are permission for owner of file
            r readable
            w writeable
            x executable (for regular file) or searchable (for dir)
          next 3 are permissions for others in the group
          last 3 are permissions for everyone else
    1 - link count (number of directory entries that point to this file)
    eggert - owner
    eggert - group
    3190 - # of bytes in the file
    Oct 6 16:45 - last-modified time
    notes.txt - file name within its parent directory
       This name is quoted if it's problematic to the shell.

    Every directory has two entries '.' (self) / '..' (parent).

Emacs convention for file names:
   FOO~ is a backup file for FOO.
   #FOO# is a last-saved version of FOO while you're editing it.
     Emacs tries to get around the problem that buffers are not persistent
       by saving the contents of a modified buffere into #FOO#.
   .#FOO is a symbolic link to nowhere (it's a string saying
       who's in the middle of editing the file now).

[break until 17:09]

Questions in Chat:
```

What is bin/zsh? It should have been "/bin/sh", a common name for the program that implements the shell command language. Type "ls -l /bin/sh" to see what it is.

what's the diff between p1, p2 and p3
They were three different processes. P1 was a C++ student program. P2 and P3 were both 'ls', but running on different directories.

what's the advantage of persistent memory, since it's so slow?
It survives when your system loses power. Banks like this, to maintain your bank account balance. I use it too, for class notes.

How do you access the source code.
C-h k K (for the key K), then go to the source file name and type ENTER>

Is there a limit to how many buffers you can have?
Not really. It is limited by available virtual memory, but you can have thousands anyway.

What format is a buffer
It's just a sequence of bytes. No other format (though there is some metainformation).

I will post these notes along with the video.

is there any difference between a normal buffer and the scratch buffer?
The *scratch* buffer is a "normal" buffer in some sense. However, its key bindings are different. More later.

Does emacs ever save (persistently) a command history? Like can you undo after re-opening emacs?
Such a feature is available somewhere. I never use it.

does .. having 4 aliases mean that there are 4 paths originating from the home directory?
Not exactly; it means there are 4 links to the file from somewhere.

Starting up Emacs

emacs
emacs FOO - start by reading FOO
emacs -nw - do not create a GUI window, just run in terminal

more shell commands:

ln A B - create a new name B for the existing file A
afterwards, A and B are "equal"; they both name the same file, and neither is more important than the other.

ln -s A B - creates a symbolic link named B that points to the name A
symbolic links are neither regular files nor directories: they're just "file name redirections"

```
  rm A - remove the name A for a file

  cat A B C - copy the contents of A, B, C to output
  cat - by itself just reads from stdin and writes to stdout
  head -N A - copy the first N lines of A to output


More Emacs characters:

   C-g - Get me out of here! Go to top level.
   C-x C-f FILE RET - visit file FILE (start editing FILE)

Why have hard links?

  - Efficient sharing of data.
      'git clone' uses this to clone Git repositories.

Why have symbolic links?

   - For metainformation (like emacs).
   - When your file has the "wrong" name.

File names in Linux are of the form

    /A/B/C/D..../Z   Each component is a directory except for Z.
    / (root directory)
    /A (subdirectory of the root)
    /A/B (sub-subdirectory of the root)
    ...
    /A/B/.../Y (sub**25 directory of the root)
    /A/B/.../Y/Z the file we want

On my machine /initrd.img is shorthand for /boot/initrd.img-4.15.0-118-generic

If you have trouble typing 'M-s', use 'ESC s' instead.

Emacs, how to copy text with a keyboard.
   C-@  (C-space) (same character)  - sets the "mark" (the other cursor)
   M-w  Copy all the text mark and current position,
   C-y  yanks the copy into the current location
   C-k  kill text to end line (copy into temp)
   C-y  yank what you just killed
   DEL  delete region (or next charactr)
        "region" - area between current location and the mark.

M-x view-lossage
  put the last N commands into *Help*


I'm typing this on 'day', which runs Ubuntu 20.04 LTS x86-64.
MS-Windows and macOS have never touched this hardware.


cat foo >output 2>errr

```

```
sends standard output to ‘output’ and standard error to ‘errr’.


UCLA CS 97 lecture 2020-10-08

Emacs and the shell are both instances of a pattern in software systems

The "Read Eval Print Loop" pattern
https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop
https://en.wikipedia.org/wiki/Read–eval–print_loop

  Aside: that’s the URL. What is this “%E2%80%93”
    It actually stand for an “en dash” - there are several characters
      that look similar -, minus, en dash, em dash, etc.
    To type on of these into Emacs, a longwinded way is:
      C-x 8 RET EN DASH RET
      – is an en dash (U+2013)
    To get details about a character ‘C-u C-x =’

    The UTF-8 encoding for U+2038 is "\xE2\x80\x93" in C
    Wikipedia describes UTF-8 pretty well.

Read-eval-print loop is simple:
  Program deals with the outside world as follows:
    1. Read a command from user input.
    2. Evaluates that command using the syntax and semantics
         of a particular language.
    3. Print out the result of the command.

Emacs REPL:

  My example is done on my Ubuntu 20.04.1 LTS [desktop (/etc/os-release),
  it has an older version of Emacs (‘emacs --version’ to see this),
  or Emacs 26.3; this sometimes disagrees with what’s running on SEASnet.
  This brings up the topic of *portability*.

C-x 4 f /etc/os-release RET
M-! emacs --version

C-j evaluates expression in *scratch*, puts results into *scratch*)
C-e evaluates it anywhere, puts result in the mode line (at bottom of window)

Portable software is designed to tolerate differences in the
underlying platforms.  E.g., if you have some Emacs Lisp code, you’ll
want it to run on both Emacs 26 and Emacs 27.  So you’ll need to know
about these differences.

    (expt 10 200)

  M-x shell RET (in Emacs) creates a buffer, runs shell in it
  C-g Quit- Interrupt what you’re doing, go back to top level
  C-c C-c in *shell* sends a interrupt to the shell.


Shell REPL:

```

```
Some shell commands

   true
   : (like ‘true’)
   false

Commands can succeed or fail, you can tell the difference by
looking at the command’s exit status (this is the integer that ‘main’
function returns).  Exit status of 0 means success, anything else
means failure.

Shell variables.
  some are builtin
    $? - exit status of the most recent command

Success or failure of a command can be used to influence later
actions, by using conditional commands.

Exit status 0 means “true”.
Exit status nonzero means “false”.

I created a *scratch* buffer by typing C-x C-c to exit Emacs,
and then I started up a new Emacs.

C-e means end of line
C-x C-e means evaluate. Emacs is modeful. Characters you type depend on context.
C-x establishes a new context.

C-h m lists your mode info - the context in which Emacs will interpret
characters that you type.

C-h b lists all your keybindings, that is, what happens when you type
any particular character.  It puts this info into *Help*.  C-x o
switches to that buffer.

   C-v scroll down
   M-v scroll up
   C-s STRING   - search for string
   M-s REGEXP   - search for a regular expression

You can use C-h b listing to find out what each command does.
It’ll give you that command’s documentation.

You can think of Emacs as having two different REPL loops.
  - At the low level of abstraction:
       - Read a character
       - Execute the function designated by the character.
       - Display resulting screen
  - In the *scratch* buffer, at a higher level of abstraction:
       - Wait for C-j
       - Read buffer before the C-j, looking for expression
       - Evaluate that expression
       - Display answer in *scratch*
```

```
ssh -X gives you the ability to run Emacs on a remote machine and display
locally, but it might be laggy.

My ~/.ssh/config contains shorthand for Penguin

   Host penguin
    HostName 131.179.64.200
    ForwardAgent yes

   DNS - Domain Name System (P. Mockapetris)
      maps 'www.ucla.edu' to IPv4 address 164.67.228.152
        (or IPv6 address 2607:f010:2e8:228:0:ff:fe00:152)


The shell also has a REPL:

Emacs language is Emacs Lisp (+ 2 3) etc.

