# Study Notes on FastAPI with Python

## Table of Contents

1. Introduction to FastAPI
2. Installation and Setup
3. Basic Concepts
   - ASGI and ASGI Servers
   - REST vs RPC
4. Creating a Basic FastAPI Application
5. Path Parameters and Query Parameters
6. Request Body and Models
7. Dependency Injection
8. Error Handling in FastAPI
9. Security Features
   - OAuth2
   - API Key
10. Middleware
11. Background Tasks
12. Testing FastAPI Applications
13. FastAPI and Databases
   - SQLAlchemy with FastAPI
14. Summary

## 1. Introduction to FastAPI
- **Definition**: FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.
- **Key Features**:
  - Fast: One of the fastest Python frameworks available (comparable to Node.js and Go).
  - Easy to use: Designed to be easy to learn and use.
  - Based on Python type hints: Enables auto-completion, validation, and interactive documentation.

## 2. Installation and Setup
- **Environment Preparation**:
  - Python Version: Ensure Python 3.6 or above is installed.
- **Installation**:
  ```bash
  pip install fastapi[all]
  ```
- **Running the application**:
  - Use an ASGI server such as `uvicorn`:
  ```bash
  uvicorn main:app --reload
  ```

## 3. Basic Concepts
### ASGI and ASGI Servers
- **ASGI (Asynchronous Server Gateway Interface)**:
  - A specification for Python web servers and applications to communicate with each other.
- **ASGI Servers**:
  - Servers such as Uvicorn or Daphne are capable of handling asynchronous connections.
### REST vs RPC

- **REST (Representational State Transfer)**:
  - Architectural style that uses standard HTTP methods.
- **RPC (Remote Procedure Call)**:
  - Calls functions through an HTTP protocol.

## 4. Creating a Basic FastAPI Application
### Basic Structure
```python
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
def read_root():
    return {"Hello": "World"}
```
### Explanation
- **FastAPI Instance**: `app = FastAPI()` creates an instance of the FastAPI application.
- **Endpoint Definition**: `@app.get("/")` defines a GET endpoint for the root path.

## 5. Path Parameters and Query Parameters
### Path Parameters
- Used to capture values from the URL.
```python
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item_id": item_id}
```
### Query Parameters
- Optional parameters that can alter the response.
```python
@app.get("/items/")
async def read_item(skip: int = 0, limit: int = 10):
    return {"skip": skip, "limit": limit}
```

## 6. Request Body and Models
### Request Body
- Send data in a structured format. Use Pydantic models for validation.
```python
from pydantic import BaseModel
class Item(BaseModel):
    id: int
    name: str
    price: float
@app.post("/items/")
async def create_item(item: Item):
    return item
```
### Pydantic Models
- **Definition**: Pydantic is a data validation and settings management using Python type annotations.
- **Features**:
  - Data validation.
  - Serialization of data.

## 7. Dependency Injection
- **Definition**: A technique to define dependencies that are reused across paths.
- **Example**:
```python
from fastapi import Depends
def get_query(param: str = None):
    return param
@app.get("/items/")
async def read_item(query_param: str = Depends(get_query)):
    return {"query": query_param}
```

## 8. Error Handling in FastAPI
- FastAPI provides built-in error handling for common exceptions.
- **Example**:
```python
from fastapi import HTTPException
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id not in data:
        raise HTTPException(status_code=404, detail="Item not found")
    return data[item_id]
```

## 9. Security Features
### OAuth2
- A protocol that allows secure authorization from third-party applications.
- FastAPI supports OAuth2 authentication out of the box.
### API Key
- Simple way to secure your API by requiring a key.
```python
from fastapi import Security, Depends
api_key: str = "your_api_key"
@app.get("/secure-data/")
async def read_secure_data(api_key: str = Depends(get_api_key)):
    return {"data": "Secure Data"}
```

## 10. Middleware
- **Definition**: Middleware is a function that is executed for every request before reaching the request path.
- **Example**:
```python
from fastapi import FastAPI
app = FastAPI()
@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    response = await call_next(request)
    response.headers["X-Process-Time"] = str(process_time)
    return response
```

## 11. Background Tasks
- Useful for non-blocking tasks.
- **Example**:
```python
from fastapi import BackgroundTasks
def write_log(message: str):
    with open("log.txt", mode="a") as log:
        log.write(message)
@app.post("/send-notification/")
async def send_notification(background_tasks: BackgroundTasks):
    background_tasks.add_task(write_log, "Notification sent")
    return {"message": "Notification sent"}
```

## 12. Testing FastAPI Applications
- FastAPI is designed to be easy to test.
- Testing using `pytest`:
```python
from fastapi.testclient import TestClient
client = TestClient(app)
def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"Hello": "World"}
```

## 13. FastAPI and Databases
### SQLAlchemy with FastAPI
- SQLAlchemy is a SQL toolkit for Python.
- Basic setup:
```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
SQLALCHEMY_DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

## 14. Summary
- FastAPI is a high-performance, easy-to-use framework for building Python APIs.
- Key components include path and query parameters, request bodies, dependency injection, error handling, and security features.
- The framework leverages Python's type hints for auto-generation of interactive documentation and validation.
- FastAPI integrates seamlessly with SQL databases using SQLAlchemy, making it suitable for production applications.
- Testing is straightforward with built-in testing capabilities, and performance benchmarks highlight FastAPI's efficiency.

End of Study Notes

# Operating System Notes
## Table of Contents
1. Introduction to Operating Systems
2. Types of Operating Systems
   - Batch Operating Systems
   - Time-Sharing Operating Systems
   - Distributed Operating Systems
   - Real-Time Operating Systems
3. Components of an Operating System
   - Kernel
   - User Interface
   - System Libraries
4. Process Management
   - Process Definition
   - Process States
   - Process Control Block (PCB)
5. Memory Management
   - Memory Hierarchy
   - Paging
   - Segmentation
6. File Management
   - File System Structure
   - File Operations
   - Access Methods
7. Device Management
   - I/O Devices
   - Device Drivers
8. Security and Protection
   - User Authentication
   - Access Control
   - Malware Protection
9. Conclusion & Summary

## 1. Introduction to Operating Systems
- **Definition:** An Operating System (OS) is software that acts as an intermediary
between computer hardware and users. It manages hardware resources and provides services
for application software.
- **Functions:**
  - Resource management
  - Process scheduling
  - Memory management
  - Device control
## 2. Types of Operating Systems
### Batch Operating Systems
- **Definition:** Executes batches of jobs without user interaction.
- **Characteristics:**
  - Efficient for high-throughput
  - Jobs are collected and processed sequentially
- **Example:** IBM 7094
### Time-Sharing Operating Systems
- **Definition:** Allows multiple users to interact with the computer simultaneously.
- **Characteristics:**
  - User programs share the processor time

  - Provides immediate response
- **Example:** UNIX
### Distributed Operating Systems
- **Definition:** Manages a collection of independent computers and makes them appear as a single coherent system.
- **Characteristics:**
  - Resource sharing
  - Transparency
- **Example:** Amoeba
### Real-Time Operating Systems
- **Definition:** Processes data as it comes in, typically without buffer delays.
- **Characteristics:**
  - Predictable response time
  - Used in embedded systems
- **Example:** VxWorks
## 3. Components of an Operating System
### Kernel
- **Definition:** The core part of the operating system that manages system resources and communication between hardware and software.
- **Functions:**
  - Process scheduling
  - Memory management
  - Device management
### User Interface
- **Types:**
  - Command-Line Interface (CLI)
  - Graphical User Interface (GUI)
- **Example:** Linux terminal (CLI), Windows desktop (GUI)
### System Libraries
- **Definition:** Collections of standard functions available to applications for performing various tasks.
- **Role:** Provide functions for handling system calls and managing processes.
## 4. Process Management
### Process Definition
- **Definition:** A program in execution. It contains the program code, current activity (represented by the value of the program counter), and process stack.
### Process States
- **States:**
  - New
  - Ready
  - Running
  - Waiting
  - Terminated
### Process Control Block (PCB)
- **Definition:** A data structure used by the operating system to store all the information about a process.
- **Contains:**
  - Process ID
  - Process state
  - Program counter
  - CPU registers
  - Memory management information
## 5. Memory Management

### Memory Hierarchy
- **Levels:**
  - Registers (fastest, smallest)
  - Cache
  - Main Memory (RAM)
  - Secondary Storage (HDD/SSD)
### Paging
- **Definition:** A memory management scheme that eliminates the need for contiguous allocation of physical memory and eliminates fragmentation.
- **Example:** A page table maps virtual addresses to physical addresses.
### Segmentation
- **Definition:** Memory management technique that divides the memory into segments based on logical divisions.
- **Example:** Code, stack, data segments.
## 6. File Management
### File System Structure
- **Definition:** A way to store and organize files on a disk.
- **Components:**
  - Directories
  - Files
  - Metadata
### File Operations
- **Common Operations:**
  - Create
  - Read
  - Write
  - Delete
### Access Methods
- **Types:**
  - Sequential Access
  - Random Access
## 7. Device Management
### I/O Devices
- **Definition:** Hardware components that allow users to interact with the computer system.
- **Types:**
  - Input Devices (e.g., keyboard, mouse)
  - Output Devices (e.g., monitor, printer)
### Device Drivers
- **Definition:** Software that allows the operating system to communicate with hardware devices.
- **Role:** Convert the OS's generic I/O instructions to device-specific codes.
## 8. Security and Protection
### User Authentication
- **Definition:** Process of verifying the identity of a user or process.
- **Methods:**
  - Passwords
  - Biometric verification
  - Multi-factor authentication
### Access Control
- **Definition:** Mechanisms that restrict access to resources based on permissions.
- **Types:**
  - Discretionary Access Control (DAC)

  - Mandatory Access Control (MAC)
### Malware Protection
- **Types:**
  - Antivirus Software
  - Firewalls
- **Goal:** Protect systems from malicious software and unauthorized access.
## 9. Conclusion & Summary
Operating Systems are crucial for managing computer hardware and software resources. Different types of OS cater to specific environments and user needs. Understanding the components of an OS, such as process management, memory management, file systems, and security measures, is vital for effective system utilization and development.

This content can be expanded upon for specific sections as needed, but provides a crisp, organized approach towards studying Operating Systems.

FastAPI Python Concepts Study Notes

Table of Contents

1. Introduction to FastAPI

2. Setting Up FastAPI

    • Installation

    • First FastAPI Application

3. Understanding Request and Response

    • Request Examples

    • Response Models

4. Path Parameters and Query Parameters

    • Path Parameters

    • Query Parameters

5. Request Body and Data Validation

    • Pydantic Models

    • Validating Input Data

6. Middleware and Dependency Injection

    • Understanding Middleware

    • Dependency Injection Concepts

7. Asynchronous Programming in FastAPI

    • Async and Await

    • Background Tasks

8. Security in FastAPI

    • Authentication

    • Authorization

9. FastAPI and Database Integration

    • Connecting to a Database

    • Using SQLAlchemy with FastAPI

10. Quick Testing and Documentation

• Testing FastAPI Applications

• Auto-generated API Documentation

11. Summary

---

1. Introduction to FastAPI

• **FastAPI**: A modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.

• **Key Features**:

• Fast: One of the fastest Python frameworks.

• Easy to use: Designed with developer productivity in mind.

• Automatic Documentation: Interactive docs via Swagger UI and ReDoc.

2. Setting Up FastAPI

### Installation

• Install FastAPI and an ASGI server (e.g., Uvicorn):

