MODULE -1

Print Statement:

What is the Print Statement?

The print statement in Python is used to display text or other output on the screen. It's one of the most basic and essential functions you'll use in Python programming. With the print function, you can show strings (text), integers (numbers), and even multiple lines of text.

The print() function is your fundamental tool for displaying text and other information in Python. It's like a megaphone that lets your program speak to the world (or more realistically, to the user's console).

Understanding with Examples

Printing Strings

A string is a sequence of characters enclosed in quotation marks. We will see strings in more detail in future modules. You can use either single quotes (' ') or double quotes (" ") for strings.

Example:

```
print("Hello, world!")
print('Python is fun!')
```

In these examples, `"Hello, world!"` and `'Python is fun!'` are strings. The `print` function displays these strings on the screen.

Printing Integers

Integers are whole numbers without a decimal point. You can directly print integers using the `print` function.

Example:

```
print(123)
print(456)
```

Here, `123` and `456` are integers that will be displayed on the screen.

Printing Multi-Line Strings

Sometimes, you might want to print text that spans multiple lines. You can do this using triple

quotes (`'''` or `"""`).

Example:

```
print("""This is a multi-line string.
It can span multiple lines.
Pretty cool, right?""")
```

In this example, the text inside the triple quotes will be printed exactly as it appears, across multiple lines.

Code Examples

Example 1: Printing a Single String

```
print("Welcome to Python programming!")
```

Example 2: Printing an Integer

```
print(2024)
```

Example 3: Printing a Multi-Line String

```
print("""This is an example of a multi-line string.
It allows you to print text
across multiple lines.
Isn't that amazing?""")
```

The `print` function in Python is a powerful tool for displaying text and numbers. By mastering the basics of printing strings, integers, and multi-line strings, you'll be well-equipped to create informative and user-friendly programs.

Feel free to practice the examples and questions provided to reinforce your understanding of the `print` statement!

MODULE-2

Introduction to Python Variables, Rules of Creating a Variable Name, Comments, and Built-In Functions

What are Variables?

Variables are used to store information that can be referenced and manipulated in a program. They act as containers for data values. For example, you can store a number, a string, or other types of data in a variable and use that variable later in your code.

Example : Imagine you're a student and you want to store information about your grades in different subjects. You can use variables to store these grades and perform calculations or comparisons later.

Rules of Creating a Variable Name

1. Names can contain letters, digits, and underscores (_).
2. Names must start with a letter or an underscore (_). They cannot start with a digit.
3. Names are case-sensitive. For example, myVar and myvar are considered two different variables.
4. Avoid using Python reserved words as variable names. These are words that have special meaning in Python (e.g., if, while, return, def).

Examples of Valid Variable Names:

- student_name
- math_grade
- total_marks
- _average_score

Examples of Invalid Variable Names:

- 2nd_place (starts with a digit)
- class (reserved word)
- student-name (contains a hyphen)

Comments

Comments are used to explain the code and make it more readable. They are ignored by the Python interpreter, so they don't affect the program's execution. Comments can be single-line or multi-line.

Single-line Comment:

```
# This is a single-line comment
student_name = "Alice" # This is an inline comment
```

Multi-line Comment:

```
"""

This is a multi-line comment.

It can span multiple lines.

Useful for providing detailed explanations.

"""
```

Example: Use comments to explain what different parts of your code do, such as calculating the average grade or printing the final result.

Built-In Functions

Python has several built-in functions that you can use to perform common tasks. Two important built-in functions are input and len.

input() Function:

Used to take input from the user.

The input is returned as a string.

Example:

```
student_name = input("Enter your name: ")
```

len() Function:

- Used to get the length of a string or other data structures like lists.

Example:

```
name_length = len(student_name)

print("The length of your name is:", name_length)
```

Understanding with Examples

Example 1: Storing and Printing Variables

```
# Store student's name and grade in variables

student_name = "Alice"

math_grade = 95


# Print the stored information
```

```
print("Student Name:", student_name)

print("Math Grade:", math_grade)
```

Example 2: Using Comments

```
# Calculate the average of three subjects

math_grade = 95

science_grade = 88

english_grade = 92


# Calculate the average grade

average_grade = (math_grade + science_grade + english_grade) / 3


# Print the average grade

print("Average Grade:", average_grade)
```

Example 3: Taking Input and Using Built-In Functions

```
# Take student's name as input

student_name = input("Enter your name: ")


# Print the length of the student's name

name_length = len(student_name)

print("The length of your name is:", name_length)
```

By understanding variables, the rules for creating variable names, comments, and built-in functions like input and len, you can write more effective and readable Python programs. These concepts are fundamental and will help you as you progress in learning Python.

Feel free to practice the examples and questions provided to reinforce your understanding of these topics!

MODULE-3

What are Data Types?

Data types specify the type of data that a variable can hold. They determine what operations can be performed on the data and how the data is stored in memory.

Built-in Data Types

Integer (int)

An integer is a whole number without any fractional part. It's commonly used to represent counts, scores, ages, and other discrete values.

Example :

- Age of a student
- Number of books in a library
- Total marks in an exam

Example:

```
age = 20

books_in_library = 5000

total_marks = 450
```

Float (float)

A float is a number that has a decimal point. It's used to represent continuous values like measurements, grades, and other precise quantities.

Real life Examples :

- GPA (Grade Point Average)
- Height of a student
- Temperature

Example:

```
gpa = 3.75

height = 5.8  # in feet

temperature = 98.6  # in Fahrenheit
```

String (str)

A string is a sequence of characters enclosed in single, double, or triple quotes. It's used to represent text.

Real life Examples :

- Name of a student
- Class name
- Address

Example:

student_name = "Alice"

class_name = "10th Grade"

address = "123 Main St, Springfield"

Boolean (bool)

A boolean represents one of two values: True or False. It's used to represent binary conditions or flags.

Example:

- Whether a student is enrolled in a course
- Whether an assignment is submitted
- Whether a student passed an exam

Example:

is_enrolled = True

assignment_submitted = False

passed_exam = True

Type Function

The type() function is used to determine the data type of a variable or value.

Example:

age = 20

print(type(age)) # Output: <class 'int'>

```
gpa = 3.75
print(type(gpa))  # Output: <class 'float'>

student_name = "Alice"
print(type(student_name))  # Output: <class 'str'>

is_enrolled = True
print(type(is_enrolled))  # Output: <class 'bool'>
```

Type Casting

Type casting is the process of converting one data type to another. In Python, this can be done using built-in functions like int(), float(), str(), and bool().