Shell language is the shell language.

    Aside: why the "Shell"?
      thin layer around "real programs"
      doesn't do much
      glorified REPL in which EP is done by other programs

Some common commands:

In shell, to interrupt:

   C-c

nondestructive (safer) commands
  true, false, :   - placeholders, for control flow
  echo a b c - write 'a b c' to output
  cat a b c - write contents of the files 'a', 'b', and 'c' to output
  exit - exit the shell
  ps - process status
  ls - list file info (default is for the current working directory '.')
  cd - change current working directory
          file names like /a/b/c are *absolute* (context independent)
         they begin with '/'
      file names like oa/b/c are *relative* (context dependent)
         they don't begin with '/'
         their meaning depends on the current working directory

To find out about a command XYZ:

  $ man XYZ    -- in the shell

  M-x man RET XYZ RET    -- in Emacs


WEEK 2
------

UCLA CS 97 lecture 2020-10-13
```

```
560
561 administration stuff:
562
563 * Don't put off listening to (preferably coming to) lectures.
564 * Take notes, even though I'm typing all this.
565     (Write down the stuff I *don't* say.)
566 * Hint from Donald Knuth: How to listen to a lecture:
567     Think about what the presenter will say *next*.
568
569 Projects will be coming up.
570 Start thinking about what project you want to do - project ideas.
571    client-server application (multiple clients, one server)
572       clients on cell phones
573       server on AWS
574    we'll ask for proposals as part of these group projects.
575    You should be designing software you want to use, or see used.
576    Do something new!
577    Don't be afraid to think of ten ideas, throw nine away.
578
579 A client-server application:
580   Several different ways to hook small parts of an app into larger ones.
581      subroutines and main program (function or method calls)
582      primary/controller and worker nodes (multiple machines)
583         one machine is in "charge"
584     other machines accept tasks from the primary, do them,
585       come back and ask for more work.
586      ...
587      client/server - single server that maintains centralized state
588                      + clients that talk to users, get requests,
589                   ship off to server, get response back,
590               show that response to user.
591       clients are "in charge" in some sense
592           Typically, client is a program, user is a person
593      e.g., my.ucla.edu uses this approach (this is managed by UCLA IT staff,
594                                            disjoint from computer science dept.)
595           Very often, the server has a database as a component,
596         but this is not absolutely required.
597
598 more details about groups later this week
599
600 (end of administration stuff)
601
602 Scripting
603   shell scripting
604   Lisp scripting
605   Python scripting
606
607 Why have three languages? Why not just one? Why not just use C++?
608
609 Thoughts?
610
611   Different strengths and weaknesses - none of them dominate
612   everywhere, none are ridiculously bad everywhere.
613
614   * Ease of / flexibility of writing and maintaining code
615   ^    ideally, a scripting language lets you write in one line
```

```
   v   what would take (say) 30 lines in C++
   * Performance
           These two goals conflict.
   * Reliability (this can be a tricky thing)
         * Bad pointers, subscript errors, similar crashes
          won't trash your entire program in a scripting language.
     * BUT, typically C++, Java, etc. have better static checking
          which means your program won't have these bugs in the
          first place.

       static checking - compiler checks your program before it runs.
            double x = NULL;
       dynamic checking - runtime checks your program as it runs.
            char *p; ...;  *p = 'x';  // p == NULL

   * If you have a 1,000,000-line application, scaling issues will
       bite you, and scripting languages are designed more for
       smaller applications

     A scaling issue arises due to diseconomies of scale.
        economy of scale - Adam Smith, "The Wealth of Nations"
           a pin factory is a very efficient way to make lots of pins
       not needed in a small village
       saves a lot of money in a large city
        diseconomy of scale - as your system grows, cost per unit goes up.
           say a typical Python program (these tend to be worse than Java)
         10 lines - easy to understand, fits in machine,
           write it in 10 minutes.
         10,000,000 lines - not so fast!
           zillions of connections between modules
        lots of things can go wrong
        writing 10 lines can take a day
          (this is average in many industries!)

   * Ease of learning (careful - inertia)


Shell scripting continued

   aside:
   POSIX - Portable Operating System Interface "Xtreme"
      derived from an operating system called UNIX
        (derived from a research/production system called Multics)
          UNIX is a stripped-down Multics
        written by two guys (Ritchie & Thompson) in their "spare" time
        Turing award winners
      very simple (compared to other OSes)
      easy to hook together applications out of code
      GNU/Linux is a "imitation" of UNIX
        GNU = "GNU's Not UNIX"
        Linux is the kernel (below your application; see CS 111)
      FreeBSD / macOS is another
      OpenBSD, NetBSD, ....

      How do you write an application that will run on these systems?
```

```
  POSIX attempts to answer this question at two levels:

     Higher level: the shell.
       There's a POSIX standard for the shell language,
       as well as for the applications and utilities that you
       can run from the shell.

     Lower level: libraries and system calls (called from C, C++)
       This specifies things like <stdio.h>, <unistd.h>,
       what they contain, and how you use their functions.

POSIX spec is at:
  https://pubs.opengroup.org/onlinepubs/9699919799/

Tension / design decision to be made here:
 A Should you try to use a single language to solve your problem?
     That is, write your whole app in one language?  (CS 31 does this.)

 B  Should you write a multilinguage app, using a language that's
     well-designed for each part of your app?

   POSIX (GNU/Linux, UNIX) is aimed at (B).
   Idea is to use the best tool for the job, whether it's a program,
      or a programming language.
   "Software Tools" approach.
   "Little languages" approach -
      C (kinda big, but smaller than C++)
      sh
      awk (text processing - sed on steroids; syntax like C)
      sed (stream editor)
             sed 's/aabx/b/'  There's a 'sed' language.
      grep (searcher)

      Perl = the union of all the above little languages

        designed by someone who wanted to unify the language
      "The Hedgehog and the Fox" - Isaiah Berlin
         Hedgehogs knows one big idea, and tries to put everything
            under that idea.
         Foxes are always running around chasing new ideas,
            don't try to integrate them.
         Berlin's point was that Leo Tolstoy was a fox who wanted
           to be a hedgehog.


   Pros and cons here - you have to learn each little language.


Take a break until 17:07

Q. would a little language approach be better for huge applications compared
   to the big language approach? Or does it just come down to preference

   Typically large apps (hard-to write apps) are multilingual anyway,
   because no single language does the trick.  So it's not a big
   deal to add a little language to the mix.
```

```
Q. Typically little languages have a similarly little featureset.

   True: they're focused.

Q. what do you mean when you say a language is larger or smaller?

   The spec is more complicated, or simpler.
      Specs for languages can be a page or two (for simple ones)
      or 1,000 pages or more (for big ones).

