1. Course Introduction and Scope

Lecturer: Paul Eggert

1.1 Software Construction vs Software Engineering

- CS 35L is about software construction.
- Prerequisite: CS 31 (basic algorithms, data structures, C++).
- CS31 covered basic, sequential code on single computer cores, which is now only one small part of modern software development.

1.2 Rapid Evolution in Software Development

- Previously, knowledge "half-life" in CS was ~15 years.
- Now decreasing due to rapid advances like machine learning.
- Students will constantly have to learn new software construction techniques throughout careers.

1.3 Generative AI and Software Construction

- Instructor is not a machine learning expert but emphasizes its growing influence.
- Tools like ChatGPT, Copilot, Cloud Code can help but cannot be fully relied upon.
- AI assistants' current ~75% accuracy is not good enough for production code.
- Students must still know how to code and verify results manually.

---

2. Course Structure

2.1 Projects and Assignments

- Group Project (5 people per team) - Practical software construction experience.
- Individual Assignments (6 total) - Cover various technologies and programming skills.

2.2 Technologies Used

- Recommended: Node.js, React (JavaScript-based stack).
- Additional topics/tools: Bash, Emacs, Make, Git, Python.

2.3 Tools and Learning Objectives

- Learn to rapidly adopt and use new software technologies.
- Build competence in pragmatic programming rather than theoretical deep-dives.

---

3. Core Topics

3.1 File Systems

- Covered via the POSIX model.
- Linux is used as the base system.
- Topic includes:
  - File abstractions
  - Storage structure
  - Permissions and metadata

Table Example:

Concept	Definition
File	A named data container in a filesystem
Directory	A file that lists other files
POSIX	Portable Operating System Interface (standard)

3.2 Scripting

Definition:

Writing programs (scripts) used to automate tasks, often with interpreted or command-based languages.

Languages Covered:

- Bash (shell scripting)
- Emacs Lisp
- Python

Tools:

- Emacs: open-source text editor and development environment.
- Interactive shell operations.

Examples:

Tool	Use Case
Bash	Automating build tasks, scripting tools
Emacs Lisp	Writing scripts in Emacs
Python	General-purpose scripting

3.3 Build and Distribution

Concepts:

- Building: Converting source into executable programs.
- Distribution: Installing programs for use on other systems.

Tools:

- make
- npm (for JavaScript projects)

Challenges:

- Consistency across systems.
- Managing dependencies.

3.4 Version Control

Key Ideas:

- Track and manage changes to code.
- Support collaboration.

Tool Used:

- Git

Concepts:

Concept	Explanation
Commit	A set of changes saved to the repository
Branch	A separate line of development
Merge	Combining changes from different branches
Tag	A named commit usually used for release versions

3.5 Low-Level Debugging and Dynamic Linking

Focus:

- Slightly higher-level than CS33 topics.
- Emphasis on understanding compiled artifacts behavior.

Tools:

- GDB (GNU Debugger)
- Dynamic linkers

3.6 Client-Server Architecture

Model:

- Server holds shared state (e.g., database, schedules).
- Clients communicate with the server but not with each other.
- Communication is indirect and goes through a server.

Example Application:

UCLA Groundskeeping Scheduler

Component	Role
Server	Stores all task and personnel schedules
Clients	Users update/view schedules

---

4. Comparison to CS130 (Software Engineering)

4.1 Topics Not Included in CS35L

Not Covered Fully	Mentioned Briefly
Scheduling Projects	Integration
Thorough Testing	Configuration
Project Forensics	Data Management
CI/CD (e.g., Docker)	Security Basics
Large-Scale Deployment	Networked Systems

4.2 Brief Lectures Promised on:

- Prompt engineering
- Data management (e.g., schemas, ACID principles)
- Security basics
- Distributed architectures (1/8 of a lecture)
- Deployment considerations

---

5. Software Construction Project

5.1 Goals

- Create a usable app with real-world potential.
- Demonstrate quick uptake of unfamiliar technologies.

5.2 Constraints and Expectations

- No C++
- JavaScript is encouraged, not mandatory.
- Node.js + React stack suggested.
- Client-server model is required.

5.3 Example: Groundskeeping App

- Task: Manage and schedule groundskeeping teams.
- Must support dynamic assignments as people call in sick or trees fall.
- Includes client-server spec and data sharing.

---

6. Tools and Languages

6.1 JavaScript/Node/React

Tool	Role
Node.js	Backend server scripting
React	Front-end UI development

6.2 Bash and Shell Use

- Used for scripting and shell operations.
- Students must be familiar with shell commands.

6.3 Emacs

- Text editor and development ecosystem.
- Written partly by Eggert.
- Includes scripting using Emacs Lisp.

6.4 Python

- Used for nondemanding scripting tasks.
- High-level general-purpose language.

---

7. Practical Infrastructure

7.1 CSNet Environment

- Students will use school's server network.

7.2 Login and Linux Usage

- SSH into CSNet servers.
- Understand command line interface (shell).
- Identify:
  - Shells: Command interpreters, e.g., Bash
  - Applications: Programs like Emacs/less

Example Commands:

Command	Action
ssh username@lnxsrv11.cs.ucla.edu	Log into CSNet
uname -r	Kernel version
cat /proc/cpuinfo	View CPU info
lsb_release -a	Show system info

---

9. Primer on Emacs and Linux Basics

9.1 Shells and Processes

- Shell: Interface to execute programs (CLI).
- Bash: Default shell used.
- Processes: Running executable programs.

Process Tree:

- systemd → bash → less/emacs/subshell

9.2 Command Line Interface (CLI)

- Command line environment pointedly non-GUI.
- Efficient for repetitive tasks, scripting.

9.3 ASCII and Control Characters

- ASCII: 7-bit character set
- Control characters: Use bit manipulation
- Example:
  - Ctrl-A = 0x01 (clear upper bits)
  - Meta key: Used to set highest bit

9.4 Basic Emacs Controls

Command	Operation
Ctrl-X Ctrl-C	Exit Emacs
Ctrl-H	Help Menu
Ctrl-H K (Key)	Show function of a Keystroke
Meta-X	Run command by name

---

10. Summary

This lecture sets the groundwork for CS 35L, a course focused on modern software construction practices beyond simple algorithm design. It frames software development in practical terms, emphasizing real-world technologies, version control, scripting, debugging, system interaction, and manageable project design using client-server architecture. It also explores how generative AI tools are beginning to change the development workflow—though caution is advised when using them. Key infrastructure such as the Linux command line, Emacs, and common Unix tools will be explored in depth. The course structure balances individual exercises with a collaborative capstone project, providing both technical experience and exposure to core skills in pragmatic software engineering.

Emacs, Shells, and Linux File Systems

1. Overview

In modern software development, users often deal with a triad of interrelated systems: scripting environments (such as shells), Integrated Development Environments (IDEs) like Emacs, and file systems (e.g., Linux file systems). This lecture takes a holistic, realistic approach to learning these tools simultaneously to mirror real-world development environments. Instead of isolating topics, systems are introduced in the overlapping, interdependent way in which they were historically developed and are actually used.

---

2. Emacs as an IDE

History and Purpose

- Initially, the instructor used Vim (and VI), but switched to Emacs due to its programmability.
- Emacs embeds a programming language (Emacs Lisp) which allows developers to build IDE-like tooling and systems efficiently.
- Emacs is programmable, extensible, and scriptable, making it suitable for heavy development work.

Emacs Features

Emacs is known for:

- Deep programmability
- Visual frame and window system
- Modality through Modes
- Internal shell invocation
- Editing many kinds of resources (e.g., files, directories, buffers)

Buffers vs. Files

- **Buffer**: A temporary in-memory representation of data (text, code, etc.)
- **File**: Stored on disk, persistent across system reboots.

Emacs distinguishes between these:

- Opening a file loads it into a buffer
- Changes are made to the buffer, not directly to the file
- Save operation (Ctrl-X Ctrl-S) writes buffer content to the file

Use case:

```
Ctrl-X Ctrl-F hello.txt       # Open or create file in buffer
Editing in buffer             # Does not affect file
Ctrl-X Ctrl-S                 # Save buffer to file
```

File Persistence in Emacs

Persistence in terms of application design means data survives crashes or reboots. Emacs approaches this with:

1. Buffers (fast, temporary)
2. Auto-save files (e.g., #hello.txt#)
3. Backup symbolic link to track editing (e.g., .#hello.txt)

Problems with persistent variables:

- Naive persistent variables (e.g., writing directly to storage each change) can degrade performance due to I/O latency.

Example:

```
persistent int numberOfStudents;
numberOfStudents++;
```

Would require saving to durable storage at every write, which is slow (milliseconds rather than nanoseconds).

Table: File Save Lifecycle in Emacs

Stage	Description
Start editing	Buffer is in memory, file untouched
Auto-save (periodic)	Saves to a temp file (#hello.txt#)
File lock (symbolic link)	Symbolic link to editing session metadata (e.g., .#hello.txt)
Manual Save (Ctrl-X Ctrl-S)	Writes buffer to hello.txt

Emacs Modes

- Emacs is a "modeful" editor
- A mode is a set of behaviors/rules for how keypresses and commands are interpreted
- Examples:
  - Text Mode
  - Dired Mode (directory editing)

Modes affect behavior:

- Typing `d` in a buffer inserts "d"
- Typing `d` in Dired marks file for deletion

List of Emacs Commands (selected):

Command	Meaning
Ctrl-X Ctrl-F	Find (open) file
Ctrl-X Ctrl-S	Save buffer contents to file
Ctrl-X 4 D	Open directory editor in new window
Ctrl-X O	Switch window (focus)
Ctrl-S	Forward search
Ctrl-R	Reverse search
Ctrl-@ (Ctrl-Space)	Set mark for region
Meta-W	Copy region
Ctrl-Y	Paste (Yank)
Ctrl-G	Abort current command (interrupt)
Ctrl-X 1	Maximize current window
Meta-X shell	Start a shell inside Emacs

Windows and Frames

Terminology:

- Window: A viewport into a buffer within a frame.
- Frame: A full Emacs window as the OS sees it.
- Multiple windows can show different buffers side-by-side.

