Don't be a bean COUNTER

10/2: Discussion 1

- `export` is used to set an environmental variable
    - Can be used to prepend to a `PATH`
- `vi`
    - `~/.profile` is the doc for your default customization in a SEASnet server
    - `i` to insert text, Esc to escape from insert
    - `:wq` to quit out of vi
- `mv` - rename a file
- `cp` - copy a file
- Moving in Emacs
    - `C-b`: back (`M-b` for one word)
    - `C-f`: forward (`M-f` for one word)
    - `C-p`: previous line
    - `C-n`: next line
    - `C-e`: end of line
    - `C-a`: beginning of line
    - `C-v`: scroll to next screen
    - `M-v`: scroll to previous screen
- Deleting in Emacs
    - `C-k`: Deletes a line, forward
- Formatting in Emacs
    - `C-o`: inserts newline
- Transferring files from server to local
    - `scp charlesx@lnxsrv06.seas.ucla.edu:/u/cs/ugrad/charlesx/file_name C:\Users\chuck\Downloads`
- I/O
    - d
- Shell Scripting
    - A shell is a UI that allows access to an OS's services
    - Common Unix Shells: Bash, zsh, sh, csh

10/6: The Command Line

- C++: objects working with operations
    - C++ programs operate in a process
    - These processes are an object in the OS, which contain the program and data necessary for the program
        - Command line commands work the same way: `ls`, `mv`, etc.
        - Processes may run the same program with different data

        - THe OS also contains files separate from running processes
            - Files contain data with no program or anything that's actually running
- High-level view: processes can be thought of as operations and files can be thought of as objects
    - Processes and files are really both objects
    - Big difference is power draw → shutting off power kills processes while file data remains
        - Processes are not persistent while files are persistent
        - Persistence requires tradeoff of efficiency in modern technology
            - Changing data would require writing to disk, slow read/write, etc.
- Applications must ideally be as persistent as possible, as efficient as possible, and understandable
- Ctrl is a mask of the least significant 5 bits of an ASCII char
- C-h k [key] tells us what key does
    - Press Enter on function name to see source code
- C-x o - switch to other buffer
- C-x 1 - look at just this buffer
- C-x 2 - split buffer in half
- C-x 3 - split buffer in half vertically
- C-x 5 - create a new window
- 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 d RET - create a buffer containing a list of what's in current directory
    - Every directory has 2 entries
        - . - current directory
            - Pointer to itself
        - .. - parent directory
            - Pointer to parent
    - Typing g refreshes it in the file system
    - Emacs automatically saves backup files when writing
- C-g - stop the current command
- C-x C-f FILE RET - start editing FILE
- Emacs is a modeful editor - actions upon typing depends on the modes Emacs is in
- M-q - reformat the current paragraph
- 2 types of files:
    - Regular files are sequences of bytes
    - Directories are mappings from file names to files
- M-x shell RET - opens the shell in Emacs
- ls -l - dumps files in directory with metainformation
    - 1st char is a file type: '-' for reg. file, 'd' for directory
    - Next 3 chars are owner permissions - 'r' for readable, 'w' for writable, 'x' for executable or searchable
    - Next 3 chars are group permissions

- Next 3 chars are public permissions
- Next is a number - number of directory entries that point to this file
    - When you create a file, its owner is the creator and the group is inherited from the group of the parent directory
- Followed by owner
- Followed by group
- Followed by file size in bytes
- Followed by last modification date
- Followed by file name within parent directory

- ls -a - includes output of files that start with .
- Emacs convention for file names
    - FOO~ is a backup file for FOO
    - #FOO# is a saved version of FOO while FOO is benign edited
        - Emacs tries to get around persistence (buffers are not persistent) by saving the contents of the buffer into the # file → autosave
- C-j evaluates commands in the scratch buffer
- ln A B - creates a new name B for the existing file A → A and B are equal afterwards
    - ln -s A B - creates a symbolic link from B to A
        - Symbolic links are neither regular files nor directories → just file name redirections
        - .#FOO is a symbolic link to nothing, just tells Emacs who's editing FOO for overwrite protection
- rm - remove a file
- cat A B C - copy the contents of A, B, C to output
- head - N A - copy the first N lines of A to output
- Why have hard links?
    - Hard links efficient sharing of data
- Why have symbolic links?
    - For metainformation
    - When your file has the wrong name

<u>10/8: REPL</u>

- Emacs and the shell are both instances of the same pattern
    - The Read-Eval-Print loop pattern
    - Program deals with the outside world as follows:
        - 1. reads a command from user input
        - 2. evaluates that command using the syntax and semantics of a particular programming language
        - 3. prints out the result of the command
- To get details about a character: C-u C-x = 'char'
- UTF-8 encoding
    - ASCII characters represent themselves → ASCII is a subset of UTF-8
    - Succeeding parts of a multi-byte encoding begin with bits 1 and 0
        - Therefore, has 6 bits available for encoding

    - Everything else is the beginning byte of a multi-byte encoding
        - Begins with as many 1s as the encoding length
    - Most commonly used chars have the lowest numerical representation for space efficiency
    - Multi-byte encodings don't overlap bounds for security reasons/ambiguity
- Emacs REPL
    - Emacs is modeful - C-x puts you in a different context for commands
    - C-h b lists all keybindings in the *Help* buffer
        - Can use to get command's documentation
    - Can think of Emacs as having 2 REPL
        - At a low level of abstraction - reads a character, executes the function designated by the character, and then displays the resulting screen
        - In the *scratch* buffer, at a higher level of abstraction - Waits for the user to type C-j, reads the contents of the buffer, looking for an expression, evaluates the found expression, then displays the answer in the *scratch* buffer
- Shell REPL
    - Some commands
        - true
        - : (like 'true')
        - false
    - Commands can succeed or fail, you can tell the difference by looking at the command's exit status → the integer that 'main' function returns (0 = success, anything else = failure)
        - You can look at this return value using shell variables
            - Specifically $? - the exit status of the most recent command
        - || - takes the first command, executes it, if it fails, executes the next command
- DNS - Domain Name System (Paul Mockapetris)
    - Maps 'www.ucla.edu' to IPv4/IPv6 address
- Emacs language is Emacs Lisp (+ 2 3) etc.
- Shell language is the shell language
    - Why the Shell?
        - You have an OS running multiple programs
        - The shell is a thin layer that connects these programs
            - It's thin because it doesn't do anything except be a gateway to the important stuff
            - It's a program that has a REPL and can parse input
            - It has its own programming system, and can be bulked up by the programmer
        - It's just another command, for convenience → takes the capabilities of the OS and connects them together
    - Some common commands:
        - Nondestructive -

- true, false, : - placeholders for control flow
- echo a b c - write ‘a b c’ to output
- cat a b c - write the contents of the files ‘a b c’ to output
- exit - exit the shell
- C-c C-c - gets to the top level of the shell
- ps - process status
    - Lists status of all statuses ps thinks you’re interested in
    - -e does something
- tr - transliterates inputs (find and replace)
- man X - gives documentation on command X
- grep - looks for patterns in input and copies lines with that input to output
- which - gives the location of a file
- ls - list file information
    - Defaults to base directory ‘.’
- cd - change directory
    - /a/b/c/ file names are absolute → context independent
        - Begin with ‘/’
    - a/b/c file names are relative → context dependent
        - Don’t begin with ‘/’
        - Meaning depends on the current working directory

<u>10/13: Scripting</u>

- Client-server application:
    - Several different ways to hook small parts of an app into larger ones
        - Subroutines and main program (function/method calls)
        - Primary/controller and worker nodes (multiple machines)
            - One machine is in charge
            - Other machines accept tasks from the primary, do them, and then come back to ask for more work
        - Client/server - single server that maintains centralized state + clients that talk to users, get requests, ship off to server, get response back, show that response to user
            - Clients are in charge in some sense, server waits passively for request to come
            - Typically the client is a program and the user is a person
            - Good approach for reasonably small applications (my.ucla.edu)
            - Very often, the server has a database as a component, but this is not requires
                - States may be kept in RAM
- Scripting
    - Shell, Lisp, and Python
    - Why have multiple languages?

    - Different strengths and weaknesses → none of them dominate everywhere, none of them are ridiculously bad everywhere
    - Ease of learning (be careful of inertia)
    - Performance
    - Ease of/flexibility of writing and maintaining code
        - Ideally a scripting language lets you write in one line what would take like 30 lines in C++
        - Flexible notation makes it easier on the programmer
    - Reliability
        - Scripting languages are going to be more reliable in terms of avoiding low level errors (bad pointers, subscript errors, etc.)
            - Can still occur, but can recover instead of core dumping
        - Typically, compiled languages have better compilers that examine the code more carefully that can catch errors that the scripting language won’t (better static checkers)
            - Dynamic checking - you never know if the program will bug out, runtime behavior varies from run to run
    - Scripting languages don’t scale as well as traditional languages
        - Designed more for smaller applications
        - Scaling issue arises due to diseconomies of scale
            - Adam Smith - a pin factory is a very efficient way to make lots of pins → not needed for a small village → saves money for a large city
            - Diseconomy of scale → the bigger your system gets, the higher cost per unit
            - Ex) a typical Python program → all languages have this problem, scripting languages have it worse
                - 10 lines - easy to understand, low memory usage
                - 10 mil lines - hard to understand, won’t fit in machine
                    - Zillions of connections between modules, lots of things that can go wrong
                    - Writing another 10 lines will take much longer (10 lines/day is a standard average rate of production)
                - Isolation of code is easier in compiled languages, so they scale a little better
    - Performance and flexibility conflict
        - Scripting gives up performance for flexibility
            - Values human time over machine time
- Shell Scripting
    - The shell has several metacharacters that need quotes to stand for themselves
        - =!#$&*()\|”’`~<>?[{; space tab newline