```
pip install fastapi uvicorn
```

### First FastAPI Application

• Code Example:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
return {"Hello": "World"}
```

Run using: uvicorn filename:app --reload

3. Understanding Request and Response

### Request Examples

- **HTTP Methods**: GET, POST, PUT, DELETE, etc.
- **FastAPI Routes**: Use decorators to define routes.

```
@app.get("/items/{item_id}")
def read_item(item_id: int):
return {"item_id": item_id}
```

### Response Models

- Use Pydantic models to define response schema:

```
from pydantic import BaseModel

class Item(BaseModel):
name: str
price: float
is_offer: bool = None

@app.post("/items/")
def create_item(item: Item):
return item
```

4. Path Parameters and Query Parameters

### Path Parameters

- Dynamic parameters in the URL.
- Example:

```
@app.get("/users/{user_id}")
def read_user(user_id: int):
return {"user_id": user_id}
```

### Query Parameters

- Optional parameters in the URL after the question mark.
- Example:

```
@app.get("/items/")

def read_items(skip: int = 0, limit: int = 10):

return {"skip": skip, "limit": limit}
```

5. Request Body and Data Validation

### Pydantic Models

- Define request body using Pydantic:

```
class User(BaseModel):

username: str

email: str

full_name: str = None
```

### Validating Input Data

- FastAPI uses Pydantic to validate data automatically. If validations fail, FastAPI returns a 422 Unprocessable Entity error.

6. Middleware and Dependency Injection

### Understanding Middleware

- Middleware processes requests before they reach the endpoint and responses before they are sent.
- Example:

```
from starlette.middleware.cors import CORSMiddleware


app.add_middleware(

CORSMiddleware,

allow_origins=["*"],
allow_methods=["*"],
allow_headers=["*"],
)
```

### Dependency Injection Concepts

- FastAPI allows defining dependencies for use in routes.
- Example:

```
def query_param(q: str = None):
return q

@app.get("/items/")
def read_items(q: str = Depends(query_param)):
return {"q": q}
```

7. Asynchronous Programming in FastAPI

### Async and Await

- Use asynchronous endpoints for better performance.
- Example:

```
import asyncio

@app.get("/items/")
async def read_items():
await asyncio.sleep(1)
return [{"item": "Item 1"}, {"item": "Item 2"}]
```

### Background Tasks

- Run long tasks in the background without blocking the main event loop:

```
from fastapi import BackgroundTasks

def write_log(message: str):
with open("log.txt", mode="a") as log:
```

```
log.write(f"{message}\n")

@app.post("/send-notification/")
async def send_notification(background_tasks: BackgroundTasks, email: str):
background_tasks.add_task(write_log, f"Email sent to {email}")
return {"message": "Notification sent"}
```

8. Security in FastAPI

### Authentication

- FastAPI supports various authentication techniques, including OAuth2.
- Example:

```
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")
```

### Authorization

- Define permissions and roles using dependencies.
- Example:

```
from fastapi import Security, HTTPException

@app.get("/users/me")
async def read_users_me(token: str = Security(oauth2_scheme)):
user = get_current_user(token) # Custom logic to validate token
return user
```

9. FastAPI and Database Integration

### Connecting to a Database

- Use ORM like SQLAlchemy for database interaction.

### Using SQLAlchemy with FastAPI

- Quick setup example:

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine = create_engine("sqlite:///./test.db")
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()

class User(Base):
__tablename__ = "users"
id = Column(Integer, primary_key=True, index=True)
name = Column(String)

Base.metadata.create_all(bind=engine)
```

10. Quick Testing and Documentation

### Testing FastAPI Applications

- Use standard Python testing frameworks (e.g., pytest) for testing.
- Example:

```
from fastapi.testclient import TestClient

client = TestClient(app)

def test_read_root():
response = client.get("/")
assert response.status_code == 200
assert response.json() == {"Hello": "World"}
```

### Auto-generated API Documentation

- FastAPI automatically generates interactive API documentation (Swagger UI and ReDoc):
- Swagger UI at: `http://127.0.0.1:8000/docs`

• ReDoc at: `http://127.0.0.1:8000/redoc`

11. Summary

• FastAPI is a powerful framework for building APIs quickly and efficiently, leveraging Python's type hints.

• Key concepts include request/response handling, data validation, asynchronous programming, security features, and integrations with databases.

• Testing and auto-generated documentation enhance development productivity.

### End of Study Notes

Keep revising each section to solidify your understanding, and practice building small applications to apply what you've learned. Happy coding with FastAPI!

# Study Notes on Football

## Introduction to Football
- **Definition**: Football is a team sport played between two teams of eleven players each, using a spherical ball. It is known as soccer in some parts of the world.
- **Objective**: The main objective of the game is to score more goals than the opposing team.
- **Popularity**: It is one of the most popular sports globally, with billions of fans and a multitude of professional leagues and tournaments.

## History of Football
### Origins
- **Ancient Games**:
  - Variants of football date back to ancient civilizations such as the Chinese (Cuju), Greeks (Episkyros), and Romans (Harpastum).
- **Modern Football**:
  - The modern rules of football were codified in England during the 19th century.
  - In 1863, the Football Association (FA) was formed.
### Evolution
- **Global Spread**:
  - The sport began to spread to other countries, leading to the establishment of international competitions.
- **FIFA Formation**:
  - The Fédération Internationale de Football Association (FIFA) was founded in 1904.

## Basic Rules of Football
### Playing Field
- **Dimensions**:
  - The field is rectangular, with a length of 100-110 meters and a width of 64-75 meters.
- **Goal**:
  - Each goal is 7.32 meters wide and 2.44 meters high.
### Game Duration
- **Match Length**:
  - A standard match is played in two halves of 45 minutes each, with a 15-minute halftime break.
- **Extra Time**:
  - If tied in knockout tournaments, matches may go into extra time (two 15-minute halves) followed by penalties if necessary.
### Players and Positions
- **Players**:
  - Each team consists of 11 players: 1 goalkeeper and 10 outfield players.
- **Positions**:
  - Goalkeeper, Defenders, Midfielders, Forwards
  - **Examples**:
    - Goalkeeper: The last line of defense.
    - Defender: Protects against opposing attacks.
    - Midfielder: Links defense and attack, controls the game's pace.
    - Forward: Primarily responsible for scoring goals.
### Offside Rule
- **Definition**: A player is in an offside position if they are nearer to the opponent's goal line than both the ball and the second last opponent when the ball is played to them, unless they are in their own half.

## Key Concepts in Football
### Skills and Techniques
- **Dribbling**:
  - Moving the ball with the feet while avoiding opponents.
- **Passing**:
  - Transferring the ball to teammates using various techniques (short pass, long pass, through ball).
- **Shooting**:
  - Striking the ball towards the goal to score.
- **Tackling**:
  - Attempting to take the ball away from an opposing player.
### Tactics and Strategies
- **Formation**:
  - The arrangement of players on the pitch (e.g., 4-4-2, 4-3-3).
- **Pressing**:
  - A strategy to regain possession quickly by applying pressure on the opposing players.
- **Possession Play**:
  - Keeping control of the ball to dictate the pace and direction of the game.
### Fouls and Violations
- **Types of Fouls**:
  - Direct Free Kick: Awarded for serious fouls (e.g., tripping, holding).
  - Indirect Free Kick: Awarded for less severe infractions (e.g., offside, dangerous play).
  - Penalty Kick: Awarded for fouls committed within the penalty area.

## Major Tournaments
### Domestic Leagues
- **Examples**:
  - English Premier League (EPL)
  - La Liga (Spain)
  - Serie A (Italy)
  - Bundesliga (Germany)
### International Competitions
- **FIFA World Cup**:
  - The premier international football tournament held every four years.
- **UEFA European Championship** (Euro):
  - A major international tournament for European national teams.
- **Copa America**:
  - A tournament for South American national teams.

## Famous Footballers
- **Pelé**:
  - Brazilian forward, known for his incredible skill and goal-scoring ability.
  - **Example**: Three-time FIFA World Cup winner (1958, 1962, 1970).
- **Diego Maradona**:
  - Argentine player, noted for his dribbling and playmaking skills.
  - **Example**: Famous for the "Hand of God" goal in the 1986 World Cup.
- **Lionel Messi**:
  - Argentine forward, regarded as one of the best players in football history.
  - **Example**: Multiple Ballon d'Or winner, renowned for his agility and vision.
- **Cristiano Ronaldo**:
  - Portuguese forward, known for his athleticism and goal-scoring prowess.
  - **Example**: Five-time Ballon d'Or winner, prolific scorer in multiple leagues.

## Summary
Football is a dynamic and exciting sport characterized by teamwork, skill, and strategy. Understanding the rules, tactics, and history enhances appreciation for the game, whether watching or playing. Key components include:
- Different positions on the pitch.
- Basic skills such as dribbling, passing, and shooting.
- Major tournaments like domestic leagues and international competitions.
- Legendary players who have shaped the sport.

These study notes provide a comprehensive overview of football, making them easy to revise and understand for students. Happy studying!

# Object-Oriented Programming (OOP) Study Notes
## Introduction to Object-Oriented Programming
- **Definition**: Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs.
- OOP focuses on using objects that combine data and functionality.
### Why OOP?
- **Modularity**: Code is organized into modules/objects.
- **Reusability**: Code can be reused across programs.
- **Scalability**: Easy to manage as systems grow.
- **Maintainability**: Changes can be made with minimal impact on existing code.

## Core Concepts of OOP
### 1. Classes and Objects
- **Class**: A blueprint for creating objects. It defines a data structure and methods applicable to that structure.
- **Object**: An instance of a class. It can hold data and perform functions.
Example:
```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
my_car = Car("Toyota", "Corolla")
```
### 2. Encapsulation
- **Definition**: The bundling of data and methods that operate on data within one unit, i.e., a class.
- **Protection**: It restricts direct access to some components, which can safeguard the object's integrity.
Example:
```python
class BankAccount:
    def __init__(self):
        self.__balance = 0  # Private variable
    def deposit(self, amount):
        self.__balance += amount
    def get_balance(self):
        return self.__balance
```
### 3. Inheritance
- **Definition**: The mechanism by which one class can inherit properties from another class.
- **Benefit**: Promotes code reusability and establishes a hierarchical relationship.
Example:
```python
class Animal:
    def speak(self):
        return "Animal speaks"
class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        return "Woof!"
my_dog = Dog()
print(my_dog.speak())  # Outputs: Woof!
```