Command Summary:

Command	Action
Ctrl+X 4 D	Open Dired in new window
Ctrl+X O	Move between windows
Ctrl+X 1	Focus single window

Emacs Editor for Directories (Dired Mode)

Dired is a mode for editing and navigating directories:

- Textual view of directory contents
- Limited editing (e.g., flagging for deletion)
- Drive changes back to the filesystem

Table: Dired Commands

Key	Action
d	Mark file for deletion
x	Execute marked delete actions
u	Unmark

Shell Access Inside Emacs

- `M-x shell`: Opens a shell inside Emacs.
- Basic commands:

- `cat file`: Print contents
- `ls -l`: List files with metadata
- `Ctrl+D`: End-of-input
- `Ctrl+C`: Interrupt
- `Ctrl+G`: Cancel Emacs command

Emacs Save Files and Metadata

Emacs creates temporary metadata files for buffers:

1. `#filename#`: Regular auto-save file.
2. `.filename~`: Backup file (not discussed in detail).
3. `.filename.swp`: May appear in other editors (not Emacs).
4. `.filename` symlink:
    - Contains editor and session metadata:
        - Username
        - Hostname
        - Emacs process ID
        - System boot timestamp
    - Enables Emacs to warn if a file is open elsewhere.

Example:

```
ls -l
#hello.txt#    # Auto-save buffer
.hello.txt     # Symbolic link containing editing info
hello.txt      # Main file
```

---

3. Linux File System Concepts

Files and File Types

In Linux:

- Files are objects storing a finite sequence of bytes
- Associated with metadata (permissions, size, timestamps)
- Common types:

Symbol	Type
-	Regular file
d	Directory
l	Symbolic link

File Metadata (shown with `ls -l`)

Example:

```
-rwxr-xr-- 1 egert egert 1234 Apr 5 10:00 hello.txt
```

Breakdown: | Component | Meaning | |---|---| | -rwxr-xr-- | Permissions (user/group/other) | | 1 | Link count | | egert egert | Owner and Group | | 1234 | Size in

bytes | | Apr 5 10:00 | Last modified time | | hello.txt | File name |

File Permissions

- Nine-bit mask:
  - Three groups: user (owner), group, others.
  - Each has read (r), write (w), execute (x).
- Use `id` command to check current user and group.
- Each file has an owner and a group field.

Section	Symbol	Octal	Role
User	rwx	7	Owner permissions
Group	r-x	5	Group permissions
Other	r--	4	Others' permissions

Common Permission Formats:

Symbolic	Octal	Meaning
rwxr--r--	744	Owner can read/write/execute
rw-r--r--	644	Owner can read/write, others read
rwxrwxr-x	775	Group shares full rights

Special Permission Bits

Linux includes 12 permission bits:

- 9 common bits (3x rwx for user/group/others)
- 3 special bits:

Name	Symbol	Effect
setuid	s	Run file as file's owner
setgid	s	Run as file's group
sticky	t	Applies to directories — restricts deletion

Example:

```
-rwsr-xr-- 1 root root 1234 su       # setuid (run as root)
drwxrwxrwt 10 root root 4096 /tmp   # sticky bit (shared dir)
```

Symbolic and Hard Links

- Symbolic Links:
  - Point to a path
  - Created with: `ln -s target linkname`
  - Can be "dangling" (point to non-existent file)
  - Used by Emacs with metadata to store process ID, editor, and system state in symlink name.
- Hard Links:

- Share inode (data) and increase link count
- Created with: `ln file1 file2`
- Both names refer to the same data

Hard Link Example:

```
touch foo
ln foo bar        # bar is another name for foo
```

Inode Confirmation with `ls -i`:

```
123456 foo
123456 bar       # same inode = same file
```

Use `rm` to remove a name; file persists until all links are removed.

Observation:

- Link count > 1 means hard link(s) exist.

Directories

- Directories are files that map names to inodes
- Always contain `.` (self) and `..` (parent)
- Directories start with link count = 2:
  - One for itself (`.`)
  - One from parent (`dirname`)
- Each subdirectory adds 1 to parent's link count (from its `..`)
  - `..` from `/` (root) points to itself.
  - Root has link count matching number of subdirectories + 1.

```
mkdir dir
ls -ld dir        # shows link count 2
mkdir dir/subdir
ls -ld dir        # link count now 3
```

Tree-Structured File System

- Root directory: `/`
- Files organized in hierarchy beneath root
- Paths:
  - Absolute: `/home/user/file.txt`
  - Relative: `./file.txt` or `../file.txt`

Diagram:

```
/
├── home/
│   └── egert/
│       ├── foo
│       └── bar
├── usr/
│   └── bin/
└── tmp/
```

Inode Numbers

- Every file has a unique inode number
- Use `ls -i` to view
- Link count: Number of names pointing to an inode

---

4. Shell Commands and Utilities

These utilities interact with the file system and processes.

Basic Shell Overview

- Default shells provide process management, file commands, and scripting features.
- Setting: Emacs can embed shell as buffer (`M-x shell`)

Control Characters

Key	Meaning
Ctrl+D	End-of-file/input
Ctrl+C	Interrupt (stop process)
Ctrl+G	Emacs interrupt (abort command)
Ctrl+@ / Ctrl+Space	Set mark for copy
Meta+W	Copy region (kill-ring)
Ctrl+Y	Paste ("yank")

---

cat

- `cat file`: Outputs content of a file.
- `cat`: With no args, reads from keyboard and echoes input.

Examples:

```
cat file1             # Print content
cat file1 file2       # Concatenate and print
```

Special behavior:

- `cat` with no arguments reads from terminal (stdin)
- Ctrl+D at start of line signals EOF

ps

- Shows process status
- `ps -ef`: Lists all processes with full info

Sample Output:

```
UID   PID   PPID  CMD
egert 22588 1     emacs
```

Process ID (PID) used in Emacs symbolic links:

```
.#hello.txt -> egert@machine.22588:timestamp
```

chmod

- Changes file permissions
- Syntax:

```
chmod 644 file.txt
chmod +x script.sh
```

ln

Creates links:

- `ln file1 file2`: Hard link
- `ln -s path link`: Symbolic link

Special cases:

- Cannot hard-link directories (except by filesystem convention for `.` and `..`)
- Emacs uses symbolic links to indicate lock/status metadata for edited files

touch, rm, and truncate

- `touch`: Creates empty file or updates timestamp
- `rm`: Removes file name (not file unless final link)
- `truncate`: Sets file size without writing content

Example:

```
truncate -s 1T bigfile   # Create 1 terabyte sparse file
ls -lh bigfile           # Shows size
cat bigfile              # Outputs null bytes (if not crashed)
```

---

5. Summary

This lecture introduced three core interconnected systems in a Unix/Linux environment: scripting via shell, file systems, and editing environments like Emacs. Emphasis was placed on realism through simultaneous learning and usage reflective of real-world software engineering scenarios. Emacs was covered in depth with its concept of buffers, windows, modes, saving mechanics, and integration with directories and the shell. File system fundamentals such as inode numbers, link counts, file permissions, symbolic vs hard links, and hierarchical structure were explored. Command-line utilities like `cat`, `ls`, `chmod`, `ps`, and `ln` helped demonstrate these principles. The engineer's need to balance persistence, performance, and understandability in system architecture was repeatedly highlighted.

Shells, File Systems, and Emacs

1. File Systems and Links

1.1. File System Overview

The file system is organized hierarchically as a tree, with the root directory `/` at its base.

- Root Directory (`/`): The entry point to the file system.
- Subdirectories: Standard directories include `/bin`, `/usr`, `/home`, etc.
- Types of Entries:
  - Regular files (e.g., `.gitignore`)
  - Directories
  - Symbolic links

Each file or directory is uniquely identified within the file system by its inode number.

1.2 Path Resolution

- Absolute path: begins with `/`.
- Relative path: begins from current working directory.
- The resolution algorithm (namei-like function):
  1. Start from `/` if absolute, otherwise current directory.
  2. Traverse each component, resolving directories.
  3. Resolve symlinks if present.

1.3. iNodes and File Identification

Each file or directory is assigned a unique identifier called an inode number.

- Inodes store metadata (ownership, permissions, timestamps), not name.
- Commands to view inode numbers:
  - `ls -i filename`
  - `ls -li` for long format with inode.

Example:

```
ls -li /bin/sh
1234567 lrwxrwxrwx 1 root root 4 Jan 1 00:00 /bin/sh -> bash
```

Where `1234567` is the inode number.

1.4. Hard Links

Definition:

- Links to the same inode from different directory entries.
- Equivalent entries pointing to the same file content.

Key Facts:

- Hard links to the same file share the same inode.
- Cannot make hard links to directories.
- Changes made through one hard link are visible through others.

Example:

```
ln file1 file2
```

Name Inode

Name	Inode
file1	97311
file2	97311

1.4. Symbolic (Soft) Links

Definition: A symbolic link (symlink) is a reference to another path in the file system. Unlike hard links, symlinks are a separate file containing a path as a string.

Key Facts:

- Symlinks have their own inode.
- They can point to non-existent targets (dangling symlinks).
- Symlinks can be relative or absolute.

Command:

```
ln -s target symlink
```

Types of symlinks:

Type	Starts with	Interpreted From
Relative	No `/`	Directory containing the link
Absolute	`/`	Interpreted from root `/`

Example:

```
ln -s /bin/sh mysh  # Absolute
ln -s ../bin/sh relsh  # Relative
```

---

2. Pathname Resolution

2.1. Absolute vs. Relative Paths

- Absolute paths start with `/` and begin resolution from the root.
- Relative paths do not start with `/` and are resolved from the current working directory.

Each process maintains a current working directory.

Change directory:

```
cd dirname
```

2.2. Symbolic Link Resolution

- If a pathname includes a symbolic link, the system replaces that component with the content of the symlink.
- If the symlink is:
  - Relative: interpreted relative to the directory containing the symlink.
  - Absolute: restart resolution from the root.

Example resolution:

`/bin/sh`

If `/bin` is a symbolic link to `usr/bin`, the path transforms to:

`/usr/bin/sh`

2.3. Common Errors and Loops in Symlinks

- Dangling symlink:

```
ln -s nowhere foo
cat foo  # No such file or directory
```

- Infinite Loop:

```
ln -s B A
ln -s A B
cat A  # Fails due to loop
```

To prevent infinite symbolic link loops, the system tracks the number of traversals (typically capped at ~20 in Linux).

2.4. Hard Links to Symbolic Links

- Multiple hard links to the same symbolic link result in shared link counts.

Example:

```
ln foo bar
```

if `foo` is a symlink.

Name	Type	Link Count
foo	symlink	2
bar	symlink	2

Removing one doesn't delete the symlink, only removes one reference.

2.5. Symbolic Links and Directory Context

Relative symlinks behave differently based on the path used to reference them.

Example:

```
mkdir e
ln foo e/.baz
```

- `e/.baz` may resolve correctly if relative from `e/`
- May not work if accessed from the parent directory.