- POSIX - Portable Operating System Interface X
    - Derived from an OS called Unix → Unix derived from research/production system caled Multics
        - Unix is a stripped down Multics
        - Multics written by 2 guys in spare time → Turing award winners
        - Very simple compared to other OSes
        - Easy to hook together applications out of code
        - GNU/Linux is a “clone”/”imitation” of Unix
            - Linux is the kernel, GNU is the OS
        - FreeBSC / macOS, OpenBSC, NetBSD are other imitations of Unix
    - How do you write an application that will run on any of these systems?
        - POSIX attempts to answer this question at 2 levels
            - Higher level - the shell → there is a POSIX standard for the shell language and for the applications and utilities that you can run from the shell
            - Lower level - the libraries and system calls → called from C/C++, specifies things like <stdio.h>, <unistd.h> what they contain, and how you use their functions
    - Tension/design decision to make here
        - Should you try to do everything in one language?
        - Should you write a multilingual application, using a language that is well-designed for each individual part of the application?
            - POSIX (GNU/Linux, Unix) is aimed at this option
            - Idea is to use the best available tool for the job, whether it’s a program or a programming language
            - “Software Tools” approach
            - “Little Languages” approach - don’t build a single language to solve every problem, use small languages for each task
                - C, sh, sed (stream editor), awk (text processing - sed on steroids, syntax is like C), grep (seacher)
                - 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

        - Hedgehog knows one big idea and tries to put everything in the world under that idea
        - Foxes are always running around chasing new ideas, don't try to integrate them
        - Point is that Leo Tolstoy was a fox who wanted to be a hedgehog
- Pros and cons - you have to learn each little language
- Typically better for large applications, since no single language is appropriate, not a big deal to add a little language to the mix
- Little languages have a similarly little featureset, the spec is simpler
- 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
    - grep 'BRE'
        - Reads input
        - If a line matches BRE, copies that to output
        - There is a little language for Basic Regular Expressions (BREs)
    - If 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
        - 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
        - [^abcdef] - matches any single character that’s not in the set abcdef
        - [^a-z] = matches any single character that’s not in the set a-z
        - 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
        - [~[:alpha:]/] - matches any single alphabetic character or ~ or /
        - ^BRE - matches any instance of BRE that starts a line
        - BRE$ - same but ends a line
        - \x - x is a special character, matches x
    - grep x - copy to stdout every input line that contains the letter ‘x’
    - grep ‘x*’ - copy stdin to stdout
    - 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
    - Early in grep development, there was a syntax dispute, so now there are 2 syntaxes in POSIX for regex
        - BRE is simpler and less powerful
        - ERE (Extended Regular Expressions) is more complex and powerful
        - ERE1 | ERE2 - either ERE1 or ERE2
        - ERE? - matches 0 or 1 instances of ERE
        - ERE+ - matches 1 or more instances

                ■ (ERE) matches ERE

- A major problem in software construction is configuration
  - If you configure, say, SAP wrong, LAUSD won't be able to pay its employees

<u>10/15: Scripting Continued</u>

- 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 large rapp
    - Your web browser
- Example 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 - used 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)
  - Not indent-sensitive
  - Bash is a heavyweight shell with a lot of features vs. a smaller, faster subset of bash (sh)
  - It's a full-fledged language - lots of stuff in it
    - Contains various control structures (while loops, for loops, if-else statements, etc.)
      - while cmd1; do cmd2; done
      - for i in $v; do echo $i; done
      - if cmd2; then cmd2; else cmd3; fi
        - if grep {thing} /etc/passwd > /dev/null; then
          - echo 'statement 1'
        - else
          - echo 'statement 2'
        - fi
    - Can define 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
      - You can put a shell script into a file and the file becomes a command
        - Use parameters with $1, $2, ... , ${10}, …
      - $(CMD) means execute CMD, capture its output, and make it a part of your shell program
        - echo "blah: $(grep {thing})"

        - eval command
            - eval “string” → treat the string as code and execute it
                - Compute the string from various methods and use eval to run it as code
        - Above methods are listed in increasing order of power
- Lisp as a Scripting Language
    - Lisp originally not intended as a scripting language
    - History lesson:
        - 2nd oldest language behind Fortran
        - Arose in the 1950s as part of AI research
            - LISP - LISt Processing - idea that the simple data structure of a dynamically allocated list could be used to implement a basic building structure for writing programs to play chess, NLP, etc.
                - Considered to be “classic AI” as opposed to modern ML
            - Lists are built from pairs and they can represent arbitrary data structures (a b c)
                - Some pairs are not part of a list → (a . b)
                - Empty list → ()
                - Can contain other lists → (a (b c) ((e)))
                    - Parentheses represent levels of the list
                - Memory was really expensive → these lists were built dynamically → concern for bloating of memory causes resistance
                - First time pointers were used very extensively
            - Many variants: Lisp 1, 1.5 (1950s - 1960s), Scheme (1970s -), Common Lisp (1980s -), Emacs Lisp (1980s -), Racket (1990s -), Clojure (2000s -) → runs atop Java Virtual Machine, Hy (2010s) → runs atop Python AST
                - All somewhat niche, none are dominant in the world today
                - Sign of success and its simplicity
    - Emacs Lisp (Elisp) data structures and functions
        - Numbers
            - Evaluates to itself
            - May have rounding errors for certain levels of precision
        - Symbols (like identifiers)
            - Have values
            - Are objects in their own right
        - Data structures
            - ‘(a (b c) ((e)))
            - The empty list ‘() is called nil
                - nil represents the empty list and false
                - Implemented as the nullptr in early implementations
        - Function Calls
            - (a (b c) ((e))) → same syntax as data structures

            - In C, you'd write "a(b(c), e()())"
                - Whatever e() returns must be a function
        - Same notation as for data structures
            - How do you tell the difference?
                - You quote it
            - Very easy to express code as data → very important strength of Lisp
    - Quoting
        - Semicolon for comments
        - Normally you're thinking of your code as something you'd be executing, the Emacs interpreter executes code
            - (cos 3) → call the cosine function
            - Treat parens as data → you quote an expression
                - '(cos 3) → create and return that data structure → 2 item list with cos as one item and 3 as the other
    - Standard functions in Elisp
        - (car L) - first item in the list L
        - (cdr L) - list of the remaining items in L (everything except the 1st item)
        - (append L_1 … L_n) - create a new list, containing the concatenation of the contents of L_1 … L_n
    - Variables
        - We can create a global variable at any time and start using it
        - (setq abc (cos -1)) → prints cos -1 and modifies global variable abc to cos -1
            - Can access by typing abc
            - setq is an assignment statement
        - Avoid variables, focus on function calls
        - Can have local variables using (let ((a 13) (b -9))
            - (+ (* a a) (* b b))) → $a^2 + b^2$
            - let is a local initialization: { int a = 13; int b = -9; return a*a + b*b ; }
    - Syntax for function calls
        - (F A B C): F is the function, A B C are argos
        - (setq a b)
            - Functions evaluate the parameters and operate on them
            - a would fail when evaluated → setq is not a function since it runs
            - setq is a "special form" → same syntax as functions, but not a function
                - In place where you'd normally find a function, you write a keyword
                - Differentiate by knowing what's a special form
            - setq is a keyword used for assignment

        - Lisp experts tend to avoid setq and prefer let
      - let is a keyword used for initialization
      - (if A B C) → if A is true, do B and yield it, otherwise evaluate C
        - Only evaluates the A part and the B part
      - (defun f (x) (+ x 1)) → sets the expression x + 1 to the function f
    - Built-in Functions → evaluate their argos and execute
      - (car L)
      - (cdr L)
      - (append A B C)
      - (cons A B) → creates a dotted pair (A . B)
        - Can use with a list of size n to create an n + 1 sized list
      - The above are standard for any Lisp implementation
    - Elisp Built-ins (C-x C-e to execute commands outside of *scratch* buffer)
      - (message "abcdef") → creates a message to display to the user in the message line
        - Emacs equivalent of printf()
      - Can execute code from within shell using Elisp with emacs -batch -eval (message "hi")
      - (current-buffer) → returns current buffer, which is an object
        - Prints a brief summary of the complex buffer object
      - (other-buffer) → returns the buffer that isn't the current buffer that the user cares the most about
      - (switch-to-buffer B) → turns Emacs' attention to buffer B
      - (point) → where are we in the buffer?
        - Returns the number character you're cursor is currently on
      - (buffer-size) → returns how many characters are in the buffer
      - (point-min), (point-max) → minimum and maximum values for point in the current buffer
        - (point-min) is always 1, unless Emacs is restricting your control of the buffer
        - (point-max) is 1 greater than the buffer size → cursor can be 1 char past the end of the buffer
      - (goto-char P) → move the cursor to position P in the buffer
      - C-h f FUNCTION → tells you about a given function
    - In any Emacs buffer, you can evaluate Elisp with C-x C-e
      - In the *scratch* buffer, evaluating is so common that C-j is added as a command
    - Errors (divide int by 0)
      - Emacs pops up a debugging window *Backtrace*
        - Lists the stack of functions being evaluated when the error occurred
      - To exit debugger → C-]
      - C-h m - tutorial for the debugger

            - Trivia: dividing 1 by 0 doesn't error since floating point arithmetic overflows to infinity

10/16: Discussion 2

- Lisp
    - High level programming language that used parenthesized prefix notation
    - Common dialects today include Racker, Common Lisp, Scheme, and Clojure
        - We're using Emacs Lisp
- Lisp Basics
    - A function call f(x) looks like (f x) in Lisp
        - (+ x_1 x_2 … x_n) → summing numbers x_1 → x_n
        - (abs x) → absolute value function
        - (max x_1 x_2 … x_n) → maximum function
- Lisp Lists
    - Three ways to create a list
        - (cons object1 object2)
            - Object1 is the head and object2 is the tail
        - (list objects)
            - Any number of objects
        - (make-list *length* object)
            - Makes a list of objects with specified length
        - (car list)
            - Returns head of list
        - (cdr list)
            - Returns tail of list
- Lisp Functions
    - Defining a function: (defun name (args) body)
    - Ex) (defun foo (ab)
        - (+ a b))
- Python
    - If-else/for/etc.
        - Indent-sensitive