Q. What's your favorite language?
A. It's the one I'm writing in.


Aside: more administration:
   Homework 2 is out. See static web pages.  Lisp + Python

The shell has several metacharacters that need to be quoted
if you want them to just stand for themselveds
    =!#&*()\|"'`~<>?;
    [{  SPACE TAB NEWLINE  Not metacharacters: a-zA-Z/@%^-_/.,]}

A little language;
    grep - defined by POSIX; GNU grep is an extension to POSIX grep
    Comes from g/re/p - Globally look for Regular Expression and Print
    T-Shirts that say "Reach out and Grep Somebody"
    grep 'BRE'
        reads input
        if a line matches BRE, copies that to output

There is a little language for Basic Regular Expressions (BREs).
    A BRE is a pattern, defined recursively as follows x
    (x is any *ordinary* character)
    a simple pattern that matches only itself
    matches any single character in a line
    BRE*      Zero or more concatenated instances of BRE
    BRE1BRE2  An instance of BRE1 followed by an instance of BRE2
    [abcdef]  Matches any single character in the set abcdef
    a[bc]*d   Matches any string of 2 or more chars, the first is a,
    the last is d, the remaining are all either b or c
    [^abcdef] Matches any single character NOT in the set abcdef
    [^a-z]    Matches any single character NOT in the set a-z
    [a^]      Matches either 'a' or '^'
    [+/%-]    Matches +, /, %, or - (don't write it [+-/%],
        in ASCII it's equivalent to [+,-./%]!)
    [[:alpha:]]   Matches any single alphabetic character.
    [~[:alpha:]/]   Matches any single alphabetic character, or ~, or /.
    [:alpha:] Equivalent to [:alph] not what you wanted
    ^BRE      Matches any instance of BRE that starts a line
    BRE      same, but ends a line
    \x        (where x is a special character) matches x
    \\        matches \

grep x   - Copy to stdout every input line that contains the letter 'x'.
grep 'x*'  - Copy stdin to stdout (like 'cat' or 'grep ''')
```

```
grep xyz  - Match only lines containing ‘xyz’ exactly
grep 'xy*z'   - Matches only lines containing x, followed by zero or more ys,
                     followed by z.
grep '^a.*z$' - matches any line that starts with a and ends with z.
grep \' - matches apostrophe the BRE is '
grep 'a'\''b' - uses the BRE a'b and matches just that 3-character sequenced.

Early on in grep development, there was a syntax dispute, so now there
are two syntaxes in POSIX for regular expressions.  BRE is simpler and
less powerful, ERE (Extended Regular Expressions) is more complex and
powerful.


With EREs you get a few more operators:

    ERE1|ERE2   either ERE1 or ERE2
    ERE?        matches zero or one instances of ERE
                    E?  == (E|)
    ERE+        matches one or more instances of ERE
                    E+  == EE*
    (ERE)       matches ERE

To use them in ‘grep’, use the -E option

    grep -E 'a(bcd|ghi)*z' ...
    grep -E '[a-z]+([a-z]+,)*[a-z]+' ...

# C identifier
id='[[:alpha:]_][[:alnum:]_]*'
# Function call
call="$id\\((id,)*id\\)"
grep -E "$call //"


A major problem in software construction - is configuration.

If you configure, say, SAP wrong, LAUSD won’t be able to pay its
employees on time.

So you end up with little languages to specify how things are configured.


Little Q&A afterwards:

If you mess up your PATH, fix it by setting PATH=/usr/bin and then
immediately edit your .profile so that it's not messed up.

More generally, if you edit .profile, don't log out! Log in again via
a separate window, to test that your .profile still works.


UCLA CS 97 lecture 2020-10-15

administration stuff first
  Homework 2 delayed until Friday next week
```

```
  aside:
     Part of my learn-by-doing philosophy
     I don't mind lecturing after the fact, to some extent.
     I don't want to take it toooo far.

Software construction via scripting
  Scripting is "just" programming - but it's also
     a way to think about gluing together large programs out of small ones.
     Maybe, the script is at the top level,
       PyTorch (ML scripting) Python script + C++ modules
     Maybe, the script is a small part of a larger app.
       Your web browser (mine is written in C++ / JavaScript via web pages)

  Examples of scripting languages
    sh - POSIX scripting language (top level is common)
    Lisp - we'll look at this as a small part of a larger app (Emacs)
    Python - (top level in our example)
    JavaScript - You'll use it both ways.
       server-side code in which JS is in "charge"
       client-side code in which JS is a "subroutine"

The shell (sh, POSIX shell - Bash as an example)
  It's a full-fledged programming language.
  Lots of stuff in it.
   control structures
     while cmd1; do cmd2; done
     for i in $v; do echo $i; done
     if cmd2; then cmd2; else cmd3; fi  # (backwards "if")

         # Bourne shell syntax
         if grep ^eggert: /etc/passwd > /dev/null; then
       echo 'eggert has a login here'
     else
       echo 'eggert cannot login'
     fi

   functions
     function f() { body of function; }
     f a b c

   Meta-execution (crucial part of software construction)
            (using software to create software)
        (program writes part of itself)
     1. You can put shell commands into a file, then the file becomes a command.
     2. $(CMD) means execute CMD, capture its output,
           make that output a part of your shell program
     3. eval "string"   - treat the string as code, and then execute it
           This is almost too much power.
    these are listed in increasing order of confusion / danger / watchout
                                          power

Lisp as a scripting language

  It wasn't originally intended as one, so this is a bit offbeat.
```

```
  Some history of Lisp (2nd oldest language still in use behind Fortran)
    Arose in the 1950s as part of artificial intelligence research
      LISP LISt Processing - idea was to use dynamically allocated
        lists as a basic building structure for writing programs
    to play chess, natural language processing, etc.
        “classic AI” as opposed to machine learning
    Many variants (sign of its success and its simplicity)
      Lisp 1, 1.5 (1950s-1960s)
        Scheme (1970s-)
    Common Lisp (1980s-)
    Emacs Lisp (1980s-)
      Racket (1990s-)
      Clojure (2000s-) runs atop the Java Virtual Machine
      Hy (2010s-) runs atop Python AST


     Aside: in any Emacs buffer, can evaluate Elisp with C-x C-e
        (cos -1)
      In the *scratch* buffer, evaluating is so common that C-j also works there.

Emacs Lisp (Elisp) data structures and functions
   numbers
   symbols (like identifiers)
      have values
      are objects
   data structures
      (a (b c) ((e)))
      nil is the empty list (nil is a very special symbol - represents
                             the empty list; it also represense ‘false’)
        nil “is” the null pointer in Lisp.
   function calls
      (a (b c) ((e))) -- same syntax as data structures
         In C, you’d write “a(b(c), e()())”
     How do you tell the difference between function calls and data?!?!
       It’s very easy to express code as data (because it *is* data!).
   Quoting.
      Normally, the Emacs interpreter executes code
          (cos 3)  - call cosine function!
      To stop it from doing that, you quote an expression
          '(cos 3)  - create and return that data structure

   Standard functions in Elisp
      (car L) - first item in the list L
      (cdr L) - list of the remaining items in L
      (append L1 ... Ln) - create a new list, containing the concatenation
                           of the contents of L1 ... Ln)

   Variables
     (setq abc (cos -1))  ; for global

     setq is an assignment statemetn
        abc = cos(-1);

     (+ 12 (let ((a 13)
                 (b -9))
             (+ (* a a) (* b b))))
```

```
951
952         let is a local initialization
953           12 + ({ int a = 13;
954               int b = -9;
955           a*a + b*b; })
956           // This uses a GNU extension - cross fertilization from Lisp to C, C++.
957
958      Syntax for function calls
959         (F A B C)    F is the function, A B C are args
960
961         (setq a b)  This is not a function: it's a *special form*.
962                   Special forms use the same syntax as functions,
963               but they're not functions.
964               In place where you'd normally write a function,
965                 you write a keyword.
966
967             setq is a keyword, used for assignment
968           Lisp experts tend to avoid setq, and prefer let
969         let is a keyword
970
971          Here are some other special forms:
972
973              (if A B C)  - if A is true, evaluate B and yield it.
974                                otherwise C
975
976             (defun f (x)
977                (+ x 1))
978
979  Aside on errors:
980
981     Emacs pops up a debugging window *Backtrace*
982       lists the stack of functions being evaluated when the error occurred.
983     To exit the debugger:
984       C-]  exits
985       C-h m    tells you what mode you're in (debugging is complicated)
986                   this gives you an intro about what you can type
987
988
989  Some builtin functions
990    (car L)
991    (cdr L)
992    (append A B C)
993    (cons A B)  - creates a pair
994    The above are standard for any Lisp implementation.
995
996  A bunch more for Emacs-specific stuff.
997
998    (message FORMAT ARGS)  - like printf
999    (current-buffer)   - returns the current buffer, which is an object
1000   (other-buffer)     - returns the "other buffer"
1001   (switch-to-buffer B) - turns Emacs's attention to buffer B.
1002   (point)            - Where are we in the current buffer?
1003      This yields 5828 for this buffer, since there are 5828 characters
1004      in the buffer on or before the cursor position.
1005   (buffer-size)       - how many characters are in the current buffer
1006   (point-min)         - minimum and maximum values for point in the current buf
```

```
  (point-max)
       (point-max)  one greater than buffer-size because you can move
                      point to just past the end of the buffer
  (goto-char P)  - move the cursor to position P


     Aside
       C-x C-b lists all your buffers
       C-h f FUNCTION RET  tells you about FUNCTION
       C-h b   lists your key-bindings

----- stoptalking script -----

#!/bin/sh

sleep $1
echo 'Time to stop talking!'


