

UNIT 2

Python Programming

Student Learning Outcomes

By the end of this chapter, students will be able to:

- Understand basic programming concepts and set up a Python development environment.
- Write and interpret basic Python syntax and structure, including variables, data types, and input/output operations.
- Use various operators and expressions in Python, including arithmetic, comparison, and logical operators.
- Implement control structures such as decision-making statements and loops in Python.
- Work with Python modules, functions, and built-in data structures like lists.
- Apply modular programming techniques and object-oriented programming concepts in Python.
- Handle exceptions, perform file operations, and apply testing and debugging techniques in Python.

Introduction

Python is a popular and easy to learn programming language. In this unit, you will learn the basics, setup tools and explore key components. Later, we will learn advanced topics like file handling, debugging and data structure.

2.1 Introduction to Python Programming

Python is a versatile and applicable to various fields, including web development, data analysis, artificial intelligence, and more. Python's straightforward syntax and clear structure make it an excellent choice for beginners, allowing them to focus on learning programming concepts rather than dealing with complex syntax rules.

2.1.1 Understanding Basic Programming Concepts

Computer programming is the process of creating a set of instructions that tell a computer how to perform a task. These instructions are written in a programming language that the computer can understand and execute.

2.1.1.1 Programming Basics

Computer programming involves the following basic steps to write a program.

1. **Write Code:** Create a set of instructions in a programming language.

2. **Compile/Interpret:** Translate the code into a form that the computer can understand.
3. **Execute:** Run the code to perform the task.
4. **Output:** Display the results or perform actions based on the code.

2.1.1.2 Setting Up Python Development Environment

The development environment refers to the process of preparing a computer to write, run, and debug Python code effectively. This involves installing and configuring the necessary software, tools, and libraries. We can download and install Python from <https://www.python.org/>. When starting with Python programming, choosing a good Integrated Development Environment (IDE) can help make coding easier.

Tidbits

When installing Python, make sure to check the box that says "Add Python to PATH." This makes it easier to run Python from the command line. We can also use online services to write and run Python program.

2.2 Basic Python Syntax and Structure

The following Python program demonstrates the simplicity and readability of the language:

```
print("This is my first page")
```

In this example, the print function is utilized to output the message enclosed in double quotation marks. This illustrates Python's straightforward syntax, where function like print is used to perform actions such as displaying text.

Python Comments

Lines that are not executed by the Python interpreter. They are used to provide explanations or notes for the code. Single-line comments start with the # symbol while multi-line comments can be created using triple quotes (""") at the beginning and the end as shown below.

```
# This is a single-line comment
print("K2 is the second-highest mountain in the world")
'''
This is a multi-line comment.
It can span multiple lines.
'''
print("Edhi Foundation is the largest volunteer ambulance network.")
```

2.2.1 Variables, Data Types and Input / Output

2.2.1.1 Variable

A variable is a storage container in a computer's memory, that allows storage, retrieval and manipulation of data. The value of a variable can change throughout the execution of

a program:

```
age = 71
print( "Ahmad lived for", age, "years")
age = 60
print( "Iqbal lived for", age, "years")
```

2.2.1.2 Variable Naming Rules in Python

Variable names in Python must adhere to the following rules:

- The name must begin with a letter (a-z, A-Z) or an underscore (_).
- Subsequent characters can include letters, digits (0-9), or underscores (_).
- Variable names are case-sensitive, meaning age and Age are considered two different variables.
- Python's reserved keywords, such as for, while, if, etc., cannot be used as variable names.

Tidbits

Always use meaningful names for variables to make your code easier to understand. For example, use age instead of a.

2.2.1.3 Creating Different Types of Variables

In Python, you can create variables of different types to store various kinds of data. Here are some common types of variables:

- **Integer (int):** Stores whole numbers. Example: age = 17
- **Floating-point (float):** Stores decimal numbers. Example: price = 19.99
- **String (str):** Stores text. Example: name = "Ali"
- **Boolean (bool):** Stores True or False. Example: is_student = True

Tidbits

- It's a good practice to use lowercase letters for variable names and underscores to separate words in variable names (e.g., student_name).