10/20: Python Scripting

- Python is often used as a string processing language
    - When you define a class/use an integral data type in languages like C++, you use a string in Python
- line = 'F,100,15.49'
    - line declared automatically, type depends on value assigned to it → can change later
- types = [str, int, float] → list of objects, you wouldn't be able to use this structure in C++ → these types are objects, Python is a dynamic language
    - A lot of notions that are compile-time in C/C++ are run-time in Python
    - spl = line.split(',') → splits the string, yields ['F', '100', '15.49'] as a list

- zspl = zip(types, spl) → yields [(str,'F'), (int,100), (float,15.49)] → list of tuples
- a = str('F') → converts argument to a string
- b = int('100') → converts argument to a 100
- c = float('15.49') → converts argument to a float
- fields = [ty(val) for (ty, val) in zspl] → iterates through zspl with the first item in each pair called ty and the second called value, and calls ty(val) on each pair, resulting in ['F', 100, 15.49]

- Python motivation and history
    - BASIC - developed in 1960s
        - Very popular teaching language: simple, like FORTRAN, scientific
        - Still popular in Microsoft
    - Problem with BASIC - very traditional and low-level language
        - Good at loops, functions, arrays
        - Not so good at other stuff
        - Project at CWI to replace BASIC in the 1980s → unteach bad habits
    - ABC - designed for 1980s computers, had its own development environment
        - Language is always properly indented because the compiler required it and IDE helped you out
        - Build simple stuff like hashing, sorting,etc. into the language so that students can write new programs, not just the same old stuff
        - Flopped
    - Perl - first general purpose scripting language that succeeded, made in US in 1980s
        - Take shell + grep + sed + awk
            - Take little languages and put them into a big language
        - Write code the way you talk
            - Hard to be a disciplined language with multiple ways to do things
    - Python - fixes Perl so there's only one way to do things
        - Indenting is done correctly
        - Think of it as Perl + ABC
        - First came out in a standard implementation that has evolved
        - C-Python → interpreter is written in C, most popular
            - There are other implementations → run atop Java Virtual Machine (from Oracle, etc.)
            - PyPy → Python running on top of Python
        - Indenting - blocks must be indented evenly, and more than their parents
        - String syntax - 'x' and "x" are the same, 'xyz' and "xyz" are the same
            - "xyz\ndef" - string of length 7, containing a newline
            - r"xyz\ndef" - raw string, string of length 8, containing a newline
        - Numbers in Python use same syntax as C, but complex nums are available
        - Python Objects
            - Every value is an object and has an identity, type, and value

    - Identity and type cannot be changed once you've created the object, but the value can be changed if the object is mutable
- id(a) - returns the identity of a as an integer
- isinstance(o, c) - returns true if o is an instance of c
- Functions
    - def f(x):
        - return x + 1
    - Defines a function and assigns it to f
    - 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, multiple inheritance
        - var = 12
        - def method(self, x, y): // self is the object this method is being called on behalf of (this)
            - return x + y + self.m2(var)
    - Variables are looked up by a depth-first, left-to-right traversal across the parent hierarchy
    - Namespace control
        - A class is an object
        - It has a member __dict__, that contains the class' members as a dictionary (data type that maps name to values)
        - By convention, names starting with __ are private → made private by mangling the names to outside
            - Names that start and end with __ are reserved for Python internal use
    - Everything is dynamic: classes are objects, you can have lists of classes, you can poke inside them, you can modify them if you know what you're doing
        - Python is willing to give up safety for flexibility
        - Find your bugs by running the program, not by compiling it
            - Like sh and Elisp
- Why was Python successful
    - Partly came from ABC's built in operations
        - Higher-level than what you'd normally see → performance isn't as valued due to the scripting nature of the language
- Major categories
    - None (special value like nullptr in C++)
    - Numbers - int, float, complex, boolean
    - Sequences - strings, lists, tuples. buffers, ranges
        - Tuples are immutable while lists are mutable
        - Buffers are like strings, but they're mutable
        - Ranges are like ranges of integers

        - Mappings - dictionaries (sets of name-value pairs)
        - Callables - functions, classes, methods
        - Internal

10/22: Python Operations

- Claim: Python standard types and operations on those types are at least partially responsible for its success
- When you talk about writing code in Python: language + library
- 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 -len(s), ... , 0, 1, 2, ..., len(s) - 1
            - s[-1] means the same thing as s[len(s) - 1], etc.
                - Goes to s[-len(s)] as s[0]
            - i can be any expression yielding an integer
            - s can be any expression yielding a sequence (may need to be parenthesized)
        - s[i:j] - returns a subsequence s[i], s[i + 1], …, s[j - 1]
            - s[1:] - everything in s except its first element, equivalent to s[1:len(s)]
            - s[:j] - equivalent to s[0:j]
            - If i == j, it's the empty sequence
            - i <= j should be the case after accounting for negative values
            - If i < 0 or j < 0, they count backwards from the end
                - s[0:-1] - all of s except its last element
        - len(s) - number of elements in s
        - min(s) - minimum value in s → uses comparison
        - max(s) - maximum value in s
        - list(s) - constructs a *list* with elements equal to those of s
            - Not every sequence is a list
            - Lists are the most convenient sequence
                - Strings are not mutable
                - Can be concatenated with '+'
            - New object is constructed by the function
    - Operations on mutable sequences
        - s[i] = v - assignment to individual element
        - s[i:j] = a - changes a subsequence to the contents of a, may change the length of s
            - a must be iterable
        - del s[i] - deletes a sequence member, shrinking the length of s by 1
        - dels[i:j] - deletes s[i], .., s[j - 1], so len(s) decreases by j - i
    - Operations on lists
        - Every list is a mutable sequence, reverse isn't true

        - s.append(v) - appends an item to a list
            - len(s) increases by 1
            - s[len(s):len(s)] = [v], but it's fast
                - Lists have a pointer to a piece of storage
                - Object also contains the length of the list and the size of the list (amount of space allocated for the list)
                    - append just has to place value at end (load/store) and increment length
                    - When list gets too long, a larger area of memory is allocated and the list object is updated
                        - Worst case: s.append(v) is O(n)
                        - s.append(v) is O(1) amortized
        - s.extend(a) - append every element of a to s
            - Costs O(len(a)) amortized
        - s.insert(i,v) - insert the value v just before s[i]
            - s[i:i] = [v], but faster
        - s.pop(i) - delete s[i], return its previous value
            - t = s[i]; del s[i]; return t
        - s.pop() - s.pop(len(s - 1)) - deletes the last element of the list and returns it
        - s.count(v) - return a count of all members of s equal to v
        - s.index(v) - returns the index of the first occurrence of v in s
        - s.remove(v) - removes first element of s equal to v
            - s.pop(s.index(v))
        - s.reverse() - trade s[0] with s[-1], s[1] with s[-2], etc.
        - s.sort() - you don't have to implement quicksort
    - Operations on strings
        - 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) - live above, except maxsplit bounds the number of words
- Mapping types
    - Dictionaries
        - Indexed by arbitrary immutable keys as opposed to sequences, indexed by integers
            - d['eggert'] = 27 // dictionaries are mutable even though keys are not
        - d['eggert'] - yields the value in the dictionary whose key is 'eggert'
            - A dictionary is a partial function from keys to value
            - Typically starts being empty, change over time through assignment
        - From a 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 doesn't see directly
            - Curly braces mean dictionary, square brackets mean lists
                - d = {} - creates an empty dictionary
                - e = { 'eggert':27, 'paul':'xyz'} - creates dictionary of size 2
                - f = {27:'eggert', 'paul':{}}
                - g = {{}:'eggert'} ← not allowed since {} is mutable
            - Why not allow keys to be mutable?
                - Since dicts are implemented with hash tables, if keys could mutate, whenever a key was changed, the hash table(s) containing the key would have to be rehashed
                - Hash tables were invented in the 1950s by an IBM programmer
                    - Hash function h(k) gives you an integer, mod that int with the hash table size, use the result as an index into an array to find the key
            - Operations
                - d[k] - look up k in d, return the corresponding value
                    - If the value is absent, a key error exception is raised
                - d[k] = v - store v as the value corresponding to k in d
                - del d[k] - remove key-value pair from d
                - len(d) - gives the number of items in the dict
                - d.clear() - discard everything from d
                - d.copy - clone the dict d
                - d.has_key(k) - true if k is a key in d
                    - Like d[k], but no KeyError
                - d.keys() - list of keys in dect
                - d.values() - list of values in dict
                - d.items() - list of key-value pairs in dict
                - d.update(d1) - merge d1 into d (d1 wins if conflicts occur)
                - d.popitem() - removes a randomish key-value pair from d and returns it
                - d.get(k[, v]) - returns d[k] if it exists, v otherwise
                    - v defaults to None

<u>10/23: Discussion 3</u>

- Regular Expressions
    - '+' - one or more
    - '*' - zero or more
    - '?' - zero or one
    - '(' and ')' - captures a group
    - '{i}' - match exactly this number of instances
    - '{i, j}' - match anywhere between i and j instances, inclusive
- What is React?
    - Open source JS library

- Used for building user interface
  - Backed by Facebook
- DOM (Document Object Model)
- Features of React
  - Declarative → you don't have to give instructions step-by-step
  - Flexible
  - Efficient → uses virtual DOM

<u>10/27: Python Modularization</u>

- Modularization and Packaging - meta-tools for writing software
  - Techniques for managing your code
  - Management is a big deal - can take a big chunk of your development costs
- Functions in Python
  - Functions are objects
    - Lots of languages are like this (not C/C++)
    - E.g., if you define a function *f*(*x*, *y*), it creates a function object and assigns it to *f*
      - *g* = *f* is valid
  - Functions have the same lifetime as a list, etc - lasts until it gets garbage collected
  - Python functions can have a varying number of arguments:
    - def printf(format, *args): // represents a placeholder for a tuple
      format will be bound to the first arg
      args will be bound to a tuple of the remaining args (args[0], args[1], …)
    - Python functions can have named arguments:
      - def arctan(x, y):
        computer the arctangent of y with respect to x
      - Can call with arctan(y = 1.5, x = 2.7)
        - Not assignment statements, just allows reordering of parameters
      - Can combine these things:
        - def foo(x, y, **rest): // represents a placeholder for dict
          …
          - x is bound to 1st arg
          - y is bound to 2nd arg
          - rest is bound to a dictionary of the remaining args
          - foo(3, 9, alpha = 0.1, beta = 9.3)
            - rest is bound to {'alpha': 0.1, 'beta': 9.3}
  - Helps Python code be more extensible
    - Can later extend foo with a new keyword argument:
      - def foo(x, y, z, **rest):
      - Callers that do this: foo(27, 19, z = 12) will work with both versions of the code