WEEK 3
------

UCLA CS 97 lecture 2020-10-20

Scripting
   sh          1970s
   Emacs Lisp  1980s
   Python      1980s

A simple example, to give you a feel:

   a = str('F')        # 'F'
   b = int('100')      # 100
   c = float('15.49')  # 15.49, with a rounding error!


   # A sample stock trade, as a string.  Strings are a big deal.
   # This declares 'line' automatically; its type depends on what
   # value is most recently assigned to it.
   line = 'F,100,15.49'  # scraped from a website

   # A list of 3 types.  You can't do this in C++, because these types are
   # objects, in the same sense that an integer or a string is an object.
   # Python is a *dynamic* language, so a lot of notions that in C/C++
   # are compile-time, are run-time.
   types = [str, int, float]

   spl = line.split(',')   # yields ['F', '100', '15.49']
   zspl = zip(types, spl)  # yields [(str,'F'), (int,'100'), (float,'15.49')]
   fields = [ty(val) for ty,val in zspl]   # ['F', 100, 15.49]

   fields = [ty(val) for ty,val in zip(types, line.split(','))]

Q. Wouldn't this be vulnerable to injections attacks?

A. An injection attack lets attacker take over the program by
```

```
    giving it a carefully formatted string, that the program
    misinterprets.  Suppose line = 'QR,37,,,,,27!'
    Python tends to work better in this situation, because
    it catches runtime errors that would crash C++.
    But it's not perfect.

Q. are those pairs just arrays w/ length 2 or some different data structure?
A. They are tuples, not lists.  They're different data types.
   (more details later) They're both sequences, so they have that in common.

end of simple example


Python motivation and history

  BASIC - originally developed in 1960s (my first language!)
     very popular teaching language: simple, like FORTRAN, scientific
  still popular in Microsoft circles
  problem, though:
    traditional, low-level language
      good at loops, functions, arrays
      not so good for fancier stuff
    project at CWI (MIT of the Netherlands) to replace BASIC in the 1980s
      faculty tired of unteaching BASIC bad habits
        (just like I'm trying to unteach C++)
      They wanted a disciplined language,
        there should be just one way to do it.
      ABC - came on a floppy disk, own development environment,
        1. Language is always properly indented, because
          the compiler required it and IDE helped you out.
    2. Build simple stuff like hashing, sorting, etc.
          into the language, so that students can write
          new programs, not just the same old stuff.
      It flopped.

    In the US in the 1980s, the language Perl was invented by Larry
      Wall in Santa Monica in his spare time.
        Perl - general-purpose scripting language
       = (sh + grep + sed + awk)   (these are little languages)
          so, one big scripting language to rule them all
        Perl took off!
        But...
       Wall is a linguist by training.
       Lots of English idioms in code, as well as computerish stuff.
           if ($x == 0) y = 3;
           y = 3 if ($x == 0);   // both work
      Write code the way you talk.
      "There's more than one way to do it." -- motto of Perl



   Python as "Perl done right" from CWI's point of view.
      * There's just one way to do it. (within limits)
      * Let's do indenting right.

Python came out in a standard implementation that has evolved.
```

```
C-Python - interpreter is written in C.
  python.org has a copy of the latest version
  I downloaded 3.9 to SEASnet, built it, installed into /usr/local/cs/bin.
  This is the most popular.
There are now other implementations.
  run atop Java Virtual Machine (from Oracle, etc.)
  PyPy  (Python implemented atop of Python)

$ python3    # on SEASNET, gives you C-Python

We are at the tail end of converting from Python 2 to 3.
SEASnet runs CentOS 7 on our lnxsrv10.seas.ucla.edu, etc.
Python 2 is standard there.  So if you run /usr/bin/python,
you get Python 2.   /usr/local/cs/bin is the same: 'python'
gives you the obsolescent Python 2, 'python3' gives you Python 3.

Compatibility issues!  Most nontrivial Python 2 programs won't run
in Python 3, unless some effort is taken.

     Python 2:  'print x'
     Python 3:  'print(x)'

Indenting in Python.

   Blocks must be indented evenly, and more than their parents.

       if a == 0:               <--- This ':' is important!
          print("it is zero")
       return 5             <--- Not indented right!
    else:                   <--- Badly indented!
    print("it is not zero") <--- Nope.

   You can use shorthand without indenting (but not recommended)

       if a == 0: print("it is zero"); return 5

String syntax in Python

   'x' and "x" mean the same thing: a string of length 1
   'xyz' and "xyz" also mean the same thing.

     '''
        This is a very
    long string
    with newlines in it
     '''

    "abc\ndef"  This a 7-character string, containing a newline.
   r"abc\ndef"  This is an 8-character string, containing \ followed by n.

Numbers in Python use syntax like C, but you also have complex numbers
if you want.


Python objects
```

```
  Every value is an object, and has:
    * an identity  (the “address” of the object)
    * type         (the “type slot” of the object - other implementations OK)
    * value        (the “value slot” of the object)

  Identity and type cannot be changed, once you’ve created the object.
  But the value can be changed if the object is *mutable*.
  Variables are *not* objects; you can assign values to them, but
     when you do that you haven’t changed an object’s contents,
     you’ve merely pointed the variable to a different object.

       a = 5
       a = 'bcd'   How does this work?  Every Python variable
                    contains the “address” of an object
       b = '100'
       a = int(b)   a and b are different objects
                   integers are immutable
       a = a + 1   constructs a new object 101 and assigns it to a
                   The old 100 is garbage collected as necessary.
              (aside - optimization for small integers)

       a is b     - Returns true if a and b are the same object,
                         i.e., they have the same identify.
       a == b     - Returns true if the objects that a and b refer to
                         have the same value
                                                        In C++
                     a is b -> a == b                   &a == &b -> a == b
                     a == b doesn’t mean that a is b    a == b doesn’t mean
                                                          &a == &b
       type(a)    - Returns the type of a, as an object
       id(a)      - Returns the identity of a, as an integer.
                          Like C++’s  (long)(a), but there’s
             no way back   (int *)(long)(a) in C++
                           no equivalent in Python
                        it wants safety
       isinstance(o,c)  - Returns true if o is an instance of c
                           Simple, basic, controversial operation
            # Weird code
            if isinstance(o, str):
           c = len(o)
        else:
           c = -1

                # Just write this:
        c = len(o)
        Maybe it’ll work, because later code will work anyway.
        If not, put it inside an exception handler
                    try:
               c = len(o)
           except:
              cleanup otherwise

Try stuff yourself!

