**Module 1 -LAB EXERCISE**

**Overview of IT Industry.**

**Q1. Write a simple "Hello World" program in two different programming languages of your choice. Compare the structure and syntax.**

**C** **language**:

#include <stdio.h>

int main() {

printf("Hello World");

return 0;

}

**Python** **language**:

print("Hello World ")

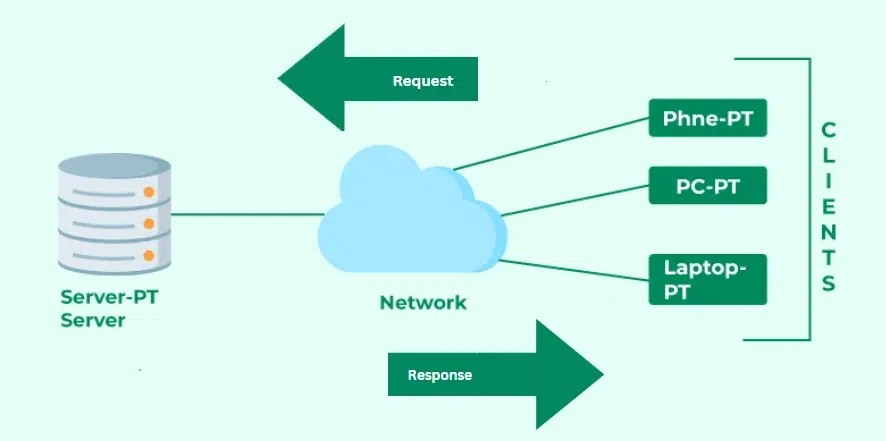
**Comparison:**

**Syntax Simplicity:** Python is simpler and more readable, needing just one line.

**Structure:** C requires more structure like #include, main() function, and return statement.

**Use Case:** Python is interpreted and best for beginners; C is compiled and better for system-level programming.

**Q2. Research and create a diagram of how data is transmitted from a client to a server over the internet.**



**Q3. Design a simple HTTP client-server communication in any language.**

A) Server(simple HTTP server):-

import socket

import argparse

import os

import datetime

import time

# parse command line arguments

parser = argparse.ArgumentParser()

parser.add\_argument("port", type=int, nargs='?', default=80 )

args = parser.parse\_args()

PORT = args.port

# create socket and lis

ten for client connections

serversocket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

serversocket.bind(("localhost", PORT))

serversocket.listen(1)

print(f"HTTP server started, serving at port {PORT}\n")

# infinite loop to check if a client is connecting

while 1:

clientsocket = None

try:

clientsocket, address = serversocket.accept()

request = clientsocket.recv(4096).decode()

# GET or PUT

method = request.split()[0]

# get the file/directory associated with request

# remove extra backslash char at beginning

request\_uri = request.split()[1][1:]

msg = str()

# process GET requests

if method.strip() == 'GET':

#check if the file exists

if os.path.exists(request\_uri):

modifiedTime = time.localtime(os.path.getmtime(request\_uri))

\_, ext = os.path.splitext(request\_uri)

#send the status

msg = (f"HTTP/1.1 200 OK\r\n"

f"Date: {str(datetime.datetime.now())}\r\n"

f"Server: simple\_server\r\n"

f"Last-Modified: {time.strftime('%Y-%m-%d %H:%M:%S', modifiedTime)}\r\n"

f"Content-Type: text/{ext[1:]}\r\n"

f"Content-Length: {str(os.path.getsize(request\_uri))}\r\n\r\n" +

open(request\_uri, "r").read())

else:

msg = (f"HTTP/1.1 404 Not Found\r\n"

f"Date: {str(datetime.datetime.now())}\r\n"

f"Server: Simple\_Server\r\n")

# process PUT requests

elif method.strip() == 'PUT':

f = open(request\_uri, "w")

if os.path.exists(request\_uri):

msg = (f"HTTP/1.1 200 OK File Created\r\n"

f"Date: {str(datetime.datetime.now())}\r\n"

f"Server: Simple\_Server\r\n")

else:

# invalid method specified by user

msg = (f"HTTP/1.1 405 Method not Allowed\r\n"

f"Date: {str(datetime.datetime.now())}\r\n"

f"Server:Simple\_Server\r\n")

clientsocket.send(msg.encode())

except KeyboardInterrupt:

# gracefully close socket on keyboard interrupt

if clientsocket:

clientsocket.close()

break

serversocket.close()

**b) Client (Simple HTTP GET Request):-**

import socket

import argparse

# this client can receive this many bytes of data at a time

BUFFER\_SIZE = 4096

# number of seconds before client socket times out

SOCKET\_TIMEOUT = 5

# parse arguments from the command line

parser = argparse.ArgumentParser()

parser.add\_argument("server", type=str, nargs='?')

parser.add\_argument("port", type=int, nargs='?', default='80')

parser.add\_argument("method", type=str, nargs='?')

parser.add\_argument("filename", type=str, nargs='?')

args = parser.parse\_args()

SERVER = args.server

PORT = args.port

METHOD = args.method

FILE = args.filename

try:

# create a client socket and connect it to the server socket

clientsocket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

clientsocket.connect((SERVER, PORT))

# formulate http request from CLI args

request = (f"{METHOD} /{FILE} HTTP/1.1\r\nHost: {SERVER}:{PORT}\r\n"

f"User-Agent: simple\_client\r\nAccept: \*/\*\r\n\r\n")

# send request to server socket

clientsocket.send(request.encode())

# set a timeout for blocking socket operations to prevent hanging up

clientsocket.settimeout(SOCKET\_TIMEOUT)

http\_response = bytearray()

try:

# keep receiving bytes until a response from server is empty

while 1:

response = clientsocket.recv(BUFFER\_SIZE)

if not response:

break

http\_response += response

except socket.timeout:

print("socket timed out, printing response")

print(http\_response.decode())

clientsocket.close()

# close gracefully on keyboard interrupt

except KeyboardInterrupt:

if clientsocket:

clientsocket.close()

**Q4. Research different types of internet connections (e.g., broadband, fiber, satellite) and list their pros and cons.**

**Types of Internet Connections**

**1. Broadband (DSL/Cable)**

* **DSL** uses telephone lines; **Cable** uses coaxial TV cables.