        - 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 - during runtime not compile time
        - a = …
        - b = …
        - return a + b
        - So, how does it work?
            - a.__add__(b) → invoking add method on a with parameter b
                - Leading and trailing __ means the Python interpreter reserves these names
                    - class c:
                      def __add__(self, other):
                      return self.name + "+" + other.name
                    - x =c()
                    - return x + y
        - So, what does it mean to have a "type error" in Python?
            - C++ - compile error, Python - runtime error
                - 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
                - def __init__(self a, b, c):
                  Used by constructors c(1, 2, 3)
                - __del__(self) when your object is deleted
                - __repr__(self) create a string representation of the object (full version)
                - __str__(self) same thing, except shorter, might be abbreviated
                - __hash__(self) used when your object is a key in a dictionary
                - __nonzero__(self) used for 'if o: …'; this calls o.nonzero__()
                - __cmp__(self, other) returns -1, 0, or -1 depending on <, =, >
                - Why does 'self' not exist in C++?

                - C++ methods work by passing a pointer to the object to the method as a hidden argument, and you can see that argument in the method using a keyword
- Software Construction Management
    - Make it easy to plug things together or tear things apart later
    - Python modules (lowest level)
        - Typical modules are a single file with 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 to access the module
                - ```
                  if x < 0:
                      import ocean
                  ```
                - Up to the caller to determine the "right time"
                - Certain things happen when "import FOO" is executed
                    - The Python interpreter creates a new namespace
                    - Read the file ocean.py and execute its code in the context of that new namespace
                    - Add a name FOO to the current namespace
                        - FOO is bound to the newly created 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 as parts of other programs
            - Still it's helpful to use this convention as a way of testing a module that isn't top-level
                - ```
                  foo.py:
                  definitions of some sort
                  if __name == '__main__':
                      test cases for foo
                  ```
                  ```

                - If foo isn't intended to be a standalone program, you turn it into one that runs test cases for foo
                    - Test-first software development → first write the test cases for a module, then write the module's code
                    - Why is this a good idea?
                        - You have to write the test cases anyways
                        - Test cases are easier to write than code
                        - Let's you debug your module design faster
                            - API has to be good enough to be tested
- 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 it look?
            - Complicated answer because it's a big configuration problem
            - One part of the answer is PYTHONPATH
                - 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, values are arbitrary strings
                - 'env' command lists your current environment
            - There is a module hierarchy as well as a class hierarchy
                - class c(a):
                    - This means c is a 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
                    - 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 higher level 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" ane 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 *
                - * in imports is considered bad style by some → package may define names that you wanted to use
                - Maybe better to use from packagename import a, b, c
            - You can use relative names
                - from . import x → import from same package
                - from .. import x → import from parent package
                - from ..z import x → import from uncle package z
    - How do you put packages in the right place in the filesystem?
        - Can be done by hand → tedious and error prone
        - Standard for Python package installation
            - Continually evolving with time, more rapidly than the rest of Python
            - Assume Python 3.9
                - $ pip install somepackage → arranges for the package's files to be put in the proper spot so that it will be found
                - $ pip uninstall somepackage → undoes install
                - $ pip list → lists your current packages
                - $ pip show --files somepackage → lists files installed as part of somepackage
    - 3 logical places to get packages from
        - Get it from the Python installation
            - Traditional default, very simple
            - Shared by everybody who runs that Python
        - Get it from a standard place under your home directory
            - Newer approach, but it still has a problem: sometimes you'll need incompatible packages just for your own stuff
        - Get it from a standard place in your application's virtual environment

                - Can install modules that disagree with each other without clashing

10/29: React and JS

- Basic idea for Python
    - We want a language core (reasonably small, general-purpose) + extensions via:
        - Python source code that you put into a library package
        - Code in some other language (C, C++, Fortran, etc.) that may be lower level → may have access to lower-level facilities or may be more efficient
        - Some combination of both
    - Sometimes not good enough
        - You can change Python (with effort)
        - PEP (python Enhancement Proposal)
            - You implement a change to Python (you have the source code) and propose it to the community → if accepted, it may appear in a future version of Python
- 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 into the Python core → dateutils
- Sources for Python packages + 3 ways to install them into the environment
    - Put it into /usr/lib/python/whatever → everybody can use package
    - Use PYTHONPATH to continue specify an alternate location, such as your home directory (per-user)
    - Virtual environments - lets you create a separate environment for each application (per application, all your procedures are on the same page)
- __pycache__ directory can contain these files to cache the result of compilation
- We're building an application by running a bunch of pip commands
    - No code, we're just figuring out what other packages we need
    - Mistakes will be made → need to be able to uninstall stuff, upgrade, tc.
    - You can save your state with pip freeze >reqs.txt
    - You can restore it later by doing pip install - r reqs.txt
        - File remembers all the stuff you needed to do to configure your application → like a spec
- 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 as a software license
        - LICENSE - big deal
        - README.md - "elevator pitch"

                - md is short for Markdown
                    - popular formatting language
            - yourpackage/code.py - source for a module (several of these)
            - yourpackage/__init__.py - code to run when your package is pulled in
            - setup.py - Python code to be run when your package is installed
                - Lots of stuff here
                - Let's focus on dependencies
            - tests/ - test cases (written in Python)
- 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
            - Example 'make' rule in a file Makefile:
                - # foo.c contains '#include "stat.h"'
                - # stat.h contains '#include "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 keep up to date
                - stat.h and sticks.h are the dependencies → the things that the target depends on
                    - You may need to build some of them because you have indirect dependencies
                - 'gcc -c foo.c' is the command - executing this command will fix any problem with food being out of date with respect to its dependencies
            - Why not just use a shell script?
                - Shell scripts are not flexible enough - they don't capture the notion of dependencies very well
                    - 'buildit' always starts from scratch
                - '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?
            - Looks at a file's timestamps
            - Looks at checksums
                - 'make' remembers checksums the last time it is used and uses the checksums instead of timestamps
                - 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
                - We have declarations of dependencies:
                    - 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”
                        - Avoid this, prevents upgrades
                    - …
                    - Q: “I depend on package S”
                    - …
                - DAG of dependencies → nodes are packages and edges are when a package depends on another
                - pip must resolve these dependencies
                    - Builds the graph
                    - Finds nodes that are already installed
                    - Installs nodes that aren’t installed in order
                        - Makes sure it doesn’t install a package unless it has installed all of its prerequisites
                - At the high level, pip just looks at the dependency graph
                - It’s just looking at declarations
                - It decides what to do
                    - You can specify code to be executed when a package is installed
                        - setup.py can contain arbitrary code, it’s bad style
                        - You want pip to have full control
                - Assumption that later versions won’t remove functionality from earlier versions
                    - Not always true
                    - Semantic versioning - version numbers indicate ow compatible a package is compared to a previous version
                        - P.Q.R. → incrementing P is a big deal - new version is incompatible with the old
                            - Incrementing Q - new features, but they’re all extensions to the old behavior
                            - Incrementing R - no visible change in the API
- React
    - 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 - 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
                - With client/server, the server can manage things
            - BitTorrent is an example
        - Primary/secondary approach
            - One piece is primary (in charge of the whole computation; decides what to do next)
            - Others are secondary:
                - Wait for instructions from the primary
                - Do the task that the primary tells you to do
                - Ship the answers back to the primary
            - “Reverse” of client/server
                - Clients are telling the server what to do next
    - Client/server performance
        - 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 the user?
                - Typical user requirement: latency < 1 ms

11/3: Client-Server Apps

- Client server computing
    - Last time - performance throughput and latency
- Correctness issues in client-server computing (arise due to common ways of attacking performance issues)

- Out-of-order execution → client and server (operating on different machines) may not be executing things in the order which you want
  - Client code and server code are not always synchronized
  - We think: C1 C2 S1 S2
  - Actually: C1 S1 C2 S2
  - Code is attempted to run as fast as possible, communication between client and server doesn't always happen
  - Possibly no serial order that makes sense → client and server may be altering shared state
  - Serialization - assume client and server code are running in parallel, explain what happens by a serial order of all their actions, resulting from interleaving the respective actions
    - Coming up with an explanation where all observable behavior by either the client or the server is explainable by the order
    - Might not be what really happened
    - Good enough implementation if works
- Out-of-date caches
  - Clients often cache server state for performance
    - What happens when the server state changes?
  - Cache validation - the client tries to keep its cache synchronized with the server cheaply (in terms of incremental updates) in a timely way

- The Internet
  - Before the Internet the most widely used form of communication was the traditional phone system → used circuit switching
    - Preallocation of circuit → N bits/sec and S ms latency
    - Lot of wasted capacity, inefficient use of the hardware resource
  - Basic idea of the Internet is to use packet switching
    - Instead of preallocating the circuit, you break up the message into packets of fixed-size (relatively small, few kB), then ship it to the network
      - Best-effort transmission for each packet
      - Small packets have more header overhead
      - Large p[ackets have less flexibility, need more memory in routers
      - Best size depends on hardware, latency, and throughput desires
    - More efficient use of the limited capacity, avoid having all the dead-time from previous implementation
    - First proposed by Paul Baran (RAND Corp)
      - Controversial because:
        - Unreliable
        - Billing
    - Packets have 2 parts: headers and payloads of a particular format
      - Headers - metainformation about the packet, overhead
      - Payloads - explains the data within the packet, data intended for the recipient