Functions in Python:
```

```
    def f(x):
       return x + 1

  defines a function and assigns it to f, functions are
  objects just like anything else (like function pointers in C++)

  You don't need to name functions

    h = lambda x: x + 10

Classes in Python:

   class a(b,c):    <--- b and c are the parent classes
      var = 12
      def method(self, x, y):     <--- self is object this method is being
        return x + y + self.m2(var)                 called on behalf of

   foo = a   # This is allowed

   Multiple inheritance, variables are looked up by
    depth-first, left-to-right traversal across
    the parent hierarchy graph


  Namespace control.
     * A class is an object.
     * It has a member __dict__, that contains the class's members,
          as a dictionary (data type that maps name to values)
     * By convention, names starting with __ are private
         (they're made private by "mangling" them)
     Names that start and end with __ are reserved for
       Python internal use.

   Everything is dynamic: class are objects, you can have lists of classes,
     you can poke inside them, you can modify them if you know what you're
     doing, and this is OK.

   Goal is flexibility, not compile-time safety.
   You find your bugs by running the program, not by compiling it.
      (like sh, Elisp)

Why did Python succeed?  There were (and still are) competitors.

One part of this came from ABC's builtin operations and types.
   They're higher-level than in C++, Java, etc.
   They're also a bit slower.

Major categories:

   None  (special value, like null pointer in C++)
   Numbers
     int float complex boolean (0 or 1)
   Sequences
     strings  'abcdef'             <--- immutable
     lists    [0, 9, -12, 'ax']    <--- mutable
     tuples   (0, 9, -12, 'ax')    <--- immutable
```

```
      buffers  like a string, but it's mutable (often used to create a string)
      ranges   ranges of integers
    Mappings
      dictionaries  (sets of name-value pairs, that map names to values)
    Callables
      functions
      classes         f(1, 3)     <--- f is a callable
      methods
      ...
    Internal
      ....

We'll look at the operations on sequences, mappings, etc. next time.


Q. I have a question about lists being mutable and tuples being immutable
What if I assign an indexed element to another value?
And how is that different than assigning an indexed element of a string?

    li = [5, 9, 12]
    st = 'axzy'
    li[2] = 15  # li is [5, 9, 15]
    st[2] = 'q' # invalid, exception occurs
    a = st[2]   # assigns 'z' to a


UCLA CS 97 lecture 2020-10-22

Some administration.
  Split into teams, and specify your projects.
  You should write a client-server application.
     Client should be something nice, you can run it on a cell phone.
     Server should let multiple clients talk to each other.
     (Homework 3 is a simple example:
      project should be more ambitious).
    You want to have some creativity involved,
      so it's fun / motivating / etc.
      It could also be important.q

  This week's discussions are more important than usual.
  You've already gotten emails.
  Please attend.  (I'll probably send one more email...)
  They'll help groups form.

  A major goal of this course is collaboration while developing software.
  Most real-world software is developed this way.
    Most teams are 3 - 5 - 10 - 50 ...
  You should build a team that really works.
    (Aside: some teams in the real world is "toxic")
            You want to avoid this in your team;
       the sum should be greater than the parts.
       This may sound like a cliche, but it's true.
      One "obvious" thing to do:
        break up project into tasks
    assign tasks to members based on their expertise
         e.g., one developer is the "database guy"
```

```
             another is the “GUI guy”
      However, don’t overdo specialization here.
      You should try to learn all the areas needed to build the project.
        (You may be quizzed on any part of your project!)

   Group size of four, roughly.
     go up to five (if it’s bigger, split)
     go down to three (be careful; it’s a bit small)
     need to survive people leaving the project

Q. can we have a mixture of people from 1A and 1B?
A. easy to mix from discussion sections held at same time.
   harder if discussion sections held at different times.
      OK if you can genuinely meet at one of the two times.

Q re midterm: are we going to have any sample test for the midterm?
A. Yes, but.... last quarter’s midterms (which I’ll give out)
                don’t match this course entirely -- watch out for that.

Q. What is the date for the presentation?
Q. Does that mean that the presentation will be held during discussion?
A. The TAs are in charge of the project details.  (Come to discussion.)
   Some of these details will be up to you.

Scripting

Python scripting
  last time: basics of the language
  started going through some of the builtin types
  claim: its builtin types and operations are at least partly
         responsible for its success
  language + library
       <<<<
    we’re edging over into the library (later in the lecture)

last time we did numbers, major categories of builtin types.

Sequences - commonly used types in Python
  sequences include lists, strings, etc.

  operations on sequences

    s[i]    Returns the ith element of s.
            Valid indexes are 0, 1, 2, ..., len(s)-1
        s[-1] means the same thing as s[len(s)-1]
           s[-2] means the same thing as s[len(s)-2], etc.
        s[- len(s)] means the same thing as s[0]
        These should be all O(1) operations; no big need
          to worry about efficiency here.
            Valid indexes are really -len(s), -len(s) + 1, ..., -2, -1,
                                  0, 1, 2, ..., len(s)-1
            i can be any expression yielding an integer
        s can be any expression yielding a sequence
           (expression may need to be parenthesized)

    s[i:j]  Returns a subsequence s[i], s[i + 1], ..., s[j - 1]
```

```
             This returns a new sequence, but the elements are
            the same as before.
             If i == j, it's the empty sequence.
          If i < 0 or j < 0, they count backwards from the end.
          i <= j should be the case (after accounting for negative)

               s[1:len(s)]  - All of s except its first element.
            s[0:-1]      - All of s except its last element
     s[i:]   s[i:len(s)]
     s[:j]   s[0:j]
     len(s)  number of elements of s
     min(s)  smallest elements of the sequence: uses comparison (see later)
     max(s)  largest   (e.g., max('foobar') == 'r')
     list(s) constructs a *list* with elements equal to those of s
                 not every sequence is a list
          lists are perhaps the most convenient sequence
            (e.g., strings are not mutable but lists are)
                (you can concatenate lists with a+b, but this is an aside)

       Q. does list(s) create a new object even though you're
          using existing values
       A. yes (see transcript)

   operations on mutable sequences

       s[i] = v        Assignment to individual element
                         -len(s) <= i < len(s)
       s[i:j] = a      Replaces a subsequence with the contents of
                         the sequence a; this can change the length of s
                  a can actually be any "iterable"
       del s[i]        Deletes a sequence member, so len(s) decreases by 1
       del s[i:j]      Deletes s[i], ..., s[j - 1], so len(s) decreases by j-i



Aside on Zoom security:
   Zoom historically didn't (and still doesn't) care as much about
   security as I do.  I used to work for a computer security company.
   Zoom usually keeps running after I press END and all the windows go away.
   So I kill it off when that happens.  I'll sometimes reboot.
     And I've reinstalled the OS.
   Run Zoom in a virtual machines to lessen the exposure, if you can.
   We'll stick with Zoom, that's what UCLA has standardized on....


Operations on lists (every list is a mutable sequence, but reverse isn't true)

   s.append(v)      Appends an item to a list (len(s) grows by 1)
                      s[len(s):len(s)] = [v] but it's fast


       This is O(1) amortized.

   s.extend(a)      Append every element of a to s.  Cost O(len(a)) amortized.
   s.insert(i,v)    Insert value v just before s[i]
                      s[i:i] = [v], but faster
```

```
    s.pop(i)         Delete s[i], returning its value.
                       t = s[i]; del s[i]; return t
    s.pop()          s.pop(len(s) - 1)
                       This helps to explain why it's called 'pop'.
                It's treating s as a stack.
                   v = s.pop()     pops s into v
               s.append(v)     pushes v onto s
    s.count(v)       return a count of all members of s equal to v
    s.index(v)       return index of first element of s that equals v
                              (raises exception if none)
    s.remove(v)      s.pop(s.index(v))  removes first elt of s equal to v
    s.reverse()      trade s[0] with s[-1], s[1] with s[-2], etc.
    s.sort()         You don't have to implement quicksort
    There's more, but let's stop here.
    These do not create a new list; they modify the existing list.

string operations

  s.join(t)    joins the strings in t, using s as a separator
  s.split(sep) splits string into a list of words, using sep as a separator
  s.split(sep, maxsplit)  like before, except maxsplit bounds the number
                          of words
  s.split([sep [, maxsplit]])
      in documentation [] denote optional arguments

  Because strings are immutable, these do not modify s;
    they create new objects.

