Lecture— “Building Software that Works ”

Real World Projects Specs are Elastic + Extras — evolves as you build it.

Unit 1 - Files, Editing, Shells

Software Quality Attributes

- **Real World Projects Constraints**
  - budget
  - customers
  - marketing (convincing your boss that this project is a good idea)
- **Project Constraints for class**
  - ***privacy & security—*** *dont build client server-app on the internet where everyone can see everyone elses stuff, needs some form of authentication and privacy in system*
  - ***performance—*** *not 23 seconds but 23 milliseconds*

- **Software Configuration Parameters:** Make sure your app is setup properly
- **Reliability:**
- **Security:**
- **Dependencies / Configuration:**
- **Understandability:** Writing understandable code will be easier to maintain, its hard to understand source code thats really long

- ***Abstraction:*** focus on higher-level problems by using simpler models of complex systems.
  - **Abstraction Types:**
    - **Data Abstraction**: defining data structures along with the operations that can be performed on them, hiding the implementation details.
    - **Procedural Abstraction:** A function name and its parameters provide a high-level interface, hiding the specific steps of the computation from the user.
    - **Control Abstraction:** hiding the flow of control in program execution and is often seen in control structures (like loops and conditionals) that abstract the details of how iteration or decision processes are implemented.

    - **Benefits:**
        - *Manageability*: Breaking down complex systems into manageable parts makes it easier to develop, understand, and maintain software.
        - *Reusability*: Abstract components like functions, classes, or modules can be reused in different parts of a program or in different projects, reducing duplication of effort.
        - *Scalability*: Abstraction allows developers to build on existing abstractions without necessarily understanding their inner workings, facilitating the development of more complex systems.
    - **Levels of Abstraction:**
        - assembly language abstracts away from machine code, high-level programming languages abstract further away from assembly, and software frameworks provide even higher levels of abstraction over programming languages.
    - **Balance**: Effective use of abstraction involves balancing the need for simplicity and manageability with the need for efficiency and control.

    - **eg., program like grep abstracts away from the complexities of this search algorithm, including the**
    - **text processing tools and programming languages provide functionalities to read from and write to files without needing to manage the complexities of the underlying file system or storage media.**

Memory

- Software attacks problems of persistence by tasking the user tell it what to do (copy from file, copy to a file)
- ***Persistent Data:*** Survives reboots (Flash Drive)
    - **Pros:**
        - Access to permanent storage, no data lass during crash
        - Survives reboots (Not on RAM)
    - **Cons:**
        - Slower than RAM
        - Flash drives are way slower than RAM (write out variable to drive each time you change the variable)
    - **Use:**
        - Data that doest get changed a lot
        - Important data should be persistent
    - **Terms:**
        - ***Persistent Variable*****:** has the same value when program exits and starts up

        - ***File*:** Persistent storage managed by OS
- ***Dynamic Data:*** RAM
    - **Pros:**
        - makes program run faster
        - its okay to lose dynamic data of a tight loop if system crashes
    - **Use:**
        - Data that changes all the time (can reclaim later)
        - dynamic data should be fast
    - **Terms:**
        - ***Buffer*:** Caches of files (sitting in RAM) “copy of file sitting in memory”
            - points to a file
            - save buffer: writes out buffer’s content —> replaces file’s content with buffer content (persistent)
            - read file: reads from a file into a newly created buffer (dynamic)
- ***Similarities***
    - Still allocating space if dynamic / persistent
    - Files and buffers are both sequences of character or byte sequences (encoded)
    - Char / Bytes are read sequentially (1 at a time)
        - ***byte data***: 8 bits, each character is represented as a sequence of bytes
            - ascii are single-byte
            - fancier characters are multibyte
            - binary files, applications interpret the binary data according to the file format and the type of data encoded.
            - binary format can represent a wide variety of data types, including executable programs, images, audio, video, and structured documents (like PDFs or Word documents). Each file format has a unique structure and set of rules for how the bytes in the file are organized and interpreted to represent the file's content.
            - encoding files allows for portability
            - The underlying representation for binary files is a sequence of bytes, where each byte is typically composed of 8 bits. Unlike text files, which are intended to store data that can be directly read and interpreted as characters by humans, binary files store data in a format that is intended for computer processing and may require specific software to interpret.

File System

files are just names linked to inodes
inodes consist of file descriptions and their content block pointers
blocks hold file contents

Memory > Controller (512 byte blocks) > O.S. Driver > Filesystem

- controller— responsible for exposing an interface of the respective memory module to the machine
- drivers— the way operating systems see and control storage, filesystems are the way they order it

Index: represents the metadata associated with a filesystem

- The sections, free or allocated to files, are also called blocks and represent physical parts of memory
- filesystem index organizes sections for fast and easy browsing

iNode: inodes consist of file descriptions and their content block pointers

- stands for index node
- Operating systems assign each inode a unique integer, so they can be used as keys to the inner filesystem structures
- points to the block which holds the file content
- glue between files the user sees and what the system actually stores about those files

Directory: Containers in a file system, store most objects including other directories

- a directory's blocks contain: a table with filenames and their associated inode numbers
- this is the only place filesystems store filenames (in symlinks too)
- each directory has the following unique associated information:
  - . (self inode reference)
  - .. (upper-level directory inode reference)
  - list of contained objects (files)
- we store most files inside directories
- Linux directories are a list of filenames to inode number mappings structure in the form of a tree

File: Persistent collection of data managed by OS

- Regular file inodes do not store their names, only their other metadata.
  - **Hard Links** create a new directory entry for a file, effectively creating

    another name for the same file on the filesystem.
        - represents another path for the iNode target
    - **Symbolic Links** (or symlinks) create a special type of file that contains a text pointer to the original file's location.
        - relative path in symlink: no hardcoded external file system structure
- Regular files have non-system contents, which we read and write via editors and other tools.

eMacs — "a programmable editor"

Why eMacs? — ***Introspective***[3]
Emacs is highly extensible and customizable, largely due to its integration with Emacs Lisp, a dialect of the Lisp programming language. This integration allows users to script and modify virtually every aspect of Emacs's behavior, making it more than just a text editor but a powerful computing environment. By providing a powerful, high-level programming language within the editor, Emacs allows users to tailor the software to their specific needs, automate virtually any task, and extend the editor in limitless ways. This has led to the development of a rich ecosystem of packages and extensions, contributed by a vibrant community of users and developers.