Example:

```
# Convert integer to float
num = 10
num_float = float(num)
print(num_float)  # Output: 10.0
print(type(num_float))  # Output: <class 'float'>

# Convert string to integer
num_str = "123"
num_int = int(num_str)
print(num_int)  # Output: 123
print(type(num_int))  # Output: <class 'int'>

# Convert integer to string
num = 456
num_str = str(num)
print(num_str)  # Output: "456"
print(type(num_str))  # Output: <class 'str'>
```

Understanding with Examples

Example 1: Creating Variables with Different Data Types

```
# Integer
age = 20

# Float
gpa = 3.75

# String
student_name = "Alice"

# Boolean
is_enrolled = True

print(age)  # Output: 20
print(gpa)  # Output: 3.75
print(student_name)  # Output: Alice
print(is_enrolled)  # Output: True
```

Example 2: Using Type Function

```
age = 20
gpa = 3.75
student_name = "Alice"
is_enrolled = True

print(type(age))  # Output: <class 'int'>
print(type(gpa))  # Output: <class 'float'>
print(type(student_name))  # Output: <class 'str'>

print(type(is_enrolled))  # Output: <class 'bool'>
```

Example 3: Type Casting

```
# Convert string to integer

roll_number_str = "1001"

roll_number_int = int(roll_number_str)

print(roll_number_int)  # Output: 1001

print(type(roll_number_int))  # Output: <class 'int'>


# Convert float to string

gpa = 3.75

gpa_str = str(gpa)

print(gpa_str)  # Output: "3.75"

print(type(gpa_str))  # Output: <class 'str'>
```

Understanding data types is crucial in programming as they determine the kind of operations you can perform on the data. Python provides several built-in data types like integers, floats, strings, and booleans. Using the type() function, you can check the data type of any variable, and with type casting, you can convert one data type to another as needed.

Feel free to practice the examples and questions provided to reinforce your understanding of these topics!

MODULE-4

Introduction to Arithmetic Operators, Assignment Operators, and Comparison Operators

What are Operators?

Operators are special symbols or keywords in Python that are used to perform operations on variables and values. They help in performing computations, comparisons, and assignments.

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations. Here are the common arithmetic operators in Python:

1. **Addition (+)**
    - Adds two values.
    - Example: a + b
2. **Subtraction (-)**
    - Subtracts the second value from the first.
    - Example: a - b
3. **Multiplication (*)**
    - Multiplies two values.
    - Example: a * b
4. **Division (/)**
    - Divides the first value by the second. The result is a float.
    - Example: a / b
5. **Floor Division (//)**
    - Divides the first value by the second and returns the largest integer less than or equal to the result.
    - Example: a // b
6. **Modulus (%)**
    - Returns the remainder when the first value is divided by the second.
    - Example: a % b
7. **Exponentiation (**)**
    - Raises the first value to the power of the second.
    - Example: a ** b

Assignment Operators

Assignment operators are used to assign values to variables. Here are the common assignment operators in Python:

1. **Equals (=)**
   - Assigns the value on the right to the variable on the left.
   - Example: a = b
2. **Add and Assign (+=)**
   - Adds the value on the right to the variable and assigns the result to the variable.
   - Example: a += b (equivalent to a = a + b)
3. **Subtract and Assign (-=)**
   - Subtracts the value on the right from the variable and assigns the result to the variable.
   - Example: a -= b (equivalent to a = a - b)
4. **Multiply and Assign (*=)**
   - Multiplies the variable by the value on the right and assigns the result to the variable.
   - Example: a *= b (equivalent to a = a * b)
5. **Divide and Assign (/=)**
   - Divides the variable by the value on the right and assigns the result to the variable.
   - Example: a /= b (equivalent to a = a / b)
6. **Modulus and Assign (%=)**
   - Applies modulus operation on the variable with the value on the right and assigns the result to the variable.
   - Example: a %= b (equivalent to a = a % b)

Comparison Operators

Comparison operators are used to compare two values. Here are the common comparison operators in Python:

1. **Equal to (==)**
   - Checks if two values are equal.
   - Example: a == b
2. **Not equal to (!=)**
   - Checks if two values are not equal.

   - Example: a != b
3. **Greater than (>)**
   - Checks if the first value is greater than the second.
   - Example: a > b
4. **Less than (<)**
   - Checks if the first value is less than the second.
   - Example: a < b
5. **Greater than or equal to (>=)**
   - Checks if the first value is greater than or equal to the second.
   - Example: a >= b
6. **Less than or equal to (<=)**
   - Checks if the first value is less than or equal to the second.
   - Example: a <= b

Understanding with Examples

Example 1: Arithmetic Operators

```
# Arithmetic Operations
a = 10
b = 5

print(a + b)  # Output: 15 (Addition)
print(a - b)  # Output: 5 (Subtraction)
print(a * b)  # Output: 50 (Multiplication)
print(a / b)  # Output: 2.0 (Division)
print(a // b) # Output: 2 (Floor Division)
print(a % b)  # Output: 0 (Modulus)
print(a ** b) # Output: 100000 (Exponentiation)
```

Example 2: Assignment Operators

```
# Assignment Operations
a = 10

a += 5
print(a)  # Output: 15

a -= 3
print(a)  # Output: 12

a *= 2
print(a)  # Output: 24

a /= 4
print(a)  # Output: 6.0

a %= 5
print(a)  # Output: 1.0
```

Example 3: Comparison Operators

```
# Comparison Operations
a = 10
b = 5

print(a == b)  # Output: False
print(a != b)  # Output: True
print(a > b)   # Output: True
print(a < b)   # Output: False
print(a >= b)  # Output: True
print(a <= b)  # Output: False
```

Operators in Python are essential for performing a variety of tasks, including mathematical calculations, value assignments, and comparisons. Arithmetic operators help with math operations, assignment operators simplify variable updates, and comparison operators allow you to compare values effectively.

Feel free to practice the examples and questions provided to reinforce your understanding of these topics!

MODULE-5

What is a List?

A list in Python is a collection of items that are ordered and changeable. Lists are one of the most common data structures used in Python. You can put any type of data in a list, like numbers, strings, or even other lists.

Understanding Lists with Examples

Imagine you have a shopping list. You write down all the items you need to buy, like eggs, milk, and bread. This shopping list is similar to a list in Python. You can look at items in the order you wrote them down, add new items, or cross out items you've bought.

Introduction to Lists and List Methods

Let's go on an adventure to understand lists and list methods in Python. Imagine you are organizing a small party for your friends. You need to keep track of various items you need to buy, tasks to complete, and even some games to play. Lists in Python can help you do all this in an organized way!