---

3. The Shell

3.1. Shell Purpose and Use

- First widely used scripting language.
- Shell's primary job is coordination: launching and configuring other programs.
- Common tasks:
    - Configuring standard input/output
    - Variable expansion
    - Command sequencing

3.2. Command Execution and Standard I/O

Redirection operators:

- `>`: Redirect standard output.
- `<`: Redirect standard input.
- `2>`: Redirect standard error.

Example:

```
cat file > output.txt 2> errors.log
```

Piping:

```
cmd1 | cmd2 | cmd3
```

Chains standard output from one command to standard input of the next.

3.3 Argument Passing

```
command arg1 arg2
```

- Argument count and values appear as:

```
int main(int argc, char** argv)
// argv[0] = command, argv[1] = arg1, ...
```

3.4. Tokenization and Quoting

Special Characters:

Characters with special interpretations unless quoted:

Character	Purpose
Space, Tab, Newline	Separators
`<`, `>`	I/O Redirection
`&`, `|`	Background & Pipe
`;`	Command separator
`#`	Comment
`$`	Variable expansion
`` ` ``	Command substitution
`(`, `)`	Grouping/Subshells

Quoting Mechanisms:

Method	Syntax	Behavior
Escape	`\\char`	Escape a single character
Single Quotes	`'text'`	Literal string
Double Quotes	`"text"`	Variable/command expansion allowed

Examples:

```
echo 'This is $HOME'  # Literal string
echo "This is $HOME"  # Expands $HOME
```

Multiline:

```
echo "Multi
Line"
```

Null Byte Limitation

Shell does not allow embedded null bytes in variables or strings.

3.5. Reserved Words

Type	Example
Conditional	`if`, `then`, `else`, `fi`
Loops	`for`, `in`, `do`, `done`, `while`, `until`
Case	`case`, `esac`
Grouping	`{`, `}`
Negation	`!`

Examples:

If Statement:

```
if cat file; then
  echo "Success"
else
  echo "Failure"
fi
```

While loop:

```
while cat file; do
  sleep 1
done
```

For loop:

```
for i in a b c; do
  echo $i
done
```

Case Statement:

```
case $x in
  *.c) gcc $x;;
```

```
  *.cpp) g++ $x;;
  *) echo "Unknown type";;
esac
```

3.6. Parameters and Variables

Shell scripts receive arguments as:

- `$0`: Script name
- `$1` to `$9`: Positional args
- `$#`: Number of arguments
- `$*`: All args as one word
- `$@`: All args as distinct words

Key Differences:

Usage	Result
`"$*"`	All args as single string
`"$@"`	All args preserved individually

Special Examples:

```
x=a b c
echo $x  # a b c
echo "$x"  # "a b c"
```

Unset variables:

```
unset x
```

Default substitution:

```
${VAR:-default}  # Use default if VAR isn't set
${VAR:=default}  # Set VAR to default if not set
${VAR:+alt}      # Use alt if VAR is set
${VAR:?error}    # Print error if not set
```

Environment variables: Use `export` to make available in subcommands:

```
export EDITOR=emacs
```

3.7. Redirection and Pipes

Standard file descriptors:

Descriptor	Number	Default
stdin	0	Keyboard
stdout	1	Screen
stderr	2	Screen

Modifiers:

```
2>file     # stderr to file
1>&2       # stdout to stderr
command &  # Run in background
```

Advanced example:

```
A 2>err1 | B 2>err2 | C
```

Splitting Output: Use tee to duplicate output:

```
cmd | tee file | another_cmd
```

3.8 Multiple Writers or Readers

- Running concurrent processes that read/write from shared pipe creates race conditions.

Example:

```
cmd1 | tee output.txt | cmd2
```

3.9. Expansion Rules

Processing order:

1. Tilde expansion (`~`, `~user`)
2. Parameter expansion (`$VAR`)
3. Command substitution (`$(cmd)`)
4. Arithmetic expansion (`$((expr))`)
5. Field splitting (split on whitespace)
6. Pathname expansion (globbing, e.g. `*.c`)
7. Redirection

Example Evaluation:

```
echo $((1 + 2)) # => 3
```

Globbing:

```
ls *.c  # Lists all .c files
```

If no matching files:

- The pattern remains unexpanded.

---

4. Summary

This lecture discusses the structure and function of POSIX-compatible file systems, focusing on file naming and linking mechanisms (both hard and symbolic). It explains the interpretation of file names through pathname resolution, including relative vs. absolute paths, and how symbolic links are resolved or can go wrong (e.g., dangling links, symlink loops). It then transitions into a comprehensive exploration of Unix shell behavior, discussing command execution, redirection, quoting rules, reserved words in scripts, and shell expansions. Special attention is paid to

scripting constructs such as conditionals (`if`, `else`), loops (`for`, `while`), variables and parameter handling (`$@`, `$*`, `$#`), background process management, process IDs, and the use of pipes for process communication. The lecture concludes with an explanation of shell expansion mechanisms and variable substitution nuances essential for writing robust shell scripts.

Pattern Matching in the Shell

Globbing

Globbing is a simplified form of pattern matching used by the shell (e.g., bash) primarily for matching file names. It's distinct from, but related to, regular expressions.

- Commonly used with commands like `echo`, `ls`, and `rm`.
- It can also be used within control structures such as `case`.
- Faster and simpler than full regex.
- Only matches filename components (no slashes).

Special Characters and Syntax

Symbol	Meaning
`*`	Matches any sequence of zero or more characters.
`?`	Matches exactly one character.
`[...]`	Matches any single character enclosed in the brackets.
`[a-z]`	Matches characters in the specified range (`a` to `z`).
`[!...]`	Matches a single character not in the specified set (negation).

- Use `-` in brackets to denote ranges. To include the `-` itself, escape ambiguity by placing it at the end.
- Directory separator `/` is not matched by `*` or `?`.
- Filenames that start with `.` (dot files) are not matched by `*` or `?` unless the pattern explicitly starts with `.`, e.g., `.??*` matches hidden files of at least three characters.

Globbing Exceptions

1. Globs never match slashes (`/`)
    - This prevents expensive recursive directory traversals.
2. File and directory names starting with `.` (dot files) are hidden and not matched by wildcards unless explicitly specified.

Globbing Examples

Pattern	Description	Match
`*`	All files except hidden ones.	`foo`, `bar.txt`, not `.hidden`
`?.txt`	One character followed by `.txt`	`a.txt`, not `ab.txt`
`[abc]*`	Files starting with `a`, `b`, or `c`.	`apple`, `cat`, not `dog`
`[!abc]*`	Files not starting with `a`, `b`, or `c`.	`dog`, not `apple`
`*.bash`	Files ending with `.bash`.	`script.bash`
`.??*`	Hidden files with name length ≥ 3.	`.xrc`, not `.x`

Example Directory: Contains files foo, bar, .hidden, a.txt, a-b.txt

Command	Matches
echo *	foo, bar, a.txt, a-b.txt
echo .*	.hidden, .., .
echo ?.txt	None (requires 1-char name)
echo [a-c]*	bar
echo [!abc]*	Anything not starting a/b/c

Globbing in Directory Hierarchies

- Directory matching needs explicit structure:
  - `*/foo*` only finds `foo*` in direct subdirectories.
  - `*/*/foo*` required for two-level search.

Globbing Exclusion

You can use bracket negation to exclude specific starting characters.

Example:

```
echo [!abc.]*
```

Matches files that do not begin with `a`, `b`, `c`, or `.`.

---

2. Shell I/O Redirection

Standard File Descriptors

Descriptor	Description
0	Standard Input
1	Standard Output
2	Standard Error

Redirection Syntax

Syntax	Meaning
`>`	Redirect `stdout` to a file (overwrite).
`>>`	Redirect `stdout` and append to file.
`2>&1`	Redirect `stderr` to where `stdout` points.
`3< file`	Open file on descriptor 3 for reading.
`3> file`	Open file on descriptor 3 for writing.
`3<> file`	Open for reading and writing on descriptor 3.

Here Documents

A "here document" is a redirect that includes text directly in the script or command.

Example:

```
cat <<EOF
line one
line two
EOF
```

This feeds the lines into `cat` as `stdin`.

- Supports variable substitution unless quoted:

```
cat << 'EOF' # prevents variable expansion
cat << EOF   # expands variables
```

Advanced Redirection Examples

```
command 3>&1 1>tempfile
```

- Redirects file descriptor 3 to the current `stdout`, and `stdout` to a file.

---

3. Shell Commands and Scripting

Exit and Return

Command	Effect
`exit`	Terminates shell (or script).
`exit 1`	Exits with status code 1.
`return`	Exits from a shell function only.

- `exit` exits the shell completely, but `return` is confined to functions.
- You can check the exit status using `$?`.

Shell Functions vs Shell Scripts

Two approaches:

Shell Function

```
g() {
  grep "$@"
}
```

- Defined in `.profile` or directly in the shell.
- Lightweight and executed in current shell.
- Faster (no new process)

Shell Script

g file:

```
#!/bin/bash
grep "$@"
```

- Saved in file in a directory listed in `$PATH`.
- Executed in a new process.
- More portable and global.

Feature	Function	Script
Scope	Local to shell	Global in environment
Overhead	Low	High (new process)
Portability	Low	High
Speed	Fast for small tasks	Better for large tasks

Aliases

- Used for simple command substitution.

```
alias g='grep'
```

- Not suitable for complex logic; use functions instead.

---

4. Regular Expressions

Design Philosophy

Regex (regular expressions) define patterns to match strings. Used with tools like grep, sed, awk, Python, etc.

- A little language tailored to string pattern matching.
- Variants exist because different tool authors chose different syntaxes.

Extended Regular Expressions (ERE)

Use `grep -E` or `egrep` to invoke EREs.

Core Syntax and Operators

Operator	Description
`.`	Any character except newline
`*`	0 or more repetitions
`+`	1 or more repetitions
`?`	0 or 1 occurrence
`|`	Alternation (OR) between regex
`()`	Grouping (changes precedence)
`{n}`	Exactly n occurrences
`{n,m}`	Between n and m repetitions
`^`	Start of line anchor
`$`	End of line anchor
`[]`	Bracket expressions (character classes)

Bracket expressions

Syntax	Description
`[abc]`	Match 'a', 'b', or 'c'
`[a-z]`	Range: a to z
`[^abc]`	Negate: anything except a, b, or c
`[[:alpha:]]`	Match alphabetical letters (locale-aware)
`[.]`, `[-]`, `[]]` tricks	Special syntax rules to include symbols

Special character handling inside `[]`:

- `-` denotes range unless at start or end
- `]` must be escaped or placed first
- `^` must be first character to negate

Examples

Pattern	Matches Example
`abc`	Only the string abc
`a.b`	a followed by any character, then b
`a*`	zero or more a's
`a+`	one or more a's
`(ab|cd)+`	ab or cd, repeated
`^xyz$`	entire line must be `xyz`
`^(.)(.)(.).\3\2\1$`	Matches six-character palindromes.

