**Exam Pattern of JOC Course**

In Proctored / Center Based Examaination you will be getting only \*\*Multiple Choice Question (MCQ)\*\* with total 50 questions and The total duration of the examination is 180 minutes⏰.

The questions will be divided into 3 sections i,e, A, B, C respectively where,

Section A : contains 10 questions and 2 marks for each.

Section B : contains 15 questions and 2 marks for each.

Section C : contains 25 questions and 2 marks for each.

Therefore, Total Marks = 100 and for passing exam you have to score at least 40 marks out of 100 or at least you have to correct 20 questions for passing the examination.

How to prepare for the Exam

Here is one trivia for all of you if you solved all the weekly quizzes then there is high probability that some questions will directly come from weekly quizzes that means \*\*100% Copy Paste.\*\*

There is also high chances of some questions that will directly come from weekly quizzes but, in that questions you will be getting different values or data.

(Note : In my opinion seeing video lectures are not that important because in exam you'll be getting hardly 1 or 2 questions from the lectures.)

Important topics for the exam

- String, List, Dictionary, Tuple, Set

- Conditional Statements (For Loop & While Loop)

- Random & Numpy Library

- Basic Scratch & Turtle Problem (Only 2 or 3)

- Graph (Finding degree of separartion & Finding number of edges questions)

## **String in Python**

### **🔹 What is a String?**

A **string** is a sequence of characters enclosed in either single (') or double (") quotes.

name = "Python"

### **🔹 String Indexing**

Access characters using indexes (starting at 0):

print(name[0]) # Output: P

print(name[-1]) # Output: n (last character)

### **🔹 String Slicing**

Get parts of a string:

print(name[0:3]) # Output: Pyt (start to 2)

print(name[2:]) # Output: thon (from index 2 to end)

### **🔹 String Methods**

Commonly used methods:

print(name.upper()) # Output: PYTHON

print(name.lower()) # Output: python

print(name.startswith("Py")) # Output: True

print(name.endswith("on")) # Output: True

print(name.replace("Py", "My")) # Output: Mython

### **🔹 Loop Through String**

for char in name:

print(char)

### **🔹 String Concatenation**

greeting = "Hello, " + name

print(greeting) # Hello, Python

### **🔹 String Formatting**

age = 25

print(f"My name is {name} and I am {age} years old.")

## **List in Python**

### **🔹 What is a List?**

A **list** is an ordered, mutable (changeable) collection of items.

fruits = ['apple', 'banana', 'cherry']

### **🔹 Accessing Items**

print(fruits[0]) # Output: apple

print(fruits[-1]) # Output: cherry (last item)

### 

### **🔹 Modifying List**

fruits[1] = 'blueberry'

print(fruits) # Output: ['apple', 'blueberry', 'cherry']

### 

### **🔹 Adding Items**

fruits.append('mango') # Add at end

fruits.insert(1, 'orange') # Add at index 1

print(fruits)

### 

### 

### **🔹 Removing Items**

fruits.remove('apple') # Removes first occurrence

fruits.pop() # Removes last item

print(fruits)

### **🔹 Looping Through List**

for fruit in fruits:

print(fruit)

### **🔹 List Length**

print(len(fruits)) # Number of elements

### **🔹 List Slicing**

print(fruits[1:3]) # Items from index 1 to 2

### **🔹 Sorting and Reversing**

fruits.sort() # Sort alphabetically

fruits.reverse() # Reverse order

### **🔹 Check Existence**

if 'mango' in fruits:

print("Mango is here!")

## **Dictionary in Python**

### **🔹 What is a Dictionary?**

A **dictionary** is a collection of key-value pairs. It is **unordered**, **mutable**, and keys must be **unique**.

student = {'name': 'Alice', 'age': 20, 'grade': 'A'}

### **🔹 Accessing Values**

print(student['name']) # Output: Alice

print(student.get('grade')) # Output: A

### **🔹 Adding or Updating Values**

student['age'] = 21 # Update age

student['city'] = 'New York' # Add new key-value pair

### **🔹 Removing Items**

student.pop('grade') # Removes 'grade'

del student['city'] # Deletes 'city'

### **🔹 Looping Through Dictionary**

for key in student:

print(key, student[key])

# OR

for key, value in student.items():

print(f"{key}: {value}")

### **🔹 Dictionary Methods**

print(student.keys()) # All keys

print(student.values()) # All values

print(student.items()) # All key-value pairs

### **🔹 Nesting Dictionaries**

students = {

"101": {"name": "Alice", "age": 20},

"102": {"name": "Bob", "age": 22}

}

print(students["101"]["name"]) # Output: Alice

## **Tuple in Python**

### **🔹 What is a Tuple?**

A **tuple** is an ordered, immutable (unchangeable) collection of items. Use it when you want a fixed collection of values.

point = (10, 20)

### **🔹 Tuple Characteristics**

* Ordered
* Immutable (can't add/remove/modify items)
* Can contain duplicate values
* Can hold different data types

### **🔹 Accessing Elements**

print(point[0]) # Output: 10

print(point[-1]) # Output: 20

### **🔹 Tuple with Multiple Data Types**

person = ("Alice", 25, True)

### **🔹 Loop Through a Tuple**

for item in person:

print(item)

### **🔹 Tuple Packing & Unpacking**

# Packing

data = ("Bob", 30)

# Unpacking

name, age = data

print(name) # Output: Bob

print(age) # Output: 30

### **🔹 Tuple Functions**

numbers = (1, 2, 3, 2, 4)

print(len(numbers)) # Output: 5

print(numbers.count(2)) # Output: 2

print(numbers.index(3)) # Output: 2

### **🔹 Nested Tuples**

matrix = ((1, 2), (3, 4))

print(matrix[0][1]) # Output: 2

Here’s a complete breakdown of **Sets** in Python, including key explanations and examples:

## **Set in Python**

### **🔹 What is a Set?**

A **set** is an unordered collection of **unique** items. It’s useful when you want to eliminate duplicates or perform set operations like union, intersection, etc.

numbers = {1, 2, 3, 2, 1}

print(numbers) # Output: {1, 2, 3}

### **🔹 Creating a Set**

my\_set = set([1, 2, 3])

print(my\_set) # Output: {1, 2, 3}

### **🔹 Adding Items**

my\_set.add(4)

print(my\_set) # Output: {1, 2, 3, 4}

### **🔹 Removing Items**

my\_set.remove(2) # Removes 2, error if not found

my\_set.discard(5) # Does nothing if 5 not found

### **🔹 Looping Through a Set**

for item in my\_set:

print(item)

### **🔹 Set Operations**

a = {1, 2, 3}

b = {3, 4, 5}

print(a.union(b)) # {1, 2, 3, 4, 5}

print(a.intersection(b)) # {3}

print(a.difference(b)) # {1, 2}

### **🔹 Check Membership**

if 2 in a:

print("2 is in set a")

### **🔹 Convert List to Set (Remove Duplicates)**

nums = [1, 2, 2, 3]

unique\_nums = set(nums)

print(unique\_nums) # {1, 2, 3}

Here’s a complete explanation of **Conditional Statements**, along with **For Loop** and **While Loop** in Python:

## **Conditional Statements & Loops in Python**

### **🔹 If, Elif, Else (Conditional Statements)**

Used to execute code blocks based on conditions.

age = 18

if age < 18:

print("Minor")

elif age == 18:

print("Just became an adult")

else:

print("Adult")

### **🔹 For Loop**

Used to iterate over sequences (like lists, strings, etc.).

# Loop through a list

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

print(fruit)

# Loop using range

for i in range(5):

print(i) # Outputs 0 to 4

#### **Nested For Loop**

for i in range(2):

for j in range(2):

print(i, j)

### **🔹 While Loop**

Repeats a block of code while a condition is true.

i = 0

while i < 5:

print(i)

i += 1

#### **Break & Continue**

# Break example

for i in range(10):

if i == 5:

break

print(i)

# Continue example

for i in range(5):

if i == 2:

continue

print(i)

### **🔹 Loop with Else**

for i in range(3):

print(i)

else:

print("Loop ended")

i = 0

while i < 3:

print(i)

i += 1

else:

print("While loop ended")

## **Random Module in Python**

The random module is used to generate random numbers or make random selections. Useful in games, simulations, testing, etc.

### **🔹 Importing the Module**

import random

### **🔹 Random Integer**

Generates a random number within a specified range (inclusive):

print(random.randint(1, 10)) # Random number between 1 and 10

### **🔹 Random Float**

Generates a random float between 0 and 1 (default):

print(random.random()) # Output: e.g., 0.6723

You can scale it to a different range:

print(random.uniform(1.5, 4.5)) # Float between 1.5 and 4.5

### **🔹 Random Choice**

Randomly picks an element from a list or sequence:

colors = ['red', 'blue', 'green']

print(random.choice(colors))

### **🔹 Shuffle List**

Randomly shuffles the elements of a list **in place**:

cards = [1, 2, 3, 4, 5]

random.shuffle(cards)

print(cards)

### **🔹 Random Sample**

Returns a **list of unique random elements** from a list:

nums = list(range(1, 11))

print(random.sample(nums, 3)) # e.g., [4, 1, 9]

## **NumPy Library in Python**

**NumPy** stands for **Numerical Python**. It is used for scientific computing, handling arrays, matrices, and performing mathematical operations efficiently.

### **🔹 Installation (if not already installed)**

pip install numpy

### **🔹 Importing NumPy**

import numpy as np

### **🔹 Creating Arrays**

a = np.array([1, 2, 3])

print(a) # Output: [1 2 3]

2D Array:

b = np.array([[1, 2], [3, 4]])

print(b)

### **🔹 Array Attributes**

print(a.shape) # Shape of the array

print(a.ndim) # Number of dimensions

print(a.dtype) # Data type of elements

### **🔹 Basic Operations**

a = np.array([1, 2, 3])

print(a + 1) # [2 3 4]

print(a \* 2) # [2 4 6]

print(a.mean()) # Average value: 2.0

print(np.sqrt(a)) # Square roots

### **🔹 Useful Functions**

zeros = np.zeros((2, 3)) # 2x3 array of zeros

ones = np.ones((3, 3)) # 3x3 array of ones

rand = np.random.rand(2, 2) # 2x2 array with random floats

eye = np.eye(3) # 3x3 identity matrix

### **🔹 Slicing and Indexing**

arr = np.array([10, 20, 30, 40])

print(arr[1:3]) # Output: [20 30]

**Turtle Problem in Python**

The **Turtle Graphics** module in Python allows for simple graphical programming using a turtle that moves around the screen to draw shapes. Let’s look at a basic problem where we use **Turtle** to draw a shape.

### **🔹 Problem: Draw a Square using Turtle Graphics**

#### **Objective: Create a program that uses Turtle to draw a square.**

### **Solution:**

1. Create a turtle object.
2. Use a loop to draw the four sides of the square.

### **Python Code Using Turtle**

import turtle

# Create a turtle object

t = turtle.Turtle()

# Draw a square

for \_ in range(4):

t.forward(100) # Move forward by 100 units

t.right(90) # Turn right by 90 degrees

# Finish drawing

turtle.done()

### **Explanation:**

* **turtle.Turtle()**: This creates a new turtle object that will be used to draw on the screen.
* **t.forward(100)**: This command moves the turtle forward by 100 units (you can adjust the number to change the size of the square).
* **t.right(90)**: This turns the turtle right by 90 degrees, which is necessary for drawing a square.
* **for \_ in range(4)**: This loop repeats the process of moving forward and turning right four times, forming a square.
* **turtle.done()**: This finishes the drawing and keeps the window open.

Here’s an explanation and examples of **Graph** related problems in Python, specifically focusing on **degree of separation** and **number of edges**:

## **Graph in Python**

In Python, graphs can be represented using various methods, including **adjacency matrices** or **adjacency lists**. We can solve problems such as finding the degree of separation and counting the number of edges in a graph.

### **🔹 Problem 1: Finding the Degree of Separation**

**Objective:** Find the degree of separation between two nodes in a graph.

#### **Graph Representation:**

Let's represent a graph using an **adjacency list**:

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

In this graph, each node is connected to others, and the list for each node represents the nodes it is connected to.

#### **Degree of Separation:**

To find the degree of separation between two nodes, we need to perform a **breadth-first search (BFS)** or **depth-first search (DFS)**.

from collections import deque

def degree\_of\_separation(graph, start, goal):

visited = set()

queue = deque([(start, 0)]) # Each element is a tuple (node, degree)

while queue:

node, degree = queue.popleft()

if node == goal:

return degree

if node not in visited:

visited.add(node)

for neighbor in graph[node]:

queue.append((neighbor, degree + 1))

return -1 # Return -1 if no path exists

# Example usage:

print(degree\_of\_separation(graph, 'A', 'F')) # Output: 2

### **Explanation:**

* **queue.append((neighbor, degree + 1))**: We append each neighboring node along with the degree (which increases by 1 with each step).
* **if node == goal**: If we reach the goal node, we return the degree of separation.

### **🔹 Problem 2: Finding the Number of Edges in a Graph**

**Objective:** Calculate the total number of edges in a graph.

#### **Graph Representation:**

We use the same **adjacency list** from the previous 1example.

#### **Finding the Number of Edges:**

To find the number of edges, we can simply count how many times each node is connected to other nodes, and divide by 2 to avoid counting the same edge twice.

def count\_edges(graph):

edge\_count = 0

for node in graph:

edge\_count += len(graph[node])

return edge\_count // 2 # Since it's an undirected graph

# Example usage:

print(count\_edges(graph)) # Output: 6

### **Explanation:**

* **edge\_count += len(graph[node])**: This adds the number of edges for each node.
* **return edge\_count // 2**: Since the graph is undirected, each edge is counted twice (once for each node it connects), so we divide by 2.

Created By,

https://github.com/Nirmalkumar7777