The Party Planning Adventure

The Party Shopping List

You decide to start with a shopping list. A list in Python is like your shopping list where you can write down items you need to buy.

```
shopping_list = ['balloons', 'cake', 'drinks']

print("Original shopping list:", shopping_list)
```

Output:

```
Original shopping list: ['balloons', 'cake', 'drinks']
```

Adding Items with append()

As you remember more items, you can add them to your list using the append() method. You realize you also need to buy party hats.

```
shopping_list.append('party hats')

print("Shopping list after adding 'party hats':", shopping_list)
```

Output:

```
Shopping list after adding 'party hats': ['balloons', 'cake', 'drinks', 'party hats']
```

Removing Items with remove()

You decide to remove an item you no longer need. Let's say you already have enough drinks at home, so you remove 'drinks' from your list.

```
shopping_list.remove('drinks')

print("Shopping list after removing 'drinks':", shopping_list)
```

Output:

```
Shopping list after removing 'drinks': ['balloons', 'cake', 'party hats']
```

Changing Your Mind with pop()

If you want to remove an item at a specific position, like the first item on your list, you can use the pop() method. Let's remove the first item, 'balloons'.

```
removed_item = shopping_list.pop(0)

print("Removed item:", removed_item)

print("Shopping list after popping the first item:", shopping_list)
```

Output:

```
Removed item: balloons

Shopping list after popping the first item: ['cake', 'party hats']
```

If you just want to remove the last item, you can call pop() without any arguments.

```
removed_item = shopping_list.pop()

print("Removed item:", removed_item)

print("Shopping list after popping the last item:", shopping_list)
```

Output:

```
Removed item: party hats

Shopping list after popping the last item: ['cake']
```

Organizing the Party with sort()

You want to organize your tasks in alphabetical order to make sure you don't forget anything important. You can use the sort() method to sort your tasks.

```
tasks = ['invite friends', 'decorate house', 'buy cake', 'send invitations']

print("Original tasks:", tasks)


tasks.sort()

print("Tasks after sorting:", tasks)
```

Output:

```
Original tasks: ['invite friends', 'decorate house', 'buy cake', 'send invitations']

Tasks after sorting: ['buy cake', 'decorate house', 'invite friends', 'send invitations']
```

Reversing the Order with reverse()

If you want to reverse the order of your tasks, you can use the reverse() method.

```
tasks.reverse()

print("Tasks after reversing:", tasks)
```

Output:

```
Tasks after reversing: ['send invitations', 'invite friends', 'decorate house', 'buy cake']
```

Tuples: Locking in Your Guest List

Now that you have your shopping list and tasks sorted, let's move on to something that shouldn't change - your guest list. A tuple in Python is like a guest list that you can't change once it's written. You can't add or remove guests from this list.

What is a Tuple?

A tuple is similar to a list, but it is immutable. This means that once you create a tuple, you cannot change its elements. Tuples are useful for storing a collection of items that should not be modified.

Creating the Guest List

```
guest_list = ('Alice', 'Bob', 'Charlie')

print("Guest list:", guest_list)
```

Output:

```
Guest list: ('Alice', 'Bob', 'Charlie')
```

Immutable Guest List

Unlike lists, you can't change the elements of a tuple. This means your guest list is fixed. You can't accidentally add or remove guests from this list.

```
# Trying to change an element (this will cause an error)

# guest_list[0] = 'Dave'  # This line will raise an error
```

MODULE-6

Introduction to Identity Operators and Membership Operators

Identity Operators

Identity operators are used to compare the memory locations of two objects. There are two identity operators in Python:

1. is : Evaluates to True if the variables on either side refer to the same object.
2. is not : Evaluates to True if the variables on either side do not refer to the same object.

Example:

```
a = [1, 2, 3]

b = a

c = [1, 2, 3]


print(a is b)   # True, because b is the same object as a

print(a is c)   # False, because c is a different object, even though it has the same content

print(a is not c)  # True, because a and c are not the same object
```

Membership Operators

Membership operators are used to test if a sequence (such as a string, list, tuple, etc.) contains a specified item. There are two membership operators:

1. in : Evaluates to True if it finds a variable in the specified sequence.
2. not in : Evaluates to True if it does not find a variable in the specified sequence.

Example:

```
student_list = ['Alice', 'Bob', 'Charlie']

print('Alice' in student_list)    # True, because 'Alice' is in the list

print('David' not in student_list)  # True, because 'David' is not in the list
```

Understanding with Examples

Identity Operators

Imagine you have two identical keys that open the same door, but they are two separate keys. Similarly, two lists with the same items are different objects in memory. However, if you give someone one of your keys and keep the other, and then later swap keys, you both still have the same key to the same door. This is akin to two variables pointing to the same list object.

Membership Operators

Think of a classroom attendance list. If you want to check if a student named "Alice" is present in class, you look through the list of students. If "Alice" is on the list, you know she is present.

Code Examples

Identity Operators

```
# Identity Operator Example
marks1 = [90, 85, 75]
marks2 = marks1  # Both variables point to the same list
marks3 = [90, 85, 75]

print(marks1 is marks2)  # True, because marks2 is the same object as marks1
print(marks1 is marks3)  # False, because marks3 is a different object
print(marks1 is not marks3)  # True, because marks1 and marks3 are not the same object
```

Membership Operators

```
# Membership Operator Example
students = ['Alice', 'Bob', 'Charlie']

print('Alice' in students)   # True, because 'Alice' is in the list
print('David' in students)   # False, because 'David' is not in the list
print('David' not in students)  # True, because 'David' is not in the list
```

Feel free to practice the examples and questions provided to reinforce your understanding of these topics!

MODULE-7

Introduction to Logical AND, OR, NOT

What are Logical Operators?

Logical operators are used to combine conditional statements. In Python, there are three logical operators:

- **AND (and)**: Returns True if both statements are true.
- **OR (or)**: Returns True if at least one of the statements is true.
- **NOT (not)**: Returns True if the statement is false (inverts the result).

Logical operators are fundamental in decision-making in programming, helping you control the flow of your program based on multiple conditions.