        - Packets are exchanged via protocols (specify packet formats, the order packets can be sent in, why the packets are being sent, etc.)
        - Cons:
            - Unreliable
                - Packets can be lost to network congestion
                - Packets can be received out of order due to routing
                - Packets can be duplicated due to misconfiguration or other low-level hardware things
    - Internet protocol suite addresses the above issues
        - Basic idea: use layers, don't try to solve it all at once
            - Lowest layer: the link layer - a point to point protocol, hardware-oriented
            - Layer 1: the Internet layer - shipping of packets with the above problems
            - Layer 2: transport layer - channels (or streams of data) implemented by sending packets
                - Transport layer only sees the channels
            - Layer 3: application layer - specific to apps (web browsers, video, etc.)
        - Core protocol is called the Internet Protocol (IP)
            - For level 1
            - Comes in various versions
            - IPv4 specifies packet formats, and it's connectionless, each packet is independent;y generated, sent, received
                - Head contains length, protocol number, source address (32-bit number expressed like 192.168.1.9), destination address, cheap checksum (let's the recipient detect data transmission errors), TTL (time to live) hop count (number of routers the packet has traversed)
        - Apps that really want packets instead of streams typically use UDP
            - User Datagram Protocol
            - Very thin layer over IP
            - "Datagram" is close to packet → one piece of data sent independently of other pieces of data
        - Transmission Control Protocol (TCP)
            - Streams of data that are:
                - Reliable
                - Ordered
                - Error-checked
            - Via:
                - Divide the stream into sequenced packets
                - Flow control
                - Retransmission and reassembly
        - Many application protocols built atop TCP/IP

        - Real-time Transport Protocol
            - Runs atop UDP, intended for real-time applications (live video, etc.)
                - Video apps care about timing, if a packet gets lost, we want to keep going
                - TCP would cause jitter
        - HyperText Transfer Protocol (HTTP) runs atop TCP
            - We want a reliable copy of someone else's webpage
- The World Wide Web
    - Invented by Tim Berners-Lee at CERN
        - A physicist's use of TCP to solve: how to present results on screen?
        - Browser (client) + web server
            - Client talks to server via an app protocol built atop TCP
    - 2 basic components:
        - HTTP protocol
            - Originally very simple
                - Client sends a "GET" request
                - Server responds with the resulting web page
            - Easy to implement, explain
            - Some problems with the protocol:
                - Security: nmo encryption, minimal checksumming
                - Uncompressed - bloated
                - No server push (client always had the initiative)
                - No pipelining (a single webpage with several components)
                    - You want the ability to send several requests before getting the responses back
                - No multiplexing (several windows talking to same server)
                    - Several connections from client to server, doing different things
                - Addressed by HTTP/2 by complicating the protocol
            - HTTP/3 (in progress)
                - Fundamental change is that HTTP/3 uses UDP instead of TCP
                    - Can do more multiplexing
                - Avoids head-of-the-line blocking delays - first packer in response is delayed
        - HTML data format
            - Borrowed from book publishers
                - SGML - Standard Generalized Markup Language
                    - Declarative language for text formatting
                    - Document Type Declaration (DTD) specified what markup elements were allowed
- HTML notions and terminology
    - HTML element

                - Node in the abstract tree that structures the HTML text document
                - Highest level is HTML element (single node root of tree)
                - Lowest level will be simple elements
                - Each element is surrounded by tags
                    - <tag> [contents] </tag>
                    - Closing tag can be omitted if obvious
                    - Void elements do not have a close by definition, never have child elements
        - HTML is a way of sending tree-structured text over the Internet
        - Documentation for HTML can be found on MDN
            - Mozilla Development Network
        - HTML started off with specific DTDs
            - A DTD is a spec for the trees you can put into a document or a grammar for the HTML language that can be used
            - These DTDs helped specify the original web
            - But, they were too limiting
                - For example, they didn't do videos
            - How do we let DTD's evolve rapidly enough to new apps?
            - Eventually gave up on DTDs as a standardization mechanism
            - Document Object Model (DOM)
                - Each HTML document is a tree
                    - Easy and standard way to walk through the tree and to modify the tree
                - Comes with APIs for traversing, updating the tree
                    - Callable from any language
                    - JS is the most common language

<u>11/10: More Client-Server Model</u>

- HTML
- DOM (Document Object Model)
    - Standard way (APIs) to manipulate what you see in a browser
    - The browser renders the DOM of the webpage
        - Static pages don't do anything
- Cascading Style Sheets (CSS)
    - Goal: separate concerns
        - Content - core part of your tree]
        - Presentation - how to present it to the user
            - Without CSS, you get a default and boring presentation
    - Styles are inherited by subtrees of DOM
        - You don't need to specify a style for everything
    - "Cascading" because a priority scheme is used
        - Get styles from ancestors, browsers, user, authors
    - Idea is to specify style without writing code to implement it
    - Allows more freedom to implement styles

- JavaScript
    - Another scripting language
    - Like Python, even more dynamic
    - Can be hooked into HTML
        - Want our browsers to be programmable
    - Can generate HTML (or DOM) to be rendered by browsers later
        - Not just a subroutine, can be thought of as main program of your webpage'
    - Relatively small and simple language
        - Has to fit into browsers (parse, compile, and execute)
        - Even today, IoT client are underpowered
        - So, you need libraries to build real apps
        - Even with libraries, it can be a pain to generate DOM/HTML
            - You have a bunch of calls to create a tree
            - You wanted to see the tree directly as an HTML-like syntax
- JSX - extension to JavaScript that allows easy generation of DOM/HTML via a syntax that looks like what you're generating
    - const header = <h1 lang="en">blah</h1>; → angle brackets signal beginning of JSX
    - Can be used anywhere in JS that a function call could be used
    - JSX produces a React element that implements the JSX
        - Can think of JSX asa preprocessor over JS
            - You have to know both JSX and JS levels to debug a system that you're building
    - Use JS inside of JSX
        - const language = "en";
const class = 'CS 97";
const n = 3;
const header = <h1 lang={language}>{class} assignment {n + 2}</h1?>;
- Efficiency issues can be understood by knowing how the browser takes/executes your code - browser rendering pipeline
    - HTML → DOM → … → pixels on screen
    - Browser starts rendering before it knows how to render everything
        - Some optimizations in this process:
            - Can skip subtrees that don't look like they appear on the screen
            - Can skip JavaScript code inside an element that's low priority
            - Decide the overall geometry of the page and then start rendering some components
                - Routine nowadays, sometimes wrong
                - Resizes are necessary, may need to re-render the page in worst cases
- Notation issue
    - Battle between HTML and JSON
    - JavaScript Object Notation

        - Notation that represents JS objects → data, not code
        - Same thing as saying it's a text format for communicating tree-structured objects
- Node.js
    - JS runtime for asynchronous events
        - Runtime - set of cooperative classes and methods for supporting a particular style of programming
        - Callbacks - user-defined functions that are called at particular points during execution
            - Cedes control from a called function back to the caller
        - Event handlers - callbacks executed when particular events occur (e.g. the arrival of a request from browser, press of a button, etc.)
        - Event loop - basic programming construct
            - int main (void)
              ```
              {
                      for (;;) {
                              Event e = waitForNextEvent();
                              handle(e); // Key point: event handler must finish
                                          "quickly", can't wait for anything else
                      }
              }
              ```
            - Event handlers don't do I/O and don't block or wait for anything
                - Can request, but can't complete
            - No locks (needed for multithreading)
                - Because event handler runs by itself
                - Avoid many race condition problems
            - So, multiple CPUs running in the same address space doesn't work - scaling via multithreading is impossible
                - Can scale with multiple computers (multiple web servers)
                - Can scale with multiple processes on the same computer
- Node.js and React are built atop these ideas
    - Uses JSON, JSX, etc.
- You can use this idea to build lots of cooperating servers or to implement code that runs in browsers
- Node.js also provides packages
    - Lots of choices - active software ecosystem
    - npm manages them
    - The packages your project is using is recorder by npm in a file
        - package.json
            - metadata about your package: name, description, author, dependencies, etc.
    - npm init
- Version control
    - Git basics

            - Git state for your project (past and future)
                - Git has an object database that contains the history of your project
                - Index file contains the plans for the future of the project
                    - Cache for the source code directories and files
                    - Creates commits
                    - Handles merges
            - Basic commands
                - git init - for projects starting from scratch
                - git clone - the more common case - cloning an existing project
                - git log - outputs log of changes made to the source
                    - Reports all changes to the project
                    - One entry per commit
                    - Commits have auto-generated IDs
                    - Abbreviations for IDs commonly used
                        - HEAD - most recent commit in the current repository
                        - HEAD^ - the just-previous commit to HEAD
                - git diff - compares 2 commits and tells you the differences
        - Where Git came from, what problems made Git?

