PYTHON PROGRAMMİNG:

FROM FUNDAMENTALS TO MASTERY

BY HASAN URAL

CONTENT

Chapter 1: Introduction to Python

In this chapter, we will cover the basics of Python and get you started with the language.

1.1 What is Python? Python is a high-level, interpreted programming language known for its simplicity and readability. It is widely used in various domains, including web development, data analysis, artificial intelligence, and more.

1.2 Installing Python To begin, you need to install Python on your computer. Visit the official Python website (https://www.python.org/) and download the latest version compatible with your operating system. Follow the installation instructions provided.

1.3 Running Python Code Once Python is installed, you can run Python code using the Python interpreter or an integrated development environment (IDE) such as PyCharm, Visual Studio Code, or Jupyter Notebook. We recommend starting with an IDE for a smoother coding experience.

1.4 Your First Python Program Let's write a simple "Hello, World!" program to get started:

```
print("Hello, World!")
```

Explanation:

- The **print()** function is used to display output to the console.
- The string **"Hello, World!"** is passed as an argument to the **print()** function.

1.5 Variables and Data Types Python is dynamically typed, meaning you don't need to declare variable types explicitly. Here's an example:

```
message = "Hello, Python!"

count = 10

pi = 3.14
```

Explanation:

- In this example, we create three variables: **message**, **count**, and **pi**.
- The variable **message** stores a string, **count** stores an integer, and **pi** stores a float.

1.6 Basic Operations Python supports various basic arithmetic operations:

Python supports various basic arithmetic operations:

```
x = 5

y = 2


addition = x + y

subtraction = x - y

multiplication = x * y

division = x / y

exponentiation = x ** y

remainder = x % y


print(addition, subtraction, multiplication, division, exponentiation, remainder)
```

Explanation:

- In this example, we perform addition, subtraction, multiplication, division, exponentiation, and modulus operations on the variables **x** and **y**.
- The results are assigned to respective variables and printed using the **print()** function.

1.7 Control Flow - Conditional Statements Conditional statements allow you to make decisions based on certain conditions. Here's an example using an **if** statement:

```
age = 18

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

Explanation:

- In this example, we use an **if** statement to check if **age** is greater than or equal to 18.
- If the condition is **True**, the first block of code is executed; otherwise, the code in the **else** block is executed.

Chapter 2: Data Types and Operators

In this chapter, we will explore different data types in Python and learn about operators that can be used to perform operations on these data types.

2.1 Numeric Data Types Python provides several numeric data types, including integers, floats, and complex numbers. Here's an example:

```
# Integers
x = 10
```

```
y = -5


# Floats

pi = 3.14

radius = 2.5


# Complex numbers

z = 2 + 3j
```

Explanation:

- In this example, we create variables to store integers, floats, and a complex number.
- Integers are whole numbers, floats are numbers with decimal points, and complex numbers have a real and imaginary part.

2.2 String Data Type Strings are used to represent text data in Python. You can create strings using single or double quotes. Here's an example:

```
name = "Alice"

message = 'Hello, ' + name + '!'


print(message)
```

Explanation:

- In this example, we create a string variable **name** and concatenate it with another string using the **+** operator.
- The resulting string is stored in the **message** variable and printed.

2.3 Lists Lists are used to store multiple values in a single variable. The elements in a list are ordered and can be of different data types. Here's an example:

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


print(fruits[0])     # Accessing the first element

print(len(fruits))   # Getting the length of the list


fruits.append('orange')      # Adding an element to the end

fruits.insert(1, 'grape')    # Inserting an element at a specific index


print(fruits)
```

Explanation:

- In this example, we create a list of fruits and perform various operations on it.
- We access elements using indexing, get the length of the list using the **len()** function, add elements using the **append()** method, and insert elements at specific positions using the **insert()** method.

2.4 Tuples Tuples are similar to lists but are immutable, meaning their elements cannot be changed once defined. Here's an example:

```
point = (3, 4)


print(point[0])     # Accessing the first element

print(len(point))   # Getting the length of the tuple
```

Explanation:

- In this example, we create a tuple representing a point in 2D space.
- We access elements using indexing and get the length of the tuple using the **len()** function.

2.5 Operators Python provides various operators to perform operations on different data types. Here are a few examples:

```
x = 5

y = 2


addition = x + y

subtraction = x - y

multiplication = x * y

division = x / y

exponentiation = x ** y

remainder = x % y


print(addition, subtraction, multiplication, division, exponentiation, remainder)
```

Explanation:

- This example demonstrates arithmetic operations using operators like **+**, **-**, *****, **/**, ****** (exponentiation), and **%** (remainder).
- The results of these operations are assigned to variables and printed.

Chapter 3: Control Flow and Loops

In this chapter, we will delve into control flow statements and loops, which allow you to control the execution flow of your program and perform repetitive tasks.