Understanding with Examples

Let's understand these operators with simple, everyday examples:

AND (and)

Imagine you are checking if you can go out to play. You need to finish your homework **and** clean your room. Both conditions need to be met.

```
finished_homework = True

cleaned_room = False


can_go_out = finished_homework and cleaned_room

print(can_go_out)  # This will print False because both conditions are not True
```

OR (or)

Now, imagine you are deciding whether to watch TV. You can watch TV if you finish your homework **or** it's a weekend.

```
finished_homework = False

is_weekend = True


can_watch_tv = finished_homework or is_weekend

print(can_watch_tv)  # This will print True because at least one condition is True
```

NOT (not)

Finally, think about a situation where you check if you should not go out because you didn't finish your homework.

```
finished_homework = False


should_not_go_out = not finished_homework

print(should_not_go_out)  # This will print True because finished_homework is False
```

Code Examples

Let's explore logical operators through some code examples:

Example 1: Logical AND (and)

Check if a student has both homework and study materials.

```
has_homework = True

has_study_materials = True


has_both = has_homework and has_study_materials

print(has_both)  # This will print True because both conditions are True
```

Example 2: Logical OR (or)

Check if a student has either completed their homework or has a test tomorrow.

```
completed_homework = False

has_test_tomorrow = True


either_or = completed_homework or has_test_tomorrow

print(either_or)  # This will print True because one of the conditions is True
```

Example 3: Logical NOT (not)

Check if a student does not have homework.

```
has_homework = False


no_homework = not has_homework

print(no_homework)  # This will print True because has_homework is False
```

Practice Question

Question: Imagine a student with variables has_homework and has_study_materials set to True or False. Use logical operators to check if the student has both homework and study materials. Print the result of this logical operation.

```
has_homework = True

has_study_materials = True
```

has_both = has_homework and has_study_materials

print(has_both) # This will print True if both variables are True

Test Question

Question: Write a program to check if a student has either completed their homework or has a test tomorrow. Print the result.

completed_homework = False

has_test_tomorrow = True

either_or = completed_homework or has_test_tomorrow

print(either_or) # This will print True if either of the conditions is True

Feel free to practice the examples and questions provided to reinforce your understanding of these topics!

MODULE-8

Introduction to If Statements, Python Indentation, Elif Statements, Else Statements

In Python, control flow is managed using conditional statements that allow you to execute certain blocks of code based on conditions. The most common conditional statements are if, elif, and else.

- **If Statements**: Used to test a condition. If the condition evaluates to True, the code block inside the if statement is executed.
- **Elif Statements**: Short for "else if," it allows you to check multiple conditions. If the if condition is False, the elif condition is checked.
- **Else Statements**: Used to execute a block of code if none of the previous conditions are true.
- **Python Indentation**: Indentation refers to the spaces at the beginning of a code line. Python uses indentation to define the scope of loops, functions, and conditionals. Proper indentation is crucial for the code to run correctly.

Understanding with Examples

If Statements

Imagine you want to check if the weather is sunny. If it is sunny, you'll go for a walk.

```
weather = "sunny"


if weather == "sunny":
    print("Let's go for a walk!")
```

Elif Statements

What if the weather can be sunny, rainy, or cloudy? You can use elif to check multiple conditions.

```
weather = "cloudy"


if weather == "sunny":
    print("Let's go for a walk!")
elif weather == "rainy":
    print("Let's stay inside.")
elif weather == "cloudy":
    print("Let's take an umbrella just in case.")
```

Else Statements

If the weather is neither sunny, rainy, nor cloudy, you can use else to cover any other possibilities.

```
weather = "snowy"


if weather == "sunny":
    print("Let's go for a walk!")
elif weather == "rainy":
    print("Let's stay inside.")
elif weather == "cloudy":
    print("Let's take an umbrella just in case.")
else:
    print("Weather condition unknown. Stay prepared!")
```

Code Examples

If Statements

Check if a number is positive.

```
number = 5


if number > 0:
    print("The number is positive.")
```

Elif Statements

Check if a number is positive, negative, or zero.

```
number = -3


if number > 0:
    print("The number is positive.")
elif number < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

Else Statements

Check if a number is even or odd.

```
number = 4


if number % 2 == 0:
    print("The number is even.")
else:
    print("The number is odd.")
```

Practice Question

Problem: Write a program that takes a person's age and checks if the person is eligible to vote or not.

Solution:

```
age = int(input("Enter your age: "))
```

```
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

Test Question

Problem: Write a program to check the grade of a student based on their marks (e.g., A, B, C, etc.).

Solution:

```
marks = int(input("Enter your marks: "))


if marks >= 90:
    print("Your grade is A.")
elif marks >= 80:
    print("Your grade is B.")
elif marks >= 70:
    print("Your grade is C.")
elif marks >= 60:
    print("Your grade is D.")
else:
    print("Your grade is F.")
```

MODULE-9

Loops in Python: For Loops, While Loops, Break, and Continue Statements

For Loops

A for loop helps us go through each item in a sequence (like a list, tuple, string, or range) and perform actions with those items.

Example: Imagine you want to greet each friend in a list of friends.

python

Copy code

```
friends = ["Alice", "Bob", "Charlie"]


for friend in friends:
    print(f"Hello, {friend}!")
```

Explanation: This code goes through each name in the friends list and prints a greeting for each one.

While Loops

A while loop keeps running as long as a certain condition is true. It's useful when we don't know beforehand how many times we need to repeat something.

Example: Suppose you want to count down from 5 to 1.

python

Copy code

```
count = 5


while count > 0:
    print(count)
    count -= 1
```

Explanation: This code starts with count at 5 and keeps subtracting 1 until count is no longer greater than 0.

Break Statement

The break statement stops a loop before it has gone through all the items. We use it when we want to exit a loop early based on some condition.

Example: You want to find a specific number in a list and stop searching once you find it.

python

Copy code

```
numbers = [1, 2, 3, 4, 5]


for number in numbers:
    if number == 3:
        print("Found the number 3!")
        break
```

Explanation: This code goes through the numbers list and stops as soon as it finds the number 3.

Continue Statement

The continue statement skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

Example: You want to print all numbers from 1 to 5, but skip printing the number 3.

python

Copy code

```
for number in range(1, 6):
    if number == 3:
        continue
    print(number)
```

Explanation: This code prints numbers from 1 to 5, but skips 3. When it reaches 3, it skips the print statement and continues with the next number.

More Code Examples

For Loops

Print each letter in the word "Python".

python

Copy code

```
word = "Python"


for letter in word:
    print(letter)
```

Explanation: This code goes through each letter in the word "Python" and prints it.

While Loops

Print numbers from 1 to 5.

python

Copy code

```
number = 1

while number <= 5:
    print(number)
    number += 1
```

Explanation: This code starts with number at 1 and keeps adding 1 until number is greater than 5.

Break Statement

Stop the loop when the number 4 is encountered.

python

Copy code

```
for number in range(1, 10):
    if number == 4:
        break
    print(number)
```

Explanation: This code prints numbers from 1 to 3 and stops when it reaches 4.

Continue Statement

Skip the number 2 in a range of 1 to 5.

python

Copy code

```
for number in range(1, 6):
    if number == 2:
        continue
    print(number)
```

Explanation: This code prints numbers from 1 to 5 but skips 2.

Feel free to try these examples to get a better understanding of loops, break, and continue statements in Python!