    - **Automating Repetitive Tasks**: Users can write scripts to automate almost any task in Emacs, from formatting text to organizing files.
    - **Creating Custom Key Bindings**: Emacs Lisp can be used to create custom keybindings for executing specific tasks or functions, allowing users to tailor their workflow.
    - **Developing Major and Minor Modes**: Major modes provide specialized editing environments for different types of files, while minor modes provide additional functionality that can be toggled on or off. Users can create their own modes to support new file types or editing techniques.
    - **Interacting with External Programs**: Emacs Lisp can call external programs and process their output, enabling integration with other tools and services.

- **text editor + ide**

- **written in low level**— partly in C but mostly in eMacs lisp
- ***keystroke***: turns into a function call

- **REPL (read>eval>print>loop)**— evaluates the function your char told it to evaluate, displays it on the screen and waits for you to type again

  1. Read[1]: It looks at the text of the function you wrote, like (add 1 2).
  2. **Tokenize**: It breaks down the text into understandable parts: (, add, 1, 2, and).
  3. **Build a List**: It puts these parts into a list structure in its memory, like a neat row of boxes, each holding a part of your function.
  4. **Evaluate**[2]: It understands what you're asking (to add 1 and 2) and performs the action, giving you the result.
  5. **Loop**[4]

- **self-documenting:** *[C-h k]* gives us sort of a summary of all the arguments that you can pass the previous line, this keystroke gives you window into source code to program the keystroke to do things— give me help about a key
- **Modeful Editor:** action it takes when you type a char depends on the context (mode)"
  - ***Modes**: affects how eMacs will behave*
    - **text**—
    - **C++**— will nicely indent according to the syntax rules
    - **Elisp**— editing eMacs lisp files
    - **Shell**— subsidiary shell in eMacs [M-x shell] — eMacs controls new shell, old shell in control of eMacs. New shell and emacs run in parallel, while old shell is waiting for eMacs to exit.
    - **fundamental**— text mode but even simpler (*text*--)
      - cant format paragraphs, doesnt think it's text
    - **dired**— directory editor mode, editing an actual directory
      - each line in current window stands for a file in directory
      - 'D' — short for delete the file in current dir
    - **Can combine modes**—
  - eMacs guesses the mode from the files content
- **one Emacs window—** is the *selected window*; the buffer this window is displaying is the current buffer
- ***Commands:***
  - **[C-h]—** help command
  - ***[C-h-m]***— explains mode, and lists the keybindings

    - ***[M-x mode]****— switch to certain mode*
    - ***[M-! CMD RET]****— run this one shell command*
    - ***[M-| CMD RET]****— run this shell command with current region as input*
    - ***[C-h f]***— describe-function
    - ***[C-x C-b]***— list buffers
    - **[M-x keymap-global-set RET key cmd RET]— set**
        - Keys with modifiers can be specified by prepending the modifier, such as ‘C-’ for Control, or ‘M-’ for Meta. Special keys, such as TAB and RET, can be specified within angle brackets as in TAB and RET.
- ***Keystroke Functions:***
    - (self-insert command)— most characters you type into emacs are bound to this command
    - some special characters bound to something else: TAB and RET
- **Region:**
    - ***point***: where the cursor is
    - **mark:** some other position in the buffer
    - **current region:** everything betweene point and mark
    - **mode line:** *cs*:*ch-fr buf   pos line*   (*major minor*)
        - *cs*— character set of text in buffer,
            - ‘-‘ —indicates no special char set handling
            - ‘=‘ —no conversion whatsoever, for files containing non-textual data
            - *other chars* represent various coding systems, ‘1’ represents ISO latin-1

eLisp

the Lisp source code and the data structures it manipulates share the same format.

- (+ 2 3) is both a piece of code that adds numbers and a list containing three elements.

    - Alright, let's pretend I'm Elisp, and you've just asked me to evaluate an expression, like (+ 2 3). Here's what I'd say:
    - "Okay, I see you've given me (+ 2 3). Let's break it down!
        - **Seeing the List**: First, I notice this is a list because it's inside parentheses. Lists usually mean you're asking me to do something.
        - **Finding the First Thing**: I look at the first thing in the list, which is

        a +. Oh, that's an instruction for me to add numbers together! When I see the + symbol at the start of the list, I remember that it's a special instruction that means 'addition'. It's like when you see a '+' sign in math class, and your teacher says it means to add numbers together. I have a list of these special instructions and what they do, so I know + means to add.
    - **Looking at the Rest**: Next, I see there are two numbers, 2 and 3. You want me to add these together.
    - **Doing the Math**: I know how to add! So, I take the 2 and the 3 and put them together. That makes 5.
    - **Giving You the Answer**: I've done the math, and the answer is 5. So, I tell you that (+ 2 3) equals 5.

- Since code is just a list, you can write Lisp code that generates, modifies, or analyzes other Lisp code.
- interpreter doesn’t need to transfrom to transform source code into a different internal representation before executing**[2]**

- Emacs Lisp primarily operates as an interpreted language within the Emacs environment, meaning that Emacs reads and executes Elisp code directly without the need for a separate compilation step. However, Emacs also supports byte-compiling Elisp code, which is a form of compilation.

    - **.el files[2]** are written in Emacs Lisp (Elisp), which is a dialect of the Lisp programming language specifically designed for the Emacs text editor. Emacs Lisp is used for scripting, customizing, and extending Emacs, allowing users to modify and enhance the editor's functionality according to their needs.
    - **Direct Execution[2]**: When you load an Emacs Lisp file (.el file) into Emacs, the Lisp interpreter within Emacs reads and executes the code directly. This mode of operation is typical for scripts and configurations that are loaded or evaluated while Emacs is running.
    - **Byte-Compiled Files**: Elisp can be compiled into bytecode, which is a more efficient, lower-level representation of the code. Bytecode files use the ***.elc extension***.
    - **Performance**: Byte-compiling Elisp code can significantly improve performance, especially for code that is executed frequently. The bytecode is executed by Emacs's byte-code interpreter, which is faster than interpreting the plain text .el files.
    - **Compatibility**: Emacs automatically chooses to load the .elc version

of a file if it is available and newer than the corresponding .el file, ensuring that users benefit from the optimized bytecode without having to manage which version of the file is loaded.