<u>11/12: Version Control</u>

- Basic Git continued
    - index - says what changes you're planning to make in the next commit/set of commits
    - You're not just writing a program, you're developing a set of changes to a program
    - You want the best changes you can come up with
        - To make the program better
        - To "sell" your changes to fellow developers
            - Make your changes convincing
            - These changes are reliable → fix a new feature, fix a bug, etc.
            - These changes do what you say they do
- A few more Git command examples
    - git ls-files → outputs all the names of files currently available
    - git grep → calls grep directly off the repository
        - More efficient than normal grep for large programs
    - git config → tells you about the configuration of the repository
    - git show → tells us more information about a commit + metainformation
        - Tool for looking at lower-level information about a commit
    - git pull → updates the current repository by copying all upstream commits
    - git blame → tells you who wrote a specific line of source code, helps find out why lines are the way they are
    - git reset → go back to the previous version, discarding all changes since then
- Git internals

        - Each revision is an object in the Git repository
            - Each stored as a set of changes from the previous revision
            - Contains pointers to previous revision
            - HEAD points to latest version in the repository
            - May not be linear, another branch making independent changes
            - Main default branch is called the master branch
        - git branch -m master master-bad → rename master branch to master-bad
        - git branch --track master origin/master → creates another branch called master to what's upstream
- Merging involves at least 4 commits
    - Common ancestor
    - Branch A
    - Branch B
    - Merged descendant
- Low-level programming
    - Commonly done in C or C++
    - Not used much to do quick/user-facing applications
        - Used to access hardware features, do system calls, bypass efficiency gotchas in Java, Python, JS
        - Used to be software components or low-level tools that end users don't see directly
    - Examples
        - C-Python interpreter, Linux kernel, Emacs interpreter - C
        - Chromium, Firefox, JavaScript V8 - C++
    - C++ builds an abstraction layer atop of C
        - Classes - inheritance, encapsulation, polymorphism
            - Not in C
            - Data objects can be abstract - we don't know how they're implemented and we don't care → more of a pain in C, not done often
        - Namespace control → better modularity
        - Overloading is easy → badly supported in C, rarely used
        - Exception handling → C has low quality exception handling
        - Heap memory build in → C, it's just functions (malloc(), realloc(), free(), <stdlib.h>)
        - cin and count → abstract ways to do I/O → C just has functions (printf(), etc.) in <stdio.h> and <unistd.h>
    - Architecture of a C/C++ environment
        - Compiling, linking, executing split into pieces
            - $ gcc -E dumb.c → just run preprocessor (expands macros)
            - $ gcc -S dumb.i → compiles from macro-free C to assembly language

11/17: Low-Level Programming

- Low level programming
  - We want to have the ability to write some low level code
  - Some of the low-level stuff isfaily advanced, as far as software construction tools go
  - Some tools are even nicer than Python, JS, etc. to some extent
    - Of course, C/C++ are far less convenient
- What's between C and the machine level?
  - Can see by looking at what gcc generates
  - Can ask to show the assembly language code
    - gcc -S -O2 *prog-name*
      - -S → generate assembly language into a .s file instead of generating an executable
      - -O2 → optimization option
        - Code is not optimal → code is just better
        - Harder to understand, longer to compile, sometimes worsens performances
  - We see that sqrt (x) is usually implemented by a single instruction, although a function exists as a fallback → efficiency
- Path between source code and what's actually running in the machine
  - foo.c (source code file - foo.cc if C++)
  - foo.s (assembly language - textual representation of machine language)
    - gcc -S foo.c
  - foo.o (object code - binary representation of the machine language)
    - gcc -c foo.c
- How does an executable work?
  - A copy of most of the program is put into main memory
  - This contains instructions and data
  - The OS jumps into that copy
  - The instructions in the program are now in charge of CPU
  - The first thing these instructions do is dynamically link in whatever libraries you asked for
    - A copy of the dynamic linker is available in the program → this like in the libraries you need using system calls
      - Ordinary function calls work by executing single instructions in your program
      - When you call a function, it may execute many instructions before returning, but these are all part of your program
        - The function can't do anything that the caller can't do
      - When your program runs, it's walled off from other programs for security and walled off from your computer for security
      - So there has to be an escape hatch, where your program can do something dangerous under the supervision of the OS
      - This is called a system call → it looks like a function call, but it's not: it's a single weird instruction that causes the program to

temporarily suspend while the OS does the real work in a safe way
    - strace is a standard utility that tells you what system calls a program does
        - Logs all the system calls

- Other tools (besides compilers) for doing low-level development
    - Operations maintenance tools (used for SEASnet ops staff)
        - ps, top, etc.
    - Developer tools (used by developers)
        - Used to find out what's wrong with your program
        - time → runs a program for you and gives time values
        - strace/ltrace (library calls)
        - valgrind → looks for probable mistakes in the execution of your program
        - GDB → debugger
        - GCC → compiler
    - DevOps says developers and op staff should be interchangeable
- Let's look at compilers
    - 2 major free compilers: GCC (GNU/Linux) and Clang (macOS)
    - We'll look at GCC
        - GCC internals manual describes how it's implemented
    - How is GCC built?
        - Portability - we want GCC to work on lots of platforms (ARM, SPARC, x86-64, etc.)
            - GCC should be able to run on SPARC architecture, for example
            - GCC should be able to generate machine code to run on ARM architecture, for example
            - GCC developers distinguish among:
                - Target - machine that GC will produce code for
                - Build - machine that you're compiling GCC on
                - Host - machine that GCC will run on
                    - Could be all different architectures, generally at most 2
            - All GCC targets have flat address spaces - all pointers are the same size and have the same interpretation (indexes into a large array of bytes, which represent our program/data)
                - Describes most machines nowadays
            - How can GCC generate code for all these architectures?
                - It'd be too much work to rewrite/maintain GCC from scratch for every target
                - GCC is split into a machine-independent part (executed regardless of target) and a machine-dependent part (specific to the target, hopefully small)
                - Does this by having GCC developers write a machine description file that describes the target machine

            - In a high level way for most things, but with hooks (calls to some machine-specific C++ code) for machine quirks
            - This machine description file is a separate programming language
            - If you write a new file, you can generate code for a new machine
        - This is backend stuff, there's a similar thing for frontend stuff
            - Which language you're compiling
            - You can specify which language you're compiling, it'll give you a parser that'll feed in to the core part of GCC, the core will use the .md file to generate the assembly language
- What is GCC useful for once it's build
    - Generate machine code of course'

<u>11/19: More Low-Level Programming</u>

- What GCC is good for, besides compiling your program
    - GCC is good for security improvement
        - Stack overflow is a classic way to break into a system
        - gcc -fstack-protector → tells GCC to generate extra code for each function in which stack overflow is a real problem
            - Compiles with canary protections
            - Slows down your code a bit
            - On by default in some systems
    - Performance improvement
        - -O optimize
        - -O2 optimize some more → make the compiler a lot slower, to make the program a bit faster
        - -Os optimize for space, not time → small code
            - Can inhibit function inlining (common optimization)
                - Caller → y = p(x)
                - Callee → p(x) is a simple function
                - Inlining → pretends the caller just did the simple function without the function call
        - Optimization should not change the meaning of a valid program
            - But in practice, many programs are not valid → larger program = more likely to be bugged
        - Program might be busted and you want to debug it
            - Debugging optimized code is tricky → compile without optimization?
        - Modifying program to improve performance
            - Best is to use a better algorithm

- Help the compiler by writing the program in a better way
- __builtin_unreachable() → GCC builtin
  - Compiler can assume that this function cannot possibly be called

- __attribute__ as a way of improving performance
  - __attribute__ ((aligned(x)))
    - Tells gcc to align based on address
    - Tells GCC to perhaps waste memory
      - Proper alignment takes advantage of how memory is accessed/cached
  - __attribute__ ((cold))
    - Function is rarely used
    - GCC can put f into "cold" sections of the code
      - Stays in RAM/Disk → more effective use of caching
  - __attribute((hot))
    - Function is used a lot
    - Cold and hot require work by the programmer → nukes performance if wrong
      - Better way → profiling
        - Run program, measure parts that are cold and hot
        - Recompile with this
- gcc -flto
  - Enables link time optimization
  - When GCC compiles a .c file into a .o file, it puts more stuff into the .o file
    - Enough so that you can inspect the .o file and reconstruct the source code easily
  - When you eventually link a bunch of .o files using this flag. GCC reconstructs the original source info and then optimizes your whole program
  - If program is large, this takes a long time, can still be worthwhile
  - This flag is less explored, can be buggier

- Static checking → debugging program before runtime
  - _Static_assert
    - Directive to the compiler, tells the compiler the expression must be true, writes a message and refuses the compiler otherwise
    - assert is done at runtime
    - More efficient (zero runtime costs), more reliable (guaranteed if your program compiles)
    - Downside: argument must be evaluated at compile-time to a constant
  - gcc -Wall → generate some sane warnings
    - Uninitialized variables

                - Includes a wide variety of options, which you can disable or enable individually
                - -Wcomment → warns about nested comments
                - -Wparentheses → warns about precedence rules in C
                - -Waddress → warns about addressing errors with pointers
                - -Wstrict-aliasing → warns about aliasing
                - -Wmaybe-uninitialized → is there a path through the function that might use a local variable without initializing it?
            - gcc -Wextra → more controversial warnings (less likely to be useful to everybody)
                - -Wtype-limits → warns about sign conventions

<u>11/24: Debugging</u>

- Using GCC, you're only ever looking at a single function
    - Don't look at a single function, look at the entire compilation unit (the whole source code file → includes .h files, etc.)
    - gcc - fanalyzer
        - Like -Wmaybe-uninitialized → look for use of uninitialized variables
            - Also finds this when crossing function boundaries
        - Interprocedural flow analysis
        - Can find more errors, can be a lot more expensive → can greatly slow down compilation
- Changing the source code to make it easier to check
    - __attribute__ ((pure))
        - Side-effect free function (no I/O, no visible storage modified), the return value depends only on the argument variables and on the contents of storage
        - Why do this?
            - Optimization → compiler can cache values into registers without worrying that the pure function has modified those values
            - Clarity / communicating the API to the users of the function
    - __attribute__ ((const))
        - Stricter version of pure
        - Writes down more of the function's API formally so that the compiler can check it and take advantage of it
    - Downsides:
        - __attribute__ is less portable → only works with GCC-compatible
        - More work to change your code, decorate it with __attribute__s
    - Point is to evolve languages gently to make them more reliable/efficient/useful
    - Set of GCC attributes evolves
    - Don't go overboard, too much work to use all of them and sometimes causes GCC to issue unnecessary warnings
        - Static checking either mises errors that are actually in the program or cry false alarms

        - Compilers cannot in general predict what a program will do
            - Halting problem is undecidable
- Dynamic checking (runtime checking)
    - gcc -fsanitize=address
        - Tells GCC to catch out-of-bounds address access
        - Generates extra code to subscript check when it can
            - Slows down program
            - Makes program crash reliably when it does the wrong thing
    - gcc -fsanitize=undefined
        - Catches undefined behavior other than address errors
            - Mutually incompatible with -fsanitize=address)
            - Ex) integer overflow
        - Ensures reliable crashes
    - gcc -fsanitize=leak
        - Catches memory leaks
    - gcc -fsanitize=thread
        - Catches common errors in multi-threaded code
            - Multiple instruction pointers for multiple threads of execution sharing the same variable
            - + performance due to parallelism
            - - logs of bugs due to race conditions
        - Operates by slowing down the application and looks for other ways to interleave threads
    - Not perfect
        - Only works for that particular run - bug may be input-dependent
        - gcc, clang, valgrind can mis some errors for efficiency reasons
        - Can be severe performance penalties
            - Not a big deal with Python, JS, etc.
- Common theme - GCC generates different code that's slower but has more checks in it
- API - Application Programming Interface
    - For a function, contract between the caller and callee
    - Caller must implement the API correctly
    - Callee must rely on the API\
- Portability checking
    - Your program works fine on Ubuntu but fails on Fedora
    - You can compile and run on one platform, but it doesn't mean it'll work on a different platform
    - How to deal with this?
        - Know what platforms you might run on/try them out
        - Cross-compile for other platforms
        - Run on other platforms
    - Not just a low-level problems
        - JS code must run on Chromium, Firefox, Edge, Safari, etc.
            - Also run on phone screens, laptop screens, etc.