**Pros:**

* Widely available in urban/suburban areas.
* Relatively affordable.
* Always-on connection.

**Cons:**

* Speeds can vary based on location and time of day.
* DSL is slower and more distance-sensitive.
* Shared bandwidth (especially cable) can lead to congestion.

**2. Fiber Optic**

* Uses light signals through fiber-optic cables for ultra-fast data transmission.

**Pros:**

* Extremely fast (up to 1 Gbps+).
* Low latency — ideal for gaming, streaming, and remote work.
* Reliable and not affected by electromagnetic interference.

**Cons:**

* Limited availability in rural areas.
* Installation may be expensive or unavailable.
* May require new infrastructure.

**3. Satellite**

* Connects via satellites orbiting Earth; useful in remote areas.

**Pros:**

* Available almost anywhere (great for rural/remote locations).
* No need for physical cables.

**Cons:**

* High latency due to signal travel distance.
* Slower speeds than fiber/cable.
* Weather can impact connectivity.
* Expensive data plans and equipment.

**4. Fixed Wireless**

* Uses radio signals from towers to homes (like 4G LTE/5G broadband).

**Pros:**

* Quick to install.
* Good for rural and suburban areas without cable/fiber.
* Decent speeds (especially with 5G).

**Cons:**

* Line of sight to the tower is often required.
* Performance can drop with bad weather or obstructions.
* Limited availability.

**5. Mobile (Cellular: 4G/5G)**

* Internet via mobile networks using SIM cards or hotspots.

**Pros:**

* Highly portable.
* Fast speeds (especially with 5G).
* No wiring needed.

**Cons:**

* Data caps and throttling are common.
* Coverage varies by region and provider.
* More expensive per GB than home broadband.

**6. Dial-Up**

* Uses traditional telephone lines with a modem to connect.

**Pros:**

* Extremely widely available.
* Very cheap.

**Cons:**

* Very slow (max ~56 Kbps).
* Ties up phone line while in use.
* Obsolete for most modern applications.

**Q5.**  **Simulate HTTP and FTP requests using command line tools (e.g., curl).**

To simulate an HTTP GET request and display the content of a web page:

**bash**

curl https://example.com/

This command will retrieve the HTML content of the specified URL and display it in the terminal.

* To simulate an FTP download, you can use the command:

**bash**

curl -O ftp://ftp.example.com/file.zip

This command downloads the file file.zip from the FTP server and saves it to the local directory with the same name, [according to Oxylabs](https://oxylabs.io/blog/curl-download-file).

**Q6. Identify and explain three common application security vulnerabilities. Suggest**

**possible solutions.**

**1. SQL Injection (SQLi)**

An attacker manipulates SQL queries by injecting malicious input into form fields or URLs. This can allow unauthorized access, data exfiltration, or even full database control.

**Solution:**

* **Use prepared statements (parameterized queries)**:
* **Input validation and sanitization**
* Use ORM tools (e.g., Sequelize, Hibernate)
* Employ a Web Application Firewall (WAF)

**2. Cross-Site Scripting (XSS)**

XSS occurs when attackers inject malicious scripts into webpages that are then executed in another user’s browser. It can steal cookies, session tokens, or perform actions on behalf of the user.

**Solution:**

* **Escape output**: Properly encode output in HTML, JavaScript, and URLs.
* **Avoid innerHTML rendering** when not necessary.

**3. Cross-Site Request Forgery (CSRF)**

An attacker tricks a logged-in user into submitting a request to a web application (e.g., transferring money) without their knowledge.

**Solution:**

* **Use CSRF tokens**: Generate a unique token per session/form.
* **Use SameSite cookie attribute** to prevent cross-origin requests:
* **Verify referrer or origin headers** in requests.

Use frameworks that include built-in CSRF protection (e.g., Django, Spring).

**Q7. Identify and classify 5 applications you use daily as either system software or application software.**

**System Software vs Application Software**

* **System Software**: Supports the computer’s basic functions (e.g., operating systems, drivers).
* **Application Software**: Helps users perform specific tasks (e.g., browsing, word processing).