Escape Sequences

Backslashes are used to escape special characters.

- Must escape metacharacters: `\* \. \( \)` etc.
- Caution: Shell may interpret before `grep` does.

To match `\`, you'll often need `\\\\`.

Basic Regular Expressions (BRE)

Used with `grep` without `-E`.

Differences from ERE:

- Metacharacters like `+`, `?`, `{}` are NOT special.
- Use `\(...\)` for grouping.
- Use `\{n,m\}` for repetition.
- `|` is not supported directly.

Backreferences (BRE only)

Syntax	Description

Syntax	Description
`\1`	Refers to first captured group.
`\2`	Refers to second captured group.

Examples

Pattern	Description
`\(abc\)\1`	Matches 'abcabc'
`\(^a.*b$\)`	Entire line starting with a, ending with b
`^\(.\)\(.\)\2\1$`	Matches 4-character palindromes.

Performance Note: Backreferences are slow and non-regular — avoid when possible.

Common Pitfalls

- Regular expressions with just a backslash (`\`) are invalid.
- Quoted expressions inside the shell need escaping.

---

5. Emacs

Philosophy

- Keyboard-driven efficiency.
- Emphasis on not taking hands off keyboard.
- Modular via modes and the mini-buffer.

Cursor and Region Operations

I/O

Command	Action
`C-x C-f`	Open file
`C-x C-s`	Save file
`C-g`	Cancel current command

Mark, Region, and Copy

Command	Action
`C-SPC` or `C-@`	Set mark at cursor (start selection)
`M-w`	Copy selected region to kill-ring
`C-w`	Cut (aka "kill") selected region
`C-y`	Yank (paste) last kill-ring
`M-y`	Paste earlier entries in kill-ring
`C-x C-x`	Exchange point and mark

The Kill Ring

- Stores multiple text entries from kills.
- Cyclic navigation with `M-y` after `C-y`.

Buffer and Window Management

- Buffers: Independent in-memory views (files, outputs, shell, etc.)
- Windows: Viewports into buffers

Command	Description
`C-x b`	Switch buffer.
`C-x C-b`	List all buffers.
`C-x o`	Switch windows.
`C-x 2`	Split window horizontally.
`C-x 3`	Split window vertically.
`C-x 0`	Close current window.
`C-x 1`	Maximize current window.

Emacs Modes

- Major modes (e.g., Fundamental, Dired) tailor behavior to the file type or buffer.
- Minor modes add auxiliary behavior (e.g., Line numbers).

Use `C-h m` to describe current modes and key bindings.

Accessing Help

Command	Description
`C-h k`	Describe key binding.
`C-h m`	Describe current mode.
`C-h i`	Info documentation browser.

Meta key (`M-`) is typically `Alt` or `Esc` key.

Mini-buffer Operations

- Executes internal commands or inputs.
- `M-x`: Execute Emacs command.
- `M-:`: Evaluate Emacs Lisp.
- `M-!`: Run shell command.
- `M-|`: Run shell command with region as input.

Examples:

- `M-! date`: Run shell date command
- `M-| sort`: Sort selected region

Key behaviors:

- Uses same movement/copy/edit commands
- Allows evaluation of Emacs Lisp: `M-:`

Summary

This lecture provided a deep and detailed exploration of Unix Shell pattern matching via globbing and regular expressions, covering the distinctions between globbing and regex, the syntax and semantics of ERE and BRE, and intricacies around quoting and special characters. It also delved into shell input-output redirection with advanced use of file descriptors and here-documents, and covered practical script design decisions between shell functions, aliases, and scripts. Finally, the lecture examined the Emacs text editor, introducing core commands for navigation, editing, buffer management, and obtaining help interactively, all designed to enable keyboard-efficient workflows during development.

Scripting Languages — Syntax, Semantics, Pragmatics, and Emacs Lisp

1. Introduction to Scripting Languages

Definitions and Context

- Scripting languages are often thought of as languages used to automate processes, configure, or extend large systems.
- Traditional classifications of programming languages include:

Category	Description
Syntax	Refers to the structure or form of code (like semicolons, braces in C++). Considered solved and relatively easy.
Semantics	Refers to the meaning or behavior of the program—what the code does.
Pragmatics	Practical concerns of programming - efficiency, configuration, usability, security, and interoperability. Most emphasized in scripting.

Scripting languages emphasize pragmatics more than traditional programming languages.

---

2. Overview of Example Scripting Languages

Shell

- One of the oldest scripting languages, designed to automate and configure Unix programs.
- Still widely used to run and link other programs.

Emacs Lisp

- Not originally a scripting language, but adapted as one.
- Originated as a general purpose language designed to write AI programs in the 1960s.
- Adopted for scripting Emacs due to its support for program generation and self-modifying code.
- Known as an extension language, designed specifically for Emacs, not general-purpose use.

Java

- Not a scripting language but contributed vital implementation ideas:
  - Bytecode abstraction
  - Interpreters and Just-In-Time (JIT) compilation
- Strong typing, performance-oriented, reliable
- Emphasizes reliability and strict compiled checking of code.

JavaScript

- Originally designed to script and extend web browsers.
- Became a language with general-purpose capabilities.
- Uses similar implementation techniques as Java (e.g., JIT, bytecode)

Python

- Now one of the world's most popular languages (estimated 25% of developer mindshare).
- Designed as a general-purpose scripting language.
- Emphasizes code readability, quick prototyping, and integration of external modules
- Supports linking external libraries (often written in C/C++)

---

3. Key Themes of Scripting Languages

Ease of Use

- Scripting languages prioritize accessibility.
- Easy to learn, fast to write short scripts.
- Example: Python makes it easy for beginners to write "Hello World!" without extensive setup.

Reliability

- Scripting languages often avoid worst-case crashes common in languages like C++ (e.g., null pointer exceptions).
- Instead, they raise exceptions or print errors.
- However, permissiveness may allow incorrect programs to execute without failure, decreasing logical reliability.

Language	Crashes on Subscript Error?	Throws Exception?
C++	Yes	No
Python	No	Yes
Shell	No	Typically alerts

Scalability

- Scripting languages are good for:
  - Small to medium programs
  - Glue code
  - Lightweight data transformation

- Problems occur with very large codebases due to:
  - Lack of strict typing
  - Fragmented tooling
- Integration often includes low-level languages (like C++) for performance and reliability in large systems.

Performance

- Generally slower than compiled languages due to runtime checks and interpreted execution.
- Tradeoff: slower performance for higher productivity
- JIT compilers and bytecode interpreters are used to mitigate this gap (e.g., JavaScript, Emacs now includes bytecode compilation).

---

4. Concepts and Philosophical Differences

General vs. Extension Languages

Language Type	Example	Description
General Purpose	Python, Java	Use it for almost anything
Extension	JavaScript, ELisp	Embedded in host applications (e.g., browser, editor) to extend functionality

Traditional vs. Scripting

Feature	Scripting Languages	Traditional Languages
Performance	Low	High
Flexibility	High	Medium
Error Handling	Graceful	Harsh (e.g. segfaults)
Compilation	Optional or runtime only	Mandatory

Design Purpose of Languages

Language	Intended Use
JavaScript	Extending browser functionality
Shell	Automated OS-level task and config scripts
Emacs Lisp	Extending the editor’s features
SQL	Query and data specification
Rust	High-performance and safe system programming

5. Emacs Lisp: An Introduction

Historical Context and Purpose

- Lisp, created for AI in the 1960s, grew to support self-modifying and program-generating code.
- Emacs Lisp is a variant tailored to the GNU Emacs text editor.
- Emacs Lisp is an extension language: enhances and customizes Emacs behavior.

Emacs Lisp as an Extension Language

- Supports deep integration with Emacs internals: buffers, windows, syntax highlighting, etc.
- Provides primitives focused on editing rather than general computing (compare to SAP's payroll-specific extension language).
- EMACS is built on:
    - Low-Level C libraries
    - Lisp Interpreter written in C
- Code entered in Emacs is interpreted or compiled into bytecode

---

6. Syntax and Semantics in Lisp

Lisp Syntax Characteristics

- Simple, uniform syntax using parenthesized prefix notation.
- All function calls look like: (function arg1 arg2 ...)
- No operator precedence, no infix notation.
- Symbols and data use homogenous parenthetical syntax.

Structuring Code as Lists

- Code and data blurred: both written as lists.
- Example:
    - Data: '(a b c)
    - Code: (+ 34 25)

Symbols and Special Forms

- Symbols: Unique named atomic values.
    - Examples: nil, t, my-var
- Special Forms:
    - Control structures and definition constructs that are not function calls.
    - Examples: defun, let, lambda, quote

Expressions vs. Data

- `'expression` prevents evaluation.
- `(eval expression)` forces evaluation even of quoted data.

Form	Meaning
'abc	Symbol `abc` as data

Form	Meaning
abc	Evaluate variable `abc`
'(a b c)	List of three symbols a, b, c
(quote (a b c))	Same as above

Interactive Evaluation (Scratch Buffer)

- Evaluated using:
  - `C-J`: Evaluate previous expression and show result below.
  - `C-x C-e`: Evaluate expression at point and show in mini-buffer.

Debugging Basics

- Emacs will drop into the debugger upon errors.
- Key command to exit debugger: `C-]` (Control + closing square bracket)

---

6. Emacs Lisp Data Structures

Numbers

- Integers: Arbitrary precision (bignums)
- Floats: IEEE 64-bit doubles
- Example: (+ 100000000000000000 1)

Strings

- Written with double quotes
- Supports newlines: "Hello\nWorld"

Symbols

- Named values that point to variables or functions
- Used for identifiers, constants, or stand-alone atoms
- Special symbols:
  - nil: false, end of list
  - t: true

Lists

- Fundamental concept in Lisp: “List Processing”
- Built-in functions for lists:
  - `cons`: Construct a new pair
  - `car`: Return first element
  - `cdr`: Return rest of list

List Example

```
(cons 'a (cons 'b nil)) ; => (a b)
```

Improper Lists

```
(cons 'a 'b) ; => (a . b)
```

Vectors

- Fixed-size, indexable arrays
- Created via: (make-vector 5 'x)

Hash Tables, Markers, Buffers

- Hash Tables: Key-value associative maps.
- Markers: Pointers to specific positions in buffers that adapt as buffer changes.
- Buffers: Editable text containers (contents of a file, scratch, etc.)