- **Parsing:** When Emacs parses an expression, particularly in the context of evaluating Emacs Lisp (Elisp) code, it internally uses a data structure known as a **list** to represent the syntactic structure of the expression. This list structure is intrinsic to Lisp languages, including Elisp, and reflects the source code's nested, parenthetical nature directly.
    - **List Representation**
        - **Trees and Lists**: In Lisp, code and data are represented using the same structure: lists. These lists can naturally form tree structures, where each list element can itself be another list. This property is especially useful for representing expressions that have nested subexpressions, as is common in arithmetic operations and function calls. *The list structure serves both as a linear collection and as a way to represent nested expressions, effectively acting like a tree*.
        - **Example**: Consider the Elisp expression (+ 1 (* 2 3)). When parsed, this expression is represented internally as a list that looks like (+ 1 (* 2 3)), where the outer list contains the operator + and its operands 1 and another list (* 2 3). This inner list represents the multiplication operation.
    - **Cons Cells**
        - **Implementation Detail**: The fundamental building block of lists in Lisp (including Elisp) is the cons cell (short for "construct" or "constructor cell"). A cons cell is a data structure with two parts, traditionally called car and cdr (originally from "Contents of the Address part of Register number" and "Contents of the Decrement part of Register number", terminology from the IBM 704 machine). The car stores the value of an element, and the cdr points to the next cons cell in the list (or nil if it's the end of the list).
        - **List Construction**: A list is a chain of cons cells, where each cell's car part holds an element of the list, and its cdr part points to the next cell. The last cell's cdr points to nil, marking the list's end.
    - **Abstract Syntax Trees (AST)**
        - While Lisp lists naturally represent both code and data, and are used internally by Emacs to parse expressions, the concept of an Abstract Syntax Tree (AST) is also relevant in many parsing contexts. An AST is a tree representation of the abstract syntactic structure of code. In Lisp-like languages, the list representation

serves a similar purpose to an AST, directly mapping the structure of the code.
  - In summary, when Emacs parses an expression, it uses lists, constructed from cons cells, to represent the syntactic structure. This representation is deeply integrated into the design of Lisp languages, allowing for a seamless transition between code as data and data as code, a hallmark feature known as homoiconicity. In Lisp, code and data share the same structure, a property known as **homoiconicity**. This means that the parsed representation of code (in lists) is directly manipulable using the language itself.

- **Arithmetic Intepretation:**
  When evaluating large floating-point numbers in Emacs Lisp (Elisp), Emacs handles them according to the limitations and capabilities of its floating-point representation, which is typically based on the IEEE 754 standard, similar to many other programming environments

  - **Parsing the Elisp expression** to understand its structure.
  - **Looking up and preparing the Elisp function** for execution.
  - **Evaluating arguments** and executing the arithmetic operation, either directly in interpreted mode or via bytecode.
  - **Translating the operation into C function calls**, which are compiled into machine code.
  - **Executing the compiled machine code** on the computer's hardware to perform the arithmetic.
  - **Returning the result** back to the Elisp environment.

  - **1. Elisp Code Parsing:**
    When you execute an arithmetic expression in Elisp, the Emacs interpreter first parses the expression. This step involves
breaking down the Elisp code into its constituent elements, such as functions and operands.
    - *eg,. the expression (+ 1 2) is parsed to identify the + function and its arguments 1 and 2.*

  - **2. Elisp Function Lookup:**
    After parsing, eMacs looks up the Elisp function corresponding to the arithmetic operation. Each arithmetic operator
    (like +, -, *, /) is implemented in Emacs as a function that can operate on numbers. Emacs knows how to handle these functions and what they are supposed to do.

- 3. **Argurment Evaluation:**
  The arguments of the arithmetic function (the numbers to be operated on) are evaluated. In the case of simple literals (like 1 and 2), this step is straightforward. However, if the arguments are themselves expressions, Emacs evaluates them recursively.

  4. Execution of the Arithmetic Operation
  Interpreted Mode: In interpreted mode, the Elisp interpreter directly executes the operation using its own internal routines. These routines translate the Elisp arithmetic operation into a series of lower-level operations that the Emacs interpreter can execute directly, often relying on the C programming language and the capabilities of the underlying hardware.
  Byte-Compiled Code: If the Elisp code has been byte-compiled, the arithmetic operation may be represented in Emacs's bytecode. The Emacs bytecode interpreter then executes the operation. Bytecode execution is faster than interpreted execution because it is closer to machine code, though still a level of abstraction away.

- **5. Translation to Machine Code**
  The execution of arithmetic operations at the interpreter or bytecode level involves calling functions written in C (as the Emacs source code is largely written in C). These C functions are compiled into machine code by a C compiler when Emacs is built from its source code.The C compiler optimizes the arithmetic operations for the target architecture, translating them into machine code that the computer's processor can execute directly. This machine code is what actually performs the arithmetic operation on the hardware level.

- **6. Result Handling**
- Finally, the result of the arithmetic operation is handled by Emacs. It is converted back into a form that can be represented in Elisp (if necessary), and then returned to the Elisp environment where it can be used by other parts of the program.

- **Macros:** writing code that transforms Lisp expressions into new Lisp expressions

Legend:

[1]: text processing— eMacs operates with a higher-level abstraction, focusing on characters and their semantic meaning, regardless of their underlying byte representation. When reading files or receiving input, Emacs decodes the byte sequences into its internal character representation, taking into account the file's or system's specified encoding (e.g., UTF-8). When writing files or sending output, Emacs encodes the internal character representation back into the appropriate byte sequence according to the target encoding.

- ◇ ***Gap Buffer:*** Dymanic array (single contigous block of memory for its elements) allowd for efficient indexing (direct access to any elements)
- ◇ gap: buffer array includes a gap of unused space that can be moved an resized as text is inserted and deleted, gap is near the current point of editing
- ◇ reduces the amount of memory movement for text insertions and deletions
- ◇ text insertion involves moving the gap > adding new text to the gap space> adjusting the gap’s size accordingly
  direct indexing for read operations

[2]: evaluate— The process of parsing in Emacs, especially regarding Emacs Lisp (Elisp) code, fundamentally relies on the list structure, which is a form of a singly linked list made up of **cons cells.**

- **Singly Linked Lists (Cons Cells)**: Each element in a Lisp list is a cons cell, a two-part structure where the car part holds a value and the cdr part points to the next element in the list. This forms a singly linked list, which, due to Lisp's syntax, can also represent tree structures.