**5 Daily Applications: Classification**

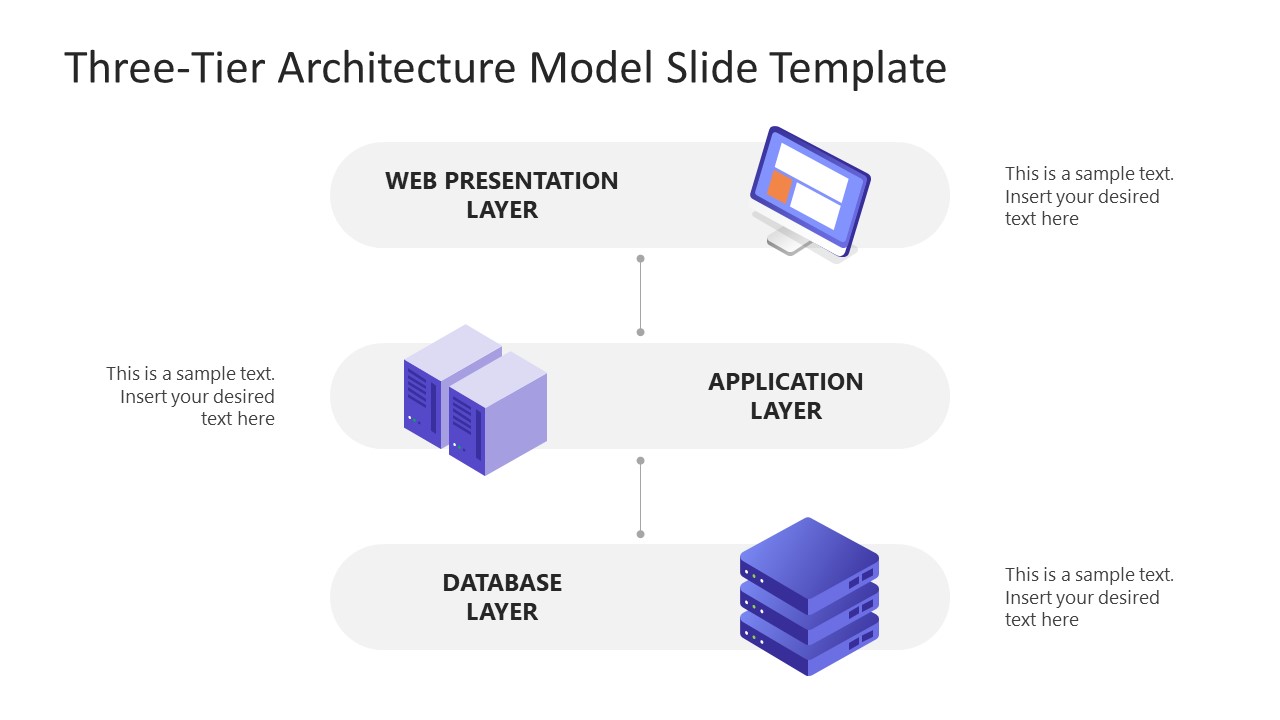
| **Application** | **Type** | **Classification** | **Purpose** |
| --- | --- | --- | --- |
| **Google Chrome** | Web Browser | Application Software | Surf the web, use web apps |
| **Microsoft Word** | Word Processor | Application Software | Create/edit documents |
| **Windows 11** (or macOS) | Operating System | System Software | Manage hardware/software resources |
| **Spotify** | Media Streaming | Application Software | Stream music and podcasts |
| **Device Drivers** | System Utilities | System Software | Allow OS to communicate with hardware |

**Summary:**

* **3 are Application Software**: Chrome, Word, Spotify.
* **2 are System Software**: Windows/macOS, Device Drivers.

**Q8. Design a basic three-tier software architecture diagram for a web application.**

* **Presentation Tier**: User interface (browser or mobile app)
* **Application Tier**: Handles business logic
* **Data Tier**: Manages data storage and retrieval

****

**Q9. Create a case study on the functionality of the presentation, business logic, and data**

**access layers of a given software system**

**Case Study: Online Food Ordering System**

**Purpose:**

Enable users to browse restaurants, place food orders, and make payments online.

**Software Architecture: 3-Layer Architecture**

**1. Presentation Layer (UI Layer)**

* **Function**: Interface between the user and the application.
* **Technologies**: React.js, HTML/CSS, Mobile App (Flutter)

**Example Functionalities:**

* Display restaurant menus
* Form for placing orders
* Login/Register screens
* Shows order history and real-time tracking

**Goal:**

Provide a user-friendly interface while handling user input/output.

**2. Business Logic Layer (Service Layer / Application Layer)**

* **Function**: Core processing of the system — handles rules, logic, and workflows.
* **Technologies**: Node.js, Django, Spring Boot (Java)

**Example Functionalities:**

* Validate user login credentials
* Calculate total price, apply discounts or delivery fees
* Match user with nearby delivery drivers
* Handle order statuses (Pending → Preparing → Delivered)

**Goal:**

Ensure all processes follow business rules consistently.

**3. Data Access Layer (Persistence Layer)**

* **Function**: Interacts with the database to fetch, insert, update, and delete data.
* **Technologies**: SQL (PostgreSQL, MySQL), ORM (Sequelize, Hibernate)

**Example Functionalities:**

* Store user data, order details, and payment information
* Retrieve restaurant menus and item prices
* Update inventory or delivery status

**Goal:**

Secure and efficient data management; abstract database logic from business logic.

**Example Workflow: Placing an Order**

**Step-by-step interaction across layers:**

| **Step** | **Layer** | **Activity** |
| --- | --- | --- |
| 1 | Presentation | User selects items and clicks "Place Order" |
| 2 | Business Logic | Validates items, calculates total, checks user balance |
| 3 | Data Access | Saves the order to the database |
| 4 | Business Logic | Sends order to kitchen or assigns driver |
| 5 | Presentation | Displays confirmation and order tracking |

**Benefits of Layered Architecture**

| **Layer** | **Benefits** |
| --- | --- |
| Presentation | Easier UI customization |
| Business Logic | Reusable and testable code |
| Data Access | Decouples data storage logic |

**Q10. Explore different types of software environments (development, testing, production). Set up a basic environment in a virtual machine.**

**Types of Software Environments**

**1. Development Environment**

* Where developers **write and debug** code.
* Often includes IDEs, version control (e.g., Git), and local servers.
* Can be highly flexible and tailored per developer.

**Tools:** VS Code, Git, Node.js, XAMPP, Docker

**2. Testing Environment**

* Used for **QA testing** to find bugs and ensure quality.
* Mimics production but with test data.
* Can include automated testing, unit testing, and integration testing tools.

**Tools:** Selenium, JUnit, Postman, Jenkins

**3. Staging Environment (optional)**

* Pre-production clone of the **production environment**.
* Used for final validation by QA or stakeholders.
* Should mirror production exactly, including configurations and data flow.

**4. Production Environment**

* The **live** system users interact with.
* Must be stable, secure, and highly performant.
* Monitored with logs and uptime tools.