---

8. Functional Programming Concepts

Functions

- First-class objects
- Created via lambda
- Using `defun`:

```
(defun add-three (x) (+ x 3))
```

- Interactive Commands:
    - Use `interactive` keyword

```
(defun greet-user ()
  (interactive)
  (message "Hello!"))
```

- Example:

```
(defun show-buffer-name ()
  (interactive)
  (message (buffer-name)))
```

- `let`: Defines local variables

```
(let ((x 3) (y 4)) (+ x y)) ; => 7
```

Lambda Expressions

- Emacs supports anonymous functions with `lambda`.

Example:

```
(lambda (x y) (+ x (* y y)))
```

- Can pass lambdas as arguments or assign to variables.

Local Bindings (LET)

- Creates local variables bound to evaluated values.

Example:

```
(let ((x 5) (y 7))
  (+ x y))
```

- Temporarily masks global variables if names overlap.

---

9. Emacs Lisp Utilities and Commands

Emacs Commands

- Functions callable via UI must be declared interactive:

```
(defun my-command ()
  (interactive)
  (message "Hello"))
```

- Can query keystrokes or input with `interactive` argument.

Accessing Documentation

- `C-h k`: Describe key bindings.
- `C-h f`: Describe function.
- `C-h m`: Describe current mode.
- Mouse over source file link (e.g., window.el) to open implementation in Emacs

---

10. Summary

This lecture introduced the concept of scripting languages with a focus on their unique emphasis on pragmatics — real-world concerns of integration, performance trade-offs, ease of use, reliability, and extensibility. It differentiated scripting languages from general-purpose and low-level languages, exemplified by Emacs Lisp and Python. The lecture gave an in-depth presentation of Emacs Lisp as an extension language, explaining its evaluation model, key data types (including symbols, lists, strings, functions), and its support for self-modifying code, which allows it to script and modify Emacs in powerful ways. The class also demonstrated fundamental Lisp programming techniques including quoting, evaluation, function definition, lambda expressions, local variable bindings, and debugging tools within Emacs.

Python Introduction & Fundamentals

1. Introduction

1.1 Audience Spectrum

The lecture is aimed at a mixed audience: some already experienced with Python and some beginners. The approach balances basic teaching with deeper insights to appeal to both groups.

1.2 Teaching Approach

- Uses live examples to build intuition.
- Encourages embracing Pythonic practices.
- Highlights common pitfalls and historical motivations behind Python features.

---

2. Writing and Executing Python

2.1 Interactive Use

Python can be run in various environments; the example uses Emacs (`Meta-X run-python`), but any environment such as Jupyter or terminal is acceptable.

2.2 Example Scenario: Parsing Stock Market Data

Given a string like:

```
"GOOG,100,153.36"
```

Goal:

- Extract the symbol ("GOOG")
- Convert 100 to int
- Convert 153.36 to float

Target Output:

```
['GOOG', 100, 153.36]
```

---

3. Python Types and Data Parsing

3.1 Python Basic Data Types

- `str`: a string
- `int`: an integer
- `float`: a floating point number

3.2 Parsing with Types

Python has constructor functions (which are actually classes) to cast strings:

```
[int("100"), float("153.36")]  # Converts strings to specified types
```

3.3 The `zip()` Function

Zips two sequences element-wise:

```
types = [str, int, float]
values = ["GOOG", "100", "153.36"]
list(zip(types, values))
# [(str, 'GOOG'), (int, '100'), (float, '153.36')]
```

- Lazily evaluated: must be cast to `list()` to see all elements.
- Reusing iterated zip objects yields empty results.

3.4 List Comprehensions

Powerful Python feature for transforming data:

Example:

```
[c(v) for c, v in zip(types, values)]
# ['GOOG', 100, 153.36]
```

Benefits:

- Concise expression of logic
- More "Pythonic" than index-based loops

---

4. Pythonic Programming Philosophy

4.1 Avoid Low-Level Thinking

Avoid thinking like in C++ (manual indexing, mutable pointers, etc.)

4.2 Think High-Level

Utilize Python abstractions such as:

- List comprehensions
- Built-in functions
- Prefer `for x in y` over `for i in range(len(y))`

---

5. Brief History of Python

5.1 Fortran → BASIC → ABC → Python

- Fortran (1950s): Popular but unforgiving
- BASIC (1960s): Beginner friendly
- ABC (1980s): Dutch origins, focused on ease, enforced indentation

5.2 ABC Concepts Carried To Python:

- Indentation used for block structure (not braces)
- Built-in data structures
- Focus on scripting, ad-hoc tasks
- Prefixes simplicity, readability, and education

- Also influenced by frustration with poor-quality scripting tools like Perl

---

6. Python Syntax and Indentation

6.1 Important Syntax Rule:

Blocks start with a colon:

```
if x > 0:
    print("Positive")
```

6.2 Copy-Paste Issues

- Copying improperly indented code can cause syntax errors.
- Maintain consistent indent spacing.

6.3 Tabs vs Spaces

- Stick to spaces for indentation.
- Avoid mixing tabs and spaces.
- Recommendation: Never use tabs to avoid compatibility issues.

---

7. Numerical Types and Operations

7.1 Integers

- Arbitrary precision
- You can do `10**1000` without overflow.

7.2 Floats

- Limited to ~10^308
- Produces `inf` when overflowed

```
float('inf') > 99999999  # True
```

7.3 Division

```
1 / 2   # 0.5 (float division)
1 // 2  # 0 (integer division)
```

7.4 Complex Numbers

- Use `j` for imaginary part:

```
1 + 2j
cmath.sqrt(-1)  # returns 1j
```

---

8. Strings in Python

8.1 Quotes

- Single `'abc'` or double `"abc"` are interchangeable.
- Python displays with single quotes by default.

8.2 Escape Sequences

```
"\n"  # newline
"\t"  # tab
```

8.3 Raw Strings

```
r"\n"  # literal backslash and n
```

8.4 Triple Quotes

Use triple quotes for multi-line strings:

```
'''This is
a multiline string'''
```

---

9. Python Object Model

9.1 Every Object Has:

- Identity → via `id(obj)`
- Type → via `type(obj)`
- Value → the object itself

9.2 Immutability

- Immutable: Value cannot change (`int`, `str`, `float`)
- Mutable: Value can change (`list`, `dict`, ...)

```
a = 12
print(id(a))
a += 1
print(id(a))  # ID has changed
```

9.3 Variables ≠ Objects

Variables are bindings to objects, not the objects themselves.

9.4 Aliasing

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)  # [1, 2, 3, 4]
```

Both refer to the same list.

---

10. Python's Built-In Data Types Overview

10.1 `None`

Singleton object representing "nothing".

10.2 Numbers

- `int`, `float`, `complex`

10.3 Sequences

- Common: `str`, `list`, `tuple`
- Indexable, iterable

10.4 Mappings

- Primarily `dict`: key/value store

10.5 Callables

- Functions, methods, classes, lambdas

---

11. Python Sequences

11.1 Indexing

```
s[i]
```

- Raises exception if out of range

11.2 Negative Indexing

```
s[-1]  # last element
```

11.3 Slicing

```
s[i:j]       # from i to j-1
s[:j]        # from beginning to j-1
s[i:]        # from i to end
s[:]         # copy entire list
```

11.4 Common Sequence Ops

Operation	Description
`len(s)`	Number of elements

Operation	Description
`min(s)`	Minimum element
`max(s)`	Maximum element
`list(s)`	Convert sequence to list

11.5 Strings Are Immutable

Use slicing and string operations to manipulate.

---

12. Mutable Sequences: Lists

12.1 Assign to Element

```
lst[i] = value
```

12.2 Assign to Slice

```
lst[i:j] = [a, b]
```

Can grow or shrink list.

12.3 Delete Items

```
del lst[i]          # Delete index
del lst[i:j]        # Delete slice
```

12.4 List-Specific Methods

Method	Description
`append(v)`	Add element to end
`extend([v])`	Add multiple elements to end
`insert(i, v)`	Insert before index i
`pop()`	Remove and return last item
`pop(i)`	Remove and return item at index i
`count(v)`	Count occurrences
`index(v)`	Index of first matching item
`sort()`	In-place sort

12.5 Efficiency of Append

- Python's `list.append()` is amortized O(1).
- Internally manages over-allocated capacity for efficiency.

Example performance analysis:

```
n total appends → total cost ≈ 2n → amortized O(1)
```

Allocates memory in powers of 2 to avoid frequent reallocation.

---

13. Summary

This lecture introduces Python from both basic and intermediate perspectives. It covers Python's core types and data structures, distinguishing between mutable and immutable objects, and demonstrates data parsing using list comprehensions and functions like `zip()`. The unique object model of Python is explained, illustrating identity, type, and value. Sequences such as strings and lists are reviewed in depth, including operations like indexing, slicing, and mutation. The efficiency of list operations such as `append()` is examined using amortized analysis. Throughout, students are encouraged to adopt Pythonic idioms and reasoning to write clean, efficient code.

Client-Server Computing, Node.js, and Event-Driven Programming

1. Introduction

This lecture covers foundational topics from distributed computing, specifically:

- Client-server models
- Introduction to Node.js
- Event-driven programming
- Performance and reliability concerns in networked systems
- Comparative discussion of different distributed computing architectures

---

2. Client-Server Architecture

Overview

A client-server model is a fundamental pattern in distributed systems where:

- A server hosts resources or services (e.g., databases)
- A client requests access to those services

Client-Server Communication

- Runs on a network
- Typically modeled visually with clients and servers connected via a line or "cloud"
- The application logic is shared between client and server

Server Responsibilities

- Maintains the authoritative "state of the system".
- Handles all significant data (e.g., enrollment records at a university).
- Enforces rules of the system (e.g., capacity limits).

Client Responsibilities

- Sends requests to the server.
- Has little or no authoritative state information.
- Acts primarily as an interface or interaction mechanism.

Real-world Example: Course Enrollment

- Students attempt to enroll in a class via a client (browser)
- The server determines whether a student is successfully enrolled
- Only servers possess the true, authoritative state

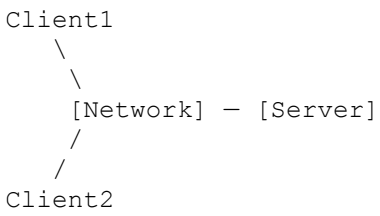
Multiple Clients, Server Mediator Model

- Many clients can exist simultaneously
- Clients communicate only with the server, not directly with each other
- Race conditions can occur (e.g., two clients attempt to enroll at the same time)

Server resolves such conflicts by:

- Picking a winner
- Rejecting both (error)
- Or mistakenly accepting both (undesired)

Diagram:



---

3. Event-Driven Programming

Definition

A different programming paradigm useful in environments where events happen spontaneously (like web servers).

Asynchronous Events

- Events arrive from the external environment.
- May include:
  - Web requests.
  - Results from external databases.
  - File read completions.

  “Events arrive, and code is run in response.”

Key Concepts

Synchronous Program	Event-Driven Program
Linear execution	Loops + Handlers
Blocking I/O	Non-blocking I/O

The Event Loop

A core concept underpinning environments like Node:

```
while (true) {
  event = getNextEvent();
  handleEvent(event);
}
```

Constraints of Event Handlers

- Should be FAST (e.g., microseconds to a few milliseconds)
- Avoid blocking functions (e.g., file.read(), network.read())

Blocking I/O causes:

- Delay in serving new requests
- Queued inputs
- Unresponsive systems

Example of bad event handler:

```
function handleEvent() {
  const result = readBigFileSync(); // This blocks - BAD!
}
```

Proper Strategy

- Split long tasks into short, fast handlers
- Schedule continuation via next event trigger

Example:

From:

```
for (let i = 0; i < n; ++i) {
   doHeavyTask(i);
}
```

To:

```
function doOne(i) {
   doHeavyTask(i);
   scheduleNext(() => doOne(i+1));
}
```

Problems in Event-Driven Code

Potential Issues:

- Higher overhead per action.

- Error handling becomes complex.
- Loss of execution order.
- Difficult debugging and reasoning.

Multi-threading vs Event-Driven

Feature	Event-Driven	Multi-threading
Concurrency Model	Single-threaded, non-blocking	Multiple threads with context switching
Shared State	No (single-thread = no races)	Yes (needs locks to avoid races)
Performance Potential	Limited CPU parallelism	High parallel performance
Safety	No race conditions	Race conditions possible
Complexity	Moderate	High due to synchronization

Key Benefit of Event-Driven

- No need for **locks**.
- Less prone to **race conditions**.

Limitations

- No true parallelism (one core at a time).
- If an event handler crashes or takes too long, the whole system halts.

---

4. Introduction to Node.js

Overview

- JavaScript runtime for event-driven applications
- Runs JavaScript outside the browser (e.g., on servers)
- Allows asynchronous I/O using callbacks/event loop
- Built on Chrome's V8 JavaScript engine

The Node Runtime

- Operates using the event loop.
- Non-blocking I/O.
- Built on the V8 JavaScript engine.

JavaScript in Node

- Originally a browser language.
- Node allows JavaScript to run on servers (no GUI required).
- Async nature fits web servers well.

  “Node is for web servers that juggle many clients.”

---

5. Writing a Basic Node.js Web Server

Source Code

```
const http = require('http');
const ip = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello CS35L World\n');
});

server.listen(port, ip, () => {
  console.log(`Server running at http://${ip}:${port}/`);
});
```

Explanation

Line	Purpose
1	Load the HTTP module.
2	Set the local IP address.
3	Define the port number.
5–9	Set up request handler (callback).
10–12	Start the server and define startup callback.

Special JS Feature:

```
console.log(`Server running at http://${ip}:${port}/`);
```

- Example of a **template literal** (uses backticks ``).
- Allows expression embedding: `${variable}`.

Demo

- Run: `node app.js`
- Access: `http://localhost:3000`
- Response: “Hello World”

---

6. Distributed Computing Models

1. Client-Server Model

- Explained earlier.
- Centralized control and data.
- Problems:
  - Scalability.
  - Single point of failure.

2. Peer-to-Peer (P2P)

- Each peer has its own data and control.
- No centralized server.

- Pros:
  - Scalable.
  - Redundancy.
- Cons:
  - Complex consistency.
  - Data propagation challenges.

3. Primary-Secondary (Master-Worker)

- One **primary/master** controls the computation.
- Many **secondary/worker nodes** perform tasks.
- Used in:
  - ML Training.
  - Distributed DBs.

Model	Use Case	Pros	Cons
Client-Server	Small to medium apps	Simple	Scalability
P2P	Decentralized systems	Scalable	Complexity
Master-Worker	Parallel computation	Control	Bottleneck at primary

---

7. Distributed Systems Challenges

A. Performance Problems

Throughput

- Number of operations per second
- Optimization methods:
  - Handle events out-of-order
  - Prioritize cached contents
  - Balance workload

Latency

- Time between request and response
- Optimization methods:
  - Cache on client
  - Use faster protocols
  - Reduce payload sizes

Latency Mitigation: Browser Cache Example

- Caches previously fetched results
- Avoids round trips to server

B. Correctness Problems

Serialization

- Ensure multiple events result in consistent output
- Even if order of execution differs
- Software must be able to "simulate" serial logic

Example:

```
Requests A, B, C → Executed in order C, A, B
But result same as A, B, C → OK (Serializable)
```

Cache Staleness

- Client caches may become stale.
- Solutions:
    1. Validate cache by comparing server state identifier (UUID)
    2. Accept that cache is stale, if app can tolerate (e.g., video games)

Condition	Strategy
Cache critical	Validate with server
Cache negligible (e.g., news)	Ignore freshness slightly

---

8. Deep Dive: Legacy Network Technology

Circuit Switching Model

- Used in traditional (1960s–1980s) telephone systems.

Mechanism:

- A **dedicated communication path** is established for each call.

Strengths:

- Guaranteed bandwidth
- Efficient for long stable conversations

Problems:

- Inefficient
- Poor scalability
- Prone to failure if any intermediate node fails
- Motivated by Cold War defense needs

Diagram:

```
Phone A → Central Office → [Switch Network] → Central Office B → Phone B
```

---

9. Packet Switching (Modern Networks)

Overview

The backbone of the modern Internet.

Mechanism:

- Data is split into **packets** (~1000 bytes).
- Each packet routes independently via routers.

Advantages:

- Resilient: if one node fails, packets find alternative routes.
- Efficient: no need to hold a full path open.
- Allows for re-routing around network failures

Key Terminology:

Term	Explanation
Packet	A bundle of data (≤1500 bytes)
Router	Device routing packets
IP	Internet Protocol
Hop	Transition to next router

Routing independence: Each packet can take a different path, allowing flexible and dynamic routing.

Conclusion & Key Takeaways

- Event-driven programming requires different thinking – design has to focus on performance AND correctness
- Node.js makes it easy to write scalable, single-threaded network apps
- Client-server is the dominant model, but alternatives should be considered
- Performance tuning (latency and throughput) involves caching, async logic, or distributing load
- Older models like circuit-switching help understand today's infrastructure evolution

---

Appendix: Requests and Status Codes Table

Code	Meaning
200	OK
404	Not Found
500	Internal Server Error

Summary

This lecture explores foundational concepts in client-server computing and networking, emphasizing the transition from traditional single-machine applications to distributed systems involving multiple interacting programs. It introduces the client-server model, detailing how clients interact with servers over networks, often relying on the server to store and manage application state, and discusses common pitfalls such as race conditions and request timing conflicts. The discussion includes introductory coverage of technologies like HTML, CSS,

JavaScript, and particularly Node.js, highlighting its event-driven, asynchronous programming model that contrasts with synchronous, sequential paradigms taught in introductory programming courses. It delves into the structure and benefits of event-driven programs, including reduced complexity from avoiding multithreading and avoiding resource locking, while also discussing their limitations, including lack of true parallelism and potential vulnerabilities if event handlers are mishandled. The lecture further explores distributed systems challenges such as latency, throughput, and correctness, examining strategies like caching, serialization, and maintaining consistency. Lastly, it contrasts modern packet switching with historic circuit switching models, explaining routing flexibility, fault-tolerance, and efficiency improvements critical to today's internet architecture.

Internet Technology and Web Services

1. Introduction to Web Technology Evolution

This lecture covers the foundations of Internet technology, key concepts around protocol layers, and how these technologies underpin the development of the World Wide Web. Topics include the problems related to packet-switched networking, layered protocols, TCP/IP stack, and core web services such as HTTP, HTTPS, and HTML.

---

2. Fundamental Concepts of Networking

2.1 Circuit Switching vs. Packet Switching

- **Circuit Switching**: Dedicated path between endpoints (e.g., traditional telephony)
- **Packet Switching**: Data is sent in small units (packets) that may take different paths to the destination (used in the Internet).

Feature	Circuit Switching	Packet Switching
Path setup	Required	Not required
Reliability	More predictable	Less predictable
Efficiency	Low (fixed path)	High (dynamic routing)

2.2 Definition and Structure of Packets

- Packet: Data unit typically 1–2 KiB in size.
- Structure:
  - **Header**: Control info (e.g., destination, protocol type); for routers
  - **Payload**: Application data; for recipient applications

Example Packet Composition:

Field	Purpose
Header	Routing, protocol type, TO/FROM
Payload	Email text, file content, etc.

---

3. Protocols and Packet Switching Challenges

3.1 What Is a Protocol?

- Protocol: Agreed rules for message transmission and processing
- Analogy: Diplomatic protocols required to speak to foreign leaders
- Each protocol layer defines what the structure and behavior of data exchanges look like.

Types of Protocol Behaviors:

- Message formats (e.g., header structures)
- Response behavior (e.g., what to do on success/failure)

3.2 Challenges in Packet Switching

Because packets travel independently, several challenges arise:

1. Packet Loss

- Common causes: router overload (buffer overflow), network congestion.
- Packets can be silently dropped.
- Loss rate varies: 1% to over 50% under high load.

2. Packets Received Out of Order

- Common due to multiple network paths.
- Can confuse applications requiring strict order (e.g., file uploads).

3. Packet Duplication

- Unusual, but possible with network misconfigurations (e.g., routers misconfigured as bridges).

4. Packet Corruption

- Bit errors due to unreliable hardware, faulty links.
- Need checksum/error-detection mechanisms.

---

4. Layered Network Architecture

4.1 Overview

To handle network complexity and address challenges modularly, protocols are structured into layers.

Layer	Functionality
Application	High-level data exchange (e.g., HTTP, HTML)
Transport	Reliable or fast delivery (e.g., TCP, UDP)
Internet	Routing across the network (e.g., IP)
Link	Data link between adjacent network nodes

Other models (e.g., OSI) define 7 layers (application, presentation, session, transport, network, data link, physical)

Notes:

- Layers act as abstractions over lower ones.
- Data encapsulation uses headers at each layer.

4.2 The Link Layer

- Hardware-specific protocols (e.g., Ethernet, Wi-Fi)
- Handles communication over a single, direct link

Example Link Layer Technologies:

Technology	Description
Ethernet	Wired LAN standard
Wi-Fi	Wireless communication
USB	Peripheral connections

4.3 The Internet Layer

- Primary protocol: IP (IPv4 or IPv6)
- Responsible for global addressing and packet routing via IP Addresses
- Stateless, best-effort delivery
- Introduces problems like loss, reordering, duplication

4.4 The Transport Layer

- Manages reliable delivery over unreliable internet layer
- Introduces concept of:
  - Data channels (logical streams)
  - Reliability
  - Ordering
  - Error Correction
- Protocols: TCP and UDP

4.5 The Application Layer

- Protocols meant for specific services (e.g., HTTP, FTP, SMTP)
- Builds on lower layers but focuses on the user's domain concerns

---

5. Internet Protocol (IP)

5.1 IPv4 Overview

- Introduced in 1983 (John Postel, UCLA alum).
- Connectionless — just delivers packets.

5.2 Fields in IPv4 Header

Field	Description
Length	Packet size in bytes
Protocol Number	Defines encapsulated protocol (e.g., TCP = 6; UDP = 17)
Source IP	32-bit address of sender
Destination IP	32-bit address of recipient
TTL	Time-to-live to prevent infinite loops, decremented at each router hop
Checksum	Simple 16-bit checksum for packet corruption detection

Notation:

- Binary: 11000000.10101000.00000000.00000001
- Decimal: 192.168.0.1

5.3 Addressing Schemes – IPv4 vs IPv6

Limitations of IPv4

- 32-bit address space (~4.3 billion addresses) proved insufficient.
- Large headers and lack of certain metadata features.

IPv6

- Introduced in 1998, addresses shortage issues.
- Features:
  - 128-bit addresses
  - More header fields
  - Additional features (not covered in detail)
- 40% adoption (still increasing)
- Efficiency trade-offs due to extra header size

Version	Address Size	Release Year	Supports
IPv4	32-bit	1983	~4.3B addresses
IPv6	128-bit	1998	2^128 addresses (future-proofing)

Adoption status:

- IPv4 still widely used (60%)
- IPv6 rising (40%) due to address exhaustion

5.4 Checksums and Reliability

- Internet checksum (16-bits) for error detection
- Cannot stop malicious actors, not cryptographic
- Used to catch accidental data corruption
- End-to-End Principle: Each layer should do its error checking

---

6. Transport Layer Protocols

6.1 User Datagram Protocol (UDP)

- Barebones protocol
- Source Port & Destination Port
- Unreliable and unordered delivery
- No guarantees: Sender must handle ordering, errors, etc.
- Used in:
  - Streaming, DNS, real-time telemetry
  - Situations where speed > reliability
- Does not resolve packet issues like loss or order

Use Case Example: IoT device sending temperature once every 10 minutes

6.2 Transmission Control Protocol (TCP)

- Reliable, connection-oriented
- Implements:
  - **Reliability**: Retransmission of lost packets
  - **Ordered data**: Reassembly of out-of-order packets
  - **Error checking**: End-to-end validation
- Guarantees:
  - In-order, lossless, error-checked delivery

Reliability Features Table:

Challenge	TCP Feature That Solves It	Description
Loss	Retransmission	Re-sends lost data
Out-of-Order	Reassembly	Orders packets before delivering to application
Duplication	Sequence numbers, ACK	Using sequence numbers to identify and discard redundant data
Overflow	Flow Control	Avoid flooding networks. Sender adjusts speed based on network conditions.

Common use cases:

- Submitting assignments
- Large data transfers
- Web traffic with HTTP(S)

---

7. Protocol Specification – How to Create a Protocol

- Define:
  - Packet formats and headers
  - Expected behaviors
- Documentation (spec), not code
- Analogy: Like C++ standard (C23, C26)

- Failing to follow spec = non-functional applications

Example:

- What headers are required?
- What order are fields?
- What responses to invalid requests?

---

8. The Web Fundamentals

8.1 Introduction and Web History

Invented by Tim Berners-Lee at CERN with two main components:

1. **HTTP** (Hypertext Transfer Protocol)
2. **HTML** (Hypertext Markup Language)

Original Goals:

- Create a simple way to share and navigate research papers.
- Relied on:
    - Simple document formatting (HTML)
    - Simple app-layer protocol (HTTP over TCP)

First Web Server:

- Hosted on Berners-Lee's NeXT workstation
- Sign: "Do not power off" — critical to early web functionality- First web server hosted on his workstation (CERN)
- Protocol: HTTP, Markup: HTML

8.2 HTTP – Hypertext Transfer Protocol

8.3 HTTP Protocol Evolution

Version	Year	Key Features	Approx Usage
HTTP/1.0	1990s	One request per connection	Legacy usage
HTTP/1.1	1999	Persistent connections, Host headers	~10%
HTTP/2	2015	Based on TCP; improves performance	~59%
HTTP/3	2022	Based on UDP via QUIC; optimized for real-time media	~32%

HTTP/1.0

- Built on top of TCP
- Request-response model (initially one request per TCP connection)

Example Request:

```
GET / HTTP/1.0
<empty line>
```

Example Response Header:

```
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2020 00:00:00 GMT
Server: Apache
Content-Length: 12345
Content-Type: text/html
Connection: close
```

HTTP Header Fields (Example)

Field	Purpose
Date	Timestamp of response
Server	Web server software (e.g., Apache)
Last-Modified	When resource last changed
E-Tag	Resource identifier/version
Accept-Ranges:	Allow clients to request only part of a resource in bytes
Content-Length	Byte count of body
Connection	Whether connection should be closed after response
Content-Type	Media type (e.g., text/html, image/jpeg)

HTTP/1.1

- Adds:
  - Persistent connections
  - Enables multiple requests per TCP connection
  - Host header support (supports multiple domains per server)
  - Improves network efficiency

HTTPS and Security

- Used today predominantly
- Encrypts HTTP using TLS
- Prevents:
  - Packet inspection (privacy)
  - Packet tampering (integrity)
- Tools: GnuTLS CLI, OpenSSL

HTTP/2

Launched 2015, 59% adoption

In HTTP/1.1, each request–response pair typically required either:

- Its own TCP connection, or
- Had to wait (block) if reusing a connection (due to head-of-line blocking). This made web pages slow to load because:
- You could only send one request at a time per connection.

- Or, you had to open lots of TCP connections, which increased overhead.

What HTTP/2 does differently with multiplexing:

- Single TCP connection: All communication between the client and server happens over one connection.
- Streams: Within this connection, many streams can be open at once — each representing a request/response.
- Interleaving: HTTP/2 breaks data into small packets called frames and interleaves them — so parts of different streams can be sent back and forth simultaneously.
- No blocking: Because of this, one slow response won't block others.

Feature	Purpose
Header Compression	Reduces size of repetitive headers
Server Push	Server sends multiple resources as a response
Pipelining	Multiple requests dispatched without waiting for responses sequentially
Multiplexing	Allows interleaved transmission of multiple streams (faster responses)

Example Multiplex Order:

Request	Response
A	B
B	D
C	A

HTTP/3

- Developed by Google, 2022
- 32% adoption
- Built atop **QUIC**, which uses **UDP**
- Motivation: avoid TCP's "head-of-line blocking"
- Enables:
  - Partial loss tolerance (key for streaming apps like Zoom)
  - Encryption by default (TLS is integral)
  - Better performance over unreliable networks

Characteristics of QUIC:

- Built by Google (Jim Roskind)
- Supports:
  - Streams with tolerable loss
  - Multiple streams between two endpoints
  - Improved performance compared to TCP

---

9. HTML – HyperText Markup Language

- Based on SGML (Standard Generalized Markup Language)
- HTML: Simplified + includes "hypertext" (links, interactivity)
- Uses angle-bracket tags:

```
<p>This is a paragraph</p>
```

- Lowercase convention (vs SGML's uppercase)

Elements:

- Tags: <p>, <a href>, <h1>, <img src>, etc.
- DOM structure: Tree of nested elements

HTML Example:

```
<html>
  <head><title>My Web Page</title></head>
  <body>
    <h1>Hello, World!</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

---

10. Additional Questions and Examples

1. Why is pipelining/multiplexing useful?
    - Reduces latency (particularly with async data)
    - Lets humans interact without waiting
2. Why can TCP be inefficient for video?
    - Retransmitted packets introduce delay/stutter. QUIC solves this.
3. Why don't all use IPv6?
    - Legacy hardware, software
    - Institutional inertia (IPv4 still works)
4. What is head-of-line blocking?
    - If one lost packet blocks all subsequent ones from being processed

---

Summary

This lecture introduced foundational concepts of Internet technology and packet-switched networking, with a focus on the importance of protocol layers in handling reliability, order, and communication abstraction. It covered key network layers (link, internet, transport, application) and exemplified how the issues of packet loss, duplication, and disorder are abstracted away primarily in the transport layer via protocols like TCP. The lecture also analyzed the evolution of the web, beginning with the HTTP and HTML protocols invented by Tim Berners-Lee, illustrating how higher-layer web services rely on and build upon the lower networking layers. Finally, it delved into modern advancements such as HTTP/2 and HTTP/3 and their performance optimizations and trade-offs.

HTML, the DOM, CSS, JavaScript, JSX, and Python Dictionaries

1. HTML: Structure and Design Principles

1.1 Origins and Goals

- HTML is based on SGML (Standard Generalized Markup Language).
- SGML was designed with two key ideas:
  - **Portability** across different publishers.
  - **Separation** of content from form (presentation).- HTML inherits the idea that:
  - **Content**: What the user wants to convey.
  - **Form**: How content is presented (appearance/style).
- HTML is declarative: it specifies desired output, not the steps required to achieve it.

1.2 Declarative vs. Imperative

- HTML is a declarative language: it describes what should appear on the screen (structure + content).
- JavaScript is imperative: describes how things should be done procedurally(logic + behavior).
- Original design philosophy:
  - Prefer HTML and CSS for content and styling.
  - Use JavaScript only when needed to control behavior.
- This philosophy wasn't always maintained in practice.

1.3 Goals in Designing HTML

- Support consistent rendering across different devices.
- Ensure that the same content can appear appropriately on laptops, phones, etc.
- Allow browsers the flexibility in rendering to improve usability and accessibility.

1.4 HTML Syntax and Elements

1.4.1 HTML Elements

- An HTML element corresponds to a node in the document's tree structure (DOM).
- Basic syntax:

```
<tagname attribute="value">Content</tagname>
```

- The tree is kept in memory (DOM) by browsers and manipulated via JavaScript.

1.4.2 Nesting and Valid Syntax

- Tags must be properly nested.
  - Invalid:

```
<a><b>…</a></b>
```

- Valid:

  ```
  <a><b>…</b></a>
  ```

- Mismatched or improperly nested tags typically render incorrectly, but browsers are forgiving.