  - When an **Emacs Lisp (.el) file evaluates a list**, it's essentially reading and executing the instructions contained in that list. Here's a simplified breakdown of what happens:
    - **Reading the File**: Emacs loads the .el file and reads its contents, which are written in Emacs Lisp code.
    - **Parsing the Code**: As it reads, Emacs parses the Lisp code, breaking it down into understandable elements or tokens. In the case of a list, it recognizes the list structure, which is delineated by parentheses ( ) and contains elements separated by spaces.
    - **Constructing Internal Representations**: The parsed code is then transformed into an internal representation, which in Lisp is typically a list structure in memory. This structure precisely mirrors the hierarchical and nested nature of the code written in the file.
    - **Evaluating the Lists**: Emacs then evaluates the lists according to

Lisp's evaluation rules. For a list, the first element is generally treated as a function or operator, and the subsequent elements are treated as arguments to that function. Emacs looks up the function (which could be a built-in function, a user-defined function, or a special form like if or let) and applies it to the arguments.

    - **Function Calls**: If the list represents a function call, Emacs executes the function with the given arguments.
    - **Special Forms**: If the list represents a special form, Emacs processes it according to the rules of that form, which may involve conditional evaluation, looping, variable binding, etc.
    - **Macros**: If the list corresponds to a macro, Emacs first expands the macro (i.e., it transforms the macro call into more Emacs Lisp code according to the macro's definition) and then evaluates the resulting code.
- **Producing a Result**: The evaluation of a list results in a value. This could be the return value of a function, the result of a computation, or the outcome of a conditional expression, among other possibilities. This value can then be used by other parts of the program, displayed to the user, or simply ignored if it's not needed.
- **Continuing the Process[4]**: Emacs continues to read, parse, and evaluate the rest of the Lisp code in the file, proceeding through the file's contents in a linear fashion but respecting the nested structure of Lisp expressions.

[3]: introspective— Emacs's introspective capabilities stem from its design and implementation, particularly through Emacs Lisp (Elisp), its extension language. Introspection in computing refers to the ability of a program to examine and modify its own structure and behavior at runtime. Emacs exhibits introspection in several ways:

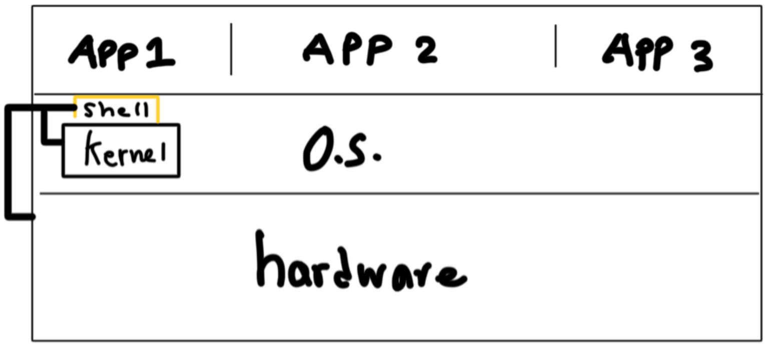
1. **Self-Awareness**: Emacs can query its own state and configuration. For example, it can list and describe all currently available functions, variables, keybindings, and active modes. This allows users and developers to understand how Emacs is currently operating and to make informed decisions about how to extend or modify it.
2. **Dynamic Evaluation**: Emacs can evaluate Elisp expressions on the fly. This means you can write code that examines aspects of the editor, modifies its behavior, or even generates and evaluates new code during an Emacs session. This dynamic evaluation capability is a powerful tool for introspection and customization.
3. **Extensible and Customizable**: Virtually every aspect of Emacs can be customized or extended through Elisp. Users can write scripts to modify editor behavior, add new features, or integrate with external tools and systems. This flexibility is possible because Emacs is designed to be introspective, allowing

scripts to access and modify the editor's internals.

4. **Documentation Access**: Emacs has extensive built-in documentation accessible through commands like describe-function, describe-variable, and describe-key, which allow users to access information about the editor's functions, variables, and keybindings directly within the editor. This is a form of introspection, as Emacs provides detailed information about its own components.
5. **Interactive Development Environment**: The Emacs environment supports interactive development and debugging of Elisp code. Developers can inspect the state of the program, evaluate expressions, and modify code in the middle of an Emacs session, which is a testament to its introspective nature.
6. **Self-Modifying**: Emacs can modify its own operation based on user input or conditions detected at runtime. For example, it can change keybindings, alter functionality, or load additional packages as needed, all based on introspective examination of its own state and configuration.

Shell

Computer System:



- **Operating System**: runs on top of hardware. Has everything you need in terms of software to run your computer and be a reasonable platform for all your applicaations to run .this design makes o.s. get bloated with new lines of source code. understanding and performance issues
- **Hardware:** x86_64, ARM 3, etc.
- **O.S. Kernel**— strip down the O.S. to kernel (small as we can make it). sits in machine always running.
  - “the boss, the master control program”
  - in charge of everything
  - can kill of shell and programs
- **Needed parts of O.S not in kernel:**

    - bin.sh— implements the shell, needed becuase shell is what starts stuff up in O.S.
    - libc.so— C library, implements low level functions like issuing kernel call fx in shell
- **Apps:** sit above O.S. and hardware — talking both to the harware and the kernel
    - mostly executing machine instructions—>hardware
    - consults with kernel—> reading input from standard data (hardeware doesnt know how to do that)

The Idea of the Shell— a thin layer sits atop the kernel & hardware, talks to both.

- Lets you type commands at it > figures out what kernel API part you want to use > does it > outputs the results.
- Simple program that just reads command and implements them using low level system calls
- relies on o.s. kernel to get work done
- use shell application to setup other applications and have them run (metaprogram)
- someone has to tell your computer I want this program to startup
- shell executes machine instructions to figure out what system call to make
- when command is running its in control, shell waits for app to exit to take control

- **sh.c source code:** $ sort

```
while (true) { # loops until exit

c = read (stdin) # read command from standard input
if (c = ("exit") break)
execlp ("usr/bin/sort", "sort", 0 ) # issue low level kernel
calls to do the command
# the kernel will arrange for the sort porgram to run
# shell waits for sort to finish before it does anything else
#REPL

}
```

- ***REPL Environment (read>eval>print>loop):*** Waits for you to type that line and that executs that line, eMacs is one character at a time.
    - The default behavior of shell environments, such as Bash, Zsh, or others found in Unix-like systems, is to process text input primarily by bytes. This approach is deeply rooted in the Unix philosophy, where the basic unit of file and stream manipulation is the byte. However,

    the distinction between processing by bytes and by characters becomes significant when dealing with multibyte character encodings, such as UTF-8.
  - ***Text Processing***
    - Shell environments are traditionally command-line interfaces (CLIs) that are designed for executing commands. These commands often require arguments and options to be specified together. Therefore, shells use a **buffer** to store the characters of a command line until the Enter key is pressed. This behavior is crucial for command parsing, where the command and its arguments are tokenized based on spaces or other delimiters. The entire line is treated as a single string (or an array of strings when parsed), which is then evaluated as a command.