3.1 Conditional Statements - If, Elif, Else Conditional statements allow you to execute different blocks of code based on different conditions. Here's an example using **if**, **elif**, and **else** statements:

```
x = 10


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

Explanation:

- In this example, we use an **if** statement to check if **x** is greater than 0.
- If the condition is true, the corresponding block of code is executed. Otherwise, the program checks the next condition using the **elif** statement.
- If none of the conditions are true, the code block under the **else** statement is executed.

3.2 Loops - For Loop A **for** loop is used to iterate over a sequence (such as a list, tuple, or string) or any iterable object. Here's an example:

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


for fruit in fruits:
    print(fruit)
```

Explanation:

- In this example, we iterate over each element in the **fruits** list using a **for** loop.
- The variable **fruit** takes on the value of each element in the list in each iteration, and it is printed.

3.3 Loops - While Loop A **while** loop is used to repeatedly execute a block of code as long as a certain condition is true. Here's an example:

```
count = 0


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

Explanation:

- In this example, the **while** loop executes as long as the condition **count < 5** is true.
- The variable **count** is initially 0, and it is incremented by 1 in each iteration until the condition becomes false.

3.4 Control Flow - Break and Continue The **break** statement is used to exit a loop prematurely, while the **continue** statement skips the rest of the current iteration and moves to the next iteration. Here's an example:

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


for number in numbers:
    if number == 3:
        continue    # Skip printing 3 and move to the next iteration
    elif number == 5:
        break       # Exit the loop when reaching 5
    print(number)
```

Explanation:

- In this example, we use the **continue** statement to skip printing the number 3 and move to the next iteration.
- The **break** statement is used to exit the loop when the number 5 is encountered.

Chapter 4: Functions

In this chapter, we will explore functions in Python, which are reusable blocks of code that perform specific tasks.

4.1 Defining and Calling Functions To define a function in Python, we use the def keyword followed by the function name and parentheses containing any parameters. Here's an example:

```
def greet(name):
    print("Hello, " + name + "!")


greet("Alice")
```

Explanation:

- In this example, we define a function called **greet** that takes a parameter **name**.
- Inside the function, we print a greeting message using the provided name.
- To call the function, we pass an argument, such as **"Alice"**, within parentheses.

4.2 Function Parameters and Return Values Functions can have parameters and return values. Parameters allow us to pass data into functions, and return values allow functions to provide output. Here's an example:

```
def square(number):
    return number ** 2


result = square(5)
print(result)
```

Explanation:

- In this example, we define a function called **square** that takes a parameter **number**.
- The function calculates the square of the provided number using the exponentiation operator ****** and returns the result using the **return** statement.
- We call the function with an argument **5** and store the returned value in the **result** variable, which is then printed.

4.3 Default Parameters In Python, we can assign default values to function parameters. These default values are used when the corresponding argument is not provided during function invocation. Here's an example:

```
def greet(name="World"):
    print("Hello, " + name + "!")


greet()           # Output: Hello, World!
greet("Alice")    # Output: Hello, Alice!
```

Explanation:

- In this example, the **greet** function has a default parameter **name** set to **"World"**.

- If no argument is passed when calling the function, it uses the default value.
- When an argument is provided, it overrides the default value.

4.4 Variable Number of Arguments Python allows us to define functions that accept a variable number of arguments. We can use the ***args** syntax to pass a non-keyworded variable-length argument list. Here's an example:

```
def multiply(*numbers):
    result = 1
    for number in numbers:
        result *= number
    return result


product = multiply(2, 3, 4)
print(product)
```

Explanation:

- In this example, the **multiply** function takes a variable number of arguments using the ***numbers** syntax.
- Inside the function, we iterate over the **numbers** tuple and multiply each number together to calculate the product.
- We call the function with arguments **2**, **3**, and **4**, and the returned product is printed.

Chapter 5: Advanced Topics

In this chapter, we will cover some advanced topics in Python, including data structures, file handling, and error handling.

5.1 Lists and List Comprehension Lists are one of the most commonly used data structures in Python. They can hold multiple items and can be modified. List comprehension is a concise way to create lists based on existing lists. Here's an example:

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


squared_numbers = [num ** 2 for num in numbers]
print(squared_numbers)
```

Explanation:

- In this example, we create a list called **numbers** containing some integers.
- Using list comprehension, we iterate over each element in **numbers**, square each element, and create a new list called **squared_numbers**.
- The resulting list, **squared_numbers**, is printed.

5.2 Dictionaries Dictionaries are another useful data structure in Python. They consist of key-value pairs and allow fast lookup and retrieval of values based on keys. Here's an example:

```
student = {
    'name': 'Alice',
    'age': 20,
    'major': 'Computer Science'
}


print(student['name'])    # Output: Alice
print(student.get('age'))  # Output: 20
```

Explanation:

- In this example, we define a dictionary called **student** with keys such as **'name'**, **'age'**, and **'major'**, and their corresponding values.
- We access values in the dictionary using square brackets and the respective key or by using the **get()** method.

5.3 File Handling Python provides built-in functions and methods for file handling. You can read from and write to files using these capabilities. Here's an example of reading from a file:

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

content = file.read()

file.close()


print(content)
```

Explanation:

- In this example, we open a file named **data.txt** in read mode using the **open()** function.
- We read the content of the file using the **read()** method and store it in the **content** variable.
- Finally, we close the file using the **close()** method and print the content.