MODULE-10

Introduction to Sets and Sets Methods

Sets

A set is an unordered collection of unique elements. In Python, sets are defined using curly braces `{}` or the `set()` function. Sets are useful when you want to store unique items and perform mathematical set operations like union, intersection, difference, and symmetric difference.

Set Methods

Sets come with various methods to perform different operations:

- **add(item)**: Adds an item to the set.
- **remove(item)**: Removes an item from the set. Raises an error if the item does not exist.
- **discard(item)**: Removes an item from the set. Does not raise an error if the item does not exist.
- **clear()**: Removes all items from the set.
- **union(other_set)**: Returns a new set with all elements from both sets.
- **intersection(other_set)**: Returns a new set with only the elements common to both sets.
- **difference(other_set)**: Returns a new set with elements in the current set but not in the other set.
- **symmetric_difference(other_set)**: Returns a new set with elements in either of the sets but not in both.

Understanding with Examples

Sets

Imagine you have a collection of unique favorite fruits.

```
favorite_fruits = {"apple", "banana", "cherry"}
```

```
print(favorite_fruits)
```

Set Methods

1. *Adding an Item*

```
favorite_fruits.add("orange")
print(favorite_fruits)
```

2. *Removing an Item*

```
favorite_fruits.remove("banana")
print(favorite_fruits)
```

3. *Union of Sets*

Suppose you and your friend have different favorite fruits.

```
my_fruits = {"apple", "banana", "cherry"}
friend_fruits = {"cherry", "orange", "grape"}

all_fruits = my_fruits.union(friend_fruits)
print(all_fruits)
```

4. *Intersection of Sets*

Find common fruits between you and your friend.

```
common_fruits = my_fruits.intersection(friend_fruits)
print(common_fruits)
```

Code Examples

Creating and Modifying Sets

```
# Creating a set
colors = {"red", "green", "blue"}
print(colors)

# Adding an element to the set
colors.add("yellow")
print(colors)

# Removing an element from the set
colors.remove("green")
print(colors)

# Discarding an element from the set (does not raise an error if the element
is not found)
colors.discard("purple")
print(colors)
```

Set Operations

```
# Union of two sets
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
print(union_set)
```

```
# Intersection of two sets
intersection_set = set1.intersection(set2)
print(intersection_set)

# Difference of two sets
difference_set = set1.difference(set2)
print(difference_set)

# Symmetric difference of two sets
symmetric_difference_set = set1.symmetric_difference(set2)
print(symmetric_difference_set)
```

Feel free to practice the examples and questions provided to reinforce your understanding of these topics!

MODULE-11

Python Dictionary

1. Introduction to Dictionary

A dictionary in Python is an unordered collection of items. Each item is a key-value pair, where each key is unique and is used to access the corresponding value. Dictionaries are incredibly useful when you need to associate pieces of data (values) with unique identifiers (keys).

Key Features:

- **Keys**: Must be unique and immutable (e.g., strings, numbers, tuples). Keys act as indexes in the dictionary, similar to how indices work in lists, but instead of being numeric and ordered, they are often strings and unordered.
- **Values**: Can be of any data type and can be duplicated. This flexibility allows dictionaries to store complex data structures.
- **Unordered**: The items do not have a defined order. This means that the order in which items are inserted does not guarantee the order of items when you iterate over them.
- **Mutable**: Dictionaries can be changed after creation. You can add, remove, or modify items.

In Other words,

A dictionary is a powerful data structure that stores data as **key-value pairs**. Think of it as a container where each item has two parts:

- **Key:** This acts like a unique identifier, like a name or a code, to access the information associated with it. It must be **immutable**, meaning it cannot be changed once assigned. Examples of valid keys include strings, numbers, and tuples.
- **Value:** This is the actual data you want to store, and it can be of any data type, including numbers, strings, lists, or even other dictionaries!

This organization allows you to access a specific value by simply using its corresponding key, making dictionaries extremely efficient for storing and retrieving data based on unique labels.

2. Understanding with Examples

Let's break down dictionaries using simple, everyday examples:

Example 1: Dictionary as a Contact Book

Imagine a contact book where you want to store names and their corresponding phone numbers. Here, the names are the keys, and the phone numbers are the values.

```
contact_book = {
   "Alice": "123-456-7890",
   "Bob": "987-654-3210",
   "Charlie": "555-666-7777"
}
print(contact_book["Alice"])  # Output: 123-456-7890
```

In this example:

- "Alice", "Bob", and "Charlie" are keys.
- "123-456-7890", "987-654-3210", and "555-666-7777" are values.
- You can retrieve Alice's phone number using contact_book["Alice"].

This setup is very similar to how you might organize a physical address book, where each contact (key) has a unique set of details (value).

Example 2: Dictionary as a Student Gradebook

Consider a grade book where students' names are keys, and their grades are values.

```
grades = {
   "John": "A",
   "Emma": "B+",
   "Sophie": "A-",
   "Max": "B"
}
print(grades["Emma"])  # Output: B+
```

In this example:

- "John", "Emma", "Sophie", and "Max" are keys.
- "A", "B+", "A-", and "B" are values.
- You can retrieve Emma's grade using grades["Emma"].

Just like in the contact book example, each student has a unique identifier (their name) associated with their grade.

Example 3:Shopping List:

When you're going grocery shopping, you can use a dictionary to keep track of the items you need:

  - **Key:** The name of the item (e.g., "Milk", "Eggs", "Bread")
  - **Value:** The quantity you need (e.g., "1 gallon", "1 dozen", "2 loaves")

```
shopping_list =
{
  "Milk": "1 Litre",
  "Eggs": "1 dozen",
  "Bread": "2 loaves"
}
```

You can easily check how many loaves of bread you need by looking up "Bread" as the key.

3. Working with Dictionaries

Creating a Dictionary

You can create an empty dictionary using {} or dict().

```
empty_dict = {}

another_empty_dict = dict()
```

You can also create a dictionary with predefined items.

```
contact_book = {
    "Alice": "123-456-7890",
    "Bob": "987-654-3210"
}
```

Adding and Modifying Items

You can add a new key-value pair or modify an existing one.

```
# Adding new key-value pairs
contact_book["David"] = "444-555-6666"

# Modifying an existing value
contact_book["Alice"] = "111-222-3333"
```

When you add a new key-value pair, if the key already exists, the value is updated. If the key does not exist, a new entry is created.

Removing Items

You can remove items using del, pop(), or popitem().

```
# Using del
del contact_book["Bob"]

# Using pop()
removed_number = contact_book.pop("Charlie")

# Using popitem() (removes the last inserted item)
last_item = contact_book.popitem()
```

- del removes the specified key-value pair.
- pop() removes the specified key-value pair and returns the value.
- popitem() removes and returns the last key-value pair added (Python 3.7+).