Q. How could python list both be able to remove and change elements in
the middle but still have the access time in O(1)?
A. Removing and changing is not O(1).
   s[i:j] = a      is not O(j-i) and not O(len(a))
                   It's O(max(len(s), len(a)))
           Because we want s[i] to be fast

Wait a second: Python is supposed to be inefficient - why do we care?
This matters mostly when things scale up.
When you use Python to deal with large arrays.

end of our intro to sequences


Mapping types

 dictionaries  (dicts for short)
  indexed by arbitrary immutable keys
    (as opposed to sequences, indexed by integers)

    d['eggert']      yields the value in the dictionary whose key is 'eggert'
                     A dictionary is partial function from keys to value.
              It typically starts being empty (no keys).
              You can change it over time by assigning to it,
              like this:

    d['eggert'] = 27     Dictionaries are mutable! even though keys are not.
    d['eggert'] = 'paul'
```

```
    del d['eggert']
    d['paul'] = 27

      Q. This may be a basic question, but can you change the key for a
       value or vice versa?  for a dict
      A. yes.

    From user's point of view, straightforward generalization of lists.
    (A list is like a dictionary where the keys are 0 .. len(s)-1.)

    They're implemented via hash tables that the Python programmer
    does not see directly.

    Curly braces mean dictionary, square brackets mean lists.
       d = {}    Creates an empty dictiory.
       e = { 'eggert':27, 'paul': 'xyz' }   Creates a dictionary with
                                          2 elements.
       f = { 27: 'eggert', 'paul': {} }
       g = { {}: 'eggert' }   <--- not allowed, since {} is mutable

       h = { (1, 2): 29, ('eggert', 19): 100}  <- OK, since tuples are immutable

   Q. Why not allow keys to be mutable?
   A. If keys could mutate, whenever you changed a key you need
       to rehash every hash table containing the key,

   Q. What's a hash table?
   A. Invented in the 1950s by IBM programmer Hans Peter Luhn,
        you can tell by its  name that people didn't think much of it.
    Corned beef hash, yuck.
       Hash function h(k) gives you a randomish integer.
          If k is a string 'eggert',
              XOR + shift all the characters in the string
          gives you a number 23424234124
            take this modulo the hash table size (511, say)
            get 256, use this as index into an array of size 511
              to find the key (if it's there, great)
          if slot is vacant (then that key is not in the hash table)
          if you find some other key then deal with collisions
            (can be via linked lists, can be other means,
             Python user should not care,
             user doesn't even know about the array or h (hash function))

Operations

    d[k]   Look up k in d, return corresponding value,
                KeyError if no value
    d[k] = v    Store v as value corresponding to k in d
    del d[k]    Remove key-value pair from d

    len(d)      Number of key-value pairs in d
    d.clear()   Discard everything from dictionary; dictionary is now empty
    d.copy()    Clone dictionary
    d.has_key(k)  True if k is a key in d (no KeyError)
    d.keys()    List of keys in dictionary (in some order)
```

```
    d.values()  List of values in dictionary
    d.items()   List of key-value pairs in dictionary
    d.update(d1)  Merge d1 into d (d1 wins if conflict)
                     Suppose d[k] = 10, d1[k] = 20
             Then, after d.update(d1),
                d[k] = 20
    d.popitem() Removes and returns a randomish key-value pair from d
    d.get(k [, v])   Returns d[k] if it exists, v otherwise.
                       v defaults to None

    for k,v in d.items():

Next time: finish up Python (modules, packages):

Please come to discussion!


WEEK 4
------

UCLA CS 97 lecture 2020-10-27

administrative stuff
  hope you went to discussion section, got a project in mind
  We'll create a CCLE spot for project proposals
    Your project proposals are not cast in stone - they can be altered
      as you - within reason - ask your TA about "reason"
    We still need a proposal now.
       aside: don't just go thru the motions and write down "just anything"
         writing down the proposal should help you advance your project;
     it shouldn't be just paperwork
    Just one copy of each proposal (put all your names on it).
    CSS? or plain HTML? - make it look good and work well.
       comment from student: "CSS is fun"

Python continued
Modularization and packaging - meta-tools for your software
  techniques for managing your code
  management of coded is a big deal -
     can take a big chunk of your development costs
  we'll do it in Python, transition to JS.
  you should know at least two methods reasonably well
Starting on React / JavaScript / etc.


Getting back to Python
  We did a lot of the basic type system, but we missed out on functions.

Functions in Python are objects, like everything else
  Lots of languages are like this (but not C, C++, ...)

E.g., if you define a function:

    def f(x,y):
       return x + y

```

It's equivalent to:

```
    f = lambda x,y: x + y
```

This creates a function object and assigns it to f.
Later on you can do this:

```
    g = f
    d = {}
    d['xyz'] = f
```

The lifetime of a function is like that of a list, etc. It lasts as
long as anybody cares about it (until it gets garbage collected - we
can make that happen as follows:

```
    g = None
    f = None
    del d['xyz']
```

Python functions can have varying number of arguments:

```
     # Call like this:  printf("%s %d\n", "xyz", -27)
     # Call like this:  printf("%s\n", "xyz")
    def printf(format, *args):
      format will be bound to the first arg
      args will be bound to a tuple of the remaining args
         args[0], args[1], ...
```

Like C's 'void printf(char const *format, ...) { code to do printf; }'.

Python functions can have named arguments:

```
   def arctan(x,y):
      compute the arctangent of y with respect to x

   arctan(y=1.5, x=2.7)   lets you call arctan in whatever order you want
```

You can combine the two notions: a function that takes a varying
number of named arguments (keyword args)

```
     def foo(x, y, **kwargs):
       ....
       x is bound to 1st arg
       y is bound to 2nd arg
       kwargs is bound to a dictionary of the remaining args

     foo(3, 9, alpha=0.1, beta=9.3)
        kwargs is bound to {'alpha':0.1, 'beta':9.3}
      kwargs['beta']
```

This helps Python code be more extensible.

```
      I can later extend foo with a new argument:
           def foo(x, y, z, **kwargs):

      Callers that do this:
```

```
          foo(27, 19, z=12)
       will work with both the old and the new version.

def bar(a, *b, **c): combines the above notions
   You might see this in Python source code, for “super duper” all-purpose
   functions - they don’t know what all the args are, they just want ’em all.

Functions can also have attributes:

        foo.secure = 1  # where foo is a function


Classes and typing.
  Recall that Python does dynamic type checking, not static
     dynamic - while the program’s running (not when you compile it)

    a = ...
    b = ...
    return a + b

So, how does it work?
    a.__add__(b)

Leading and trailing __ means the Python interpreter reserves these names.

    class c:
       def __add__(self, other):
          return (self.name + " plus " + other.name);

    x = c()
    return x + y

So, what does it mean to have a “type error” in Python
        char *p, *q;
    return p + q;
    Compiler will yell at you.
   With Python, it’s a runtime error: there’s no __add__ method!

This is called “duck typing” - if you want to add numbers,
   run x+y, and if it works, this means x and y both waddled
   and quacked like ducks, so they must be ducks.