2.2.1.4 Input and Output Operations

Input and output operations allow you to interact with the user. You can ask the user to enter data (input) and display information to the user (output).

- **Input:** Use the input () function to get user input. The input () function displays a message on the screen and waits for the user to type something and press Enter. The text entered by the user is then stored in a variable. For example:

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

- **Output:** Use the print () function to display information on the screen. The print () function takes one or more arguments and displays them. For example:

```
print("Hello, " + name + "!")
```

2.2.1.5 Handling Integer and Float Inputs

To handle numeric inputs, you typically use the `int()` or `float()` functions to convert input strings to integers or floating-point numbers, respectively.

Integer Inputs

```
# Example : Handling integer input
```

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

```
print("Your age is:", user_age)
```

Float Inputs

```
# Example: Handling float input
```

```
user_height = float(input("Enter your height in meters: "))
```

```
print("Your height is", user_height, "meter")
```

2.3 Operators and Expressions

Operators are symbols that perform operations on variables and values. An expression is a combination of variables, operators, and values that produces a result.

2.3.1 Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations such as addition, subtraction, multiplication, division, modulus, exponentiation, and floor division as shown in the following code.

```
# Define variables
```

```
a= 10, b= 3
```

```
# Perform all arithmetic operations
```

```
print(a, "+", b, "=", a + b) # Output: 10+3=13
```

```
print(a, "*", b, "=", a * b) # Output: 10 * 3 = 30
```

```
print(a, "/", b, "=", a / b)
```

```
# Output: 10 / 3 = 3.3333333333333335
```

```
print(a, "//", b, "=", a // b)
```

```
# Output: 10 // 3=3
```

```
print(a, "%", b, "=", a % b)
```

```
# Output: 10 % 3 = 1
```

```
print(a, "**", b, "=", a ** b)
```

```
# Output: 10**3= 1000
```



A tutorial on Python is available at
<https://docs.python.org/3/tutorial/>

2.3.2 Comparison Operators

Comparison operators are used to compare two values or expressions. They determine the relational logic between them, such as equality, inequality, greater than, less than, and so on. These operators return a boolean value (True or False) based on the comparison result.

```

# Define variables
x, y = 10, 5
# Greater than
print (x, ">", y, "=", x > y) # Output: 10 > 5 = True
# Less than
print (x, "<", y, "=", x < y) # Output: 10 < 5 = False
# Equal to
print (x, "==", y, "=", x == y) # Output: 10 == 5 = False
# Not Equal to
print (x, "!=", y, "=", x != y) # Output : 10 != 5 = True
# Greater than or equal to
print (x, ">=", y, "=", x >= y) # Output: 10 >= 5 = True
# Less than
print (x, "<=", y, "=", x <= y) # Output: 10 <= 5 = False

```

2.3.3 Assignment Operators

Assignment operators are used to assign values to variables. The most common assignment operator is the equal sign (=), which assigns the value on the right to the variable on the left. There are also compound assignment operators like +=, -=, *=, and /=, which combine arithmetic operations with assignment.

```

# Define initial values
a = 10
b = 5
# Assignment
assignment = a; print ("a =", assignment) # Output: a = 10
# Addition assignment
a += b; print ("a after addition =", a) # Output: a = 15
# Subtraction assignment
a -= b; print ("a after subtraction =", a) # Output: a = 5
# Multiplication assignment
a *= b; print ("a after multiplication =", a) # Output: a = 50
# Division assignment
a /= b; print ("a after division =", a) # Output: a = 2.0
# Modulus assignment
a %= b; print ("a after modulus division =", a) # Output: a = 2
# Exponentiation assignment
a **= b; print ("a after Exponentiation =", a) # Output: a = 100000