Accessing Keys, Values, and Items

You can access all keys, values, or items (key-value pairs) using keys(), values(), and items() respectively.

```
keys = contact_book.keys()
values = contact_book.values()
items = contact_book.items()
```

These methods return view objects that reflect the current state of the dictionary.

Iterating Over a Dictionary

You can iterate over the keys, values, or items in a dictionary.

```
# Iterating over keys
for key in contact_book:
    print(key, contact_book[key])

# Iterating over values
```

```
for value in contact_book.values():
    print(value)


# Iterating over items
for key, value in contact_book.items():
    print(key, value)
```

MODULE-13

Python Functions

1. Introduction to Functions

A function in Python is a reusable block of code that performs a specific task. Functions help to organize code, reduce repetition, and improve readability. They allow you to break down complex problems into smaller, manageable parts.

Key Features:

- **Reusability**: Functions allow you to write a piece of code once and reuse it multiple times throughout your program.
- **Modularity**: Breaking code into functions helps in organizing and maintaining the codebase.
- **Abstraction**: Functions allow you to abstract complex operations, making the code easier to understand and use.
- **Parameters**: Functions can accept inputs, known as parameters or arguments, to customize their behavior.
- **Return Values**: Functions can return a value after execution, which can be used further in the code.

In other words, functions are like mini-programs within your program. They take inputs, process them, and produce outputs.

2. Understanding with Examples

Example 1: Simple Function to Add Two Numbers

Imagine you want to add two numbers repeatedly in different parts of your program. Instead of writing the addition code each time, you can create a function.

```
def add_numbers(a, b):
    return a + b


result = add_numbers(5, 3)
print(result)  # Output: 8
```

In this example:

- def add_numbers(a, b): defines a function named add_numbers that takes two parameters a and b.
- return a + b returns the sum of a and b.
- result = add_numbers(5, 3) calls the function with arguments 5 and 3, and stores the result in result.

Example 2: Function to Convert Temperatures

Consider a function to convert temperatures from Celsius to Fahrenheit.

```
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32


temp_f = celsius_to_fahrenheit(25)
print(temp_f)  # Output: 77.0
```

In this example:

- def celsius_to_fahrenheit(celsius): defines a function named celsius_to_fahrenheit that takes one parameter celsius.
- return (celsius * 9/5) + 32 converts the Celsius temperature to Fahrenheit.

- temp_f = celsius_to_fahrenheit(25) calls the function with 25 degrees Celsius and stores the result in temp_f.

Example 3: Function as a Greeting Message Generator

Think about a function that generates a greeting message for a user.

```
def greet_user(name):
    return f"Hello, {name}!"


greeting = greet_user("Alice")
print(greeting)  # Output: Hello, Alice!
```

In this example:

- def greet_user(name): defines a function named greet_user that takes one parameter name.
- return f"Hello, {name}!" returns a greeting message.
- greeting = greet_user("Alice") calls the function with the argument "Alice" and stores the result in greeting.

3. Working with Functions

Defining a Function

To define a function in Python, use the def keyword, followed by the function name, parentheses (with optional parameters), and a colon. The function body is indented.

```
def function_name(parameters):
    # function body
    return value  # optional
```

Calling a Function

To call a function, use the function name followed by parentheses containing any required arguments.

```
result = function_name(arguments)
```

Function Parameters

Functions can take multiple parameters. You can specify default values for parameters, making them optional.

```
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

print(greet("Alice"))  # Output: Hello, Alice!
print(greet("Bob", "Hi"))  # Output: Hi, Bob!
```

In this example:

- greet(name, greeting="Hello") defines a function with a default parameter greeting.

Returning Values

Functions can return values using the return statement. If no return statement is used, the function returns None by default.

```
def square(number):
    return number * number

result = square(4)
print(result)  # Output: 16
```

Scope and Lifetime of Variables

Variables defined within a function are local to that function and cannot be accessed outside of it. These variables exist only for the duration of the function execution.

```
def local_scope_example():
    local_var = "I am local"
    print(local_var)

local_scope_example()
# print(local_var)  # This will cause an error because local_var is not defined outside the function.
```

MODULE-14

1. Introduction to Lambda Functions

Lambda functions, also known as anonymous functions, are small, unnamed functions defined using the lambda keyword. They are designed for situations where you need a simple function for a short period of time. Unlike regular functions defined with the def keyword, lambda functions are typically used for short, throwaway tasks.

Key Features:

- **Anonymous**: Lambda functions do not have a name.
- **Single Expression**: They are limited to a single expression, unlike regular functions that can have multiple statements.
- **Compact Syntax**: Lambda functions are defined in a single line of code.
- **Use Cases**: Often used in situations requiring a small function for a short duration, such as in functional programming, with functions like map(), filter(), and reduce().

In other words, lambda functions are like small, disposable tools that you can quickly create and use without the need to define a full-fledged function.

2. Understanding with Examples

Example 1: Lambda Function for Basic Arithmetic

Imagine you need a simple function to add two numbers, but you don't want to define a full function.

```
add = lambda x, y: x + y

print(add(3, 5))  # Output: 8
```

In this example:

- lambda x, y: x + y defines a lambda function that takes two parameters x and y and returns their sum.
- add(3, 5) calls the lambda function with arguments 3 and 5.

Example 2: Lambda Function in Sorting

Consider a list of tuples representing students and their scores. You want to sort this list based on the scores using a lambda function.

```
students = [("John", 85), ("Emma", 92), ("Sophie", 78), ("Max", 89)]

sorted_students = sorted(students, key=lambda student: student[1])

print(sorted_students)  # Output: [('Sophie', 78), ('John', 85), ('Max', 89), ('Emma', 92)]
```

In this example:

- lambda student: student[1] defines a lambda function that takes a tuple student and returns the second element (the score).
- sorted(students, key=lambda student: student[1]) sorts the list based on the scores.

Example 3: Lambda Function in Filtering

Imagine you have a list of numbers, and you want to filter out the even numbers using a lambda function.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even_numbers)  # Output: [2, 4, 6, 8, 10]
```

In this example:

- lambda x: x % 2 == 0 defines a lambda function that returns True if x is even.
- filter(lambda x: x % 2 == 0, numbers) filters the list to include only even numbers.

3. Working with Lambda Functions

Defining a Lambda Function

A lambda function is defined using the lambda keyword, followed by the parameters, a colon, and the expression.