- ***file metainformation [ls -l]:***
  - ***Filetype:***
    - ***'-' : regular file***
    - ***'d': directory***
    - ***'l': symlink***
  - ***Permissions:*** *9 characters long,*
    - *charachters represent 9 bits of a number. Group of 3 bits representing permissions for the owners(users), groups, and others.*
      - *eg,. 110100100 = rw-r—r— (commmon for temp files)*
        - *owner can read and write the file but cannot execute it (rw-)*
        - *other people in the owners group can read the file but cannot execute it (r—)*
        - *anybody elsse can read the file but not execute it*
  - ***last modified time***: old files only list day/month/year
  - **file size:** size of the file in bytes, how many btes the file takes up
  - **user:** name of owner
  - **group:** name of owner's group
- **byte convention:**
  - **' . ' : is the file name that is the current directory**
  - **' .. ' : the file name that is the current directory**

In what sense does the shell "glue" smaller programs together? Which parts of the shell language make it especially suitable for this purpose (compared to something like Python)?

The shell, particularly Unix-like shells such as Bash, is often described as "gluing" smaller programs together because it **excels at combining simple, single-purpose commands and utilities into more complex workflows or processes.** This capability is central to the Unix philosophy of building small, modular tools that do one thing well and then combining those tools in various ways to perform complex tasks. Here's how the shell achieves this and why it's particularly suited for such tasks, especially compared to more general-purpose programming languages like Python:

Control Flow:

- **Piping (|):**
    - **Mechanism**: The pipe operator allows the output of one command to be used as the input to another command. This enables chaining commands in a way that the output of one is seamlessly passed to the next.
        - **Example**: grep 'pattern' file.txt | sort | uniq searches for 'pattern' in file.txt, sorts the occurrences, and filters out duplicates.
- **Redirection (>, >>, <):**
    - **Mechanism**: Redirection operators are used to direct the output of commands to files, or to read input from files, instead of standard input/output streams.
        - **Example**: echo "Hello, World!" > hello.txt writes "Hello, World!" to hello.txt.
- **Command Substitution ($(...))**
    - **Mechanism**: Allows the output of a command to be substituted in place, where a command can be used as part of an argument to another command.
        - **Example**: echo "Today is $(date)" includes the output of the date command in the message echoed to the screen.
- **Background Execution (&)**
    - **Mechanism**: Allows commands to run in the background, freeing the terminal for other tasks.
        - **Example**: long_running_process & starts long_running_process in the background.
- **Scripting and Control Structures**
    - **Mechanism**: Shell scripting allows for loops, conditionals, and functions, enabling complex logic and flow control in combining commands.
        - **Example**: Using loops to process files or conditionals to execute commands only if certain conditions are met.
        - do-while:
            - **do:** This marks the beginning of the loop's body, where the

        script processes each line
        - **case:** inititates a case statement
        - **esac:** This closes the case statement. It's the word case spelled backward,
        - **done < myfile.txt:** This ends the while loop and redirects the contents of myfile.txt to be the standard input for the while read command. This means that instead of reading from the keyboard, the loop will read the lines of

- **Why Shell is Particularly Suitable Compared to Python:**
    - **Conciseness**: Shell syntax is designed for concise specification of command execution and composition. Tasks that require multiple lines of code in Python can often be accomplished with a single line in a shell script.
    - **Built-in Operations**: Many of the operations needed to glue programs together (like piping and redirection) are built directly into the shell's syntax, making them more straightforward to use compared to Python, where additional modules or more verbose syntax might be needed.
    - **Environment Integration**: The shell is tightly integrated with the Unix/Linux environment, making it easy to invoke system utilities, manipulate the filesystem, and manage processes. While Python can do these things, it often requires additional modules and more boilerplate code.
    - **Interactive Use**: The shell is designed for both interactive use and scripting, allowing users to quickly test commands in an interactive session before scripting them. Python is primarily a general-purpose programming language, and while it can be used interactively, its design is more focused on writing and executing complete programs.

In summary, the shell's design and features make it exceptionally well-suited for gluing smaller programs together to create complex workflows, particularly for tasks related to system administration, process management, and file manipulation.

Test Command: ‘ [’

- most often invoked by the if statement: if [ ]

The **test** utility evaluates the expression and, if it evaluates to true, returns a zero (true) exit status; otherwise it returns 1 (false). If there is no expression, **test** also returns 1 (false).

Commands

ls - l	List all files and details in current directory. Details are metainformation of the file, which is other associated file information thats not part of the contents.
^D	end of file, “no more input.” will log shell out if given at prompt (shell was reading from the file)
^C	terminate currently running process, gives prompt ($) back
^Z	suspends current process (shell goes catatonic), process is frozen. gives prompt ($). SIGSTOP until SIGCONT
fg *optional* %n	brings suspeneded background process back up, ‘foreground.’ n is process# listed by command jobs
jobs	list all suspended processes
ps	process status — lists all processes, running programs attached to terminal, alot of options to find out processes you want
kill PID	issues a kernel call to kill process (like typing ^C at that process)
command —help	reminds you of options you forgot about
info CMD	large documentation on CMD
ls -i	shows inode #

- **gives prompt ($) back—** brings control back to shell
- **end of file—** whatever terminal is reading from the program will get end of file.
- **ps columns—** (PID) (CPU TIME) (CURRENT CMD RUNNING) — # of column depends on flags
- **process—** running program
- **program—** static notion (file sitting in a directory)
- **signal flavors—** used by user and kernel to control program
  - INT 3 interrupt signal
  - KILL 9 kill signal

- TERM 15 termination signal
- HUP 1 hangup signal, sent to program when ssh session dies and program still running
- STOP 17 stop the process
- CONT 18 resume execution if stopped
- SIGIO 29 I/O now possible

Chmod: In the context of the chmod command in Unix-like operating systems (Linux, macOS, etc.), octal numbers are used to represent file permissions. These permissions determine who can read, write, or execute a file or directory. Octal numbers are a concise way to specify these permissions using a three-digit code, where each digit represents a different aspect of the permission set.

- 4: Read permission
- 2: Write permission
- 1: Execute (or search) permission
- ex:
  - Read permission only: 4
  - Write permission only: 2
  - Execute (search) permission only: 1
  - Read and write permissions: 4 + 2 = 6
- To give read and write permissions to the owner and read-only permissions to the group and others for a file, you would use the octal code 644:

```
chmod 644 myfile.txt
```

2>&1

```
grep "[^0-9]" > /dev/null 2>&1
```

This part redirects the standard error (stderr, file descriptor 2) to the same destination as the standard output (stdout, file descriptor 1). &1 references the target of the standard output redirection

Interpretation:

- Double quotes allow variable expansion and interpret some escape sequences, whereas single quotes preserve the literal value of the enclosed text.