```

2.3.4 Logical Operators

Logical operators are used to combine multiple conditions or expressions in a program. The most common logical operators are *and*, *or* and *not*. They are used to perform

logical operations and return boolean values based on the evaluation of the expressions involved.

```
# Define variables
x = True
y = False
# Logical AND
logical_and = x and y
print(x, "and", y, "=", logical_and) # Output: True and False = False
# Logical OR
logical_or = x or y
print(x, "or", y, "=", logical_or) # Output: True and False = True
# Logical NOT
logical_not_x = not x
print(x, "not", x, "=", logical_not) # Output: Not x = False
```

2.3.5 Expressions

An expression is a combination of variables, operators, and values that produces a result. For example, $3 + 4$ is an expression that results in 7. More complex expressions can use parentheses $()$ to control the order of operations. For example:

```
result = (3 + 4) * 2    # result is 14
```

Class Activity

Write a program to calculate Body Mass Index (BMI). Ask the user for their weight and height, then compute and display their BMI and classification. The Body Mass Index (BMI) is calculated using the formula given below.

$$\text{BMI} = \frac{\text{weight}}{\text{height}}$$

where:

- weight is in kilograms (kg)
- height is in meters (m)

2.3.6 Operator Precedence in Python

Operator precedence determines the order in which operations are performed in an expression. In Python as well as in Mathematics, certain operators have higher precedence and are evaluated before others.

- **Parentheses $()$:** Highest precedence. Operations inside parentheses are performed first. $(3 + 2) * 4$ evaluates to 20.
- **Exponentiation:** Performs power operations next. 2^3 evaluates to 8.

- **Multiplication '*', Division '/', and Modulus '%':** These operations come next. $4*3$ evaluates to 12, $10/2$ evaluates to 5.0 and $11\%3$ evaluates 2.
- **Addition '+' and Subtraction '-':** These have lower precedence compared to multiplication and division.
 $5 + 2$ evaluates to 7, and $10-4$ evaluates to 6.

Class Activity

Compute the following expressions and compare results with your class fellows and class teacher.

1. $10 + 3*2**2-5/5$
2. $(10 + 3)*(2**(2 - 1))/5$

DO YOU KNOW?



Using parentheses can help clarify complex expressions and ensure the operations are performed in the desired order.

2.4 Control Structures

In programming, we often need to control the flow of our program based on different conditions or repeat certain actions multiple times. There are two main types of control structures, Decision Making and Looping:

2.4.1 Decision Making

Decision making in programming allows the program to choose different actions based on conditions. Python provides a variety of conditional statements to implement decision making.

2.4.1.1 If Statement

The if statement allows us to make decisions based on conditions. If the condition is true, it runs a block of code.

Syntax of if statement if condition:

if condition:

code to run if the condition is true

Example: If the temperature is above 30 degrees, we print a message.

```
temperature = 35
if temperature > 30:
    print("It is a hot day")
```

2.4.1.2 If-else Statement

The if-else statement allows us to execute one block of code if a condition is true and another block if the condition is false.

Syntax of if-else statement if condition:

if condition:

code to run if the condition is true else :

else:

code to run if the condition is false

Example:

```
temperature = 15
if temperature > 30:
    print("It's a hot day")
else:
    print("It's not a hot day")
```

2.4.1.3 Short Hand if-else Statement

Python also allows a short-hand if-else statement that can be written in a single line.

Syntax of short hand if-else statement
action_if_true if condition else action_if_false

```
temperature = 15
m = "It's a hot day" if (temperature > 30)
else "It's not a hot day"
print(m)
```

Class Activity

Write an if-else statement and a short-hand if-else statement to check if a number is even or odd and print the appropriate message.

2.4.1.3 if-elif-else Statement

The if-elif-else statement allows us to check multiple conditions and execute different blocks of code for each condition.

Syntax of if-elif-else statement
if condition1:
 # code to run if condition1 is true
elif condition2:
 # code to run if condition2 is true
else:
 # code to run if none of the conditions are true

Example:

```
weather = "cloudy" # The output depends on the value stored in the variable weather
if weather == "sunny":
    print("Wear sunglasses")
elif weather == "rainy":
    print("Take an umbrella")
else:
    print("Enjoy your day!")
```

Class Activity

Write an if-elif-else statement to check if a number is positive, negative, or zero.