**Tools:** Nginx, Apache, Kubernetes, AWS, monitoring tools like Grafana or Datadog

**Set Up a Basic Environment in a Virtual Machine**

Let’s create a **basic development environment** using **Ubuntu Linux** in a VM (VirtualBox example):

**Prerequisites:**

* Download and install [**VirtualBox**](https://www.virtualbox.org/)
* Download an **Ubuntu ISO** (e.g., Ubuntu 22.04 LTS)

**Step-by-Step Setup**

**1. Create a VM:**

* Open VirtualBox → New
* Name: DevEnv
* Type: Linux
* Version: Ubuntu (64-bit)
* RAM: 2048 MB or more
* Hard Disk: Create a virtual hard disk (20 GB)

**2. Install Ubuntu:**

* Start VM → Select Ubuntu ISO
* Follow on-screen instructions to install Ubuntu

**3. Install Development Tools:**

After installation, open the terminal in Ubuntu and run:

bash

CopyEdit

sudo apt update

sudo apt install build-essential git curl vim

**4. Install a Language Runtime (e.g., Python):**

bash

CopyEdit

sudo apt install python3 python3-pip

**5. Install a Web Server (optional):**

bash

CopyEdit

sudo apt install apache2

**6. Set Up a Sample Project:**

bash

CopyEdit

mkdir my\_project

cd my\_project

echo "print('Hello from VM!')" > app.py

python3 app.py

**Snapshot Your Environment**

Once the setup is complete, **take a snapshot** in VirtualBox to preserve the state for quick restore.

**Final Result:**

You now have a **basic development environment** running in a virtual machine, isolated from your host system — ideal for testing, experimenting, or building safely.

Q11. **Write and upload your first source code file to GitHub.**

Step 1: Create a Simple Source Code File

Step 2: Create a New Git Repository

Step 3: Create a Repository on GitHub

Step 4: Connect Local Repo to GitHub

**Q12.Create a GitHub repository and document how to commit and push code changes**.

**Step 1: Create a GitHub Repository**

* + Go to <https://github.com/new>
  + Enter a repository name (e.g., my-first-project)
  + Choose **Public** or **Private**
  + **DO NOT** check “Initialize with README” (important for command line setup)
  + Click **Create repository**

**Step 2: Set Up Locally and Push Code**

#### A. Open Terminal/Command Prompt

bash

CopyEdit

mkdir my-first-project

cd my-first-project

#### B. Create a Simple File

bash

CopyEdit

echo "print('Hello GitHub')" > hello.py

#### C. Initialize Git & Commit Code

bash

CopyEdit

git init # Initialize Git repo

git add hello.py # Stage the file

git commit -m "Initial commit" # Commit with a message

#### D. Connect to GitHub Repo & Push

bash

CopyEdit

git remote add origin https://github.com/your-username/my-first-project.git

git branch -M main

git push -u origin main

**Q13 .Create a student account on Github and collaborate on a small project with a classmate.**

**Step 1: Create a Student GitHub Account**

* + Visit <https://github.com/join>
  + Fill in your **username**, **email**, and **password**, then create your account.
  + After verification, apply for **GitHub Student Developer Pack**:  
     <https://education.github.com/pack>
    - * + Use your **college/university email** (if available)
        + Upload proof of student status (ID card, fee receipt, etc.)

**Step 2: Collaborate on a Project**

#### One Person Creates the Repo

* + Go to [GitHub.com](https://github.com), click **+** → **New repository**
  + Name it something like mini-project
  + Choose **Private** (if only for classmates) or **Public**
  + Initialize with a README.md if you like

#### Add Collaborator

* + Go to your repo → **Settings** → **Collaborators**
  + Enter your classmate’s GitHub **username**
  + Click **Add collaborator** → they’ll receive an email invitation

**Step 3: Work on the Project Together**

Both collaborators can now:

bash

CopyEdit

# Clone the repo

git clone https://github.com/username/mini-project.git

# Add or change files

echo "print('Collaborating!')" > hello.py

# Commit and push changes

git add .

git commit -m "Added hello.py"

git push

**Q14. Create a list of software you use regularly and classify them into the following categories: system, application, and utility software.**

* + - * System software
* Operating System: Windows, macOS, Linux, Android, iOS
  + - * Application software
* Productivity: Microsoft Word/Excel/PowerPoint, Google Docs/Sheets/Slides, Notion, email clients
* Browsers: Google Chrome, Mozilla Firefox, Microsoft Edge, Safari
* Communication: Zoom, Slack, Microsoft Teams, WhatsApp, Skype
* Multimedia: VLC Media Player, Spotify, YouTube, Netflix
* Image/Video Editing: Adobe Photoshop, Canva, CapCut
  + - * Utility software
* System Maintenance/Optimization: Disk Cleanup, CCleaner, Norton Utilities, Windows Disk Defragmenter
* Security: Antivirus software (e.g., Norton, Avast, McAfee)
* File Management: File Explorer (Windows), Finder (macOS), WinRAR (file compression)

**Q15 .Follow a GIT tutorial to practice cloning, branching, and merging repositories.**

**Step 1: Create a Repository on GitHub**

* + Go to <https://github.com/new>
  + Name it: git-practice
  + Check **"Add a README file"**
  + Click **Create repository**

**Step 2: Clone the Repository**

bash

CopyEdit

git clone https://github.com/your-username/git-practice.git

cd git-practice

**Step 3: Create and Switch to a New Branch**

bash

CopyEdit

git checkout -b feature-branch

**You’re now on a new branch called feature-branch**.

**Step 4: Make a Change**

bash

CopyEdit

echo "This is a test feature file." > feature.txt

git add feature.txt

git commit -m "Add feature.txt"

**Step 5: Merge the Branch Back to Main**

bash

CopyEdit

git checkout main

git merge feature-branch

**If there are no conflicts, it merges cleanly**.

**Step 6: Push Changes to GitHub**

bash

CopyEdit

git push origin main

**Now check your GitHub repo to see the merged changes**

**Q16. Write a report on the various types of application software and how they improve**

**productivity.**

**1. Productivity Suites**

* **Word processors, spreadsheets, presentation tools (e.g., MS Word, Excel, PowerPoint): streamline document creation, data analysis, and visual communication** [**Wikipedia+14Fingent+14Web and Crafts+14**](https://www.fingent.com/blog/a-detailed-guide-to-types-of-software-applications/?utm_source=chatgpt.com)**.  
  Benefit: Saves time and improves output quality.**

**2. Enterprise & Business Apps**

* **ERP (Enterprise Resource Planning): integrates accounting, procurement, and more (e.g., Oracle, Microsoft Dynamics)** [**Fingent**](https://www.fingent.com/blog/a-detailed-guide-to-types-of-software-applications/?utm_source=chatgpt.com)**.**
* **CRM/BPM: manage sales pipelines, workflows, process automation .  
  Benefit: Automates business processes, reduces manual tasks, enhances scalability** [**toolpioneers+3Investopedia+3Wikipedia+3**](https://www.investopedia.com/how-ai-is-used-in-business-8611256?utm_source=chatgpt.com)**.**

**3. Project & Resource Management**

* **Project tools (Asana, Trello): plan, assign, track progress .**
* **Resource/time trackers (Harvest, ClickTime): logs hours, boosts accountability** [**Wikipedia**](https://en.wikipedia.org/wiki/Workflow_application?utm_source=chatgpt.com)**.  
  Benefit: Promotes on-time, on-budget delivery and virtual collaboration.**

**4. Database Software**

* **DBMS (Oracle, MySQL, PostgreSQL): organize and query large datasets** [**Quickbase+12Fingent+12Wikipedia+12**](https://www.fingent.com/blog/a-detailed-guide-to-types-of-software-applications/?utm_source=chatgpt.com)**.  
  Benefit: Centralizes data, enabling fast access and better reports.**

**5. Web & Multimedia Apps**

* **Browsers (Chrome, Firefox): access online tools, research .**
* **Photo/video editors (Photoshop, VLC, MX Player): create/consume media .  
  Benefit: Speeds up content creation and information consumption.**

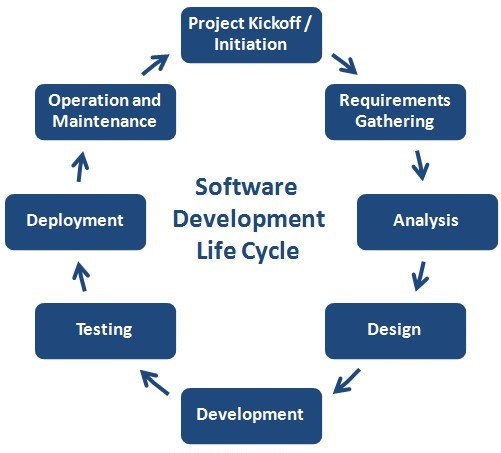
**6. Educational & Simulation Software**

* **Learning platforms (Google Classroom, TalentLMS) and simulation tools: support training and modeling** [**Fingent**](https://www.fingent.com/blog/a-detailed-guide-to-types-of-software-applications/?utm_source=chatgpt.com)**.  
  Benefit: Personalizes learning, enables virtual experimentation.**

**7. Custom & Developer Tools**

* **Bespoke apps: tailored workflows and integrations .**
* **Dev tools/IDEs (Visual Studio, Android Studio): streamline coding/testing .  
  Benefit: Increases domain efficiency, lowers adaptation friction.**

**Q17.Create a flowchart representing the Software Development Life Cycle (SDLC).**

****

**Q18. Write a requirement specification for a simple library management system**.

**Software Requirements Specification (SRS)**

**Project Title: Simple Library Management System**

**Version**: 1.0  
**Prepared by**: [Ansh shah]  
**Date**: [15/07/2025]

**1. Introduction**

**1.1 Purpose**

The purpose of this document is to outline the functional and non-functional requirements for a **Simple Library Management System (LMS)** that enables librarians to manage books, users, and borrowing transactions efficiently.

**1.2 Scope**

The system will be designed for a small library to:

* Manage books (add/update/delete/search)
* Register and manage members
* Track book checkouts and returns
* Apply basic fines for overdue returns

**2. Overall Description**

**2.1 Users of the System**

* **Admin (Librarian)** – full control over all features
* **Members (Students/Users)** – limited view, can search books and view borrowing history

**2.2 Assumptions and Constraints**

* System will run on desktop or web
* Single-location library
* Simple fine policy (fixed daily rate for overdue items)

**3. Functional Requirements**

**3.1 Book Management**

* Add new books with details (title, author, ISBN, category)
* Edit or delete existing books
* Search books by title, author, or ISBN
* Track the availability status of books (available/borrowed)

**3.2 Member Management**

* Register new members
* Edit or delete member details
* View a member’s borrowing history

**3.3 Borrowing & Returns**

* Allow admin to issue books to members
* Allow admin to mark returned books
* Prevent issuing more than a set number of books per member
* Calculate and display overdue fines

**3.4 Reports**

* Generate reports for:
  + List of borrowed books
  + Overdue items and fines
  + Member borrowing statistics

**4. Non-Functional Requirements**

**4.1 Usability**

* Simple, intuitive user interface
* Accessible to users with basic computer skills

**4.2 Reliability**

* System must maintain accurate records
* Data integrity must be preserved during updates

**4.3 Performance**

* Should handle up to 1000 books and 500 members efficiently

**4.4 Security**

* Login required for admin actions
* Data should be protected from unauthorized access

**5. System Features Summary**

| **Feature** | **Description** |
| --- | --- |
| Book Management | CRUD operations for book records |
| Member Management | CRUD operations for members |
| Issue/Return System | Manage book loans, returns, and availability |
| Fine Calculation | Calculate overdue charges |
| Reporting | Generate standard library reports |