~~~
### 4. Polymorphism
- **Definition**: The ability of different classes to be treated as instances of the same class through a common interface.
- **Types**:
  - **Method Overriding**: A subclass provides a specific implementation for a method that is already defined in its superclass.
  - **Method Overloading**: Multiple methods in the same scope with the same name but different parameters (not strictly supported in all languages).
**Example**:
```python
class Shape:
    def area(self):
        pass
class Rectangle(Shape):
    def area(self, length, width):
        return length * width
class Circle(Shape):
    def area(self, radius):
        return 3.14 * radius * radius
shapes = [Rectangle(), Circle()]
for shape in shapes:
    print(shape.area(5))  # The right area method is called based on the object type
~~~

## Advanced Concepts of OOP
### 1. Abstraction
- **Definition**: The concept of hiding the complex reality while exposing only the necessary parts.
- **Implementation**: Often achieved through abstract classes and interfaces.
Example:
```python
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Square(Shape):
    def __init__(self, side):
        self.side = side
    def area(self):
        return self.side ** 2
```
### 2. Composition
- **Definition**: A design principle where one class contains instances of other classes as its members, establishing a "has-a" relationship.
- **Benefit**: Promotes code reuse and offers flexibility through delegation.
Example:
```python
class Engine:
    def start(self):
        return "Engine starts"
class Car:
```

```
    def __init__(self):
        self.engine = Engine()  # Car has an Engine
my_car = Car()
print(my_car.engine.start())  # Outputs: Engine starts
```

## OOP Principles
### 1. DRY Principle (Don't Repeat Yourself)
- Aim to reduce repetition of code by using abstraction, inheritance, and composition.
### 2. SOLID Principles
- **Single Responsibility Principle**: A class should have one, and only one, reason to change.
- **Open/Closed Principle**: Software entities should be open for extension but closed for modification.
- **Liskov Substitution Principle**: Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
- **Interface Segregation Principle**: No client should be forced to depend on methods it does not use.
- **Dependency Inversion Principle**: High-level modules should not depend on low-level modules, but both should depend on abstractions.

## Practical Considerations
### Best Practices
- Use descriptive naming for classes and methods.
- Keep classes focused on a single responsibility (SRP).
- Favor composition over inheritance when applicable.
- Document code effectively to maintain clarity.
### Common Languages Supporting OOP
- **Python**: Dynamic and flexible for OOP.
- **Java**: Strongly typed with comprehensive OOP support.
- **C++**: Combines both procedural and object-oriented features.
- **C#**: Built on the Common Language Runtime (CLR) with rich OOP capabilities.

## Summary
- **OOP Overview**: OOP is about organizing software design using objects that encapsulate data and functionalities.
- **Key Concepts**: Classes and objects, encapsulation, inheritance, polymorphism, abstraction, and composition.
- **Principles**: Emphasis on principles such as DRY, SOLID, and best practices to create effective and maintainable code.
- **Applicability**: Widely used in many modern languages, improving code organization and reusability.

These consolidated notes reflect foundational concepts in object-oriented programming and serve as a study guide to understand OOP principles effectively.

Nachiket Abhay Vaidya

Test ID: 450011854000582 | 7066131709 | nachiket.vaidya24@vit.edu

Test Date: May 13, 2025

Logical Ability	Quantitative Ability (Advanced)	English Comprehension
54 /100	78 /100	68 /100

Logical Ability

54 / 100

Inductive Reasoning	Deductive Reasoning	Abductive Reasoning
55 / 100	55 / 100	53 / 100

Quantitative Ability (Advanced)

78 / 100

Basic Mathematics	Advanced Mathematics	Applied Mathematics
76 / 100	78 / 100	81 / 100

English Comprehension

68 / 100 | CEFR: **C1**

Grammar	Vocabulary	Comprehension
71 / 100	73 / 100	61 / 100

1 | Introduction

About the Report

This report provides a detailed analysis of the candidate's performance on different assessments. The tests for this job role were decided based on job analysis, O*Net taxonomy mapping and/or criterion validity studies. The candidate’s responses to these tests help construct a profile that reflects her/his likely performance level and achievement potential in the job role

This report has the following sections:

The **Summary** section provides an overall snapshot of the candidate’s performance. It includes a graphical representation of the test scores and the subsection scores.

The **Insights** section provides detailed feedback on the candidate’s performance in each of the tests. The descriptive feedback includes the competency definitions, the topics covered in the test, and a note on the level of the candidate’s performance.

The **Learning Resources** section provides online and offline resources to improve the candidate's knowledge, abilities, and skills in the different areas on which s/he was evaluated.

Score Interpretation

All the test scores are on a scale of 0-100. All the tests except personality and behavioural evaluation provide absolute scores. The personality and behavioural tests provide a norm-referenced score and hence, are percentile scores. Throughout the report, the colour codes used are as follows:

🟢 Scores between 67 and 100

🟡 Scores between 33 and 67

🔴 Scores between 0 and 33

2 | Insights

English Comprehension

68 / 100 | CEFR: **C1**

This test aims to measure your vocabulary, grammar and reading comprehension skills.

You have a good understanding of commonly used grammatical constructs. You are able to read and understand articles, reports and letters/mails related to your day-to-day work. The ability to read, understand and interpret business-related documents is essential in most jobs, especially the ones that involve research, technical reading and content writing.

Logical Ability

54 / 100

Inductive Reasoning

55 / 100

This competency aims to measure the your ability to synthesize information and derive conclusions.

You are able to work out rules based on specific information and solve general work problems using these rules. This skill is required in data-driven research jobs where one needs to formulate new rules based on variable trends.

Deductive Reasoning

55 / 100

This competency aims to measure the your ability to synthesize information and derive conclusions.

You are able to work out rules based on specific information and solve general work problems using these rules. This skill is required in data-driven research jobs where one needs to formulate new rules based on variable trends.

Abductive Reasoning

53 / 100

Quantitative Ability (Advanced)

78 / 100

This test aims to measure your ability to solve problems on basic arithmetic operations, probability, permutations and combinations, and other advanced concepts.

It is commendable that you are able to understand and solve complex arithmetic problems. You are able to solve basic problems of probability, logarithms, permutations, and combinations. This skill will help you in jobs where one needs to work with statistical data and make probabilistic predictions.

3 | Learning Resources

English Comprehension

Read opinions to improve your comprehension	🏷	🌐	📖
Improve your vocabulary of terms used in business	$	🌐	📖
Improve your knowledge of Business English	🏷	🌐	📖

Logical Ability

Test your application of inductive logic!	🏷	🌐	📖
Play Sudoku and develop your skills of deduction!	🏷	▷	📖
Learn about the art of deduction	$	🌐	📖

Quantitative Ability (Advanced)

Learn about the real life applications of logarithms	🏷	🌐	📖
Learn about the application of Bayes' Theorem in varied fields	🏷	W	📖
Learn about Fermet's last theorem	🏷	W	📖

Icon Index

🏷	Free Tutorial	$	Paid Tutorial	▶	Youtube Video	🌐	Web Source
▶	Wikipedia	📖	Text Tutorial	🎬	Video Tutorial	▷	Google Playstore

# Football Study Notes
## Table of Contents
1. Introduction to Football
2. History of Football
3. Types of Football
   - Association Football (Soccer)
   - American Football
   - Canadian Football
   - Australian Rules Football
4. The Rules of Football
   - Basic Rules
   - Positioning
   - Scoring
5. Key Skills in Football
   - Dribbling
   - Passing
   - Shooting
   - Defending
6. Popular Tournaments and Competitions
   - FIFA World Cup
   - UEFA Champions League
   - NFL Super Bowl
7. The Importance of Football
8. Summary

## 1. Introduction to Football
- **Definition**: Football, often referred to as soccer in the United States and Canada, is a team sport played between two teams of eleven players each. The primary aim is to score goals by moving a ball into the opposing team's net.
- **Objective**: The game seeks to outscore the opponent by advancing the ball and creating scoring opportunities.
## 2. History of Football
- **Origins**: Football has ancient origins, dating back to over 2,000 years ago with various forms played in China, Egypt, Greece, and Rome.
- **Modern Football**:
   - The modern game began in the mid-19th century.
   - The Football Association (FA) was formed in England in 1863, establishing standardized rules.
- **Global Popularity**:
   - The first FIFA World Cup took place in 1930 in Uruguay, cementing football's status as the world's most popular sport.
## 3. Types of Football
### Association Football (Soccer)
- Played on a rectangular field with a goal at each end.
- Most popular globally.
- Key tournaments include the FIFA World Cup and UEFA Champions League.
### American Football
- Predominantly played in the United States.
- Two teams of eleven players each, aiming to advance an oval ball into the opposing team’s end zone.
### Canadian Football
- Similar to American Football but with a larger field and more players.
- Canadian Football League (CFL) governs the sport in Canada.