2.4.2 Looping Constructs

Loops help us repeat actions, making our code more efficient and easier to read. There are two main types of loops in Python: while loops and for loops.

2.4.2.1 while Loop

A while loop runs as long as a condition is true. It checks the condition before each iteration and stops running when the condition is no longer true.

Syntax of while loop while condition:

code to run while the condition is true

Example: Add 1 to a number until it reaches 10.

```
number = 1
while number < 10:
    print(number)
    number += 1
```

Class Activity

Write a Python program that prints even and counts the odd numbers from 1 to 20 using a while loop.

2.4.2.2 for Loop

A for loop repeats a block of code a specific number of times. It is commonly used to iterate over a sequence (like a list, tuple, or string).

Syntax of for loop

for variable in sequence:

code to run for each element in the sequence

Example 1: Say "Hello" to each friend in a list of friends.

```
friends = ["Ahmad", "Ali", "Hassan"]
for friend in friends:
    print("Hello", friend)
```

Explanation: In this example, the code goes through each friend in the list and prints a greeting message for each one.

Class Activity

1. Write a for loop using range() to print the even numbers from 2 to 10.
2. Write a Python program that prints the first 10 multiples of 3 using a for loop and the range() function.

2.5 Python Modules and Built-in Data Structures

Python offers an extensive standard library that includes numerous built-in modules and data structures. A data structure refers to a particular format or method for organizing and storing data. For example, a list is a data structure that we have previously utilized. In this section, we will examine the utilization of functions, modules, and libraries within Python.

2.5.1 Functions and Modules

Functions and modules in Python are key to writing efficient and organized code. Functions allow you to encapsulate reusable blocks of code, while modules help you structure your program by grouping related functions together.

2.5.1.1 Defining and Invoking Functions

Functions are defined using the `def` keyword, followed by the function name and parentheses which may include parameters. The body of the function contains the code to be executed and must be indented.

`def function_name(parameters):`

`# code to be executed`

Example: Define a function to greet a person.

```
def greet(name):  
    print("Hello", name)  
# Function invoking means call the function by name and  
perform the required task For example.  
greet('Ali')
```

2.5.1.2 Function Parameters and Return Values

Functions can take multiple parameters and return values.

Example: Define a function to add two numbers.

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

**DO YOU
KNOW?**



You can call a function multiple times with different arguments to reuse the same code for different inputs.

2.5.1.3 Default Parameters

Functions can have default parameter values, which are used if no argument is provided during the function call.

Example: Define a function with a default parameter.

```
def greet(name = "Student"):  
    return "Hello" + name + "!"  
print(greet()) # Output: Hello, Student!  
print(greet("Umer")) # Output: Hello, Umer!
```

Class Activity

Define a function that takes a list of numbers and returns the maximum value.

2.5.2 Using Libraries and Modules

In Python, libraries and modules are like toolboxes, full of useful tools that help you solve different problems without having to build everything from scratch. In this section, we will explain how to import and use both standard and third-party libraries in your Python programs.

2.5.3 Importing and Using Libraries

Libraries are like pre-built toolkits that you can use without having to write all the code yourself.

Example: Import the random library to generate random numbers.

```
import random
# Generate a random number between 1 and 10
number = random.randint(1, 10)
print("The random number is:", number)
```

```
Import datetime
# Get the current date and time
current_time = datetime.datetime.now()
print("Current date and time:", current_time)
```

```
import statistics
# Calculate the mean of a list of numbers
data = [23, 45, 67, 89, 12, 44, 56]
mean_value = statistics.mean(data)
print("The mean value is:", mean_value)
```

2.5.3.1 Package Structure

To manage large projects, you can organize modules into packages. A package is simply a directory containing related modules. For example, if you're building an e-commerce platform, you could create a package named ecommerce with modules like products.py, customers.py, and orders.py.