**6. User Interface Requirements**

* **Admin Dashboard**: Links to manage books, users, and view reports
* **Book Search Page**: Search box with filters (title, author, category)
* **Borrow/Return Page**: Member ID input, book ID selection, due dates, and return actions

**7. Hardware/Software Requirements**

**7.1 Software**

* Frontend: HTML/CSS or GUI framework (e.g., React, Tkinter)
* Backend: PHP/Python/Node.js or Java
* Database: MySQL, SQLite, or PostgreSQL

**7.2 Hardware**

* Runs on a standard PC (4GB RAM, 100GB HDD, Windows/Linux)

**8. Future Enhancements (Optional)**

* Barcode scanning for issuing/returning
* SMS or email reminders for due dates
* Integration with online book catalog

**Q19. Perform a functional analysis for an online shopping system**.

**Functional Analysis: Online Shopping System**

**1. System Overview**

The online shopping system allows users to browse products, manage a cart, make purchases, and track orders. Admin users manage inventory, view sales data, and process customer orders.

**2. Stakeholders**

| **Stakeholder** | **Role and Interest** |
| --- | --- |
| **Customers** | Browse, shop, and pay for products |
| **Admins** | Manage products, orders, and users |
| **Delivery Team** | View orders assigned for delivery |
| **Payment Gateway** | Securely process customer payments |

**3. Primary Functional Requirements**

**3.1 User Management**

* Register new users
* User login/logout with authentication
* Edit user profile (email, password, address)
* Admin login with higher privileges

**3.2 Product Management**

* View product catalog (with images, price, stock)
* Search and filter by category, price, or keyword
* View detailed product information
* Admins can add/edit/delete products
* Update stock quantities (automatically after purchases)

**3.3 Shopping Cart and Wishlist**

* Add/remove products to cart
* Update quantity in cart
* Save products to wishlist
* View subtotal and total pricing in cart

**3.4 Checkout and Payment**

* Enter shipping and billing information
* Choose payment method (credit/debit card, PayPal, etc.)
* Integration with payment gateway (e.g., Stripe, PayPal)
* Generate and display confirmation receipt
* Apply discount codes or promo vouchers

**3.5 Order Management**

* Place an order after payment confirmation
* View order history and details
* Cancel or return orders (within limits)
* Admin view of all customer orders
* Update order status: Processing → Shipped → Delivered

**3.6 Inventory and Stock Management**

* Automatically decrease stock after purchases
* Admin alerts for low inventory
* Track popular products

**3.7 Customer Support and Feedback**

* Contact support via form or chatbot
* Submit product reviews and ratings
* Admin moderation of reviews

**4. Use Case Diagram (Text Format)**

pgsql

CopyEdit

[Customer]

├── Register/Login

├── Browse Products

├── Add to Cart / Wishlist

├── Checkout / Make Payment

├── View Order History

└── Leave Review

[Admin]

├── Manage Products

├── Manage Orders

├── Manage Users

└── View Reports

[System]

├── Send Confirmation Emails

├── Process Payments

└── Track Inventory Automatically

**5. Non-Functional Functional Interactions**

While the analysis focuses on *functions*, some system features supporting these include:

* **Security**: Encrypted login and payment handling
* **Performance**: Quick product search and real-time cart updates
* **Scalability**: Ability to handle growing numbers of users and products

**6. Key Functional Dependencies**

| **Function** | **Depends On** |
| --- | --- |
| Checkout | User login, cart, payment system |
| Order Processing | Inventory, payment confirmation |
| Product Listing | Admin product management |
| Customer Reviews | Purchase completion |

**7. Summary of Core Functional Areas**

| **Category** | **Key Functions** |
| --- | --- |
| User Operations | Registration, login, profile, order tracking |
| Product Management | Add/edit products, update stock, search/filter |
| Shopping & Checkout | Cart, wishlist, payment, address input, promo codes |
| Order Lifecycle | Create orders, process returns, track status |
| Admin Tools | Manage catalog, view reports, oversee transactions |

**Q20 . Design a basic system architecture for a food delivery app.**

**Food Delivery App: Basic System Architecture**

**1. Overview**

The food delivery app connects **Customers**, **Restaurants**, and **Delivery Drivers**. Users can browse menus, place orders, and track deliveries in real-time.

**2. Key Components**

**a) Client Applications**

* **Customer App** (iOS/Android/Web)
* **Restaurant App** (for order management)
* **Delivery Driver App** (for delivery tracking)

**b) Backend Server**

* Handles business logic, user management, order processing, notifications, and payment integration.

**c) Database**

* Stores user data, restaurant info, menus, orders, and delivery details.

**d) Third-party Services**

* Payment Gateway (Stripe, PayPal)
* Map and Geolocation APIs (Google Maps, Mapbox)
* Push Notifications (Firebase, OneSignal)

**3. Layered Architecture**

css

CopyEdit

[Client Apps] <---> [API Gateway / Backend Services] <---> [Database + External Services]

**4. Detailed Architecture Components**

| **Component** | **Description** |
| --- | --- |
| **User Interface** | Apps for customers, restaurants, and drivers with appropriate features and authentication |
| **API Gateway** | Entry point for all client requests; routes to backend services |
| **Authentication Service** | Manages user registration, login, and session management |
| **Order Service** | Handles order creation, status updates, and history |
| **Restaurant Service** | Manages menus, availability, and order notifications |
| **Delivery Service** | Assigns drivers, tracks location, and manages delivery statuses |
| **Payment Service** | Integrates with payment gateways for processing transactions |
| **Notification Service** | Sends real-time updates via push notifications or SMS |
| **Database** | Relational DB for structured data (PostgreSQL/MySQL) or NoSQL DB for flexibility |

**5. Data Flow Example: Order Placement**

1. **Customer** selects food and places an order via app.
2. Request sent to **API Gateway**, then forwarded to **Order Service**.
3. **Order Service** validates order, sends notification to **Restaurant Service**.
4. **Restaurant** confirms the order; **Delivery Service** assigns a driver.
5. **Payment Service** processes payment.
6. **Delivery Driver App** receives order details and tracks delivery.
7. Customer receives real-time updates via **Notification Service**.
8. After delivery, order status updated and stored in **Database**.