5.4 Error Handling - Try, Except, Finally Error handling is important to handle exceptions and prevent program crashes. In Python, we use **try**, **except**, and **finally** blocks for error handling. Here's an example:

```
try:

    result = 10 / 0
```

```
except ZeroDivisionError:
    print("Error: Division by zero")
finally:
    print("The 'try-except' block is finished")
```

Explanation:

- In this example, we attempt to perform a division by zero, which raises a **ZeroDivisionError**.
- The code inside the **try** block is executed, and since an error occurs, the program jumps to the **except** block where we handle the specific error.
- The **finally** block always executes, regardless of whether an exception occurred or not.

Chapter 6: Object-Oriented Programming (OOP)

In this chapter, we will dive into the concepts of object-oriented programming (OOP) in Python, which is a powerful paradigm for organizing and structuring code.

6.1 Classes and Objects A class is a blueprint for creating objects, while an object is an instance of a class. Classes define the properties (attributes) and behaviors (methods) that objects of that class can have. Here's an example:

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width
```

```
rect = Rectangle(5, 3)
print(rect.area())   # Output: 15
```

Explanation:

- In this example, we define a class called **Rectangle** with two attributes: **length** and **width**.
- The **__init__** method is a special method called the constructor, which is invoked when creating a new instance of the class.
- We define a method called **area** that calculates and returns the area of the rectangle.
- We create an object **rect** of the **Rectangle** class with length 5 and width 3, and then call the **area** method on **rect** to compute and print the area.

6.2 Inheritance Inheritance allows us to create a new class (derived class) based on an existing class (base class). The derived class inherits the attributes and methods of the base class and can also add new attributes and methods. Here's an example:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclass must implement the speak method")

class Dog(Animal):
    def speak(self):
        return "Woof!"


class Cat(Animal):
    def speak(self):
        return "Meow!"


dog = Dog("Buddy")
cat = Cat("Whiskers")


print(dog.speak())   # Output: Woof!
print(cat.speak())   # Output: Meow!
```

Explanation:

- In this example, we define a base class called **Animal** with an attribute **name** and a method **speak**, which is marked as **NotImplementedError** since it should be implemented in the derived classes.
- We create two derived classes, **Dog** and **Cat**, which inherit from the **Animal** class and implement their own **speak** methods.
- We create objects **dog** and **cat**, and call their **speak** methods to get the respective animal sounds.

6.3 Encapsulation Encapsulation is a principle of OOP that involves bundling data (attributes) and methods that operate on that data within a class. It helps in data hiding and abstraction. Here's an example:

```
class BankAccount:
    def __init__(self, account_number, balance):
        self._account_number = account_number
        self._balance = balance

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if amount <= self._balance:
            self._balance -= amount
        else:
            print("Insufficient balance")

    def get_balance(self):
        return self._balance

account = BankAccount("1234567890", 1000)
account.deposit(500)
account.withdraw(200)
print(account.get_balance())   # Output: 1300
```

Explanation:

- In this example, we define a **BankAccount** class that encapsulates the account number and balance as private attributes using a single underscore prefix (convention).
- We provide methods

such as ‘deposit’, ‘withdraw’, and ‘get_balance’ to interact with the account.

The methods modify or retrieve the account balance, but the underlying attributes (‘_account_number’ and ‘_balance’) are not directly accessible outside the class.

We create an object ‘account’ of the ‘BankAccount’ class, perform deposit and withdrawal operations, and finally print the updated balance.

6.4 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class. It allows methods in different classes to have the same name but different implementations. Here's an example:

```
class Shape:
    def area(self):
        raise NotImplementedError("Subclass must implement the area method")


class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width


class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius ** 2


shapes = [Rectangle(5, 3), Circle(2)]


for shape in shapes:
    print(shape.area())
```

Explanation:

- In this example, we define a base class **Shape** with an **area** method that raises a **NotImplementedError** since it should be implemented in the derived classes.
- We create two derived classes, **Rectangle** and **Circle**, which inherit from the **Shape** class and provide their own implementations of the **area** method.
- We create a list **shapes** containing objects of different shapes.
- Using a loop, we call the **area** method on each shape object, and polymorphism allows the appropriate **area** method to be invoked based on the actual type of the object.