System runlevels are a concept used in Unix-like operating systems to represent different states of the system, each with a specific purpose and set of processes and services that are allowed to run. They provide a straightforward way to organize and control the behavior of a system during startup, operation, and shutdown. Here's a brief overview of the standard runlevels and their typical uses:

- **Runlevel 0**: System halt or shutdown. Used to safely turn off the system.
- **Runlevel 1** (or **S** or **s**): Single-user mode. Only the root user can log in, and network services are not started. This mode is used for maintenance tasks.
- **Runlevel 2**: Multi-user mode without networking. Allows multiple users to log in but does not configure network interfaces and does not start network services.
- **Runlevel 3**: Full multi-user mode with text-based interface. Networking is enabled, allowing multiple users to log in over the network. This mode does not start the graphical user interface (GUI).
- **Runlevel 4**: Undefined/unused. Reserved for user-defined or system-specific needs.
- **Runlevel 5**: Full multi-user mode with graphical interface. Similar to runlevel 3 but starts the GUI by default, allowing users to log in via graphical login prompts.
- **Runlevel 6**: System reboot. Used to safely restart the system.

Positional Parameters ($N): Positional parameters are variables in a shell script that store values passed to the script as arguments, and you can access and manipulate them using the described syntax and built-in commands.

- A positional parameter is a variable that is represented by one or more digits, except for the single digit 0.
- These parameters are initially assigned values from the arguments passed to the shell when it is invoked.
- You can reference positional parameters using the notation \`${N}\`, where \`N\` represents the position or index of the parameter. For single-digit parameters, you can use the shorthand notation \`$N\`.
- Positional parameters cannot be assigned new values using assignment statements; they are typically modified using specific built-in commands like \`set\` and \`shift\`.
- The \`set\` and \`shift\` built-in commands are used to set and unset positional parameters. \`set\` is used to assign values to them, and \`shift\` is used to shift their values.
- When a shell function is executed, the positional parameters are temporarily replaced with the function's arguments, if any.
- If you want to expand a positional parameter that consists of more than a single digit, you must enclose it in braces \`{}\` to ensure the correct parameter is referenced

- example:

```
$ ./script.sh John 25
```

Script:

```
#!/bin/bash

# Check if there are at least two arguments
if [ $# -lt 2 ]; then
    echo "Usage: $0 <name> <age>"
    exit 1
fi

# Assign positional parameters to variables
name=$1
age=$2

# Display the values of the positional parameters
echo "Name: $name"
echo "Age: $age"

# You can also use the shorthand notation
echo "Using shorthand notation:"
echo "Name: $1"
echo "Age: $2"


[ output: ]

Name: John
Age: 25
Using shorthand notation:
Name: John
Age: 25
```

  - $# represents the number of positional parameters passed to the script.
  - We check if there are at least two arguments provided.
  - We assign the first and second positional parameters to the name and age variables, respectively.
  - We then display the values of these variables as well as using the shorthand notation to directly access the positional parameters.

Regular Expressions— Strings that can be matched

- string is a sequence of chars
- *Pros:* maximize the power of the searching capabilities, while keeping the SEARCH programs small and fast

Basic Regular Expressions (BRE)*[1]*

Extended Regular Expressions (ERE)*[2]*

Extended Regular Expressions add additional functionality to BREs for enhanced pattern matching without needing to escape certain characters for special functions.

Symbols

.*	any sequence char ***except newline***
P_1P_2	concatenation of single character regular expressions eg. ‘abc’ (any string S_1S_2 such that S_1 matches P_1 and S_2 matches P_2)
P*	matches 0 or more occurrences of p
P_1 \\| P_2	anything matched by p1 or matched by p2
P+	matches 1 or occurences of p
\(P1)	any match of p
[abc]	any single character in a set
[a-eq-z0-9]	matches letters from A through E, all the letters from q through Z, and all the characters from zero through nine. So this matches any digit and any lowercase letter except the letters F through P. ‘-‘ (ascii minus)
[aeiou]*	match any sequence of lower case values
\\\\	\\
[^a-e]	negate charachter set, match any single char thats not a -e

[^0-9]	match any single char thats not a digit (will match a newline,*not a digi*t)
[^a-z]	
[^^]	any char but ^
[^-]	any char but ‘^-‘
[ ]a-z]	contains ‘[‘ ‘a-z’ (has to be this] convention)
[^]a-z]	matches everything but ‘[‘ ‘a-z’
[]	
^p	match only at the start of line
p$	has to match at the end of a line
[[:alpha:]]	match any single alphabetic character (character sets)
[[:ascii:]]	match any single ascii in the set of 128 ascii chars (0-127)
[^a-e]	[ABCDE…Z]
[a^]	matches either ‘a’ or ‘^’ (^ is special anywhere but the end)
[]az-]	putting it at the end treats ‘-‘ as literal (-)
[014-9]	all numbers from 0-9 except 2 and 3
emacs: [a^Q^J]	a or newline in emacs

- **‘space’** — is significant, if you put a space, space will be searched
- ‘**.***’ — matches any char but a new line
- newlines are still matched in [^a-z0-9A-Z]
- **‘ \ ’** has no special character inside a char set
- ‘**[[:set:]]’** — thisis a more compact notation as opposed to typing out all 128 ascii chars for example
- **[ ]** — char set
- **‘^’** — circumflex. “if you’re in the middle of the line you cant possibly match at the start of the line right after”
- **‘\n’** — is a char
- The] is considered metacharacter **only when it is used to close character set** [...]. If before] there is no unclosed and unescaped opening square bracket [then] is as simple literal which doesn't require escaping (but *allows* it, which is why your IDE gives you "*warning*" instead of *error*).
- ^Q — quote next char in emacs regex search

- **Terms**
    - ***interactive***: as you type your regexp its constantly searching for every expression you found
    - ***failed search:*** no string (sequence of chars) found from current position to end of buffer
    - ***fixed string*****:** string literal search, no regexp symbols — helpful for finding symbols without treating it like a regexp
    - ***longest-leftmost:*** "leftmost longest" or "first match longest" match rule ("longest leftmost match" principle).
    - “ 1. First Match, 2. Longest Match”
        - **First Match**: When performing a search with a regex, the engine starts at the beginning of the text and scans towards the end, looking for a section of the text that matches the pattern defined by the regex. The first portion of text that matches the pattern is considered the first match. This is the "leftmost" part because regex engines typically operate from left to right in the string.
        - **Longest Match**: Among potential matches that start at the same position in the text, the regex engine will prefer the one that consumes the most characters, or the "longest" match. This preference exists because regex engines typically employ a greedy matching strategy by default, where they try to match as much of the text as possible before considering a match successful.
            - eg., consider the text "aaa" and the regex pattern a+. Here, a+ means "one or more a characters."
                - **First Match**: The regex engine starts at the first a and notes that it matches the pattern.
                - **Longest Match**: Since the pattern a+ is greedy, it doesn't stop at the first a. Instead, it continues to consume the second and third a characters because they also match the pattern, resulting in the longest possible match at the leftmost position.