**6. System Diagram**

plaintext

CopyEdit

+-------------------+ +--------------------+ +-----------------+

| Customer App | | API Gateway | | Database |

+--------+----------+ +---------+----------+ +--------+--------+

| | |

+--------v----------+ +---------v----------+ +--------v--------+

| Restaurant App | <-------> | Backend Services | <------> | External Services|

+-------------------+ +--------------------+ +-----------------+

^

|

+--------v----------+

| Delivery Driver App|

+-------------------+

**7. Technology Suggestions**

| **Layer** | **Technologies** |
| --- | --- |
| Client Apps | React Native, Swift, Kotlin, ReactJS |
| Backend | Node.js, Django, Spring Boot |
| Database | PostgreSQL, MongoDB |
| Payment Gateway | Stripe, PayPal |
| Maps API | Google Maps API, Mapbox |
| Notifications | Firebase Cloud Messaging |

**Q21. Develop test cases for a simple calculator program.**

**Test Cases – Simple Calculator Program**

| **Test Case ID** | **Description** | **Input** | **Expected Output** | **Remarks** |
| --- | --- | --- | --- | --- |
| TC001 | Add two positive numbers | 5 + 3 | 8 | Basic addition |
| TC002 | Add positive and negative number | 10 + (-4) | 6 | Addition with negative |
| TC003 | Subtract two numbers | 9 – 4 | 5 | Basic subtraction |
| TC004 | Subtract larger from smaller | 3 – 5 | -2 | Negative result |
| TC005 | Multiply two positive numbers | 4 × 3 | 12 | Basic multiplication |
| TC006 | Multiply with zero | 6 × 0 | 0 | Zero multiplication |
| TC007 | Divide two numbers | 8 ÷ 2 | 4 | Basic division |
| TC008 | Divide by zero | 7 ÷ 0 | Error / Exception | Division by zero check |
| TC009 | Divide resulting in decimal | 7 ÷ 2 | 3.5 | Floating-point result |
| TC010 | Add two decimal numbers | 2.5 + 1.5 | 4.0 | Floating-point addition |
| TC011 | Input non-numeric values | "a" + 2 | Error / Invalid Input | Input validation |
| TC012 | Chained operations (if supported) | 2 + 3 × 4 | 14 | Operator precedence |
| TC013 | Negative result in division | -8 ÷ 2 | -4 | Negative division |
| TC014 | Multiply negative and positive | -3 × 6 | -18 | Sign handling |
| TC015 | Very large numbers | 1,000,000 × 1,000,000 | 1,000,000,000,000 | Performance test |

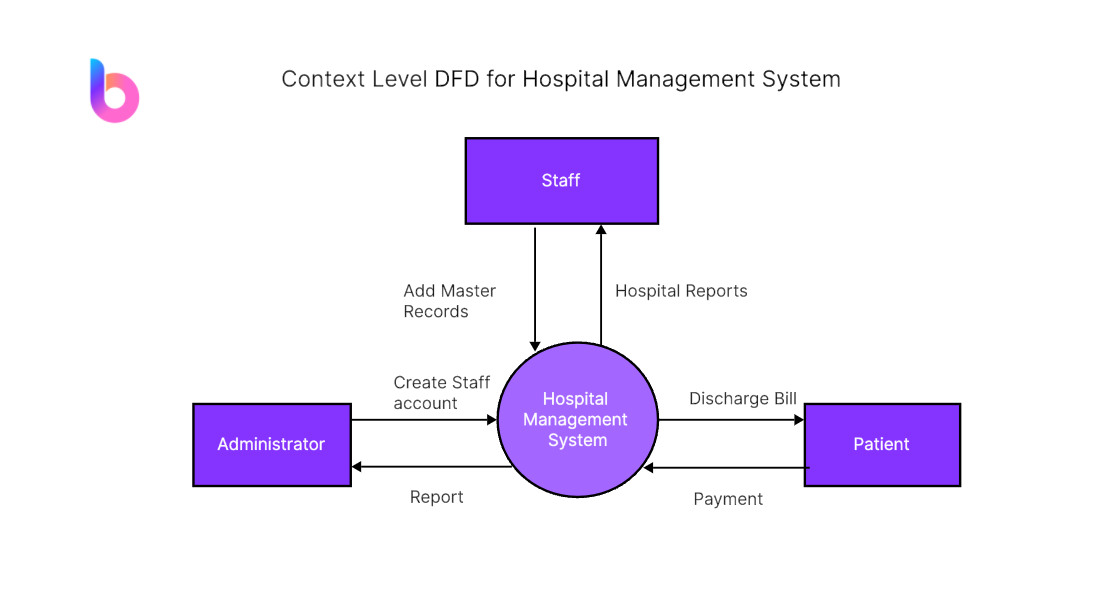
**Q 22. Document a real-world case where a software application required critical**

**maintenance.**

**3. Rogers Telecom Maintenance Failure (Canada, July 2022)**

* **What happened: A routine network maintenance update deleted routing filters, causing a nationwide outage across internet and cellular networks. 9‑1‑1 and debit payment services went down for ~15 hours.**
* **Fix: Emergency reinstatement of routing configurations, reciprocal roaming support from other providers.**
* **Lesson: Multi-layered redundancy and mutual aid agreements are essential for critical infrastructure resilience**