### Australian Rules Football
- Combines elements of rugby and soccer.
- Played on an oval field with 18 players per team.
- Popular in Australia, governed by the Australian Football League (AFL).
## 4. The Rules of Football
### Basic Rules
- **Match Duration**: Typically, two 45-minute halves with a 15-minute halftime.
- **Players**: Each team has 11 players (including a goalkeeper).
- **Offside Rule**: A player cannot be offside when receiving the ball unless there are fewer than two opponents between him and the goal line.
### Positioning
- **Forward**: Primarily responsible for scoring goals.
- **Midfielder**: Links defense and offense, maintaining possession.
- **Defender**: Protects the goal from the opposing team.
- **Goalkeeper**: Last line of defense, protects the goal.
### Scoring
- A team scores when the entire ball crosses the goal line between the goalposts and under the crossbar.
## 5. Key Skills in Football
### Dribbling
- **Definition**: The act of controlling and advancing the ball using a series of small touches.
- **Example**: Players like Lionel Messi are known for their exceptional dribbling skills.
### Passing
- **Definition**: Moving the ball to a teammate.
- **Types**: Short pass, long pass, through ball.
- **Example**: Xavi Hernandez exemplified precise short passing.
### Shooting
- **Definition**: Attempting to score a goal by kicking the ball towards the goal.
- **Techniques**: Instep drive, volley, header.
- **Example**: Ronaldo's powerful shots are iconic.
### Defending
- **Definition**: Actions taken to prevent the opposing team from scoring.
- **Techniques**: Tackle, interception, marking.
- **Example**: Paolo Maldini is renowned for his defensive prowess.
## 6. Popular Tournaments and Competitions
### FIFA World Cup
- Held every four years.
- Involves teams from around the world competing for the title of world champion.
### UEFA Champions League
- Annual club competition in Europe.
- Features the top club teams competing in a knockout format.
### NFL Super Bowl
- Championship game of the National Football League (NFL).
- Highly viewed sporting event in the United States.
## 7. The Importance of Football
- **Social Significance**: Brings together communities, fostering a sense of belonging.
- **Health Benefits**: Encourages physical fitness and teamwork.
- **Economic Impact**: Generates revenue through ticket sales, television rights, and sponsorships.
## 8. Summary
Football is not just a sport; it is a global phenomenon that influences cultures and societies around the world. From its rich history and various types of football to skill

development and major tournaments, the importance of football transcends beyond the pitch. It fosters community, encourages healthy lifestyles, and unites people across different backgrounds through a shared love of the game.

FastAPI Python Concepts Study Notes

Table of Contents

1. Introduction to FastAPI
    - What is FastAPI?
    - Key Features
2. Installation
3. Basic Concepts
    - Request and Response
    - Path Parameters
    - Query Parameters
    - Request Body
4. Data Validation
    - Pydantic Models
    - Validation in FastAPI
5. Dependency Injection
    - What is Dependency Injection?
    - Creating Dependencies
6. Middleware
    - Definition of Middleware
    - How to Create Middleware
7. Routes and Routers
    - Creating Routes
    - Using Routers for Modular Design
8. Asynchronous Programming
    - Understanding Asynchronous in Python
    - Async Functions in FastAPI

9. Error Handling

• Exception Handling

• Custom Error Responses

10. Summary

---

1. Introduction to FastAPI

### What is FastAPI?

• FastAPI is a modern web framework for building APIs with Python 3.6+ based on standard Python type hints.

• It is designed to create high-performance APIs.

### Key Features

• **Fast**: High performance, comparable to NodeJS and Go.

• **Easy**: Designed for simplicity and ease of use.

• **Pythonic**: Leverages Python type hints for easy validation.

• **Automatic documentation**: Interactive API documentation (Swagger UI).

• **Asynchronous Support**: Built-in support for async and await.

---

2. Installation

To install FastAPI with the supporting ASGI server, use pip:

pip install fastapi[all]

• `fastapi`: The core library.

• `uvicorn`: ASGI server used to run FastAPI apps.

---

3. Basic Concepts

### Request and Response

- **Request**: The data sent by the client (e.g., a web browser).
- **Response**: The data returned by the server.

### Path Parameters

- Dynamic parameters in the route path.
- Example of defining a path parameter in FastAPI:

```
@app.get("/items/{item_id}")

async def read_item(item_id: int):

return {"item_id": item_id}
```

### Query Parameters

- Key-value pairs sent in the URL to further specify the request.

```
@app.get("/items/")

async def read_items(skip: int = 0, limit: int = 10):

return {"skip": skip, "limit": limit}
```

### Request Body

- Used to send complex data structures (e.g., JSON) in POST requests.

```
from pydantic import BaseModel

class Item(BaseModel):

name: str

description: str = None

price: float

tax: float = None

@app.post("/items/")

async def create_item(item: Item):
```

```
return item
```

---

4. Data Validation

### Pydantic Models

- FastAPI uses Pydantic for data validation and settings management.
- Helps ensure that data is of the expected format.

### Validation in FastAPI

- FastAPI validates the data automatically based on specified models.

Example:

```
from pydantic import BaseModel

class User(BaseModel):
name: str
age: int
email: str

@app.post("/users/")
async def create_user(user: User):
return user
```

---

5. Dependency Injection

### What is Dependency Injection?

- A way to manage dependencies and reduce tight coupling in your code.

### Creating Dependencies

- Use `Depends` to declare dependencies in FastAPI.

Example:

```
from fastapi import Depends

def get_query_param(q: str = None):
return q

@app.get("/items/")
async def read_items(q: str = Depends(get_query_param)):
return {"query": q}
```

---

6. Middleware

### Definition of Middleware

- Middleware is a function that runs before or after every request.

### How to Create Middleware

- Use `starlette.middleware`.

Example:

```
from starlette.middleware.cors import CORSMiddleware

app.add_middleware(
CORSMiddleware,
allow_origins=["*"],
allow_credentials=True,
allow_methods=["*"],
allow_headers=["*"],
)
```

---

7. Routes and Routers

### Creating Routes

- Define endpoints using decorators.

Example:

```
@app.get("/health/")
async def health_check():
return {"status": "healthy"}
```

### Using Routers for Modular Design

- Use `APIRouter` for organizing routes.

Example:

```
from fastapi import APIRouter

router = APIRouter()

@router.get("/items/{item_id}")
async def get_item(item_id: int):
return {"item_id": item_id}

app.include_router(router)
```

---

8. Asynchronous Programming

### Understanding Asynchronous in Python

- Allows writing concurrent code using the `async` and `await` syntax.

### Async Functions in FastAPI

- Define async functions to handle requests for better performance.

Example:

```
@app.get("/async-items/")

async def async_get_items():

await asyncio.sleep(1) # Simulates asynchronous processing

return ["item1", "item2"]
```

---

9. Error Handling

### Exception Handling

- FastAPI allows you to manage exceptions and provide custom responses.

Example:

```
from fastapi import HTTPException

@app.get("/items/{item_id}")

async def get_item(item_id: int):

if item_id > 10: # Example condition

raise HTTPException(status_code=404, detail="Item not found")

return {"item_id": item_id}
```

### Custom Error Responses

- You can customize error responses globally.

Example:

```
from fastapi.responses import JSONResponse

@app.exception_handler(HTTPException)
```

```
async def http_exception_handler(request, exc):
return JSONResponse(
status_code=exc.status_code,
content={"message": exc.detail},
)
```

---

10. Summary

- FastAPI is an efficient framework for building RESTful APIs.
- It integrates data validation using Pydantic and handles asynchronous requests gracefully.
- Dependency injection promotes clean code architecture.
- Middleware can be used for cross-cutting concerns, and error handling can be customized.
- FastAPI auto-generates interactive API documentation, facilitating ease of use.

FastAPI is a powerful tool that allows developers to create high-performance APIs with minimal effort while maintaining code clarity and simplicity.

Use these notes to understand the concepts and functionalities of FastAPI, and revise them for better retention of knowledge!

# Football Study Notes
## 1. Introduction to Football
### 1.1 Definition
Football, also known as soccer in some countries, is a team sport played between two teams of eleven players using a spherical ball.
### 1.2 Importance
- **Global Popularity**: Football is the world’s most popular sport, played and watched by millions.
- **Cultural Impact**: It brings people together, transcending cultural and geographic barriers.
- **Economic Influence**: Football generates significant revenue through leagues, sponsorships, and major tournaments.
## 2. History of Football
### 2.1 Origin
- **Ancient Roots**: Variants of football date back to ancient civilizations (e.g., China, Greece, and Rome).
- **Modern Football**: The rules of modern football were codified in 1863 in England with the formation of the Football Association.
### 2.2 Evolution
- **Formation of FIFA**: Founded in 1904, responsible for international competitions.
- **World Cup**: The first World Cup was held in Uruguay in 1930.
## 3. The Game Structure
### 3.1 Field of Play
- **Dimensions**: Rectangular field between 90-120 meters long and 45-90 meters wide.
- **Key Areas**:
  - **Goal Area**: 6-yard box in front of the goal.
  - **Penalty Area**: 18-yard box where fouls can result in penalty kicks.
### 3.2 Duration of the Game
- **Match Length**: Two halves of 45 minutes each, with a half-time break of 15 minutes.
- **Injury Time**: Additional time added at the end of each half to compensate for stoppages.
## 4. Equipment
### 4.1 Football
- **Size**: Official match ball is size 5.
- **Material**: Typically made from synthetic leather.
### 4.2 Player’s Gear
- **Jersey and Shorts**: Each team wears a distinctive kit.
- **Shin Guards**: Worn to protect the legs.
- **Footwear**: Cleats designed for traction on the field.
## 5. Rules of the Game (Laws of the Game)
### 5.1 Basic Rules
- **Offside Rule**: A player is offside if they are nearer to the opponent's goal line
than both the ball and the second last opponent at the moment the ball is played.
- **Fouls and Free Kicks**: Direct and indirect free kicks awarded for fouls.
### 5.2 Scoring
- **Goal**: A goal is scored when the entire ball crosses the goal line between the goalposts.
- **Points System**: Teams earn 3 points for a win, 1 for a draw, and 0 for a loss.
## 6. Key Positions and Roles
### 6.1 Goalkeeper (GK)
- **Role**: Protects the goal and the only player allowed to use hands within the penalty area.
### 6.2 Defenders