Legend:
[1]:

- **^**: Matches the beginning of the line.
- **$**: Matches the end of the line.
- **.**: Matches any single character.
- **[...]**: Matches any one of the enclosed characters. A hyphen (-) can specify a range.
- **[^...]**: Matches any one character not enclosed.
- \: Escapes the next character, indicating it should be treated literally, or enables special sequence if it is a special character.
- *****: Matches zero or more occurrences of the previous character.

- **\{m,n\}**: Matches from m to n occurrences of the preceding element.
- **\{m,\}**: Matches m or more occurrences of the preceding element.
- **\{m\}**: Matches exactly m occurrences of the preceding element.
- **\{0,1\}** or **\?** (in some implementations): Matches 0 or 1 occurrence of the preceding element. Not part of the original POSIX BRE but commonly used.
- **\(... \)**: Groups expressions as a single element and captures the matched substring for backreferencing.
- **\n**: Refers to the nth captured substring in the pattern.

[2]:

- **+**: Matches one or more occurrences of the previous character (ERE).
- **?**: Matches zero or one occurrence of the previous character (ERE).
- **|**: Indicates an alternation (logical OR) between patterns (ERE).
- **()**: Groups expressions without needing backslashes (ERE).

Common Characters and Sequences

- **\d**: Matches any digit (0-9). Not part of POSIX but widely used in other regex flavors.
- **\w**: Matches any word character (alphanumeric characters plus underscore). Not part of POSIX but commonly used elsewhere.
- **\s**: Matches any whitespace character (spaces, tabs, etc.). Not in POSIX but widely used.

POSIX Character Classes

For locale-aware character matching, POSIX defines several character classes that can be used within square brackets:

- **[:alnum:]**: Alphanumeric characters
- **[:alpha:]**: Alphabetic characters
- **[:cntrl:]**: Control characters
- **[:digit:]**: Digits
- **[:graph:]**: Visible characters (not space)
- **[:lower:]**: Lowercase letters
- **[:print:]**: Visible characters and spaces
- **[:punct:]**: Punctuation characters
- **[:space:]**: Whitespace characters
- **[:upper:]**: Uppercase letters
- **[:xdigit:]**: Hexadecimal digits

Dictionary:

Buffer**:** Caches of files sitting in RAM, “copy of file sitting in memory” that points to a file *[p. Memory]*

Byte Data**:** 8 bits, each character is represented as a sequence of bytes *[p. Memory]*

Control Flow[1]: Control flow in programming refers to the order in which individual statements, instructions, or function calls are executed or evaluated within a script or program.

Dynamic Data: RAM *[p. Memory]*

File**:** Persistent storage managed by OS *[p. Memory]*

Locale:
- 'ascii' : upper case comes before lower case

Parsing: Parsing is the process of analyzing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar. The goal is to determine the syntactic structure of the input, often converting it into a tree-like representation that shows the hierarchical relationships between elements

Persistent Data: Survives reboots (Flash Drive) *[p. Memory]*

REPL Environment: The Read-Eval-Print Loop (REPL) is a simple, interactive programming environment that takes single user inputs (reads), executes them (evaluates), and returns the result to the user (prints). This process repeats in a loop (hence the name). Both Emacs and shell environments use this concept, but they apply it in slightly different contexts**.** Both environments use the REPL process to facilitate interactive computing, but they do so in the context of their respective domains: Emacs Lisp for extending and interacting with the Emacs editor, and the shell for interacting with the operating system and executing commands. *[p. eMacs] [p. Shell]*

- **eMacs Lisp REPL:** Emacs is a highly extensible text editor, and its extension language is eMacs Lisp. The Emacs environment includes an interactive Emacs Lisp REPL, typically accessed through the ***scratch*** buffer or via the command **M-x ielm** for an interactive Emacs Lisp mode.
    1. **Read**: When you type an expression in the *scratch* buffer or in the ielm mode and press Enter (or use C-j in the *scratch* buffer), Emacs

reads the Lisp expression.
2. **Evaluate**: Emacs Lisp evaluates the expression according to its rules. This might involve calculating a value, modifying a data structure, executing a function, etc.
3. **Print**: The result of the evaluation is then printed in the buffer, directly after the expression or in the minibuffer.
4. **Loop**: The environment waits for you to input the next expression, and the process repeats.

- **Shell REPL**: The shell (such as Bash, Zsh, or Fish) operates as a command line interface for interacting with the operating system. Here, the REPL process is more about executing commands and scripts rather than evaluating expressions in a programming language (though shell scripts and commands can get quite complex).
    1. **Read**: The shell reads a command that you type into the command line interface and press Enter.
    2. **Evaluate**: The shell evaluates the command, which might involve executing a built-in command, running an executable file, or evaluating a script.
    3. **Print**: The output or result of the command execution is printed to the terminal. This could be textual data, an error message, or the output of a program.
    4. **Loop**: After executing the command, the shell displays a new prompt, waiting for the next command to be entered, continuing the loop.

Legend:

[1]: Elements of Control Flow—

1. Sequential Execution:

- The default mode where code is executed line by line, one after the other, from the top down.

2. Conditionals:

- **If-Else Statements**: Allow your program to execute certain blocks of code only if a specified condition is true (or false for the else part). It's like making a decision: "If it's raining, then take an umbrella, otherwise wear sunglasses."

3. Loops:

- **For Loops**: Repeat a block of code a specific number of times. For example, "For each day of the week, print the day."
- **While Loops**: Keep executing a block of code as long as a condition is true. For instance, "While there are items in the list, remove one and print it."

4. Switch/Case Statements (in some languages):

- Let you choose between multiple paths based on the value of a variable. It's a more streamlined way of doing multiple if-else checks on the same variable.