Chapter 7: Modules and Libraries

In this chapter, we will explore the concept of modules and libraries in Python. Modules are files containing Python code that can be imported and used in other programs. Libraries, on the other hand, are collections of modules that provide additional functionality.

7.1 Importing Modules To use code from a module, we need to import it into our program. Python provides various ways to import modules. Here's an example:

```
import math
```

```
radius = 5

circumference = 2 * math.pi * radius


print(circumference)   # Output: 31.41592653589793
```

Explanation:

- In this example, we import the **math** module, which provides mathematical functions and constants.
- We use the **math.pi** constant to calculate the circumference of a circle with a radius of 5.

7.2 Creating and Importing Custom Modules In addition to built-in modules, we can create our own modules. A module is simply a Python file containing functions, classes, or variables. Here's an example: **module.py**

```
def greet(name):
    print("Hello, " + name + "!")
```

main.py

```
import module


module.greet("Alice")   # Output: Hello, Alice!
```

Explanation:

- In this example, we create a module named **module.py** with a **greet** function that prints a greeting message.
- In the **main.py** file, we import the **module** module and call the **greet** function.

7.3 Using Library Modules Python provides a rich set of libraries that extend its functionality. These libraries can be installed using package managers like **pip** and then imported into our programs. Here's an example using the **requests** library:

```
import requests


response = requests.get("https://www.example.com")
print(response.status_code)   # Output: 200
```

Explanation:

- In this example, we import the **requests** library, which allows us to send HTTP requests and handle responses.
- We use the **requests.get()** function to send a GET request to "https://www.example.com" and store the response in the **response** variable.
- We access the **status_code** attribute of the response to check the HTTP status code.

7.4 Exploring Library Documentation

When using external libraries, it's essential to refer to their documentation for information on available functions, classes, and usage examples. Documentation often includes tutorials, API references, and code samples to help us understand and utilize the library effectively.

Chapter 8: Further Learning and Resources

Congratulations on completing the previous chapters of our Python tutorial! As you continue your journey to advance your Python skills, here are some suggestions for further learning and resources:

8.1 Online Learning Platforms There are several online platforms that offer comprehensive Python courses and tutorials. Some popular ones include:

- Coursera (www.coursera.org)
- Udemy (www.udemy.com)
- Codecademy (www.codecademy.com)
- edX (www.edx.org)

These platforms provide structured courses with video lectures, quizzes, coding exercises, and projects to help you practice and reinforce your Python skills.

8.2 Official Python Documentation The official Python documentation is an excellent resource for understanding the language and its standard library. It provides detailed explanations, examples, and references for all Python features and modules. You can access it at: docs.python.org

8.3 Python Community and Forums Engaging with the Python community can be highly beneficial for learning and getting support. There are various forums and communities where you can ask questions, share ideas, and collaborate with other Python enthusiasts. Some popular platforms include:

- Python subreddit (www.reddit.com/r/Python)
- Python.org community (www.python.org/community)
- Stack Overflow (stackoverflow.com) - a Q&A platform for programming-related questions

8.4 Python Books There are numerous books available on Python programming that cater to different levels of expertise. Here are a few highly recommended ones:

- "Python Crash Course" by Eric Matthes
- "Automate the Boring Stuff with Python" by Al Sweigart
- "Fluent Python" by Luciano Ramalho
- "Python Cookbook" by David Beazley and Brian K. Jones

8.5 Python Projects and Challenges To solidify your Python skills, it's essential to practice by working on projects and solving coding challenges. Websites like Project Euler (projecteuler.net) and LeetCode (leetcode.com) offer a wide range of programming problems to solve in Python.

Remember, practice is key to mastering Python. Experiment with code, build projects, and seek opportunities to apply your knowledge.

Chapter 9: Conclusion

Congratulations on reaching the end of our Python tutorial! Throughout the chapters, we covered the fundamentals of Python programming, explored various concepts and topics, and provided examples to reinforce your learning. By now, you should have a solid foundation in Python and be well-equipped to tackle programming tasks and projects.

Python is a versatile and powerful programming language with a vast ecosystem of libraries and frameworks. It can be used for web development, data analysis, machine learning, scientific computing, and much more. As you continue your programming journey, don't hesitate to explore different areas and applications of Python that interest you.

Remember, programming is a continuous learning process. To further enhance your skills and stay up to date with the latest developments, it's crucial to keep practicing, building projects, and exploring new concepts. Embrace challenges, seek opportunities to collaborate with others, and contribute to the Python community.

Lastly, if you ever need assistance or have questions in the future, feel free to come back and ask. We're here to help you on your Python programming journey.

Happy coding!