Type checking is done by runtime behavior checking: if you don’t
 get an error it must be OK.
This gives you a lot of flexibility: your code can work in a lot
  of environments.
It also encourages error-prone code, as errors can easily slip through.

Many other builtin method names.

  class c:
     def __init__(self, a, b, c):
        Used by constructors c(1, 2, 3)

     __del__(self)  when your object is deleted  ‘del x[i]’ calls x[i].__del__()
```

```
                        assuming that's the last use of the object
    __repr__(self)  create a string representation of the object (full version)
    __str__(self)   same thing, except shorter, might be abbreviated
    __hash__(self)  used to implement dictionaries
    __nonzero__(self)   used for 'if o: ...' ; this is like
                              'if o.__nonzero__():'
    __cmp__(self, other)  returns -1, 0, 1 depending <, =, >
                          This explains seq.sort(): it invokes __cmp__
              as needed for members of the sequence.
            Like strcmp in C.
    There are several others like this, each bound to a builtin
    notion of the Python interpreter.

    Recall from last time: c.__dict__ is a dictionary of the names
      defined in the class.

Aside: why is 'self' necessary?
The reverse question: Why is 'self' missing in C++ methods?
  The way these methods work, is a pointer to the object is
  passed as hidden argument to the method, and you can see that
  argument in the method using a keyword.
Python attitude: let's write this down in the callers
  and not keep it a secret; you can call it whatever you want.


Now, turn our attention to software construction management.

Python modules (lowest level)

Module: (typically) a single file that contains Python code to be executed
  at the right time.

     ocean.py  file contains:
        abc = 27
    def f(x):
      return x + 2
    class c:
      def __init__(self):
        self.val = 0
      def bar(self, y):
        return self.val + y
          ...

  The "right time" occurs when you execute a statement saying
  "I want this module now!".

     if x < 0:
        import ocean

  Up to the caller to determine the "right time".

  Certain things happen when "import FOO" is executed:

     * Create a new namespace
     * Read the file ocean.py, and execute its code
         in the context of the new namespace.
```

```
      * Add a name FOO to the current namespace.
          FOO is bound to the newly create namespace.

How to run a module from the top level (when Python starts up).

  $ python3 modulename a b c ...
    imports module named 'modulename' with __name__ == '__main__'
  Lots of modules are not intended to be top-level programs;
  they're intended to be used only as parts of other programs.
  Still, it's helpful to use this convention of testing a module
  that isn't a top-level program>

      foo.py:
      definitions of some sort
      ...
      if __name__ == '__main__':
         test cases for foo

   I.e., if foo isn't intended to be used as a standalone program,
   you turn it into a standalone program that runs test cases for foo.

     aside: test-first software development likes this style.
        First you write the test cases for a module.
    *Then* you write the module's code.
     Why is this a good idea?
       1. You'll write test cases anyway, because you're
            a competent software engineer, and you know
        tests are essential
       2. Test cases are easier to write than code.
            Sometimes thought to be boring, because they're so easy.
       3. This lets you debug your module design faster.
            In particular, your API has to be good enough to be tested.

Q. if I import a module within a function, is it only defined in the
scope of the function, or within the whole file?
A. Try it.

Q. could you explain modulename a b c again
A. See next section.


Searching for modules.

   Where to look for modules when you do import?
   A Python installation consists not just of /usr/bin/python executable,
   but also of a bunch of files somewhere in the filesystem;
   where should Python look?
   Answer is complicated, partly because it's a big configuration problem.
   One part of the answer is PYTHONPATH.

PYTHONPATH is an environment variable in POSIX systems.
Environment variables are global variables, set in the shell,
and their names and values are exported to subsidiary programs.
Names are arbitrary shell identifiers, value are arbitrary strings.
'env' command lists your current environment.
PATH is a commonly used environment variable, for where to look
```

```
for executables.

PYTHONPATH is to Python modules as PATH is to Linux executables.

E.g., PYTHONPATH='/home/eggert/pylib:/usr/share/xonas/pylib' says
where to find a module 'foobar'.

There is a module hierarchy as well as a class hierarchy.

   class c(a):
     This means c is subclass of a

   class d(c):
     D's grandparent is a.

   There's a tree of classes, in which the parent node is the parent class.

  In modules:

     import ocean.island
     import ocean.island.hawaii
        Acts by reading 'ocean/island/hawaii.py' from some directory
      in your PYTHONPATH
     so we also have a hierarchy in modules, because the directory
     hierarchy is a tree and modules live in that tree.

  Keep these two hierarchies distinct in your mind:
    - Class hierarchy is about behavior:
        child objects act sort-of-like parent objects (they should be compatible)
    - Module hierarchy is about maintenance:
        It's typical for a single dev org to be in charge of a particular
    directory of the module hierarchy, and the module will be
    upgraded as a unit.

      You could have a.b.c be a subclass of d.e.f:
        a/b.py contains:
          class c(d.e.f):


Python packages (a level higher than modules)

  A package is implemented by having a directory
  contains module files (m1.py, m2.py, etc.)
  along with one extra metafile (__init__.py):
  that tells Python "this is a package" and is read
  whenever you import the package.  It could be empty.
  More commonly it at least defines
     __all__ = [list of modules to be imported when the user
                 wants to grab them all]
    User does this by saying
        from packagename import *

   aside: "*" in imports is considered to be bad style by some.
          because it puts your code at the mercy of the package;
      it can define names that maybe you wanted to use on your own.
      so a better style might be:
```

```
           from packagename import a, b, c  # Gives you control
   aside: you can use relative names

        from . import x   (import module x from same package)
        from .. import y  (import module y from parent package)
        from ..z import w (import module w from aunt/uncle package z)

So far, I've mentioned how packages are implemented when Python runs.
I haven't yet said how to place the packages in the filesystem.
You can do it by hand:
    mkdir pydir
    mkdir pydir/a
    cp myfile.py pydir/a/myfile.py
    export PYTHONPATH=$PWD/pydir
    ...
This is fairly error-prone; we want something more convenient, and
less error-prone.

"Standard" for Python package installation
  "in quotes" - it's evolving with time, more rapidly than the rest of this
     lecture.
  We'll assume Python 3.9 (installed as /usr/local/cs/bin/python3 on SEASnet)

How package installation is used -- should be reasonably simple.

  $ pip install somepackage
     # arranges for the package's files to be put in the proper spot
     # so that, e.g., default PYTHONPATH will find them.
  $ pip uninstall somepackages # undoes this
  $ pip list # lists your current packages
  $ pip show --files somepackage # lists files installed for that package.

Three logical places to get packages from.

  1. Get it from the Python installation.
       Traditional default, very simple;
       shared by everybody who runs that Python.
  2. Get it from a standard place under your home directory.
       Newer approach, but it still has at least one problem:
         sometimes you'll need incompatible packages just for your own stuff.
     You build app A that wants module M version 27
                app B that wants module M version 28
              These two module versions aren't compatible.
  3. Get it from a standard place in your app's virtual environment.
         Look for "venv" under Python documention.


UCLA CS 97 lecture 2020-10-29

apologies for the noise - construction next door

last time - Python modules and packages
  (basic units of construction in Python)
  some commonalities between Python, JS, etc.

Basic idea for Python
```

```
  language core (reasonably small, general-purpose)
+ extensions via
   *some Python source code that you put into a library package
   *code in some other language (C, C++, Fortan, ...)
      that may be lower level, can use lower-level facilities
      or may be more efficient
   *combination of the above

sometimes even this isn't enough
  You can change Python! (it's not that easy)
  PEP (Python Enhancement Proposal)
    you implement a change to Python (you have the source!)
    you propose it to the community
  This feature can appear in Python 3.22!