5. Function Calls:

- Direct the flow to another section of code and back again. "Go do this task (in a function), and then come back and continue from where you left off."

6. Jump Statements:

- **Break**: Exit out of a loop early.
- **Continue**: Skip the rest of the loop body and start the next iteration.
- **Return**: Exit from a function, optionally returning a value.

7. Exception Handling:

- **Try-Catch-Finally**: Allows you to attempt to execute code that might fail (try), handle errors gracefully (catch), and clean up resources or finalize tasks (finally), ensuring the program can continue running or exit cleanly even if an error occurs.

- **tr with brackets**: Change these letters to those.
  - define set of characters that you can edit
  - characters of set 1 with char of set 2
- **grep with brackets**: Find lines with any of these letters.
  - & is a pointer

Unit 4

Client-Server Software

Infastructure— comes before java script

- **interent**
- **www**
- **html**

Problems

Performance Issues:

- **network latency—** sending a message and coming back: “roundtrip delay”
  - “network transmission delays, keyboard going to sever and back”
  - “slows you down rather than working on a local machine

        - **Solution:** Caching [:07]
            - pretending to talk to a server but not really
            - doing stuff locally
            - client chaces server data
            - stale cache
                - ‘’10 minute old bank balance”
- **Throughput—** number of problems per second
    - server throughtput: number of requests per second that the server can answer
    - **Solution:**
        - Multiple servers ruunning in parallel OR multiple threads
            - each server is in charge for a region, each client knows what servers to talk to
            - improves overall throughpout
        - Front-End with paralllel back end servers
            - layered approach: gluing together cilent server arechitecture with primary server architecture
        - out-of-order requests responses:
            - responses from this group don't come back in the same order that the requests were sent to
            - **parallel issues:** serialazation problems—
                - series of requests dont make sense in the order they came
                - do the actions’ results make sense
                - explain a result by consructing a **serial order** for the actions
                - reasoning— unless you can justify the out-of-order response/ requests
                - parallel server’s
                - sending a node to another server
            - **mutliple threads**
                - faster

Correctness Issues

Stale Cache:

- **solution: Cache Validation**
    - refresh cache (not ideal)— ‘check entire weathermap’
    - checksum— ask server whats the checksum
        - “take your original data, apply formula to it”
        - 32 bit value that is a function of the template and it's scrambled off all the bytes input
        - it's a standard function that the client and server agreed upon, that converts this long string into a single and then you ask the server.

```
unsigned sum =0
while (c=getchar)
```

- timestamp— ask the server whats the timestamp

Internet:

Before the internet: 1962

- **circuits switching**: telephone systems
  - pros: guaranteed performance
  - cons:waisted capacity when call are stalled— expensive charges incurring
- **packet switcing:** solves waisted capacity
  - breaks information into little packets, approx 1000 bytes each
  - packets sent to centrol office
  - pros:
    - less waisted capacity, not sending packets when phone call is stalled
    - more reselient— cant blow it up, packets can redirect delivery
    - switches keep track of which neighbors are up/down
      - keeps a mental model of what the internet looks like
      - neighbor talking to neighbor
  - cons:
    - some packets get dropped
    - longer delay— move packets around before rendering them into analog
    - certain things cannot tolerate packet loss

Packets— low level

- headers (used by network to stuff out)
- payloads (data sent by user)
  - are exchanged via protocols (agreement between sender + receipient)
    - you can think of these are agreements
    - protocol helps with
  - **problem:**
    - 1. packet loss— usually due to server overload
    - 2. packets can arrive out of order
    - 3. packets can be duplicated— recipient can get the same packet twice even though sent once
    - faulty router makes false alarm and sends twice

- **solution:** protocols to deal with it at high level
  - **layers—** suite of protocols — 4high level protocols (js, etc.) so the client doesnt have to worry
    - **link layer**— lowest layer of protocol
      - point to point — one router to the next, tranmitter and reciever (wifi)
      - hardware oriented
    - **internet layer—** individual packets across the internet
      - internet protocol (iPv4) (1983)
        - packet oriented
        - header length— number of bytes in the packet
        - protocol number
          - end-to-end check, link layer checksum is reliable — important property of the internet
        - IP Adress: 32 bit a 4byte dotted sequence
        - TTL: Time to Leave (hop count)
          - packet and once you send a copy of that packet somewhere else, subtract one from the TTL field and then when it drops to zero that means the packet is no longer alive
          - each router change the number by 1
        - checksum (16-bit)
        - |__header | data ______ |
      - **IPV6 (1996)**
        - 128-256 IP adresses
        - more than 4 billion nodes on the internet
    - **transport layer—** data channels (conventions implemented by hardware/software on how to communicate)
      - **UDP— User Diagram Protocol (thin layer over IP)**
      - **TCP— Transmission Command Protocol**
        - **Data Streams—** one level up from packtets
          - **1) reliable (data not lost)**
          - **2) ordered (data recieved in order)**
          - **3) error checked**
        - **divide stream into packets**
        - **assembly—** reassemble out-of-order packets
        - **retransmission—** notifies sender of packet drop, sender resends it
          - **sender is tasked with resending that packet later on (complicating)**
        - **flow control—** dont send packets too quickly

      - one level of abstraction above internet
      - sending multiple packets through data channels
      - like circuit switching back then
    - **application layer—** protocols designed for a common set of application
      - Protocols:
        - HTTP:(Highest Transfer Protocol) (sits atop TCP)
          - ensures reliable data
          - TCP says data must arrive in order
          - The World Wide Web (WWW):
            - simple protocol for sharing webpages
              - HTTP —> (v1):
                - create connected —> client sends reeqeusts server via TCP approach—> server recieves request computes payload —> sends HTML response back to client   [1:44] —> connection closes
                - Protocol Syntax:
                  - GET /abc.html HTTP/1.0
              - Modern HTTP —> Data is encrypted (HTTPS, doesnt close connection
              - HTTP —> 2015
                - improved HTTP/ HTTP2
                - header compression

            - simple format for webpages
        - RTP: Real-time Transfer (Sits atop UDP)
          - TCP would cause jitter
          - ZOOM uses WEBRTC (audio-video) + SIP
            - video stream comes with dropped packets
            - transmission problems over functionality issues
            - SIP (Session Initiation Protocol)
              - like circuit switching— old phone system on steroids
              - packet switching at bottom level— top level more like circuit switching
      - build your own application protocol
      - web apps — built atop the lower levels

Internet Request For Comment: Standard for sending packets , packets wont get through for not following rules

Midterm: