Python Programming Syllabus

A Complete Guide for Beginners and Self-Learners

Target Audience: School Students, College Beginners, Coding Institutes

Level: Beginner to Intermediate

Prepared For: Comprehensive Exam & Practical Readiness

Table of Contents

Course Overview

Course Level	Beginner to Intermediate
Target Learners	High school students, college freshmen, self-taught programmers, and coding institute enrollees.
Total Duration	Approx. 40 Hours
Prerequisites	Basic computer literacy. No prior coding experience required.
Teaching Method	Theory instruction, followed by immediate practical execution and mini-projects.
Course Outcomes	Ability to write basic to intermediate Python scripts, understand core programming logic, manipulate data structures, and build basic real-world applications.

Module Planner

Module No.	Module Name	Est. Hours	Key Outcomes
1-2	Introduction & Setup	2 Hrs	Understand Python and run the first program.
3-5	Basics, Operators & I/O	5 Hrs	Declare variables, use operators, handle user input.
6-7	Control Flow	6 Hrs	Make logic decisions and iterate using loops.
8-10	Data Structures	8 Hrs	Manage data using Strings, Lists, Tuples, and Dicts.
11-13	Functions, Files, Exceptions	6 Hrs	Write modular code, store data, handle errors.
14-15	OOP Basics & Modules	4 Hrs	Understand classes, objects, and standard libraries.
16-17	Practical Programs & Projects	6 Hrs	Build standard algorithms and mini-applications.
18-20	Assessments & Revision	3 Hrs	Prepare for exams and practical vivas.

Module 1: Introduction to Python

Duration: *1 Hour* | ***Suggested Assessment:*** *Oral Q&A on Python's features*

Introduction

Python is a widely-used, high-level programming language known for its simplicity and readability. Created by Guido van Rossum in 1991, it has become the standard for data science, web development, and automation.

Learning Objectives

- Understand what Python is and its brief history.
- Identify the key features that make Python universally popular.
- Explore real-world applications and job opportunities.

Module Topics

- **What is Python?** A general-purpose, interpreted, high-level language.
- **History:** Conceived in the late 1980s, officially released in 1991.
- **Features:** Easy to read, dynamically typed, massive standard library, open-source.
- **Applications:** Web development (Django/Flask), AI/ML, Automation scripts, Game Development.
- **Why learn Python?** High market demand, beginner-friendly syntax, versatile use cases.

Practical Activity

Task: Research and note down 3 popular applications built using Python (e.g., Instagram, Spotify).

> **Tip:**
>
> Python is designed to emphasize readability. It uses plain English words instead of complex punctuation marks, making it the perfect starting point for beginners.

Revision Points

- Python is interpreted, meaning code is executed line by line.
- It is widely used in AI, Machine Learning, and Web Services.

Module 2: Python Installation and Setup

> ***Duration:*** *1 Hour* | ***Suggested Assessment:*** *Lab Setup Verification*

Introduction

Before writing code, we must set up our environment. This module covers downloading Python, installing it safely, and choosing an Integrated Development Environment (IDE).

Learning Objectives

- Successfully download and install Python on a computer.
- Configure the system PATH variables.
- Write and execute a "Hello World" script in an IDE.

Module Topics

- **Downloading Python:** Using python.org to get the latest stable version.
- **Installing Python:** Windows/Mac installation steps.
- **Python IDEs:** Introduction to IDLE, VS Code, and PyCharm.
- **Running first Python program:** Interactive mode vs. Script mode.

> **Common Mistake:**
>
> Forgetting to check the "Add Python to PATH" box during Windows installation. Without this, your computer won't recognize the `python` command in the terminal.

Practical Activity

Open IDLE or VS Code, create a new file named `hello.py` , and execute the following code:

```
print("Hello, World! I am learning Python.")
```

Revision Points

- IDLE comes pre-installed with Python.
- The default file extension for Python scripts is `.py` .

Module 3: Basics of Python Programming

Duration: *2 Hours* | ***Suggested Assessment:*** *Code Tracing Exercise*

Introduction

Understanding the fundamental building blocks of Python, such as variables, data types, and syntax rules, is crucial for writing structurally valid code.

Learning Objectives

- Understand Python's syntax and indentation rules.
- Declare variables and assign different types of values.
- Use comments to document code.

Module Topics

- **Python syntax:** No semicolons, relies on strict indentation for logic blocks.
- **Comments:** Single-line (`#`) and multi-line (`'''` or `"""`).
- **Variables:** Containers for storing data. Python is dynamically typed.
- **Keywords:** Reserved words (e.g., `if` , `while` , `class` , `def`).
- **Data types:** Integer (`int`), Float (`float`), String (`str`), Boolean (`bool`).

Practical Activity

Create a script that stores personal details in variables and prints them.

```
# This is a comment age
= 15     # int
height = 5.9   # float
name = "Alice"# str
is_student = True # bool

print("Name:", name)
print("Age:", age)
```

Revision Points

- Variables cannot start with a number.
- Indentation (usually 4 spaces) is mandatory in Python.

Module 4: Operators in Python

Duration: *2 Hours* | ***Suggested Assessment:*** *Mathematical Logic Quiz*

Introduction

Operators are special symbols used to perform operations on variables and values. Python supports a wide array of mathematical and logical operators.

Learning Objectives

- Perform mathematical calculations using arithmetic operators.
- Compare values using relational operators.
- Combine conditional statements using logical operators.

Module Topics

- **Arithmetic operators:** `+` , `-` , `*` , `/` , `//` (floor division), `%` (modulus), `**` (exponent).
- **Relational operators:** `==` , `!=` , `>` , `<` , `>=` , `<=` .
- **Logical operators:** `and` , `or` , `not` .
- **Assignment operators:** `=` , `+=` , `-=` .
- **Membership operators:** `in` , `not in` .

> **Remember:**
>
> The single equals sign `=` is for assignment, while the double equals sign `==` is for comparison.

Practical Activity

Write a script to demonstrate different arithmetic operations:

```
a = 10
b = 3
print("Division:", a / b)          # 3.3333
```

```
print("Floor Division:", a // b) # 3 print("Modulus
(Remainder):", a % b) # 1
```

Revision Points

- `**` is used for powers (e.g., 2**3 = 8).
- Logical operators return True or False.

Module 5: Input and Output Functions

> ***Duration:*** *1 Hour* | ***Suggested Assessment:*** *Build an interactive greeting script*

Introduction

To make programs interactive, we need to take input from the user and display results back to them. This module covers the core Input/Output (I/O) functions.

Learning Objectives

- Take input from users dynamically.
- Convert data types using type casting.
- Format output strings for clean, readable displays.

Module Topics

- **print():** Printing variables, text strings, and combining them using commas.
- **input():** Taking user input. It always captures data as a String by default.
- **Type Casting:** Using `int()` , `float()` , `str()` to convert input data.
- **Formatting output:** Using f-strings (e.g., `f"Hello {name}"`).

Practical Activity

Create a script that asks for a user's birth year and calculates their age.

```
name = input("Enter your name: ")
birth_year = int(input("Enter your birth year: ")) # Type casting

age = 2024 - birth_year
# Using f-string for formatted output print(f"Hello
{name}, you are {age} years old.")
```

Revision Points

- `input()` always returns a string. You must cast it for math.
- f-strings start with an `f` before the quotes and use `{}` for variables.

Module 6: Conditional Statements

Duration: *3 Hours* | ***Suggested Assessment:*** *Student Grading Program*

Introduction

Programs often need to make decisions based on specific conditions. Conditional statements allow code to branch into different logical paths depending on user input or variables.

Learning Objectives

- Use if, elif, and else statements to control program flow.
- Understand code indentation blocks.
- Create nested conditions for complex logic.

Module Topics

- **if statement:** Executes a block of code if the condition is True.
- **if else statement:** Provides an alternative block if the condition is False.
- **elif ladder:** Checks multiple conditions sequentially until one is True.
- **nested if:** Placing an if statement inside another if statement.

Practical Activity

Write a program to check if a number is positive, negative, or zero.

```
num = float(input("Enter a number: "))

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

Revision Points

- Colons `:` are required at the end of if/elif/else statements.
- Indentation defines what code belongs inside the condition.

Module 7: Loops in Python

Duration: *3 Hours* | ***Suggested Assessment:*** *Number Pattern Generation*

Introduction

Loops allow us to execute a block of code multiple times without rewriting it. They are essential for iterating over data and automating repetitive tasks.

Learning Objectives

- Differentiate between `for` and `while` loops.
- Use the `range()` function for iteration.
- Control loop execution with `break` and `continue` .

Module Topics

- **for loop:** Used for iterating over a sequence (like a list or a range of numbers).
- **while loop:** Repeats a block of code as long as a condition remains True.
- **break statement:** Instantly exits the loop entirely.
- **continue statement:** Skips the current iteration and moves to the next one.

Practical Activity

Use a loop to print the multiplication table of a given number.

```
table_num = 5
for i in range(1, 11):
    print(f"{table_num} x {i} = {table_num * i}")

# Loop control example
for i in range(5):
    if i == 3:
        break # Stops at 3
    print(i)
```

Revision Point

- `range(start, stop)` stops one number BEFORE the 'stop' value.
- Be careful with `while` loops to avoid infinite loops (always update your counter).

Module 8: Strings in Python

Duration: *2 Hours* | ***Suggested Assessment:*** *String Manipulation Exercises*

Introduction

Strings are sequences of characters used to represent text. Python provides a rich set of built-in functions to manipulate and search text data easily.

Learning Objectives

- Create and manipulate text data.
- Extract substrings using string slicing.
- Apply common string methods (upper, lower, replace).

Module Topics

- **String creation:** Using single (`' '`) or double (`" "`) quotes.
- **String slicing:** Accessing parts of a string using indices `[start:stop:step]` .
- **String functions:** `len()` , `upper()` , `lower()` , `replace()` , `find()` , `split()` .
- **String operations:** Concatenation (`+`) and Repetition (`*`).

Practical Activity

Take a sentence, convert it to uppercase, and extract the first 5 characters.

```
text = "Python Programming"

# Slicing
print("First 6 characters:", text[0:6])

# Methods
print("Uppercase:", text.upper())
print("Replaced:", text.replace("Python", "Java"))
```

Revision Points

- Strings are immutable (they cannot be changed in place; methods return a new string).
- Indexing starts at 0. Negative indexing starts at -1 from the end.

Module 9: Lists in Python

> ***Duration:*** *2 Hours* | ***Suggested Assessment:*** *List Sorting and Filtering Lab*

Introduction

Lists are ordered, mutable (changeable) collections of items. They can hold items of different data types and are one of Python's most powerful built-in data structures.

Learning Objectives

- Create and modify lists dynamically.
- Use list methods to add, remove, and sort elements.
- Work with nested lists (2D arrays).

Module Topics

- **List creation:** Defined using square brackets `[]` .
- **List methods:** `append()` , `insert()` , `remove()` , `pop()` , `sort()` .
- **Nested lists:** Lists within lists to represent grids or matrices.
- **List operations:** Indexing, slicing, concatenation, and checking membership (`in`).

Practical Activity

Create a shopping list, add items, remove an item, and sort the list.

```
cart = ["Apples", "Milk", "Bread"]
cart.append("Eggs") # Adds to end cart.insert(1,
"Butter")      # Adds at index 1 cart.remove("Milk") #
Removes specific value cart.sort()  # Alphabetical
sort

print("Final Cart:", cart)
```

Revision Points

- Lists are mutable, meaning you can change elements using their index (e.g., `cart[0] = "Juice"`).
- `pop()` removes and returns the last item by default.

Module 10: Tuples and Dictionaries

> ***Duration:*** *2 Hours* | ***Suggested Assessment:*** *Data Structure Identification Test*

Introduction

While lists are mutable, tuples are immutable collections. Dictionaries, on the other hand, store data in key-value pairs for extremely fast data retrieval without relying on numerical indices.

Learning Objectives

- Understand the difference between lists and tuples.
- Create, access, and modify dictionaries.
- Iterate through dictionary keys and values.

Module Topics

- **Tuple basics:** Creation using parentheses `()`, immutability concepts.
- **Dictionary creation:** Using curly braces `{}` with `key: value` mappings.
- **Dictionary methods:** `keys()`, `values()`, `items()`, `get()`, `update()`.

Practical Activity

Create a dictionary storing student grades, update a grade, and print all keys.

```
# Tuple (Immutable)
dimensions = (1920, 1080)

# Dictionary (Key-Value)
student = {
    "name": "John Doe",
    "age": 16,
    "grade": "A"
}

student["age"] = 17 # Update value
```

```
print("Student Name:", student.get("name"))
print("All keys:", student.keys())
```

Revision Points

- Use tuples for data that should never change (like geographic coordinates).
- Dictionary keys must be unique and immutable (strings, numbers, tuples).

Module 11: Functions in Python

Duration: *3 Hours* | ***Suggested Assessment:*** *Write modular utility scripts*

Introduction

Functions are isolated blocks of reusable code that perform a specific task. They help organize large scripts, make code readable, and follow the DRY (Don't Repeat Yourself) principle.

Learning Objectives

- Define custom functions using the `def` keyword.
- Pass arguments to functions safely.
- Return calculated values from functions.

Module Topics

- **Defining functions:** Syntax and indentation.
- **Function arguments:** Positional parameters, keyword arguments, and default values.
- **Return statement:** Sending results back to the caller instead of just printing.
- **Recursive functions basics:** A brief look at functions that call themselves.

Practical Activity

Write a function that calculates the area of a rectangle with a default width.

```
def calculate_area(length, width=5):
    area = length * width
    return area

# Calling the function
result1 = calculate_area(10, 10)
result2 = calculate_area(10) # Uses default width 5

print("Area 1:", result1)
print("Area 2:", result2)
```

Revision Points

- Variables declared inside a function are local and cannot be accessed outside.
- A function ends immediately when a `return` statement is executed.

Module 12: File Handling in Python

Duration: *2 Hours* | ***Suggested Assessment:*** *Read from and write to a `.txt` file*

Introduction

File handling allows a Python program to read data from local files and write data to files, enabling persistent data storage beyond the runtime of the script.

Learning Objectives

- Open and close files safely.
- Read string data from text files.
- Write and append data to files without data loss.

Module Topics

- **Opening files:** Using the `open()` function and modes ('r', 'w', 'a').
- **Reading files:** `read()` , `readline()` , `readlines()` .
- **Writing files:** Overwriting content using 'w' mode.
- **Appending data:** Adding content to the end of a file using 'a' mode.

> **Tip:**
>
> Always use the `with open(...) as file:` syntax. It automatically closes the file for you, even if your program crashes, preventing memory leaks!

Practical Activity

Create a file, write a sentence to it, and read it back.

```
# Writing to a file
with open("notes.txt", "w") as file:
    file.write("Python file handling is easy!\n")

# Reading from the file
with open("notes.txt", "r") as file:
```

```
    content = file.read()
    print("File Content:", content)
```

Revision Points

- 'w' mode will completely erase the existing file before writing.
- 'a' mode keeps existing text and adds new text to the end.

Module 13: Exception Handling

Duration: *1 Hour* | ***Suggested Assessment:*** *Debugging broken code lab*

Introduction

Errors often occur during execution (like dividing by zero, or a file not being found). Exception handling helps manage these runtime errors gracefully so the program doesn't abruptly crash.

Learning Objectives

- Identify common Python exceptions.
- Use try-except blocks to catch and handle errors.
- Ensure execution of cleanup code using `finally` .

Module Topics

- **try:** The block of code to be monitored for errors.
- **except:** The block of code that executes if an error occurs in the try block.
- **finally:** The block of code that executes regardless of whether an error occurred or not.

Practical Activity

Write a division program that handles division by zero and invalid inputs.

```
try:
    num = int(input("Enter a number to divide 100 by: "))
    result = 100 / num
    print("Result:", result)
except ZeroDivisionError:
    print("Error: You cannot divide by zero!")
except ValueError:
    print("Error: Please enter a valid integer.")
finally:
    print("Operation finished.")
```

Revision Points

- You can have multiple `except` blocks for different error types.
- The `finally` block is often used to close files or release resources.

Module 14: Object-Oriented Programming Basics

Duration: *2 Hours* | ***Suggested Assessment:*** *Create a blueprint Class*

Introduction

Object-Oriented Programming (OOP) is a paradigm based on the concept of "objects", which contain data (attributes) and code (methods). It models real-world entities.

Learning Objectives

- Understand the difference between a class and an object.
- Define a class and instantiate objects from it.
- Use constructors for initializing attributes.

Module Topics

- **Class and object:** A class is a blueprint; an object is a physical instance.
- **Constructor:** The special `__init__` method used for setup.
- **Encapsulation basics:** Bundling data and methods together; basic private variables.

Practical Activity

Create a class for a Car with attributes and a method.

```
class Car:
    # Constructor
    def init  (self, brand, color):
        self.brand = brand self.color
        = color

    # Method
    def start_engine(self):
             print(f"The {self.color} {self.brand} is starting.
Vroom!")
```

```
# Creating Objects
car1 = Car("Toyota", "Red")
car1.start_engine()
```

Revision Points

- `self` refers to the specific object being created/used.
- Classes use PascalCase naming convention (e.g., `StudentProfile`).

Module 15: Modules and Libraries

Duration: *2 Hours* | ***Suggested Assessment:*** *Math/Random script creation*

Introduction

Modules are pre-written Python files containing useful functions and variables. Libraries are collections of modules. Reusing code via libraries is a core philosophy of Python development.

Learning Objectives

- Import standard standard Python modules.
- Utilize functions from the math and random libraries.

Module Topics

- **Import statement:** Using `import module_name` or `from module import func`.
- **math module:** Accessing `sqrt()`, `pow()`, `pi`, `ceil()`.
- **random module:** Using `randint()`, `choice()` for game logic.

Practical Activity

Create a script that simulates rolling a 6-sided dice and calculates the square root of a number.

```
import math
import random

# Using Random
dice_roll = random.randint(1, 6)
print(f"You rolled a {dice_roll}")

# Using Math
number = 25
print(f"Square root of {number} is {math.sqrt(number)}")
```

Revision Points

- Using `from math import pi` lets you use `pi` directly without typing `math.pi` .
- Python has thousands of external libraries available via `pip install` .

Module 16: Practical Programs

This module provides beginner-friendly standard programs to solidify logic building. These are frequently asked in lab exams.

1. Addition of Two Numbers

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
sum_val = num1 + num2
print(f"The sum is: {sum_val}")
```

Note: Teaches basic I/O, type casting, and arithmetic operations.

2. Check Even or Odd

```
num = int(input("Enter a number: "))
if num % 2 == 0:
    print(f"{num} is Even")
else:
    print(f"{num} is Odd")
```

Note: Teaches the modulus operator and if-else logic branching.

3. Factorial of a Number

```
num = int(input("Enter a positive integer: "))
fact = 1
for i in range(1, num + 1):
    fact = fact * i
print(f"The factorial of {num} is {fact}")
```

Note: Demonstrates the use of a for-loop and accumulating a product.

4. Check Prime Number

```
num = int(input("Enter a number: "))
is_prime = True

if num > 1:
    for i in range(2, num):
        if num % i == 0:
            is_prime = False
            break

if is_prime and num > 1:
    print("Prime Number")
else:
    print("Not a Prime Number")
```

Note: Teaches loop breaking, flags, and advanced conditional logic.

5. Fibonacci Series

```
n = int(input("How many terms? "))
n1, n2 = 0, 1

print("Fibonacci Sequence:")
for _ in range(n):
    print(n1, end=" ")
    nth = n1 + n2
    n1 = n2
    n2 = nth
```

Note: Shows sequence generation and parallel variable updating.

6. Calculator Program

```
def add(x, y): return x + y
def subtract(x, y): return x - y
def multiply(x, y): return x * y
def divide(x, y): return x / y if y != 0 else "Error!"

print("Select operation: 1.Add 2.Subtract 3.Multiply 4.Divide")
choice = input("Enter choice (1/2/3/4): ")
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

if choice == '1': print("Result:", add(num1, num2))
elif choice == '2': print("Result:", subtract(num1, num2))
elif choice == '3': print("Result:", multiply(num1, num2))
elif choice == '4': print("Result:", divide(num1, num2)) else:
print("Invalid Input")
```

Note: Merges function definitions, user menus, and conditional execution.

7. File Handling Program

```
filename = "data.txt"

# Write data
with open(filename, "w") as f: f.write("Welcome
    to Python programming!\n") f.write("File
    handling is essential.")

# Read data
print("Reading from file...")
with open(filename, "r") as f:
    for line in f:
        print(line.strip())
```

Note: Explains creating files, writing lines, and reading line-by-line using a loop.

Module 17: Python Mini Projects

Applying concepts to small projects bridges the gap between theory and application.

1. Student Management System

- **Purpose:** Manage student records dynamically using classes.
- **Concepts Used:** Object-Oriented Programming (OOP), Lists, Loops.
- **Suggested Steps:** Create a `Student` class with attributes (name, roll, marks). Create an empty list to store objects. Build a while loop menu offering options to "Add Student" and "Display All".
- **Extension Idea:** Add a feature to save the student list to a CSV file.

2. Simple Calculator

- **Purpose:** Create an interactive CLI tool for continuous arithmetic.
- **Concepts Used:** Functions, While Loops, Conditional Statements.
- **Suggested Steps:** Define basic math functions. Wrap the input and logic inside a `while True:` loop so the user can perform multiple calculations until they type "exit".
- **Extension Idea:** Add advanced math features using the `math` library (e.g., square roots or exponents).

3. Quiz Game

- **Purpose:** Test user knowledge and keep a running score.
- **Concepts Used:** Dictionaries, For Loops, Input/Output.
- **Suggested Steps:** Create a dictionary where keys are questions and values are answers. Loop through the dictionary, ask the user the question, check their answer, and add +1 to a score variable if correct.
- **Extension Idea:** Randomize the question order using `random.shuffle()` .

4. To-Do List

- **Purpose:** Create a daily task tracker.
- **Concepts Used:** Lists, While Loops, List Methods.

- **Suggested Steps:** Initialize an empty list `tasks = []` . Present a menu: 1. Add Task, 2. View Tasks, 3. Delete Task, 4. Exit. Use `tasks.append()` and `tasks.remove()` based on user input.
- **Extension Idea:** Implement Exception Handling to prevent errors if a user tries to delete a task number that doesn't exist.

Module 18: Viva Questions and Important Questions

Theory Questions

1. What is Python? Who created it and when?
2. Explain the difference between compiled and interpreted languages. Which category does Python belong to?
3. What are the primary differences between a List and a Tuple in Python?
4. How does a Dictionary store data? Explain with a real-world analogy.
5. Explain the fundamental difference between a `for` loop and a `while` loop.
6. What is the function of `break` and `continue` statements inside loops?
7. Differentiate between the assignment operator (`=`) and the equality operator (`==`).
8. What is the default return type of the `input()` function? How do you accept integer inputs?
9. What are modules in Python? Mention two standard modules you have used.
10. Define Object-Oriented Programming (OOP). What are classes and objects?

Practical / Scenario-based Questions

11. Write a single line of Python code to find the length of the string `text = "Programming"` .
12. If you have `x = "123"` , how do you mathematically add 5 to it without causing a TypeError?
13. What happens if you try to open a non-existent file in `'r'` mode versus `'w'` mode?
14. How would you use a `try-except` block to handle a user inputting a letter instead of a number for age?
15. Given a list `fruits = ["Apple", "Banana"]` , show two different ways to add "Mango" to it.

Module 19: MCQs and Practice Questions

Multiple Choice Questions (MCQs)

1. **Which keyword is used to define a function in Python?**
   a) func b) define c) def d) function
   Ans: c

2. **Which of the following data types is immutable?**
   a) List b) Dictionary c) Set d) Tuple
   Ans: d

3. **What is the output of `print(2 ** 3)` ?**
   a) 6 b) 8 c) 9 d) 12
   Ans: b

4. **Which extension is used for Python files?**
   a) .py b) .python c) .pt d) .txt
   Ans: a

5. **What does `type(5.0)` return?**
   a) int b) float c) double d) string
   Ans: b

6. **Which block executes regardless of errors?**
   a) try b) except c) finally d) catch
   Ans: c

7. **How do you start a single-line comment?**
   a) // b) /* c) <!-- d) #
   Ans: d

8. **In a dictionary, elements are stored as:**
   a) Indices b) Key-Value pairs c) Nodes d) Arrays
   Ans: b

9. **Which method adds an element to the end of a list?**
   a) insert() b) push() c) append() d) add()
   Ans: c

10. **What library contains the `sqrt()` function?**
    a) random b) math c) os d) sys
    Ans: b

11. **Which symbol performs floor division?**
    a) / b) // c) % d) **

Ans: b

12. **What is the output of `len("Code")` ?**
    a) 3 b) 4 c) 5 d) Error
    Ans: b
13. **How do you read the entire content of a file at once?**
    a) readlines() b) read() c) readline() d) get()
    Ans: b
14. **Which operator checks if a value exists in a list?**
    a) exists b) in c) == d) find
    Ans: b
15. **What does the `continue` statement do?**
    a) Exits program b) Exits loop c) Skips current iteration d) Pauses execution
    Ans: c

Short Answer Questions

1. Write a python program to check if a person is eligible to vote (Age >= 18).
2. Define dynamic typing in Python.
3. Give an example of string slicing to reverse a string.
4. Explain the difference between `append()` and `insert()` in lists.
5. How do you handle a `ValueError` in Python?
6. Write the syntax to open a file named 'data.txt' for writing.
7. What is the purpose of the `__init__` method in a class?
8. How does the `random.randint()` function work?

Long Answer Questions

1. Describe the various data types available in Python with examples.
2. Explain Control Flow in Python. Detail the use of if, elif, else, and nested if statements.
3. Compare Lists, Tuples, and Dictionaries based on mutability, syntax, and use cases. Provide code examples.
4. What are Functions? Explain parameters, default arguments, and return values with a complete example program.
5. Discuss Exception Handling. Why is it important, and how do try, except, and finally blocks operate together?
6. Explain Object-Oriented Programming principles focusing on Classes, Objects, and encapsulation in Python.

Module 20: Revision Notes

Important Python Concepts

- **Indentation:** Python entirely relies on whitespace to define block scopes instead of curly braces `{}` .
- **Mutability:** Mutable types (Lists, Dicts, Sets) can be altered in-place. Immutable types (Strings, Tuples, Ints) cannot.
- **Zero-Indexing:** When accessing elements in sequences, the first element is always at position `0` .
- **Everything is an Object:** Integers, strings, lists, and functions are all objects under the hood in Python.

Quick Syntax Summary

Concept	Syntax / Example
Variable	`age = 25`
List	`colors = ["red", "blue"]`
Dictionary	`data = {"key": "value"}`
If Statement	`if x > 10: print("High")`
For Loop	`for i in range(5): print(i)`
While Loop	`while x > 0: x -= 1`
Function	`def greet(name): return f"Hi {name}"`
Class	`class User: def __init__(self): pass`
File Reading	`with open('f.txt', 'r') as file:`

Exam Preparation Tips

- **Write code on paper:** Exams often require hand-writing syntax without IDE autocomplete. Practice indentations clearly.

- **Memorize exact method names:** Remember `append()` for lists (not add), and `keys()` for dictionaries.
- **Trace variables:** Practice predicting the output of nested loops and conditions step-by-step.

One-Page Revision Bullets

- Python created by Guido van Rossum in 1991.
- `input()` always captures a string.
- `//` is floor division, `%` is modulus (remainder).
- Lists use `[]`, Tuples use `()`, Dictionaries use `{}`.
- Functions are defined with `def`. Return values using `return`.
- File modes: `'r'` (read), `'w'` (write/overwrite), `'a'` (append).
- Handle errors using `try` and `except`.
- Classes define attributes and methods. `self` represents the instance.

Happy Coding!

Learning programming is a journey of continuous problem-solving. Do not be discouraged by errors—they are essential stepping stones to mastery. Keep practicing, build your own projects, and explore the endless possibilities of Python. You've got this!