1.4.3 Optional and Void Tags

- Closing tags can sometimes be omitted if the structure is unambiguous.
- Void elements: represent standalone elements with no children.
  - Example:

    ```
    <br>
    ```

  - Equivalent to:

    ```
    <br></br>
    ```

    But HTML recommends using the shorter form.

1.5 Attributes

- Elements can have attributes: name-value pairs.
- Syntax:

  ```
  <img src="image.jpg" width="500" height="300">
  ```

- The attribute name and value are strings and must be unique per element.
- Attributes must appear in the opening tag.

1.6 Types of Elements

Element Type	Characteristics	Example Tags
Normal	Can contain mixed content and children	`<p>`, `<div>`
Raw Text	Can only contain text (rare)	Could be `<script>`
Void	Cannot have any content or children	` `, `<img>`

1.7 DTD and HTML Versions

- Early HTML specified strict rules using DTD (Document Type Declaration).
- DTD specifies:
  - Which elements are allowed.
  - Their types (void, raw, normal).
  - Allowed attributes.
  - Valid nesting and element hierarchy.
- HTML 1–4 used this system.
- Issues:
  - Too rigid and slow to adapt to evolving browser innovation.
  - Standards couldn't keep pace with browser capabilities.

1.8 HTML5 and Living Standards

- HTML5 introduced in ~2008 with a "living standard" approach, abandoning rigid versioning.
- Living standard: constantly updated without version freezes.
- Specification now maintained online at w3.org.
- Practical and responsive to web development pace.
- Allows for more flexibility and quicker browser evolution.

1.9 Generous Syntax Parsing

- Browsers are forgiving: they parse incomplete or incorrect HTML to maximize content rendering.
- This ensures users see as much of the page as possible despite developer errors.
- Contrast to strict languages like C++ which halt on syntax errors.

---

2. XML vs HTML

2.1 XML Overview

- XML = Extensible Markup Language.
- Similar to HTML but stricter syntax.
  - Closing tags are mandatory.
- Domain-specific extensions.
- XML is declarative and used in data-exchange scenarios.
- XHTML is XML-based HTML.

2.2 XML Use Cases

- Used in government and enterprise applications for structured data.
- Useful when correctness and validation are important (e.g., social security data). Primarily used for Data transmission.

2.3 XML vs HTML Syntax

Feature	HTML	XML
Closing Tags	Often optional	Required
Error Tolerance	Very forgiving	Strict
Usage	Web pages (UI)	Structured data, APIs

- XHTML is an XML variant used for well-formed HTML pages.
- Data governance is more reliable in XML (useful in critical fields).

---

3. DOM: Document Object Model

3.1 DOM Overview

- DOM: Tree of HTML content represented as objects.
- JavaScript accesses and manipulates this tree.
- Nodes correspond to elements, text, attributes, etc.

3.2 Key Properties

- Object-Oriented API to interact with the structure of a document.
- Represented in RAM by browsers.

3.3 Operations with DOM

- Navigation: locate elements (`getElementById`, etc.).
- Traversal: visit nodes to search or extract data.
- Modification: change structure, add or remove elements.

3.4 DOM APIs

- Provided by browsers.
- Language-agnostic (usable beyond JavaScript).
- JavaScript is the dominant language using DOM today.

---

4. CSS: Cascading Style Sheets

- Separates content (HTML) from presentation (style).
- Avoids embedding style details directly in HTML.

4.1 Problem with Early HTML Styling

- Direct styling embedded in content:

```
<i>italics</i> <b>bold</b>
```

- Mixed content and presentation, making adaptation difficult.

4.2 Modern Semantic Styling

- Use semantic tags:

```
<em>emphasis</em> <strong>strong emphasis</strong>
```

- Use CSS to define presentation.
- Advantages:
  - Makes pages accessible (e.g., screen readers for visually impaired).
  - Adapts better to various devices.

4.3 CSS Inheritance and Cascading

- Styles apply hierarchically.
- Child elements inherit parent's styles unless overridden.

4.4 Sources of Style

Source	Description
Browser	Default styles
User	Custom overrides via browser settings

Source	Description
Author	Defined in the webpage/CSS files

4.5 Style Declaration

- Inline:

```
<span style="font-variant: small-caps;">Styled Text</span>
```

- External:

```
<link rel="stylesheet" href="styles.css">
```

4.6 CSS Syntax Example

```
<span style="font-variant: small-caps;">Sample Text</span>

selector {
  property: value;
}
```

4.5 CSS Usage

- Designed for visual presentation (more design-focused).
- Collaboration between developers (logic) and designers (presentation).

---

5. JavaScript and JSX

5.1 JavaScript Basics

- Dynamic programming language embedded in HTML.
- More dynamic than Python (e.g., runtime decisions).
- Used for interactiveness and logic in webpages.

5.2 Embedding JavaScript in HTML

Two methods:

1. External File:

```
<script src="hello.js"></script>
```

2. Inline Code:

```
<script>
  // code here
</script>
```

5.3 Reasons to Use External Scripts

- Pros:
  - Modularity and code reuse.
  - Easier updates.

    - Support for 3rd-party libraries (no copy-and-paste).
    - Better for copyright compliance.
- Cons:
    - One extra HTTP request
    - More complicated than inline scripts

---

6. JSX: JavaScript XML

6.1 What is JSX?

- A syntax extension to JavaScript used with React.
- Allows writing HTML-like code within JavaScript.
- Transpiled to standard JavaScript behind the scenes.

6.2 JSX Example

```
const header = <h1 lang={"en"}>CS35L Homework</h1>;
```

- JSX expressions can include variables:

```
const course = "CS35L";
const header = <h1>{course} Homework</h1>;
```

6.4 JSX Rendering

```
ReactDOM.render(header, document.getElementById("root"));
```

- Attaches the JSX-generated element to the DOM.

6.5 Nesting and Flexibility

- JSX supports deeply nested elements.
- JavaScript expressions can be embedded within curly braces.

---

7. JSON vs XML

7.1 JSON: JavaScript Object Notation

- Lightweight data-interchange format.
- Commonly used in web APIs and async data loading.
- Readable and writable directly in JavaScript.

JSON (JavaScript Object Notation)

- Trees where labels apply to arcs (key-value pairs).
- Serialized JavaScript object representation.

Example:

```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        { "value": "New", "onclick": "CreateNewDoc()" },
        { "value": "Open", "onclick": "OpenDoc()" }
      ]
    }
  }
}
```

7.4 JSON vs XML Tree Representation

Feature	JSON	XML
Label Position	On arcs (keys)	On nodes (element names)
Syntax	Curly braces/arrays	Tags with attributes
Parsing	Native in JavaScript	Requires XML parser

7.5 Comparison to XML

Feature	JSON	XML
Syntax Simplicity	Simpler	Verbose
Readability	Easier horizontally	Harder
Data Orientation	Keys on arcs	Tags on elements (nodes)
Parsability	Built into JavaScript	Requires external library

Format	Description	Example Usage
XML	Verbose, hierarchical	Government, legacy systems
JSON	Lightweight, JavaScript-native	Modern web applications

---

8. Browser Rendering Pipeline

8.1 Stages

1. Parse HTML → create DOM
2. Apply CSS → compute layout
3. Render → display pixels

Optimization Strategies

- Lazy rendering: Start showing content before full parsing completion.
- Skip off-screen elements: Don’t render or execute their JS until visible.
- Parallel execution: JavaScript may run while parsing and rendering continues.
- Browsers render partially parsed HTML for responsiveness.

8.3 JavaScript and Rendering

- JavaScript execution may be delayed:
  - If DOM is not visible (e.g., below-the-fold elements).

    - If rendering is incomplete.
- JavaScript and rendering can interleave.
    - Can lead to race conditions if DOM is mutated before render.

Issue	Explanation
Self-modifying DOM	JavaScript modifies the DOM during load
Deferred execution	JS scripts for hidden elements may not run immediately

8.4 Performance Tip

- Be careful where `<script>` tags are placed in HTML.
- Improper placement affects load speed and user experience.

8.5 Testing Concerns

- Fast local networks may hide timing issues.
- Must test on variable latency connections for robustness.

---

9. Python: Dictionaries and Data Types

9.1 Mapping Types – `dict` (Dictionaries)

- Stores key-value pairs.
- Keys must be hashable and immutable.
- Values can be of any type (mutable/immutable).

9.2 Syntax & Operations

```
d = {"a": 1, "b": 2}
d["c"] = 3            # Add item
print(d["a"])         # Access value
del d["b"]            # Remove item
```

9.3 Core Dictionary Operations

Method	Description
`len(d)`	Number of key-value pairs
`d.clear()`	Remove all pairs
`d.copy()`	Shallow copy of dictionary
`d.items()`	View object of (key, value) tuples
`d.keys()`	View object of keys
`d.values()`	View object of values

9.4 Views and Mutability

- `.items()`, `.keys()`, & `.values()` return view objects:
    - Dynamic: reflect changes in dictionary.
    - Not independently mutable.

9.6 Copying vs Assignment

Assignment:

```
e = d  # e refers to the same dict as d
```

Copy:

```
e = d.copy()  # new dict, shared keys/values
```

9.7 Shallow vs Deep Copy

- Shallow copy:
  - Dict is copied.
  - Mutable values (like lists) are shared.
  - `.copy()` is a shallow copy.
- Deep copy (via `copy.deepcopy()`):
  - Recursively copies everything, including values.

9.8 Dictionaries Are Ordered (Python 3.7+)

- Insertion order is preserved.
- Reassigning a key does not affect its position.
- Deleting and re-adding a key puts it at the end.

9.9 Iterating Over Dictionaries

Example:

```
for key in d:
    print(key, d[key])
```

Or:

```
for key, value in d.items():
    print(key, value)
```

Summary

This lecture detailed the foundational concepts, history, and technical implementation of HTML, XML, the DOM, CSS, JavaScript, and related technologies. It explored declarative and imperative paradigms of web applications, described HTML syntax rules, different types of elements (void, raw, normal), and discussed the evolution from DTD-based HTML to HTML5's living standard model. The lecture introduced the DOM as an object-oriented representation of HTML/XML trees, explained the role and application of cascading style sheets (CSS) to separate content from presentation, and covered JavaScript and JSX as mechanisms for imperative logic and efficient DOM manipulation. Lastly, it compared data-exchange formats XML and JSON, outlined the browser rendering pipeline for performance optimization, and provided a comprehensive overview of Python dictionaries, focusing on their operations, immutability, view objects, and performance characteristics.