```
lambda parameters: expression
```

MODULE-15

Python File Handling

1. Introduction to File Handling

File handling is an essential part of any programming language, enabling you to store and retrieve data from files on your computer. Python provides built-in functions and methods to create, read, write, and manipulate files. Understanding how to work with files is crucial for tasks such as data storage, configuration management, logging, and data analysis.

Key Concepts:

- **File Operations**: Python supports various file operations such as opening a file, reading from it, writing to it, and closing it.
- **File Modes**: Files can be opened in different modes, including read, write, append, and binary modes.
- **Context Management**: Using the with statement ensures that files are properly closed after their suite finishes, even if an exception is raised.

2. Opening and Closing Files

To work with files, you first need to open them using the open() function, which returns a file object. After performing the necessary operations, you should close the file to free up system resources.

```
# Opening a file

file = open('example.txt', 'r')


# Closing a file

file.close()
```

File Modes

- 'r': Read mode (default). Opens the file for reading.
- 'w': Write mode. Opens the file for writing (creates a new file or truncates an existing file).
- 'a': Append mode. Opens the file for appending new data to the end.
- 'b': Binary mode. Used with other modes to handle binary files (e.g., 'rb', 'wb').

3. Reading from a File

You can read the contents of a file using various methods, such as read(), readline(), and readlines().

read()

Reads the entire content of the file as a string.

```
file = open('example.txt', 'r')

content = file.read()

print(content)

file.close()
```

readline()

Reads a single line from the file.

```
file = open('example.txt', 'r')

line = file.readline()

print(line)

file.close()
```

readlines()

Reads all lines and returns them as a list of strings.

```
file = open('example.txt', 'r')

lines = file.readlines()

print(lines)

file.close()
```

4. Writing to a File

You can write data to a file using the write() and writelines() methods.

write()

Writes a string to the file.

```
file = open('example.txt', 'w')

file.write('Hello, World!')

file.close()
```

writelines()

Writes a list of strings to the file.

```
lines = ['First line\n', 'Second line\n', 'Third line\n']

file = open('example.txt', 'w')

file.writelines(lines)

file.close()
```

5. Using with Statement

Using the with statement for file operations ensures that the file is properly closed after its suite finishes, even if an exception is raised. This is considered a best practice for file handling in Python.

```
with open('example.txt', 'r') as file:

    content = file.read()

    print(content)

# File is automatically closed here
```

6. Appending to a File

To add content to the end of an existing file, open the file in append mode ('a').

```
with open('example.txt', 'a') as file:

    file.write('\nAppending a new line.')
```

7. Working with Binary Files

Binary files, such as images or executables, require binary mode for reading and writing.

Reading Binary Files

```
with open('image.jpg', 'rb') as file:
    binary_data = file.read()
```

Writing Binary Files

```
with open('output.jpg', 'wb') as file:
    file.write(binary_data)
```

8. File Methods and Attributes

Python provides several methods and attributes to interact with file objects.

tell()

Returns the current file position.

```
with open('example.txt', 'r') as file:
    print(file.tell())
```

seek()

Changes the file position.

```
with open('example.txt', 'r') as file:
    file.seek(5)
    print(file.read())
```

name

Returns the name of the file.

```
file = open('example.txt', 'r')
print(file.name)
file.close()
```

mode

Returns the mode in which the file was opened.

```
file = open('example.txt', 'r')
print(file.mode)
file.close()
```

9. Exception Handling in File Operations

Handling exceptions is crucial to manage errors during file operations gracefully.

```
try:
    with open('non_existent_file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist.")
except Exception as e:
    print(f"An error occurred: {e}")
```

10. Practical Examples

Example 1: Reading a Text File

```
with open('data.txt', 'r') as file:
    for line in file:
        print(line.strip())
```

Example 2: Writing to a Text File

```
data = ['Line 1', 'Line 2', 'Line 3']
with open('output.txt', 'w') as file:
    for line in data:
        file.write(line + '\n')
```

Example 3: Copying a Binary File

```
with open('source_image.jpg', 'rb') as src_file:
    with open('destination_image.jpg', 'wb') as dest_file:
        dest_file.write(src_file.read())
```

MODULE-16

Try, Except Block

Introduction to Try, Except Block

The try and except blocks in Python are used to handle exceptions, which are errors that occur during the execution of a program. By using try and except, you can prevent your program from crashing and handle errors gracefully.

Understanding with Examples

Imagine you're making a smoothie, and you ask your friend to bring bananas. But what if they bring apples instead? You can handle this situation without your smoothie-making process crashing by using try and except.

Code Examples

Here’s a simple example of using try and except to handle division by zero:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Finally Block

Introduction to Finally Block

The finally block is used to execute code regardless of whether an exception is thrown or not. It is typically used for cleanup actions, like closing files or releasing resources.

Understanding with Examples

Think of the finally block as cleaning up the kitchen after making a smoothie, no matter if the smoothie turned out well or you spilled it all over.

Code Examples

Here’s an example with a finally block:

```
try:
    file = open('example.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print("The file was not found.")
finally:
    file.close()
    print("File has been closed.")
```

Raising Exceptions

Introduction to Raising Exceptions

You can raise exceptions in your code to signal that an error has occurred. This is done using the raise statement.

Understanding with Examples

Imagine you are the smoothie master, and you see a rotten banana. You can raise an exception to stop anyone from using that banana.

Code Examples

Here’s how you can raise an exception:

```
age = -1
if age < 0:
    raise ValueError("Age cannot be negative!")
```

Custom Exceptions

Introduction to Custom Exceptions

Custom exceptions allow you to define your own error types for situations specific to your application. This can make your error handling more precise and descriptive.

Understanding with Examples

Imagine your smoothie shop has a rule that only ripe bananas can be used. You can create a custom exception for unripe bananas.

Code Examples

Here’s how you can define and use a custom exception:

```
class UnripeBananaException(Exception):
    pass


def make_smoothie(banana_ripeness):
    if banana_ripeness < 5:
        raise UnripeBananaException("Banana is not ripe enough!")
    return "Smoothie made!"


try:
    make_smoothie(3)
except UnripeBananaException as e:
    print(e)
```

MODULE-17

Classes

Introduction to Classes

A class in Python is a blueprint for creating objects. It defines a set of attributes and methods that the objects created from the class can use. Think of a class as a template for creating objects with similar properties and behaviors.

Understanding with Examples

Imagine you are designing different types of smoothies. A class would be like a recipe that defines what ingredients (attributes) and steps (methods) are needed to make a smoothie.

Objects

Introduction to Objects

An object is an instance of a class. When a class is defined, no memory is allocated until an object of that class is created. Objects are the real-world entities that use the blueprint defined by the class.

Understanding with Examples

Using the smoothie example, an object would be a specific smoothie made using the recipe. Each smoothie made from the recipe can have different ingredients or amounts (attributes).

Constructor

Introduction to Constructor

A constructor is a special method called when an object is instantiated. The __init__ method in Python is the constructor method that initializes the object's attributes.

Understanding with Examples

In the smoothie example, the constructor would be like the initial setup of your blender and ingredients when you start making a smoothie.

Code Examples

Classes, Objects, and Constructors

```
# Defining a class

class Smoothie:

  def __init__(self, fruits, liquid):

    self.fruits = fruits

    self.liquid = liquid
    def blend(self):
        return f"Blending {self.fruits} with {self.liquid}."