Example: In ecommerce/products.py:

```
def List_products () :
    return ["Laptop", "Mobile", "Tablet"]
```

In your main Script

```
from ecommerce import products
available_products = products.List_products()
print(available_products)
# Output :
# ['Laptop', 'Mobile', 'Tablet']
```

Explanation: In this case, ecommerce is the package, and products.py is the module. This structure helps you keep your code organized and manageable.

Tidbits

Organizing your modules into packages is like organizing books into sections of a library—it makes finding and maintaining your code much easier.

2.6 Built-in Data Structures

Python provides several built-in data structures that are essential for organizing and manipulating data efficiently. These include lists, tuples, and dictionaries, each offering unique features to handle various types of data and perform common operations.

2.6.1 Lists

In Python, a list is a versatile data structure that can hold a collection of items. You can create, access, and modify lists easily.

2.6.1.1 Creating, Accessing, and Modifying Lists

A list is created by placing items inside square brackets `[]`, separated by commas. Lists can contain items of different types, such as numbers, strings, or even other lists.

Example: Create a list of your favorite fruits.

```
fruits = ["Mango", "Apple", "Banana"]
print(fruits)
# Output: ['Mango', 'Apple', 'Banana']
```

2.6.1.2 Accessing List Items

You can access items in a list by referring to their index, starting from 0. Example: Access and print the second item from the list of fruits.

```
fruits = ["Mango", "Apple", "Banana"]
print(fruits[1])
# Output: Apple
```

Explanation: The code initializes a list 'fruits' containing 'Mango', 'Apple', and 'Banana', then prints the second item, 'Apple', using the index '1'.

2.6.1.3 Modifying a List

You can modify list items by accessing them via their index and assigning a new value.

Example: Change the first item in the list to "Orange" and add a new fruit "Pineapple".

```
fruits = ["Mango", "Apple", "Banana"]
fruits[0] = "Orange"
fruits.append("Pineapple")
print(fruits)
# Output: ['Orange', 'Apple', 'Banana', 'Pineapple']
```

Explanation: The code modifies the first element of the 'fruits' list to 'Orange', appends 'Pineapple' at the end, and prints the updated list.

2.6.1.4 Methods and Operations on Lists

Python provides several built-in methods to work with lists. Here are a few useful ones:

- `append (item)` - Adds an item to the end of the list.
- `remove (item)` - Removes the first occurrence of an item from the list.
- `sort ()` - Sorts the list in ascending order.
- `reverse ()` - Reverses the order of the list.

Example: Add a new student to the list of students and then sort the list.

```
students = ["Ahmed", "Sara", "Ali"]
students.append("Hina")
students.sort ()
print(students)
# Output : ['Ahmed', 'Ali', 'Hina', 'Sara']
```

Explanation: The code creates a list of students, adds 'Hina' to it, and sorts the list alphabetically.

2.6.1.5 List Operations

Lists also support various operations, such as slicing and concatenation.

Example: Slice a portion of the list and concatenate it with another list.

```
numbers = [1, 2, 3, 4, 5]
slice = numbers [1:4] # Gets items from index 1 to 3
extra_numbers = [6, 7]
combined = slice + extra_numbers
print(combined)
# Output : [2, 3, 4, 6, 7]
```

Explanation: The code slices the 'numbers' list from index 1 to 3, combines it with 'extra_numbers', and prints the resulting list '[2, 3, 4, 6, 7]'.

Example: Sort a list of student names and remove a specific name.

```
student_names= ["Ahmed", "Sara", "Ali", "Hina"]
student_names.sort ()
student_names.remove("Sara")
print(student_names)
# Output: ['Ahmed ', 'Ali', 'Hina ']
```

Explanation: The code sorts the list 'student_names' alphabetically, removes 'Sara' from the list, and then prints the updated list.

Class Activity

Imagine you are maintaining a list of your favorite books: ["To Kill a Mockingbird", "1984", "The Great Gatsby", "Pride and Prejudice"]. Perform the following tasks using Python:

1. Add a new book "Moby Dick" to the list.
2. Replace "1984" with "Brave New World".
3. Remove "The Great Gatsby" from the list.
4. Merge this list with another list of books: ["War and Peace", "Hamlet"].
5. Print the final list of books.