- **Roles**:
  - **Center-back**: Positioned centrally; main role is to block opposing forwards.
  - **Full-back**: Positioned on the flanks; supports both defense and crosses to forwards.
### 6.3 Midfielders
- **Roles**:
  - **Central Midfielder**: Links defense and attack.
  - **Attacking Midfielder**: Positioned closer to forwards to create scoring opportunities.
### 6.4 Forwards
- **Roles**:
  - **Striker**: Main goal scorer, often the most advanced player on the field.
  - **Winger**: Positioned on the flanks; provides width and crosses.
## 7. Tactics and Strategies
### 7.1 Formations
- **4-4-2**: Four defenders, four midfielders, and two forwards.
- **4-3-3**: Four defenders, three midfielders, and three forwards.
### 7.2 Game Strategies
- **Pressing**: A defensive tactic where players attempt to win back possession high up the pitch.
- **Counter-attack**: A strategy focusing on quick transitions from defense to attack.
## 8. Major Competitions
### 8.1 International Competitions
- **FIFA World Cup**: Held every four years, the premier competition for national teams.
- **UEFA European Championship**: Held every four years for European national teams.
### 8.2 Club Competitions
- **UEFA Champions League**: Annual tournament involving top-division European clubs.
- **Domestic Leagues**: Each country has its league system (e.g., Premier League, La Liga).
## 9. Famous Players and Legends
### 9.1 Historical Icons
- **Pelé**: Brazilian forward, three-time World Cup winner.
- **Diego Maradona**: Argentine forward, known for his "Hand of God" goal and incredible dribbling ability.
### 9.2 Current Stars
- **Cristiano Ronaldo**: Known for his goal-scoring ability and athleticism.
- **Lionel Messi**: Renowned for his dribbling and playmaking skills.
## 10. Summary
Football is a dynamic and globally significant sport characterized by its rich history, strategic complexity, and the excitement of competition. Understanding the rules, player roles, tactics, and major players contributes to a deeper appreciation of the game. Engaging in football, whether as a player or a fan, fosters community and cultural exchange, making it a vital part of global culture.
### Key Takeaways
- Football has ancient roots and has evolved significantly to become a structured and highly organized sport.
- Understanding the laws of the game is essential for players and fans alike.
- Player roles and strategies play a crucial part in the success of a team.
- Major competitions provide a platform for teams and players to showcase their skills on an international level.

By grasping these concepts, you'll enhance your knowledge and enjoyment of football as both a sport and a cultural phenomenon.

# C++ Study Notes
## Introduction to C++
- **Definition**: C++ is a high-level programming language developed by Bjarne Stroustrup in 1979, known for its performance and flexibility, and is widely used for system/software development and game programming.
- **Paradyms**: It supports procedural, object-oriented, and generic programming paradigms.
### Key Features of C++
- Object-oriented programming (OOP)
- Low-level manipulation
- Rich library support
- Portability
- Performance
- Memory management

## Setting Up C++
### Installing a Compiler
- **Compilers**: Easily work with a popular compiler like GCC or Visual Studio.
- **IDE**: Integrated Development Environments (IDE) like Code::Blocks, Eclipse, or Visual Studio to write and manage code.
### First Program
```cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```
- **Explanation**:
  - `#include <iostream>`: Preprocessor directive that includes the input-output stream library.
  - `using namespace std;`: Enables the use of standard library names without the `std::` prefix.
  - `cout`: Standard output stream.
  - `endl`: Ends the line and flushes the output buffer.

## C++ Basics
### Variables and Data Types
- **Definition**: Variables are containers for storing data values.
- **Data Types**:
  - `int`: Integer Type
  - `float`: Floating Point Type
  - `double`: Double Precision Float
  - `char`: Character Type
  - `bool`: Boolean Type
### Operators
- **Arithmetic Operators**: +, -, *, /, %
- **Relational Operators**: ==, !=, <, >, <=, >=
- **Logical Operators**: && (AND), || (OR), ! (NOT)
- **Bitwise Operators**: &, |, ^, ~, <<, >>
### Control Structures
#### Conditional Statements

- **if Statement**:
```cpp
if (condition) {
   // code to execute if condition is true
}
```
- **switch Case**:
```cpp
switch (variable) {
   case value1:
      // code
      break;
   case value2:
      // code
      break;
   default:
      // default case
}
```
#### Loops
- **for Loop**:
```cpp
for (initialization; condition; increment) {
   // code
}
```
- **while Loop**:
```cpp
while (condition) {
   // code
}
```
- **do-while Loop**:
```cpp
do {
   // code
} while (condition);
```

## Functions in C++
### Definition
- **Function**: A block of code designed to perform a specific task.
- **Syntax**:
```cpp
returnType functionName(parameters) {
   // function body
}
```
### Passing Arguments
- **By Value**: Copies the actual value of an argument into the function.
- **By Reference**: Uses references to access and modify the original variable.
### Example
```cpp
```

```cpp
int add(int a, int b) {
    return a + b;
}
```

## Object-Oriented Programming (OOP) Concepts
### Classes and Objects
- **Class**: A blueprint for creating objects (contains data and methods).
- **Object**: An instance of a class.
#### Example
```cpp
class Car {
public:
    string brand;
    void drive() {
        cout << "Driving a car" << endl;
    }
};
Car myCar; // Object creation
myCar.brand = "Toyota";
myCar.drive();
```
### Encapsulation
- **Definition**: Bundling the data and methods that operate on the data within one unit (class).
### Inheritance
- **Definition**: Mechanism where a new class derives properties and behavior from an existing class.
#### Example
```cpp
class Vehicle {
public:
    void honk() {
        cout << "Honk!" << endl;
    }
};
class Bike : public Vehicle { }; // Bike inherits Vehicle
Bike b;
b.honk(); // Outputs: Honk!
```
### Polymorphism
- **Definition**: Ability to process objects differently based on their data type or class.
#### Example
```cpp
class Animal {
public:
    virtual void sound() {
        cout << "Animal makes sound" << endl;
    }
};
class Dog : public Animal {
public:
    void sound() override {
        cout << "Woof!" << endl;
    }
};
Animal* a = new Dog();
a->sound(); // Outputs: Woof!
```

## Advanced Concepts
### Templates
- **Definition**: Enables writing generic programs.
#### Example
```cpp
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```
### Exception Handling
- **Definition**: Mechanism to handle runtime errors.
#### Example
```cpp
try {
    // code that may throw an exception
} catch (exception &e) {
    // handle exception
}
```

## Conclusion & Summary
- **C++ Overview**: A versatile language that supports multiple programming paradigms.
- **Core Components**: Variables, data types, functions, control structures, and OOP principles.
- **Advanced Features**: Templates and exception handling allow for robust and reusable code.

Remember: Practice coding, understand the logic behind each construct, and gradually explore more complex topics as you become more comfortable with the language!

Nachiket Abhay Vaidya

Test ID: 450011854000582 | 7066131709 | nachiket.vaidya24@vit.edu

Test Date: May 13, 2025

Logical Ability	Quantitative Ability (Advanced)	English Comprehension
54 /100	78 /100	68 /100

Logical Ability

54 / 100

Inductive Reasoning	Deductive Reasoning	Abductive Reasoning
55 / 100	55 / 100	53 / 100

Quantitative Ability (Advanced)

78 / 100

Basic Mathematics	Advanced Mathematics	Applied Mathematics
76 / 100	78 / 100	81 / 100

English Comprehension

68 / 100 | CEFR: **C1**

Grammar	Vocabulary	Comprehension
71 / 100	73 / 100	61 / 100

1 | Introduction

About the Report

This report provides a detailed analysis of the candidate's performance on different assessments. The tests for this job role were decided based on job analysis, O*Net taxonomy mapping and/or criterion validity studies. The candidate’s responses to these tests help construct a profile that reflects her/his likely performance level and achievement potential in the job role

This report has the following sections:

The **Summary** section provides an overall snapshot of the candidate’s performance. It includes a graphical representation of the test scores and the subsection scores.

The **Insights** section provides detailed feedback on the candidate’s performance in each of the tests. The descriptive feedback includes the competency definitions, the topics covered in the test, and a note on the level of the candidate’s performance.

The **Learning Resources** section provides online and offline resources to improve the candidate's knowledge, abilities, and skills in the different areas on which s/he was evaluated.

Score Interpretation

All the test scores are on a scale of 0-100. All the tests except personality and behavioural evaluation provide absolute scores. The personality and behavioural tests provide a norm-referenced score and hence, are percentile scores. Throughout the report, the colour codes used are as follows:

- 🟢 Scores between 67 and 100
- 🟡 Scores between 33 and 67
- 🔴 Scores between 0 and 33

2 | Insights

English Comprehension

68 / 100 | CEFR: **C1**

This test aims to measure your vocabulary, grammar and reading comprehension skills.

You have a good understanding of commonly used grammatical constructs. You are able to read and understand articles, reports and letters/mails related to your day-to-day work. The ability to read, understand and interpret business-related documents is essential in most jobs, especially the ones that involve research, technical reading and content writing.

Logical Ability

54 / 100

Inductive Reasoning 55 / 100

This competency aims to measure the your ability to synthesize information and derive conclusions.

You are able to work out rules based on specific information and solve general work problems using these rules. This skill is required in data-driven research jobs where one needs to formulate new rules based on variable trends.

Deductive Reasoning 55 / 100

This competency aims to measure the your ability to synthesize information and derive conclusions.

You are able to work out rules based on specific information and solve general work problems using these rules. This skill is required in data-driven research jobs where one needs to formulate new rules based on variable trends.

Abductive Reasoning 53 / 100

Quantitative Ability (Advanced)

78 / 100

This test aims to measure your ability to solve problems on basic arithmetic operations, probability, permutations and combinations, and other advanced concepts.

It is commendable that you are able to understand and solve complex arithmetic problems. You are able to solve basic problems of probability, logarithms, permutations, and combinations. This skill will help you in jobs where one needs to work with statistical data and make probabilistic predictions.