- Debugging
    - Don't do it if you can avoid it - it's an inefficient way to find and fix bugs
        - In badly-run projects, it can consume more than 50% of your developers' time
        - Debugging efficiently can give you an advantage on your competition
        - Need to be proactive
            - Prevent bugs from happening in the first place
            - Make bugs easier to detect when they do happen
            - Technology: static checking, dynamic checking, test cases
            - Especially true for low-level code
            - Use a better platform, port to worse platform later
                - Or pick a better language
        - Defensive programming
            - Traces and logs (print statements, output put into log files)
            - Checkpoint restart
                - Every now and then, save entire app state into file in a format so that you can later restore that state
            - Assertions

<u>12/1: GDB</u>

- Defensive Programming
    - Trying to prevent disasters
    - Shouldn't limit self to single technique
    - Traces and logs
        - Put in print statements
        - Organize and analyze the output
        - Web server logs, for example
    - Checkpoint/restart
        - Program periodically saves its entire state into a file
        - Program has an option to start with a saved state
        - This lets you restart the program if it failed
            - Hardware failure - restart with fixed hardware
            - Software crash - it may crash the same way, run it a bit differently
    - Assertions
        - Have to be relatively efficient, can't have side-effects
    - Exception handling
        - Try … catch …
    - Barricades
        - Divide your data into 2 major regions
            - Safe data - generated internally, can be trusted
            - Tainted data - data from the outside world
                - Can't be sure what the data is
        - Set of software routines that convert the tainted data to safe data
            - Filter out/edit issues

            - Must be systematic
    - Interpreters
        - When you don’t entirely trust your program
        - Can check each statement as it’s executed and catch errors, preventing bad behavior
        - Adding checks slows the interpreter down, interpreters are slow anyways
        - Defense against bugs in your programs
            - Ex: Throw a bug on a runtime error and let your program recover/do something more useful than crashing
    - Virtual machines
        - Enlist hardware to build interpreters that are faster while still having safety
        - Insulate the real machine from your application → barricade
- Defensive techniques do not always work
- Debugging strategy - assumption is: your program doesn’t work
    - Don’t guess at random what the bug is
        - Horribly inefficient, especially for larger programs
        - Your program’s features combine
        - When debugging, you’re exploring a combinatorially-explosive space
    - A more systematic approach for larger systems with bugs:
        - Stabilize the failure - make it reproducible
            - Many failures are randomish
        - Locate the failure’s source
            - From symptoms to cause
            - Requires real understanding of how the program works
            - Debuggers can help here
            - Terminology:
                - Error - mistake made by the developer
                - Fault - latent problem in the program
                    - In principle, it’s static
                - Failure - observed bad behavior by program during a run
                    - Prevent failures by fixing faults and errors
- Debuggers
    - Your program runs under a debugger’s control
        - Either a virtual environment like GDB or interpreter
    - The debugger can stop your program’s execution and examine your program’s state (contents of variables, registers, ip, sp, etc.)
    - The debugger can change your program’s state
    - Since the state includes the ip (instruction pointer), it can run program’s code that otherwise wouldn’t run
    - If your program is optimized, it can be harder to debug because machine code is in a different order than the source code
- GDB
    - r ARGS <INPUT >OUTPUT - starts a program

    - start ARGS <INPUT >OUTPUT - does the same as r, but sets a breakpoint on main
    - attach PID - lets GDB take control of an already running program
      - detach - lets the program run free again
  - set env PATH “PATH” - setting up environment variables
  - set cwd “DIRECTORY” - sets current working directory
  - set disable-randomization on/off - modifies Address Space Layout Randomization (ASLR)
    - Causes low-level primitives to create objects at random-ish locations
    - Makes it harder to exploit bugs in your program
    - Commonly enabled in modern era, makes program behavior irreproducible
    - Disabled by default
  - bt - generates a backtrace
  - b LOCATION - set a breakpoint at LOCATION, execution stops if it reaches there
    - LOCATION can be a function name, filename, etc.
    - info break - gives information on existing breakpoints
  - c - continue execution
  - step - single step to the next line, descending into any subroutines and stopping there
  - next - single step to the next line, let the subroutines execute
  - fin - continue until the current function finishes, then stop
  - u LOCATION - continue until we exit the current function or we reach LOCATION
  - stepi - like step, except executes just one machine code instruction
  - rc - reverse continuation: like c, except it goes backwards in your execution history
  - watch EXPR - stop the program if EXPR changes
    - Can slow down execution greatly, but if you keep EXPR simple, it’ll run at full speed
  - checkpoint - saves the state and outputs ID for the saved state
  - restart CHECKPOINT_ID
  - p EXPR - prints the value of an expression
    - p VAR = VALUE - assigns VALUE to VAR, prints result
  - p/x EXPR - print the value of EXPR in hex
- Debugging targets
  - You can use GDB on one machine X to debug a program running on a different machine Y
    - X and Y don’t need to have the same architecture
  - Common for IoT or embedded systems, where GDB is too big and unwieldy to run on a small device
- Back to version control
  - DevOps - combination of developers and operations staff
    - Traditionally different people
    - Nowadays, same person can have both rules

                - Break down the boundaries between these 2 organizations
    - Version control becomes more important in this world
        - You are developing both programs and program configurations at the same time
        - Use version control to make sure this stuff stays in sync
    - Version control needs
        - Backups in case source code gets messed up
        - History for code/configuration historians
            - Developers or ops staff wondering why the code is the way it is
            - Looking here is often the most efficient way to understand a program

<u>12/3: DevOps and Version Control</u>

- Version control needs
    - Backups - you've messed things up
    - History - for knowing why the code the way it is
        - Relationship between bug reports and feature requests and changes you make to the source code
            - Bug report → … → weird source code
        - What's the relationship[between 2 projects that get integrated/split apart
        - Review of the source code by the original developers, but they forgot why they did things
            - The why is important, the what is easy
    - The future
        - In software development, there are many possible evolution paths
            - Sometimes future features don't interact, so the development group can work on them independently
            - Too often, they do interact
                - You need plans to minimize problems due to these interactions
                - A good version control tool will support planning for the future, as well as looking into the past
- Backups and disaster recovery
    - You must be prepared for bad things to happen
    - Data/systems will be lost, minimize the damage when that happens
    - Periodically make a copy of everything, restore that from copy on disaster
    - In some sense, inverse of caches
        - Caches help performance by making throwaway copies in faster memory
        - Backups hurt performance by making permanent copies in stable storage, gaining reliability
    - If you're designing a backup system:
        - You must have a failure model
            - What things can go wrong
            - How likely they are to go wrong

        - To prioritize when and what to back up
        - Examples:
            - Your flash drive fails in your laptop
            - A disk drive fails in your servers
            - You delete or trash files by mistake
            - An outside/inside attacker trashes files on purpose
- A failure model with a few more details:
    - Flash drives AFR (annualized failure rate)
        - Say it's 1% that the whole drive fails
    - If you have a backup policy: "I back up every file once per day", what's the possibility you'll lose data sometime in the next year?
    - Make sure your key assumptions are valid
    - Your failure model must be end-to-end, which means you must worry about recovery
- What to backup?
    - File data - just the contents
    - File metadata - information about each file
        - Last-modified time, last-accessed time, ownership, permissions
        - Directory organization
    - Filesystem metadata
        - Keeps track of the OS infrastructure lying underneath all the files on your system
    - Alternate possibility: just worry about the underlying hardware - just copy all the hardware blocks of data on the hardware
- When to backup
    - Do you back up every change to the system, or just some changes?
        - You can just back up every now and then, omitting intermediate states
        - You can back up just a part of your system, omitting less-important parts of the machine state
    - If you generate a lot of backups, when do you reclaim storage that's no longer needed?
- How to do backups cost-effectively
    - Do them less often
    - Do them to a cheaper device with poorer performance
        - Flash → disk
        - Disk → Magtape
    - Remote backups → let someone else do the work
        - Some trust issues here
    - Incremental backups
        - Instead of making a copy of all your files, just copy the changes you've made since the previous backup
            - Need a way of computing the delta from the previously backed-up system

- You can use timestamps for this (file metadata), backup all files newer than the previous backup
- You can do something fancier than timestamps by having a delta that records just the changes
- Idea is that the delta is smaller than the originals
  - You do incur the cost of computing the delta
- The edit scripts can do the following say:
  - Insert some lines at location N
  - Delete some lines at location M
  - Replace = insert + delete
  - These 2 commands generally suffice

- Deduplication is an optimization of a backup policy
  - Suppose the data are organized as a sequence of blocks
    - cp bigfile bigcopy
      - Supposed to copy all the data from bigfile to bigcopy
      - The file system notices whenever you write a block of data that happens to equal a block of data iot already contains
      - It cheats in that case by using a pointer to the already-existing block → hashing
      - cp at the low-level simply creates a file whose metadata are pointers to the other file
  - Problem with this approach → if a block is corrupted, several files may be corrupted
- Compression
  - gzip → backup can be smaller than original
- Encryption
  - For security in case an attacker gets hold of the backups
- Multiplexing (single backup device for many systems)
- Staging
  - Primary copy backed up to secondary copy to tertiary copy, etc.