In short, Python is evolving
  Every successful software technology is evolving.
  You need to adapt by knowing the evolution techniques.
  In the Python world, PEP is an extreme because it lets you change the language.
  More commonly, you extend Python instead of changing it.
  It's a continuum in practice
    - some packages get used so much that they migrate in Python core
    - example: dateutils (needed for Python 3.8 and earlier
                          to do timestamp computation)
       In 3.9 (as a result of a PEP) the functionality has migrated
       into Python.

Last time I mentioned several sources for Python packages,
plus three ways that you can install them into your environment.
   1. Put it into /usr/
   lib/python/whatever (everybody can use package).
        (per system).
   2. Use PYTHONPATH to specify an alternate location,
        such as your home directory (per-user).
   3. Virtual environments
        Lets you create a separate environment for each application.
        (per application - all your procedures are on same page)

Virtual Environment tutorial
https://docs.python.org/3/tutorial/venv.html

As an aside:
   .pyc files are "compiled" .py files  .pyc is not machine code,
      but they're a portable alternative (a machine-independent set
      of virtual instructions, helps Python load and start up faster,
      because it can skip the syntax checking and scanning).

   __pycache__ directory can contain these files to cache the result
   of compilation.[

Virtual environments are newer than what we'll talk about next

Installing packages in a Python system

https://packaging.python.org/tutorials/installing-packages/
```

```
    It can be tricky -longish tutorial

We're building an application by running a bunch of 'pip' (and other)
commands.  This can take a while, and you'll make mistakes, and you'll
need to uninstall stuff, upgrade, etc.
You can save your state with 'pip freeze >requirements.txt'.
You can restore it later by doing 'pip intall -r requirements.txt'.
A requirements.txt file is a spec for your application's requirements.

How do you *create* packages?
  You have to write some code.
  This code will have some requirements, which you'll have to tell users.
  Usually, your code will have some legal requirements,
    such a software license. (e.g., GNU GPL)

  https://packaging.python.org/tutorials/packaging-projects/

   LICENSE - big deal
   README.md - your "elevator pitch"
      .md is short for Markdown
         popular formatting language
     markdownguide.org/cheat-sheet
   yourpackage/code.py      your source for a module (several of these)
   yourpackage/__init__.py  code to be run when your package is pulled in
   tests/ - test cases (written in python)
   setup.py - Python code to be run when your package is installed
      lots of stuff here
      let's focus on dependencies

Q. When you import setuptools.py, wouldn't that itself need a setup.py
and this makes a loop?

A. No, because each package has its own setup.py, so you're OK so long
as your package dependencies aren't circular.

Dependency management (when one part of your software assumes another part)
It is important in many phases of software construction.

-------------
*Build-time dependencies* in traditional Linux/Unix apps.
'make' does this (also in 'ant' for Java, etc.)
An example 'make' rule in a file 'Makefile':

  Makefile:
        # foo.c contains '#include "stat.h"',
        # stat.h contains '#include "sticks.h"'.

        sticks.h: sticks.h.in
            sed 's/VERSION/3.4/g' sticks.h.in >sticks.h

        foo.o: foo.c stat.h sticks.h
             gcc -c foo.c       # simple way to build foo.o

    foo.o is the *target* - the file that you want to exist,
                             and to be up to date
```

```
    foo.c, stat.h, sticks.h are the *dependencies*
       They're the things that the target depends on.
       You may need to build some of them,
          because you may have indirect dependencies.

    'gcc -c foo.c' is the *command* - executing this command
        will fix any problem with foo.o being out-of-date
    with respect to its dependencies.

     $ make foo.o
     ...
     $ edit sticks.in.h
     $ make foo.o
     sed 's/VERSION/3.4/g' sticks.h.in >sticks.h
     gcc -c foo.c

This seems pretty obvious: why not just use a shell script?

   buildit:
     sed 's/VERSION/3.4/g' sticks.h.in >sticks.h
     gcc -c foo.c

   Shell scripts don't capture the notion of dependencies.
   'buildit' always starts from scratch.

     $ edit foo.c
     $ make foo.o
     gcc -c foo.c

   'make' operates incrementally; it does the minimal set of
   commands needed to satisfy the dependencies.  It can restart
   from a partially failed computation, without doing all the
   work all over again.

How does 'make' record whether a file is up-to-date?
It 'cheats': it looks at the file's timestamps:
'foo.o: foo.c stat.h sticks.h' means foo.o is up up-to-date
if its timestamp is newer than max(foo.c, stat.h, sticks.h)'
I.e., 'make' relies on metadata from the file system.
This doesn't work if the metadata are wrong.
There's also a problem if the timestamps are exactly equal;
whether these are up-to-date is debatable.

Another approach is to use checksums of file contents.
'make' remembers checksums the last time it as used,
and uses the checksums instead of timestamps.
But now, where do you store the checksums?

-------------
*Installation-time dependencies* in Python / JavaScript / Unix/Linux/etc.

You have a package P, that depends on package Q already being installed.
With pip, you say this in setup.py. (more details in tutorial etc.)

   P: "I depend on package Q, version 3 or later."
   P: "I depend on package R, version 2 or later, but version 4 or earlier."
```

```
   ...
   Q: “I depend on package S.”

   declarations of dependencies

You have a directed acyclic graph (DAG) of dependencies
   nodes are packages
   arc when A depends on B.

pip must resolve these dependencies: builds the graph,
   finds what nodes are already installed
   installs the remaining ones, in order
   so that every package is installed after its prerequisites

At the high level, pip just looks at dependency graph
It’s just looking at declarations.
It decides what to do.
(Aside: You can specify code to be executed when a package is installed.
   setup.py can contain arbitrary Python code, after all
   But it’s bad style to do arbitrary stuff there.)


Aside: How to do version numbers, so that dependencies work well?
P: “I depend on package R, version 2 or later, but version 4 or earlier.”
   This set of constraints is not good, because it prevents
   upgrades to package R.
P: “I depend on package R, version 2 or later.” is better
      presumably because it uses version 2 features not in version 1
   There’s an assumption that later versions won’t materially degrade
   existing functionality.
   This assumption is not always true!  Packages will sometimes
     withdraw functionality.
   *Semantic versioning* - version numbers indicate how compatible
     a package is, compared to a previous version.
        version number 3.1.4
      P.Q.R (in general)
        Incrementing P is a big deal - it means the new
        version is incompatible with the old.

            Incrementing Q means - new features, but they’re
           all extensions to the old behavior

            Incrementing R means - no API-visible changes
           (bug fixes, performance improvements)


Getting started with React - big picture -

Client-server applications.
  Basic model:
    Application is split into cooperating pieces.
      Each piece runs independently on its own “computer” (might be virtual).
      One distinguished piece is called the “server”.
         Central part of the application.
     Application’s state (contents of variables, files,
                          that tell you the state of the system)
```

```
            is centrally controlled by the server
      The other pieces are called “clients”.
          Peripheral to the application.
      Often talk to human users.
      Clients have a GUI, touchscreen, etc.
      Typically they do this:
         wait for user request
         format it
         send formatted request to server
         get response back
         display it to user
 There are other ways to do distributed applications
     peer-to-peer applications
        every client talks to every other client
    no central server
    more complicated management than client/server
       (server can manage things in client/server)
        BitTorrent is an example
     primary/secondary
        one piece is primary (in-charge of computation; decides what to do next)
        others are secondary:
           wait for instructions from primary
       do the task that the primary tells you to do
       ship answers back to primary

Client-server performance issues

  Throughput
    How many actions per second can your application do?
    You can support more users if your throughput is higher.

  Latency
    What’s the delay between a user request, and the response back to user?
      Typical user requirement: latency < 1 ms


Discussion will talk about midterm
  Reminder: Midterm a week from today.
     2-hour midterm done within a 24-hour window.

--------------------------------------------------------------------------------
```