3 | Learning Resources

English Comprehension

Read opinions to improve your comprehension	Free Tutorial	Web Source	Text Tutorial
Improve your vocabulary of terms used in business	Paid Tutorial	Web Source	Text Tutorial
Improve your knowledge of Business English	Free Tutorial	Web Source	Text Tutorial

Logical Ability

Test your application of inductive logic!	Free Tutorial	Web Source	Text Tutorial
Play Sudoku and develop your skills of deduction!	Free Tutorial	Google Playstore	Text Tutorial
Learn about the art of deduction	Paid Tutorial	Web Source	Text Tutorial

Quantitative Ability (Advanced)

Learn about the real life applications of logarithms	Free Tutorial	Web Source	Text Tutorial
Learn about the application of Bayes' Theorem in varied fields	Free Tutorial	W	Text Tutorial
Learn about Fermet's last theorem	Free Tutorial	W	Text Tutorial

Icon Index

Free Tutorial	Paid Tutorial	Youtube Video	Web Source
Wikipedia	Text Tutorial	Video Tutorial	Google Playstore

Tutorial No 5

Name : Sarthak Shrikant Bagul

Branch : ENTC

Div :- A

PRN :- 12410555

Roll No :- 25

Subject :- Computer Graphics (MDM)

Title / Problem Statement:

2D Transformations in Computer Graphics – Translation, Scaling, Rotation, Reflection, and Shearing

Understand and apply 2D transformations on graphical objects such as points, lines, and polygons. Explore different types of transformations and their effects on the shape, position, and orientation of objects in 2D space.

Objectives: By the end of this tutorial, students should be able to:

- Define and explain the different types of 2D transformations.
- Apply transformations such as **translation, scaling, rotation, reflection, and shearing** to 2D objects.
- Represent transformations mathematically using **homogeneous matrices**.
- Understand and analyze the effects of transformations on 2D objects in computer graphics.

Prerequisites:

- Knowledge of coordinate geometry (points, lines, polygons).
- Basic linear algebra (matrices, matrix multiplication).
- Basic trigonometry (sine, cosine for rotation).
- Understanding of computer graphics basics (pixels, raster displays).

Theory Overview:

1. **2D Transformations:**
   Transformations are operations that alter the position, size, orientation, or shape of graphical objects in 2D space.

2. **Types of Transformations:**
   (a) **Translation:** Moves an object from one location to another without changing its shape or orientation.

- Formula:

$$x' = x + tx, \quad y' = y + ty$$

(b) **Scaling:** Changes the size of an object relative to a fixed point (usually origin).

- Formula:

$$x' = Sx \cdot x, \quad y' = Sy \cdot y$$

- Scaling around a pivot requires moving the object to origin, scaling, and moving back.

(c) **Rotation:** Rotates an object around a fixed point (usually origin) by angle θ.

- Formula:

$$x' = x\cos\theta - y\sin\theta, \quad y' = x\sin\theta + y\cos\theta$$

(d) **Reflection:** Produces a mirror image of an object about a line (x-axis, y-axis, y=x, etc.).

(e) **Shearing:** Slants the shape of an object along x or y axis.

- X-shear: x' = x + $sh_x \cdot$ y
- Y-shear: y' = y + $sh_y \cdot$ x

3. **Homogeneous Coordinates:**

- Transformations can be represented by 3×3 matrices to combine multiple operations conveniently:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- This allows **composite transformations** like scaling + rotation + translation in a single operation.

4. **Order of Transformations:**

- The sequence of transformations matters:
    - **Rotation then translation ≠ Translation then rotation**
- Composite transforms often use a **pivot point** to control rotation or scaling.

Algorithm / Steps:

A. Translation:

1. For each point (x, y), calculate:

$$x' = x + tx, y' = y + ty$$

2. Plot the new points.

B. Scaling:

1. Select scale factors Sx,Sy.
2. For each point (x, y), calculate:

$$x' = x \cdot Sx , \ y' = y \cdot Sy$$

3. Plot scaled points.

C. Rotation:

1. Select rotation angle θ.
2. For each point (x, y), calculate:

$$x' = x\cos\theta - y\sin\theta, y' = x\sin\theta + y\cos\theta$$

3. Plot rotated points.

D. Reflection:

1. Choose the axis of reflection (x-axis, y-axis, y=x).
2. Apply the corresponding reflection formula.

E. Shearing:

1. Select shear factors sh_x or sh_y.
2. For each point (x, y), apply:

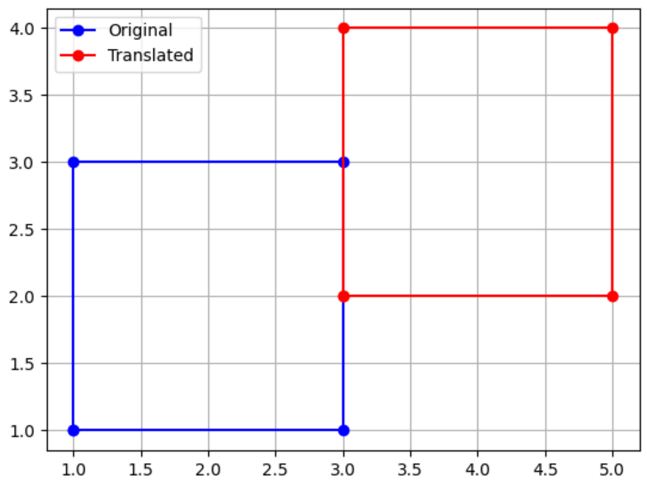
$$x' = x + sh_x \cdot y, \quad y' = y + sh_y \cdot x$$

3. Plot sheared points.

Code Implementation (Python + Matplotlib)

(i) Translation:

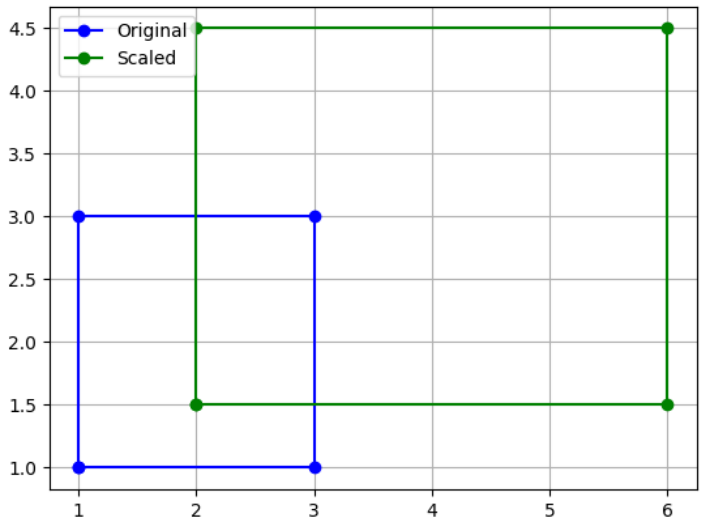
```
import matplotlib.pyplot as plt

def translate(points, tx, ty):

return [(x + tx, y + ty) for x, y in points]

# Original square

square = [(1,1),(1,3),(3,3),(3,1),(1,1)]

translated_square = translate(square, 2, 1)

x_orig, y_orig = zip(*square)

x_trans, y_trans = zip(*translated_square)

plt.plot(x_orig, y_orig, 'b-o', label='Original')

plt.plot(x_trans, y_trans, 'r-o', label='Translated')

plt.legend()

plt.grid(True)

plt.show()
```



(ii) Scaling:

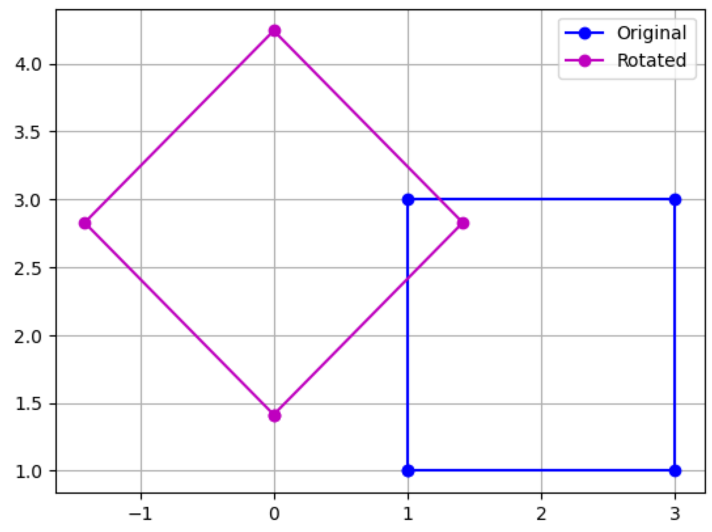
```
def scale(points, sx, sy):

    return [(x * sx, y * sy) for x, y in points]
```

```
scaled_square = scale(square, 2, 1.5)

x_scaled, y_scaled = zip(*scaled_square)

plt.plot(x_orig, y_orig, 'b-o', label='Original')

plt.plot(x_scaled, y_scaled, 'g-o', label='Scaled')

plt.legend()

plt.grid(True)

plt.show()
```



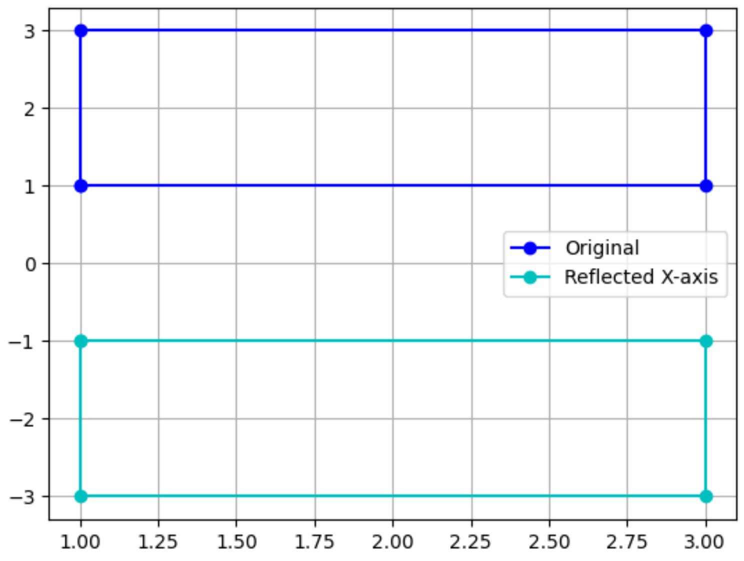
(iii) Rotation:

```
import math

def rotate(points, theta):

   rad = math.radians(theta)

   return [(x*math.cos(rad) - y*math.sin(rad), x*math.sin(rad) + y*math.cos(rad)) for x,y in points]

rotated_square = rotate(square, 45)

x_rot, y_rot = zip(*rotated_square)

plt.plot(x_orig, y_orig, 'b-o', label='Original')

plt.plot(x_rot, y_rot, 'm-o', label='Rotated')

plt.legend()

plt.grid(True)

plt.show()
```