# Creating an object
my_smoothie = Smoothie("bananas and strawberries", "milk")


# Using an object
print(my_smoothie.blend())
```

MODULE-18

Introduction to Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called a **subclass** or **derived class**) to inherit attributes and methods from another class (called a **superclass** or **base class**). This promotes code reusability and establishes a natural hierarchy between classes.

Understanding with Examples

Imagine you have a general blueprint for a **Vehicle**. All vehicles have some common features like wheels, engine, and seats. Now, let's say you want to create a more specific blueprint for a **Car** and a **Bike**. Instead of creating these blueprints from scratch, you can inherit the common features from the **Vehicle** blueprint and add specific features like air conditioning for the **Car** and pedals for the **Bike**. This is similar to how inheritance works in programming.

Code Examples

Let's explore inheritance with some Python code.

Example 1: Person and Student

Here, Person is the base class, and Student is the subclass.

python

Copy code

```
# Base class-class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
# Subclass-class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def display_student_info(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Student ID: {self.student_id}")
# Creating an instance of Student
student = Student("Alice", 20, "S12345")
student.display_student_info()
```

Practice Question

Task

Define a base class Person with attributes name and age. Define a subclass Student that adds an attribute student_id and a method display_student_info that prints all attributes. Create an instance of Student and call the method.

Solution

python

Copy code

```
# Base classclass Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
# Subclass-class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def display_student_info(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Student ID: {self.student_id}")
# Creating an instance of Student
student = Student("Bob", 21, "S67890")
student.display_student_info()
```

Test Question

Task

Define a base class Employee with attributes name and salary. Define a subclass Manager that adds an attribute department and a method display_manager_info that prints all attributes. Create an instance of Manager and call the method.

Solution

```
# Base classclass Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
# Subclassclass Manager(Employee):
    def __init__(self, name, salary, department):
        super().__init__(name, salary)
        self.department = department

    def display_manager_info(self):
        print(f"Name: {self.name}")
        print(f"Salary: {self.salary}")
        print(f"Department: {self.department}")
# Creating an instance of Manager
manager = Manager("Charlie", 90000, "IT")
manager.display_manager_info()
```

Single Inheritance: A subclass inherits from a single base class.

```
class Parent:
    pass
class Child(Parent):
    pass
```

Multiple Inheritance: A subclass inherits from multiple base classes.

```
class Father:
    pass
class Mother:
    pass
class Child(Father, Mother):
Pass
```

Multilevel Inheritance: A subclass inherits from another subclass.

```
class Grandparent:
    pass
class Parent(Grandparent):
    pass
class Child(Parent):
    pass
```

Hierarchical Inheritance: Multiple subclasses inherit from a single base class.

```
class Parent:
    pass
class Child1(Parent):
    pass
class Child2(Parent):
    pass
```

Hybrid Inheritance: A combination of two or more types of inheritance.

```
class Base:
    pass
class Derived1(Base):
    pass
class Derived2(Derived1):
    pass
class Derived3(Base):
    pass
```

Understanding inheritance will help you create more structured and maintainable code, promoting code reuse and reducing redundancy.

MODULE-19

Encapsulation and Polymorphism in Python

Introduction to Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class. It also restricts direct access to some of the object's components, which is a means of preventing accidental interference and misuse.

Understanding with Examples

Think of encapsulation as a capsule or a protective shield around your smoothie ingredients. The capsule ensures that the ingredients are not accessed or modified directly but only through specific methods.

Code Examples for Encapsulation

Here’s an example of encapsulation in Python:

```
class Smoothie:
  def __init__(self, ingredients):
    self._ingredients = ingredients # Private attribute

  def get_ingredients(self):
    return self._ingredients

  def add_ingredient(self, ingredient):
    self._ingredients.append(ingredient)

my_smoothie = Smoothie(['banana', 'strawberry'])
print(my_smoothie.get_ingredients())
my_smoothie.add_ingredient('milk')
print(my_smoothie.get_ingredients())
```

Introduction to Polymorphism

Polymorphism is another core concept in OOP. It refers to the ability of different objects to respond, each in its own way, to identical messages (or methods). It allows for methods to be used interchangeably, regardless of the specific type of object.

Understanding with Examples

Think of polymorphism as the ability to use the same action, like 'blend,' on different types of ingredients, whether it's fruits, vegetables, or ice.

Code Examples for Polymorphism

Here's an example of polymorphism in Python:

```
class Fruit:

  def blend(self):

    return "Blending a fruit!"


class Banana(Fruit):

  def blend(self):

    return "Blending a banana!"


class Strawberry(Fruit):

  def blend(self):

    return "Blending a strawberry!"


def make_smoothie(fruit):

  return fruit.blend()


banana = Banana()

strawberry = Strawberry()

print(make_smoothie(banana))
print(make_smoothie(strawberry))
```

Practice Exercise

Define a class `Employee` with private attributes `_name` and `_salary`. Provide methods `get_name`, `set_name`, `get_salary`, and `set_salary` to access and update the attributes. Create an instance of the class and test the methods.

```
class Employee:

  def __init__(self, name, salary):

    self._name = name

    self._salary = salary


  def get_name(self):

    return self._name


  def set_name(self, name):

    self._name = name


  def get_salary(self):

    return self._salary


  def set_salary(self, salary):

    self._salary = salary


employee = Employee("John Doe", 50000)

print(employee.get_name())

print(employee.get_salary())

employee.set_name("Jane Doe")
employee.set_salary(60000)

print(employee.get_name())

print(employee.get_salary())
```

Test Exercise

Define a class `Course` with private attributes `_course_name` and `_students` (a list of student names). Provide methods to add a student, remove a student, and print the list of students. Create an instance of the class and test the methods.

```
class Course:

  def __init__(self, course_name):

    self._course_name = course_name

    self._students = []


  def add_student(self, student_name):

    self._students.append(student_name)


  def remove_student(self, student_name):

    self._students.remove(student_name)


  def print_students(self):

    print(f"Students in {self._course_name}: {', '.join(self._students)}")


course = Course("Python Programming")

course.add_student("Alice")

course.add_student("Bob")

course.print_students()
course.remove_student("Alice")

course.print_students()
```