- How do you know your backups are working
  - Test them by doing test restores
  - If you restore the data, how do you know the restored copy is correct?
    - Checksum your data (store them somewhere else)

- Backups for software developers
  - We're not just talking about code
  - It's for data, documentation, configuration, etc. (you're writing down everything anyways)
  - File systems with versioning
    - Version as part of the file name, application decide when there's a new version
      - open() or write() doesn't change version, close() does

            - More typically, you'll need a new API to control this
            - Not every application wants to do this
            - Every version is a separate file
            - There's a limit to a number of outstanding versions a file can have
            - You can preen a filesystem by removing older versions of stuff
        - Snapshot approach - filesystem periodically decides to make a snapshot of itself (of a single point in time for all the files)
            - SEASnet → WAFL
            - Backups are often written to different media, while snapshots are often done on the same media, for efficiency
            - Snapshots work better for procedural/human/software failures, not hardware failures
            - Snapshots are commonly implemented via copy-on-write
                - You pretend you made a copy, but you simply create a pointer to the original data, and you don't actually copy until someone modifies either the original or the copy
- Version control systems
    - Originally intended mostly for software development
    - Basic idea is to be smarter than versioning file systems
        - More efficient
        - More useful
            - Navigating through a pile of old versions of software
    - Useful and/or necessary features for version control systems
        - Keep histories indefinitely (at least for source code)
        - Record metadata as well as data
            - Not just file system metadata (last-modified time), but also the why for a change
        - Suppose for example you rename a file
            - You need to be able to link the 2 names together
        - Metadata about the history, not just about the source code
            - Ex: in Git, branch names belong to the history of the source code
        - Atomic commits
            - Change several files or objects all at once so that the history never records just some of the changes, you either have the old version or the new one
            - Essential for collaboration, you don't want to be messing with other developers' code
                - You want the source code to always be in a consistent state, not some random mix of 2 versions
        - Pre-commit hooks
            - Run just before commit
            - Can be used to sanity-check a commit
        - Post-commit hooks
            - Run just after a commit

                - Can be used to update your other files
            - Signed commits
                - Cryptographic signature that is hard to forge that only you can create with the possession of a secret key
            - Format conversion
            - Navigation and visualization of complex histories of source code
- History of version control systems
    - SCSS (Source Code Control System)
        - Prototype written in AIish language Snobol
        - Rewritten in C for Unix, mutated as it went
        - Each file F in your source code has a corresponding history file s.F which contained all of F's history
        - Bulk of the history was at the end of the file, containing all of the lines that ever appeared in F
        - Start of the s.F file contained the metadata for F's history
            - This metadata included locations of line numbers in the bulk history
        - To construct tan old version of F
            - Read s.F's copy of the metadata
            - Deduce where all the old version's lines are
            - Left-to-right scan through the bulk data, grabbing just the lines you want in order - 1 sequential pass through the data
        - At the end of s.F, there's a checksum
        - Cost to extract F is O(|s.F|)
    - Several successors
        - RCS (Like SCCS, but free software)
            - Its bulk data put the most recent version of F first
            - Cost to extract latest F was O(|F|)
        - CVS
            - RCS front end
            - Key differences:
                - Single commits that crossed file boundaries
                - Client/server
        - Linux kernel started out using CVS
            - It didn't scale to lots of developers
            - Switched to Subversion
            - Switched again to BitKeeper (proprietary from SCCS, etc.)
                - Not open source, so controversial
        - Git arose when BitKeeper started costing money
            - Design a new system from scratch
            - Do the data structure first

<u>12/8: Git Internals</u>

- Git implementation/internals

- Versioning filesystems are built into the OS
    - Hard to change how they work - you must change the OS
- Git is all user-mode code
    - It's a program that you could've written given the time
    - You can evolve it without evolving the OS
- Linux was originally criticized for having too much in the kernel
    - To some extent, this criticism was right
- Built atop the Linux filesystem hierarchy (directories, files) atop compression (save space, can save time in some cases)
    - Brief aside about compression technologies (vast field)
        - 2 basic ideas used by Git:
            - Huffman coding - represent more-common symbols by shorter bit strings, rarer symbols by longer bit strings
                - Compress a string of symbols into a string of bits
                - Ex) Normal English has a lot of e's
                - ASCII uses up 7 bits for each char - let's represent E via a shorter bit string, etc.
                - Shortest way to represent English using this encoding? → look at probability of each English character occurring, build tree structure
                - Downside: assumes we have good estimates for probabilities of letters
                    - Cross-language probabilities will fall apart
                    - Static probabilities vs. dynamic probabilities
                - We can do better if we want to devote more resources to the problem
            - Dictionary coder - requires more memory in the sender and recipient
                - Create a dictionary of commonly-used words, then transmit and receive indexes into the dictionary
                - If sender and receiver have the same dictionary and words average more than 2 bytes each, this will be a win
                - Harder problem than Huffman coding - what's the best dictionary to use?
                - Static dictionaries vs. dynamic dictionaries
                - Dynamic - the sender builds the dictionary as content is processed, dictionary gets sent to recipient as it's updated
            - These 2 approaches can be combined
                - We can Huffman code the output of the dictionary coder
                - Approach is taken by gzip, zlib
                - Ordinary text in any language will compress well

            - Not so much for audio/video

- Git internals
  - Git history is stored into read-only “objects” files, which can be shared among repositories on the same machine
    - Speeds up git clone, which needn’t copy these files, it can just link to them
  - .git subdirectory
    - Branches directory - obsolescent
    - config - configuration information for this particular repository
      - Overrides your Git configuration in ~/.gitconfig
    - HEAD - tells you where you are
    - hooks - scripts that Git executes at crucial points (during a commit, etc.)
    - index - your proposed next commit (“staging area”)
      - “Future” of your current branch
      - Developers are not writing programs, they’re writing changes to programs
        - You want to make the best set of patches you can to improve software/explain the improvement
        - You want to use the staging area to not waste time
    - info/exclude - like .gitignore
    - logs - contains reflogs, records of where branch tips used to be/are. controlled by git configuration
    - objects - where the actual history is kept
    - packed-refs - optimized version of refs
    - refs - keeps track of where your branches are, as well as tags
- Git objects
  - Git repository is a user-mode filesystem built atop Linux/POSIX filesystem
  - It solves many of the same problems ordinary filesystems do
  - It also addresses problems specific to software development
    - Metainformation about changes to the software
    - Commits that atomically change lots of different files simultaneously
  - Simplest Git object - blob of bytes
    - Any sequence of bytes - Git doesn’t care
    - SHA-1 checksum - fingerprint for data, reliance on the hash being unique
  - Trees of objects
    - A Git tree is like a POSIX directory (blobs are like POSIX files
    - A tree contains a list of entries, each have a name (of the subtree/blob), type (blob or tree), mode (octal number that represents Linux permissions of the corresponding file), and SHA-1 hash of the entry

<u>12/10: Git Externals</u>

- blobs
  - Take the SHA-1 checksum of the blob
  - Compress the blob string using zlib (Huffman + dictionary coding)

- Write that compressed string into a file
- Branches
  - What branches are used for:
    - Mainline and maintenance releases
      - Merging maintenance fixes to master
        - You won't merge every fix - some of the maintenance fixes aren't appropriate
        - One convention (Emacs development) - "Do not merge to master" in a maintenance commit (skip this merge)
        - "Cherry-picking" or "backporting" from mainline to maintenance branches
    - Alternate visions of the future
      - Do we do A or B? Do both, see which is better
      - You can still do merging, cherry-picking, etc. here, but neither branch is "main"
    - Feature branches (more common, cheaper)
      - Let's implement feature A without worrying about feature B + vice versa
      - Teams work somewhat independently, whoever finishes first gets an easy merge into the master
        - Whoever finishes second may get a harder merge → must consider all interactions between A and B
        - Overhead → work of the second merge, some changes may collide
    - Forking because of disputes among developers
  - 2 ways to keep master consistent - merge and rebase
    - Merge makes connection to old versions much clearer
- How do branches work?
  - A branch is a lightweight moveable pointer to a commit
    - It can be changed to point towards a different commit
    - Normally moved by installing a new commit whose parent is pointed to by the branch
    - So, a branch keeps track of the latest commit in a logical sequence
- Merging
  - git merge X - merges (typically a branch) X into the current branch C
  - C and X will have a common ancestor A
  - First, we deduce A, then look at contents of A, C, X
  - Use this to infer the contents of the new commit that contains the changes in both C and X
    - Can do this by computing the difference between A and C and between A and X
    - Find the consensus between all 3 differences, marking merge conflicts
      - This is just a heuristic, just checking for textual overlaps
      - Conflicts fixed manually

        - This is just for a 2-way merge, can be generalized for an N-way merge
- Rebasing
    - git rebase X - prepare all the changes from common ancestor and X and apply them to the current branch
    - git rebase -i X - interactive rebase (lets you treat each change differently if you want)
    - Should never rebase a branch that is shared with other developers
    - Can split/merge patches to help understanding
- Git is technology for preparing patches
    - git diff should generate nice output because many developers review patches by reading them
- Remote repositories
    - Git is a distributed version control system
    - No single repository is in charge
    - Repositories for the same project can disagree or can become out of sync
        - This happens all the time
        - Use different branches to minimize this problem
            - Private branches in your own repository → can rebase whenever
            - When you're ready, install into master and push upstream
    - git remote - lists your remote repositories (can be on the same machine)
    - git remote -v - lists a lot more detail
    - git remote show origin - even more
    - git fetch REMOTE - fetch changes from the remote repository into yours, do not merge
    - git pull - git fetch; git merge
        - Merge can make lots of changes, may be dangerous
    - git push - push changes from current repository into the remote
        - Publishing your changes upstream