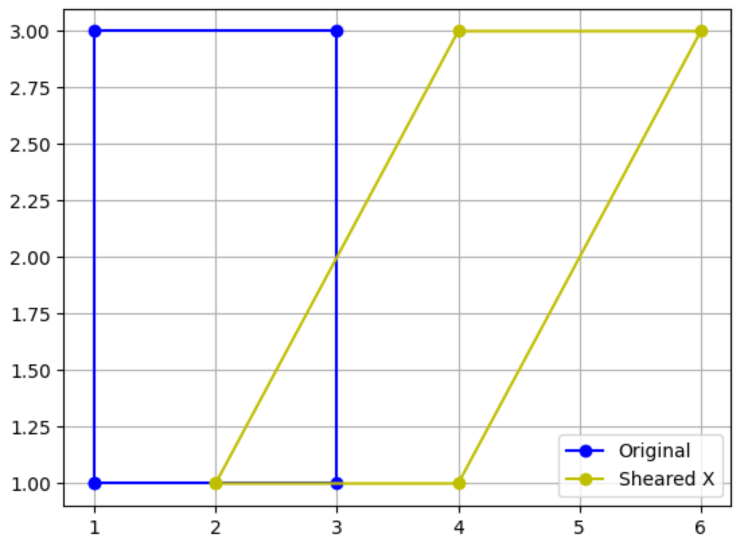
(iv) Reflection (about x-axis):

```
def reflect_x(points):
    return [(x, -y) for x, y in points]
reflected_square = reflect_x(square)
x_ref, y_ref = zip(*reflected_square)
plt.plot(x_orig, y_orig, 'b-o', label='Original')
plt.plot(x_ref, y_ref, 'c-o', label='Reflected X-axis')
plt.legend()
plt.grid(True)
plt.show()
```



(v) Shearing (x-direction):

```
def shear_x(points, shx):
  return [(x + shx*y, y) for x, y in points]
sheared_square = shear_x(square, 1)
x_shear, y_shear = zip(*sheared_square)
plt.plot(x_orig, y_orig, 'b-o', label='Original')
plt.plot(x_shear, y_shear, 'y-o', label='Sheared X')
plt.legend()
plt.grid(True)
plt.show()
```



Result / Observation / Conclusion:

- **Translation** moves the object without changing shape.
- **Scaling** changes object size; non-uniform scaling distorts shape.
- **Rotation** changes orientation while keeping shape and size intact.
- **Reflection** creates mirror images along specified axes.
- **Shearing** slants the object, altering angles but not parallelism.
- **Homogeneous matrices** allow combining multiple transformations efficiently.
- These transformations demonstrate how mathematical operations directly manipulate pixel positions on raster displays.

# Study Notes on Cristiano Ronaldo
## Introduction
Cristiano Ronaldo is one of the most famous and successful football (soccer) players in the world. Known for his incredible skills, work ethic, and dedication, he has become a global sports icon.

## 1. Early Life
### 1.1 Birth and Family
- **Full Name:** Cristiano Ronaldo dos Santos Aveiro
- **Date of Birth:** February 5, 1985
- **Place of Birth:** Funchal, Madeira, Portugal
- **Family Background:**
  - Youngest of four children
  - Father: José Denis Aveiro, a municipal gardener
  - Mother: Maria Dolores dos Santos Aveiro, a cook
### 1.2 Childhood Interests
- Developed an interest in football at a young age.
- Joined local club Andorinha at age 8.
- Moved to Nacional, a larger club in Madeira, before joining Sporting Lisbon.

## 2. Professional Career
### 2.1 Sporting Lisbon
- **Years Active:** 1997-2003
- Joined the youth academy at age 12.
- Made his professional debut in the Primeira Liga at 17.
### 2.2 Manchester United (First Stint)
- **Years Active:** 2003-2009
- **Transfer Fee:** £12.24 million
- Key Achievements:
  - Premier League Champion (3 times)
  - UEFA Champions League Winner (2008)
  - FIFA World Player of the Year (2008)
### 2.3 Real Madrid
- **Years Active:** 2009-2018
- **Transfer Fee:** €94 million (world record at that time)
- Key Achievements:
  - La Liga Champion (2 times)
  - UEFA Champions League Winner (4 times)
  - UEFA Best Player in Europe (3 times)
### 2.4 Juventus
- **Years Active:** 2018-2021
- Extended his legacy in Serie A.
- Key Achievements:
  - Serie A Champion (2 times)
  - Coppa Italia Winner
### 2.5 Manchester United (Second Stint)
- **Years Active:** 2021-2022
- Returned to United and faced contrasting fortunes.
### 2.6 Al Nassr
- **Years Active:** 2023-Present
- Joined Saudi club and continues to showcase his talent on the global stage.

## 3. Playing Style

### 3.1 Position
- Primarily plays as a forward or winger.
### 3.2 Characteristics
- **Speed:** Exceptional pace and agility.
- **Dribbling:** Known for his close control and ability to beat defenders.
- **Shooting:** Powerful shot, known for scoring from long distances and penalties.
- **Heading:** Strong aerial presence and effectiveness during set-pieces.
### 3.3 Work Ethic
- Renowned for intense training and dedication to fitness.
- Committed to maintaining top physical condition throughout his career.

## 4. Achievements and Awards
### 4.1 Club Achievements
- Numerous league titles across England, Spain, and Italy.
- Multiple UEFA Champions League titles.
### 4.2 Individual Awards
- **Ballon d'Or Wins:** 5 (2008, 2013, 2014, 2016, 2017)
- **FIFA World Player of the Year:** 1 (2008)
### 4.3 International Success
- Led Portugal to victory in Euro 2016 and the UEFA Nations League in 2019.

## 5. Off the Pitch
### 5.1 Philanthropy
- Involved in various charitable activities.
- Donated millions to hospitals and humanitarian causes.
### 5.2 Business Ventures
- Launched CR7 brand for various products including clothing, shoes, and fragrances.
- Engaged in real estate investments.
### 5.3 Personal Life
- Father to four children and has had high-profile relationships.
- Known for his disciplined lifestyle.

## 6. Impact and Legacy
### 6.1 Influence on Football
- Regarded as one of the greatest footballers of all time.
- Inspired countless young athletes around the world.
### 6.2 Cultural Icon
- Considered a global ambassador for football.
- Strong presence on social media, with millions of followers.

## 7. Key Concepts
### 7.1 Work Ethic
- Importance of consistent training and dedication.
### 7.2 Adaptability
- Evolution of playing style and ability to excel in different leagues.
### 7.3 Brand Building
- How athletes can leverage success to create a brand.

## Summary
Cristiano Ronaldo's journey from a humble beginning in Madeira to becoming one of the most celebrated athletes worldwide showcases his incredible talent, dedication, and resilience.
His on-pitch success is matched by his off-pitch endeavors, making him a true icon in both sports and global culture. With numerous records, awards, and philanthropic efforts,

Ronaldo’s impact on football and society continues to inspire future generations.

These detailed notes encompass the life, career, and legacy of Cristiano Ronaldo, providing a structured format for easy revision and understanding of his influence and achievements in football.

Tutorial No 5

Name : Sarthak Shrikant Bagul

Branch : ENTC

Div :- A

PRN :- 12410555

Roll No :- 25

Subject :- Computer Graphics (MDM)

Title / Problem Statement:

2D Transformations in Computer Graphics – Translation, Scaling, Rotation, Reflection, and Shearing

Understand and apply 2D transformations on graphical objects such as points, lines, and polygons. Explore different types of transformations and their effects on the shape, position, and orientation of objects in 2D space.

Objectives: By the end of this tutorial, students should be able to:

- Define and explain the different types of 2D transformations.
- Apply transformations such as **translation, scaling, rotation, reflection, and shearing** to 2D objects.
- Represent transformations mathematically using **homogeneous matrices**.
- Understand and analyze the effects of transformations on 2D objects in computer graphics.

Prerequisites:

- Knowledge of coordinate geometry (points, lines, polygons).
- Basic linear algebra (matrices, matrix multiplication).
- Basic trigonometry (sine, cosine for rotation).
- Understanding of computer graphics basics (pixels, raster displays).

Theory Overview:

1. **2D Transformations:**
   Transformations are operations that alter the position, size, orientation, or shape of graphical objects in 2D space.
2. **Types of Transformations:**
   (a) **Translation:** Moves an object from one location to another without changing its shape or orientation.

- Formula:

$$x' = x + tx, \quad y' = y + ty$$

(b) **Scaling:** Changes the size of an object relative to a fixed point (usually origin).

- Formula:

$$x' = Sx \cdot x, \quad y' = Sy \cdot y$$

- Scaling around a pivot requires moving the object to origin, scaling, and moving back.

(c) **Rotation:** Rotates an object around a fixed point (usually origin) by angle θ.

- Formula:

$$x' = x\cos\theta - y\sin\theta, \quad y' = x\sin\theta + y\cos\theta$$

(d) **Reflection:** Produces a mirror image of an object about a line (x-axis, y-axis, y=x, etc.).

(e) **Shearing:** Slants the shape of an object along x or y axis.

- X-shear: $x' = x + sh_x \cdot y$
- Y-shear: $y' = y + sh_y \cdot x$

3. **Homogeneous Coordinates:**

- Transformations can be represented by 3×3 matrices to combine multiple operations conveniently:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & tx \\ c & d & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- This allows **composite transformations** like scaling + rotation + translation in a single operation.

4. **Order of Transformations:**

- The sequence of transformations matters:
    - **Rotation then translation ≠ Translation then rotation**
- Composite transforms often use a **pivot point** to control rotation or scaling.

Algorithm / Steps:

A. Translation:

1. For each point (x, y), calculate:

$$x' = x + tx, y' = y + ty$$

2. Plot the new points.

B. Scaling:

1. Select scale factors Sx,Sy.
2. For each point (x, y), calculate:

$$x' = x \cdot Sx ,\ \ y' = y \cdot Sy$$

3. Plot scaled points.

C. Rotation:

1. Select rotation angle θ.
2. For each point (x, y), calculate:

$$x' = x\cos\theta - y\sin\theta, y' = x\sin\theta + y\cos\theta$$

3. Plot rotated points.

D. Reflection:

1. Choose the axis of reflection (x-axis, y-axis, y=x).
2. Apply the corresponding reflection formula.

E. Shearing:

1. Select shear factors sh_x or sh_y.
2. For each point (x, y), apply:

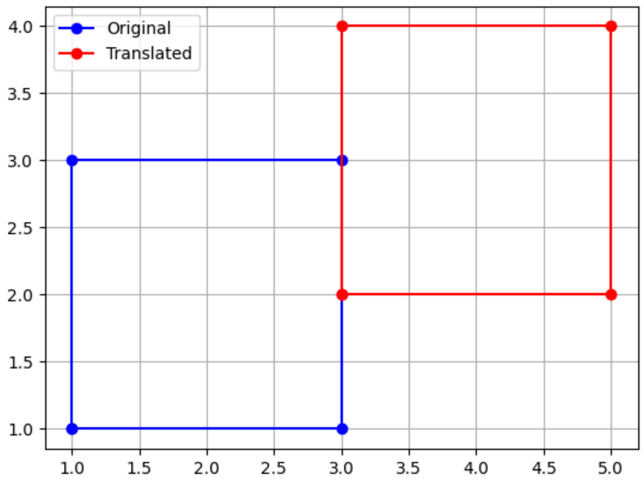
$$x' = x + sh_x \cdot y, \;\; y' = y + sh_y \cdot x$$

3. Plot sheared points.

Code Implementation (Python + Matplotlib)

(i) Translation:

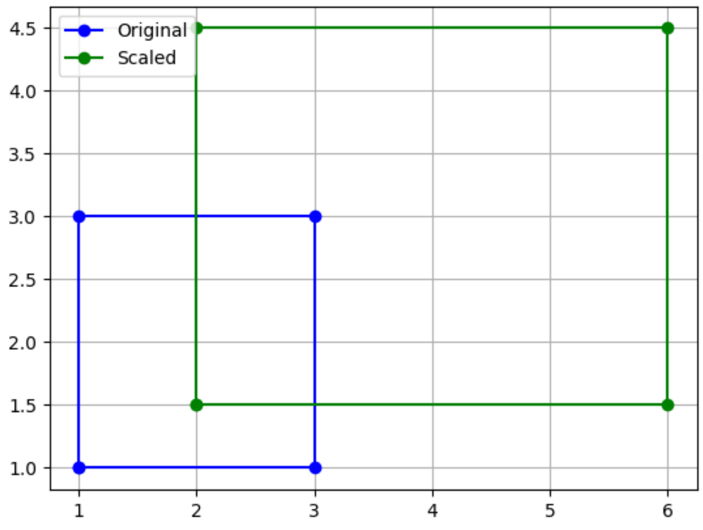
```
import matplotlib.pyplot as plt

def translate(points, tx, ty):

return [(x + tx, y + ty) for x, y in points]

# Original square

square = [(1,1),(1,3),(3,3),(3,1),(1,1)]

translated_square = translate(square, 2, 1)

x_orig, y_orig = zip(*square)

x_trans, y_trans = zip(*translated_square)

plt.plot(x_orig, y_orig, 'b-o', label='Original')

plt.plot(x_trans, y_trans, 'r-o', label='Translated')

plt.legend()

plt.grid(True)

plt.show()
```



(ii) Scaling:

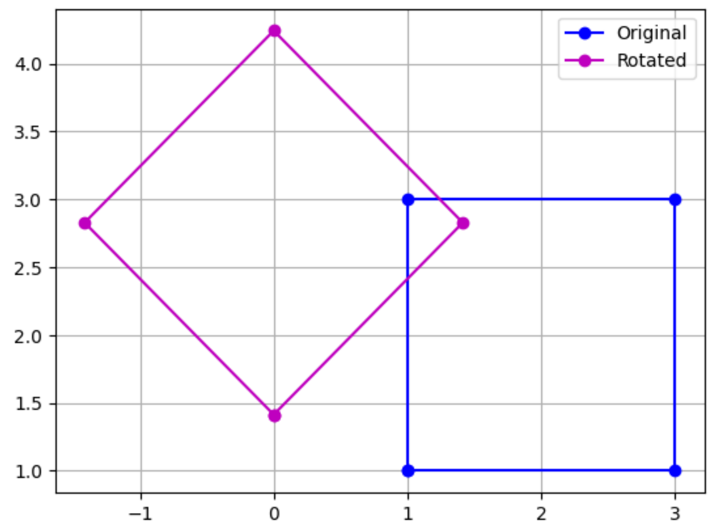
```
def scale(points, sx, sy):

  return [(x * sx, y * sy) for x, y in points]
```

```
scaled_square = scale(square, 2, 1.5)
x_scaled, y_scaled = zip(*scaled_square)
plt.plot(x_orig, y_orig, 'b-o', label='Original')
plt.plot(x_scaled, y_scaled, 'g-o', label='Scaled')
plt.legend()
plt.grid(True)
plt.show()
```



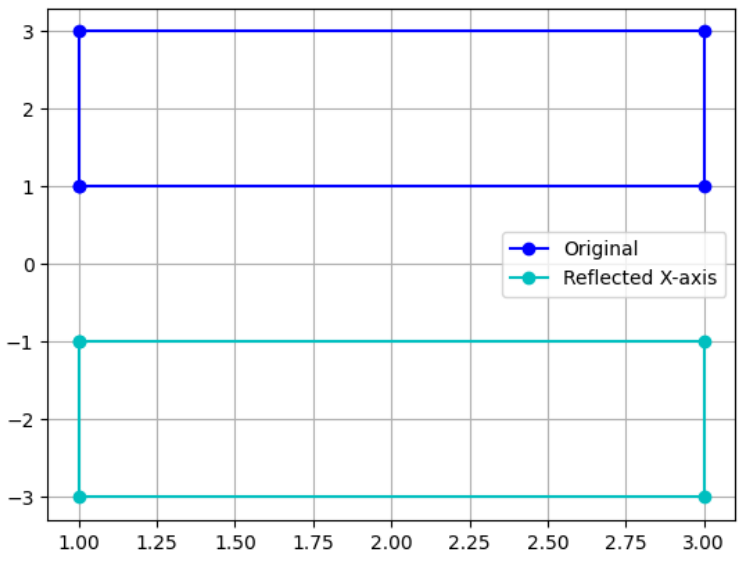
(iii) Rotation:

```
import math
def rotate(points, theta):
   rad = math.radians(theta)
   return [(x*math.cos(rad) - y*math.sin(rad), x*math.sin(rad) + y*math.cos(rad)) for x,y in points]
rotated_square = rotate(square, 45)
x_rot, y_rot = zip(*rotated_square)
plt.plot(x_orig, y_orig, 'b-o', label='Original')
plt.plot(x_rot, y_rot, 'm-o', label='Rotated')
plt.legend()
plt.grid(True)
plt.show()
```



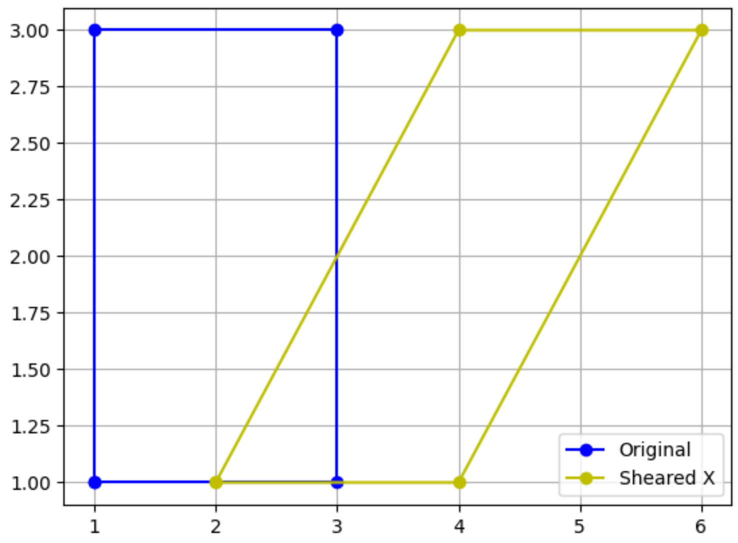
(iv) Reflection (about x-axis):

```
def reflect_x(points):
    return [(x, -y) for x, y in points]
reflected_square = reflect_x(square)
x_ref, y_ref = zip(*reflected_square)
plt.plot(x_orig, y_orig, 'b-o', label='Original')
plt.plot(x_ref, y_ref, 'c-o', label='Reflected X-axis')
plt.legend()
plt.grid(True)
plt.show()
```



(v) Shearing (x-direction):

```
def shear_x(points, shx):
    return [(x + shx*y, y) for x, y in points]
sheared_square = shear_x(square, 1)
x_shear, y_shear = zip(*sheared_square)
plt.plot(x_orig, y_orig, 'b-o', label='Original')
plt.plot(x_shear, y_shear, 'y-o', label='Sheared X')
plt.legend()
plt.grid(True)
plt.show()
```



Result / Observation / Conclusion:

- **Translation** moves the object without changing shape.
- **Scaling** changes object size; non-uniform scaling distorts shape.
- **Rotation** changes orientation while keeping shape and size intact.
- **Reflection** creates mirror images along specified axes.
- **Shearing** slants the object, altering angles but not parallelism.
- **Homogeneous matrices** allow combining multiple transformations efficiently.
- These transformations demonstrate how mathematical operations directly manipulate pixel positions on raster displays.