**Q23. Create a DFD for a hospital management system.**



**Q24. Build a simple desktop calculator application using a GUI library.**

**Code :-**

from tkinter import \*

expr = "" # Global expression string

def press(key):

global expr

expr += str(key)

display.set(expr)

def equal():

global expr

try:

result = str(eval(expr))

display.set(result)

expr = ""

except:

display.set("error")

expr = ""

def clear():

global expr

expr = ""

display.set("")

if \_\_name\_\_ == "\_\_main\_\_":

root = Tk()

root.configure(bg="light green")

root.title("Simple Calculator")

root.geometry("270x150")

display = StringVar()

entry = Entry(root, textvariable=display)

entry.grid(columnspan=4, ipadx=70)

# Number buttons

btn1 = Button(root, text='1', fg='black', bg='red', command=lambda: press(1), height=1, width=7)

btn1.grid(row=2, column=0)

btn2 = Button(root, text='2', fg='black', bg='red', command=lambda: press(2), height=1, width=7)

btn2.grid(row=2, column=1)

btn3 = Button(root, text='3', fg='black', bg='red', command=lambda: press(3), height=1, width=7)

btn3.grid(row=2, column=2)

btn4 = Button(root, text='4', fg='black', bg='red', command=lambda: press(4), height=1, width=7)

btn4.grid(row=3, column=0)

btn5 = Button(root, text='5', fg='black', bg='red', command=lambda: press(5), height=1, width=7)

btn5.grid(row=3, column=1)

btn6 = Button(root, text='6', fg='black', bg='red', command=lambda: press(6), height=1, width=7)

btn6.grid(row=3, column=2)

btn7 = Button(root, text='7', fg='black', bg='red', command=lambda: press(7), height=1, width=7)

btn7.grid(row=4, column=0)

btn8 = Button(root, text='8', fg='black', bg='red', command=lambda: press(8), height=1, width=7)

btn8.grid(row=4, column=1)

btn9 = Button(root, text='9', fg='black', bg='red', command=lambda: press(9), height=1, width=7)

btn9.grid(row=4, column=2)

btn0 = Button(root, text='0', fg='black', bg='red', command=lambda: press(0), height=1, width=7)

btn0.grid(row=5, column=0)

# Operator buttons

plus = Button(root, text='+', fg='black', bg='red', command=lambda: press('+'), height=1, width=7)

plus.grid(row=2, column=3)

minus = Button(root, text='-', fg='black', bg='red', command=lambda: press('-'), height=1, width=7)

minus.grid(row=3, column=3)

mult = Button(root, text='\*', fg='black', bg='red', command=lambda: press('\*'), height=1, width=7)

mult.grid(row=4, column=3)

div = Button(root, text='/', fg='black', bg='red', command=lambda: press('/'), height=1, width=7)

div.grid(row=5, column=3)

# Other buttons

eq = Button(root, text='=', fg='black', bg='red', command=equal, height=1, width=7)

eq.grid(row=5, column=2)

clr = Button(root, text='Clear', fg='black', bg='red', command=clear, height=1, width=7)

clr.grid(row=5, column=1)

dot = Button(root, text='.', fg='black', bg='red', command=lambda: press('.'), height=1, width=7)

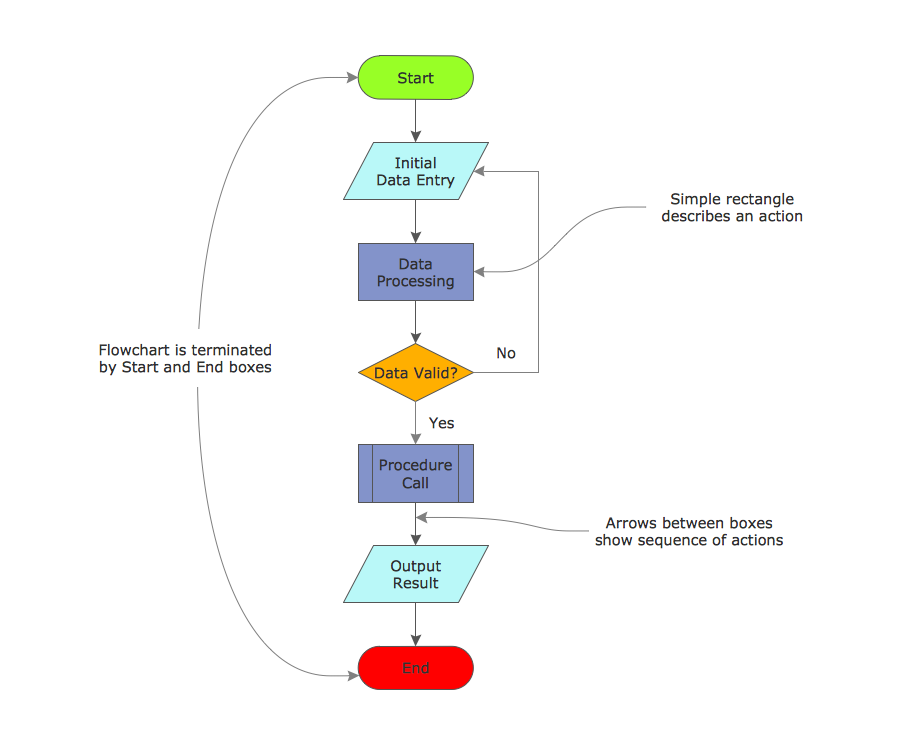
dot.grid(row=6, column=0)

root.mainloop()

**Explanation:**

* **press(key)**appends input to expr and updates the display, **equal()** evaluates expr and shows result or error, **clear()** resets everything.
* Sets up the main window with title, size, background and a **StringVar**display linked to an entry widget for showing input/output.
* Numeric buttons (0–9) arranged in a grid, each calls **press()**with its number.
* Operator buttons (+, -, \*, /) placed in the grid, each calls **press()** with the operator symbol.
* Additional buttons: = triggers calculation, Clear resets and **. adds** a decimal point.
* Uses**grid()** layout for neat positioning buttons have uniform size and color, runs the Tkinter main event loop.

**Q25. Draw a flowchart representing the logic of a basic online registration system.**

****