Tidbits

Use list methods like `append()` and `remove()` to efficiently manage and modify your lists. For larger projects, organizing data in lists helps keep your code clean and manageable.

2.6.2 Tuples

In Python, tuples are a type of data structure used to store an ordered collection of items, similar to lists, but with a key difference: tuples are immutable, meaning their values cannot be changed after creation.

Example

```
# Creating a tuple
my_tuple = (1, 2, 3, "Hello", 4.5)
# Accessing elements by index
print(my_tuple[0]) # Output: 1
print(my_tuple[3]) # Output: Hello
# Tuple length
print(len(my_tuple)) # Output: 5
```

2.6.3 Indexing and Slicing

Indexing and slicing are essential techniques in Python for accessing and manipulating sequences such as lists, tuples, and strings.

2.6.3.1 Indexing

Indexing allows you to access individual elements in a sequence. Python uses zero-based indexing, meaning the first element has an index of 0, the second element has an index of 1 and so on.

2.6.3.2 Slicing

Slicing allows you to access a subset of a sequence. The syntax for slicing is `sequence[start: stop: step]`, where `start` is the starting index, `stop` is the ending index (not inclusive), and `step` is the step size.

2.6.3.3 Indexing and Slicing with Negative Indices

Negative indices count from the end of the sequence. For example, `-1` refers to the last element, `-2` refers to the second last element, and so on.

Example: Indexing and slicing with both positive and negative indices on a list.

```
# Create a list of fruits
fruits = ["Apple", "Banana", "Cherry", "Date", "Elderberry " ]
# Indexing
print("First fruit:", fruits [0]) # Positive index
print("Last fruit:", fruits [-1]) # Negative index
# Slicing with positive indices
print("Fruits from index 1 to 3:", fruits[1:4])
# Slicing with negative indices
print("Fruits from index -4 to -1:", fruits [-4 : - 1] )
```

Explanation: This code demonstrates list operations in Python: creating a list of fruits, accessing elements using positive and negative indexing, and slicing the list with both positive and negative indices.

Class Activity

Consider the following list, tuple, and string:

```
# List: [10, 20, 30, 40, 50, 60, 70, 80]
# Tuple: ("Math", "Science", "English", "History", "Geography")
# String: "Python Programming"
```

Perform the following operations:

1. Access and print the third element from each sequence (list, tuple, and string).
2. Slice and print elements from index 2 to 5 from the list and the tuple.
3. Slice and print characters from index 7 to the end of the string.
4. Use negative indexing to print the last two elements from the list and the tuple.
5. Use negative slicing to print characters from the second last to the last character of the string.

Write the Python code to perform these operations and print the results.

Tidbits

Indexing and slicing are powerful tools for working with sequences in Python. Practice these techniques to become more proficient in manipulating data and accessing specific parts of sequences.

2.7 Modular Programming in Python

Modular programming is a technique used to divide a program into smaller, manageable, and reusable pieces called modules. By breaking a program into modules, developers can work on different parts independently and reuse code efficiently. This approach simplifies managing complex programs and promotes code reuse.

The main Function

The main function in Python defines where the program should start. It's usually placed in a block that checks if the script is being run directly or imported as a module.

Example: Here's a simple example:

```
# main.py
def main():
    print("This is the main function.")
if __name__ == "__main__":
    main()
```

Explanation: In this example, the `main()` function will only run if the script is executed directly, not when it's imported elsewhere. This setup is useful in larger projects that have multiple modules.

Tidbits

Using the `main` function with modules helps keep your code organized, making it easier to maintain. Always use the `main` function to define the starting point of your program, and use modules to separate different parts of your code.

DO YOU KNOW?



Python's standard library is made up of hundreds of modules that you can be used to perform common tasks, like working with dates, generating random numbers, or reading files.

Class Activity

Create a Python module named `calculator.py` that includes two functions:

1. `add(a, b)` - This function should return the sum of two numbers.
2. `subtract(a, b)` - This function should return the difference between two numbers. Then, write a script named `main.py` that imports your `calculator` module and uses these functions to perform the following:
 1. Print the result of adding 15 and 8.
 2. Print the result of subtracting 10 from 25.

Make sure to run your `main.py` script and verify that the output is correct.

2.8 Object-Oriented Programming in Python

Object-Oriented Programming (OOP) is a way of designing and organizing code to make it easier to manage and understand.

2.8.1 Class and Objects

A class is like a template for creating things, and an object is an actual thing created from that template. Imagine you want to make a toy car. You first need a blueprint or a template that describes how the toy car should look and function. This template includes details like:

- Color
- Size
- Number of wheels
- Type of material

The template is not an actual toy car; it's just a plan and it represents a class. Using the template, you can create multiple toy cars. Each object is an instance of the class, meaning it follows the plan to have its own specific characteristics.

2.8.1.1 Defining Classes and Creating Objects

In programming, we use classes as concepts to define what an object should be like.

```

# Define a class called ToyCar
class ToyCar:
    # The __init__ method initializes the object with specific attributes
    def __init__(self, color, size, wheels):
        self.color = color    # Color of the toy car
        self.size = size      # Size of the toy car
        self.wheels = wheels  # Number of wheels in the toy car
    # Method to describe the toy car
    def describe(self):
        return f"This toy car is {self.color}, size {self.size}, and has {self.wheels} wheels."
# Create objects of the ToyCar class
car1 = ToyCar("red", "small", 4)
car2 = ToyCar("blue", "large", 6)

# Print descriptions of the toy cars
print(car1.describe())
print(car2.describe())

```

Explanation:

Class Definition: The "ToyCar" class is like the template for making toy cars. It describes what attributes a toy car should have: color, size, and wheels.

Creating Objects: "car1" and "car2" are specific toy cars object created using the ToyCar template. Each has its own unique attributes.

Using Methods: The describe () method allows us to get a description of the toy car.

Self: self is a convention used in Object-Oriented Programming (OOP) to represent the instance of a class within its methods.

2.9 Advanced Python Concepts

Advanced Python concepts extend the foundational knowledge and empower programmers to handle more complex tasks effectively. This section covers key topics such as exception handling, which deals with managing errors gracefully, and file handling, which involves reading from and writing to files. Mastering these concepts is essential for developing robust and efficient Python applications.

2.9.1 Exception Handling

Exception handling is a mechanism to manage errors that occur during program execution. It allows a program to continue running or gracefully terminate if an error occurs, ensuring more robust and error-resilient code.

2.9.1.1 Try-Except Blocks

In Python, the try block lets you test a block of code for errors, and the except block lets you handle errors if occur.

Example:

```
Input a
try :
    result = 10/a # This line creates error if the value of 'a' is 0
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Explanation: In this example:

- The try block contains code that might cause an error.
- The except block catches the ZeroDivisionError and handles it by printing a message.

2.9.1.4 File Handling

File handling involves reading from and writing to files. It is essential for storing data persistently.

2.9.1.5 Opening, Reading, and Closing Files

To read a file, open it using the open() function, read its contents, and then close the file to free up resources.

```
# Open and read a
with open("example.txt", "r") as file:
    content = file.read ()
    print(content)
```

Explanation: In the above code:

- The with statement ensures that the file is properly closed after its suite finishes, even if an error occurs.
- The file is opened in read mode (r), read contents into content, and then printed.
- The file opened using 'with' is automatically closed.

2.9.1.6 Writing to Files

To write to a file, open it in write mode (w) and use the write () method. To append data, use append mode (a).

```
# Writing to a file
with open("example.txt", "w") as file:
    file.write("As-Salaam-Alaikum, World!\n")
# Appending to a file
with open("example.txt", "a") as file:
    file.write("Appending new line.\n")
```

Explanation: In the above code:

- The file is opened in write mode (w) to overwrite its contents and write new data.
- The file is opened in append mode (a) to add data without overwriting existing content.

2.10 Testing and Debugging in Python

In Python programming, testing and debugging are essential practices to ensure that your code works correctly and efficiently.

2.10.1 Testing

Testing is the process of running your code with various inputs to check if it behaves as expected. The goal is to find and fix any issues before the code is used in real-world applications.

2.10.1.1 Types of Testing

- **Unit Testing:** Tests individual parts of the code (like functions or classes) in isolation. Python's `unittest` module is commonly used for this.
- **Integration Testing:** Checks how different parts of the code work together.
- **Functional Testing:** Validates that the software behaves as expected from the user's perspective.
- **Regression Testing:** Ensures that new changes don't break existing functionality.

2.10.1.2 Debugging

Debugging is the process of finding and fixing errors (bugs) in your code. It involves identifying the root cause of problems and making the necessary changes.

2.10.1.3 Common Debugging Techniques

- **Print Statements:** Adding print statements to check the values of variables at different stages of the code.
- **Debugging Tools:** Using tools like `pdb` (Python Debugger) to step through the code, inspect variables, and understand the flow of execution.
- **Error Messages:** Reading and interpreting error messages to locate the source of the problem.

EXERCISE

Multiple Choice Questions

1. An action needed during Python installation to run from the command line easily:

- a) Uncheck "Add Python to PATH"
- b) Choose a different IDE
- c) Check "Add Python to PATH"
- d) Install only the IDE

2. A valid variable name in Python is:

- a) variable1
- b) 1variable
- c) variable-name
- d) variable name

3. Output of the following piece of code is:

```
age = 25
```

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

- a) Age: 25
- b) 25
- c) Age
- d) age

4. The operator used for exponentiation in Python is:

- a) *
- b) **
- c) //
- d) /

5. A loop used to iterate over a collection such as lists is:

- a) while
- b) for
- c) do-while
- d) repeat

6. A range() function used to generate a sequence of numbers:

- a) Generates a list of numbers
- b) Creates a sequence of numbers
- c) Calculates the sum of numbers
- d) Prints a range of numbers

7. A keyword used to define a function in Python:

- a) define
- b) function
- c) def
- d) func

8. The Output of the following code is:

```
temperature, humidity, wind_speed = 25, 60, 15
```

```
print("Hot and humid" if temperature > 30 and humidity > 50 else
```

```
"Warm and breezy" if temperature == 25 and wind_speed > 10 else
```

```
"Cool and dry" if temperature < 20 and humidity < 30 else
```

```
"Moderate")
```

- a) Hot
- b) Warm
- c) Cool
- d) Nothing

9. The operation used to combine two lists in Python:

- a) combine()
- b) concat()
- c) +
- d) merge()

Short Questions

1. Explain the purpose of using comments in Python code.
2. Describe the difference between integer and float data types in Python. Provide an example of each.
3. Define operator precedence and give an example of an expression where operator precedence affects the result.

- 
4. How does the short-hand if-else statement differ from the regular if-else statement?
 5. Explain the use of the range() function in a for loop.
 6. Explain how default parameters work in Python functions.
 7. Explain why modular programming is useful in Python.
 8. Explain the difference between a class and an object in Python.

Long Questions

1. Evaluate the following Python expressions.
 - (a) $(18 / 3 + 4 ** 2) - (2 * (7 - 3)) / (9 * 7, 4)$
 - (b) $(25 + 3 * 4 ** 2 - 6) / (2 ** 3 + 1) - 7$
 - (c) $(12 + 6 * (5 - 2)) ** 2 / ((4 ** 2 - 7) + 10)$
 - (d) $45 / (2 ** 2 + 3 * 4) + 8 * (7 - 3)$
2. Translating the following mathematical expressions to Python syntax
3.
 - (a) $5x(3 + 2^3)$
 $6 - 2x3$
 - (b) $7 + 2^2$
4. Explain the concept of variables in Python.
5. Write a Python program that takes a number as input and checks whether it is positive, negative, or zero using an if-elif-else statement.
6. Write a Python program using a while loop that prints all the odd numbers between 1 and 100. Also, count and print the total number of odd numbers.