Gufran Ahmad

PYTHON PROGRAMMING: A QUICK OVERVIEW

Introduction to Python

Python is a high-level, interpreted programming language designed to be easy to understand and use. It is known for its clean and readable syntax.

Why Python?

- Easy to Learn: Python is often referred to as a beginner-friendly programming language.
- Versatile: It’s used in web development, data science, AI, machine learning, automation, and much more.
- Huge Community: Python has a large, supportive community, making it easier to find solutions and resources online.

Lesson 1: Setting Up Python

Before we begin writing code, let's install Python.

1. Download and Install Python:
    - Go to https://www.python.org/downloads/
    - Download the latest stable version of Python (Python 3.x).
    - Install Python and make sure to check the box that says "Add Python to PATH" during installation.
2. IDEs (Integrated Development Environments): You can write Python code in several environments:
    - IDLE: Comes pre-installed with Python.
    - VSCode: A popular code editor.
    - Jupyter Notebooks: Great for data science and experimentation.
3. Running Python Code: You can run Python in a terminal, an IDE, or a Jupyter notebook.

Lesson 2: Your First Python Program – "Hello, World!"

Let’s start with the classic first program in any language: Hello, World!

```
# This is a comment
# Comments start with a # symbol and are ignored by Python

print("Hello, World!")  # This command prints text to the screen
```

Explanation:

- `print("Hello, World!")`: This tells Python to display the text inside the quotes to the screen.
- Anything after `#` is a comment, which is not executed.

Lesson 3: Variables and Data Types

In Python, you store data in variables. A variable is just a name that refers to a value.

```
# Example of variables
name = "Alice"  # String data type
age = 25        # Integer data type
height = 5.7    # Float data type
is_student = True  # Boolean data type

print(name)
print(age)
print(height)
print(is_student)
```

Common Data Types:

1. Strings: Text (e.g., `"Alice"`, `"Hello, World!"`)
2. Integers: Whole numbers (e.g., `10`, `-5`)
3. Floats: Decimal numbers (e.g., `3.14`, `-0.5`)
4. Booleans: True or False values (e.g., `True`, `False`)

Lesson 4: Operators in Python

Operators are symbols that perform operations on variables.

Types of Operators:

1. Arithmetic Operators (addition, subtraction, etc.):
2. x = 10
3. y = 5
4. print(x + y) # Addition
5. print(x - y) # Subtraction
6. print(x * y) # Multiplication
7. print(x / y) # Division
8. print(x % y) # Modulo (remainder)
9. Comparison Operators (compare values):
10. print(x == y) # Equals
11. print(x != y) # Not equals
12. print(x > y)   # Greater than
13. print(x < y)   # Less than
14. Logical Operators (combine boolean expressions):
15. a = True
16. b = False
17. print(a and b) # AND
18. print(a or b)   # OR
19. print(not a)   # NOT

Lesson 5: Control Flow (Conditional Statements)

Control flow allows us to make decisions in our code using `if`, `elif`, and `else`.

```
age = 20

if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

Explanation:

- `if` checks if a condition is true.
- `elif` (else if) is used to check additional conditions.
- `else` runs if none of the conditions are true.

Lesson 6: Loops in Python

Loops are used to repeat a block of code multiple times.

For Loop:

```
for i in range(5):  # range(5) gives numbers from 0 to 4
    print(i)
```

While Loop:

```
i = 0
while i < 5:
    print(i)
    i += 1  # This is shorthand for i = i + 1
```

Lesson 7: Functions

A function is a block of reusable code that performs a specific task.

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
greet("Bob")
```

Explanation:

- `def` is used to define a function.
- `name` is a parameter that gets passed to the function when it’s called.

Lesson 8: Lists, Tuples, and Dictionaries

In Python, you can store multiple items in collections.

Lists: Ordered, mutable collection of items.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")  # Add an item to the list
print(fruits)
```

Tuples: Ordered, immutable collection of items.

```
coordinates = (10, 20)
# coordinates[0] = 30  # This would raise an error because tuples are immutable
```

Dictionaries: Unordered collection of key-value pairs.

```
student = {"name": "Alice", "age": 20}
print(student["name"])
```

Lesson 9: Working with Files

Python allows you to read from and write to files.

Reading from a file:

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

Writing to a file:

```
with open('example.txt', 'w') as file:
    file.write("Hello, Python!")
```

Lesson 10: Object-Oriented Programming (OOP)

Python supports object-oriented programming, which allows you to model real-world entities with classes and objects.

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

    def speak(self):
        print(f"{self.name} says Woof!")

my_dog = Dog("Buddy", 3)
my_dog.speak()
```

Key Concepts:

1. Class: A blueprint for creating objects.
2. Object: An instance of a class.
3. Method: A function defined inside a class.

Lesson 11: Modules and Libraries

In Python, you can import external libraries to extend the functionality of your program.

```
import math

print(math.sqrt(16))  # Square root of 16
```

Python has a rich ecosystem of libraries for tasks like web development, data analysis, AI, etc.

Lesson 12: Advanced Topics

- Decorators: Functions that modify the behavior of other functions.
- Generators: Functions that allow you to iterate over data lazily.
- Exception Handling: Handling errors with `try`, `except`.
- Regular Expressions: Pattern matching in text.

1. Decorators in Python

What are Decorators?

A decorator is a function that modifies or enhances the behavior of another function or method without changing its source code. Decorators are a very Pythonic way to add functionality to your code in a clean and reusable manner.

How Decorators Work:

A decorator is a function that takes another function as an argument and returns a new function that adds some kind of functionality.

Basic Example of a Decorator:

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_hello():
    print("Hello!")

# Apply decorator
decorated_function = my_decorator(say_hello)
decorated_function()
```

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

Explanation:

1. `my_decorator` takes the `say_hello` function as input.
2. `wrapper` is an inner function that adds some extra behavior before and after calling the original function.
3. The decorator returns the `wrapper` function.

Using the `@` Symbol for Decorators

Python provides a shorthand syntax using the `@` symbol, so you don’t need to manually assign the decorated function.

```
@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

This is equivalent to writing:

```
say_hello = my_decorator(say_hello)
say_hello()
```

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

Decorators with Arguments

You can create decorators that accept arguments. Here's how you can modify the decorator to pass arguments to the function:

```
def repeat_decorator(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat_decorator(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

Output:

```
Hello, Alice!
Hello, Alice!
```

```
Hello, Alice!
```

In this example, the `repeat_decorator` takes a parameter `n` and repeats the call to the `greet` function `n` times.

Lesson 13: Advanced Topics

2. Generators in Python

What are Generators?

A generator is a special type of iterator in Python that allows you to iterate over a sequence of values lazily, meaning the values are generated one at a time and only when required. Generators are more memory-efficient than using lists when working with large datasets.

Creating Generators with `yield`

Generators are written using the `yield` keyword, which allows you to produce a series of values lazily.

Basic Generator Example:

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

# Using the generator
counter = count_up_to(5)
for num in counter:
    print(num)
```

Output:

```
1
2
3
4
5
```

Explanation:

1. The function `count_up_to` is a generator that yields numbers starting from 1 up to `n`.
2. Each time `yield` is called, the function's state is saved, and the next time it’s called, execution continues from where it left off.

Why Use Generators?

- Memory Efficiency: Generators don’t store the entire list in memory. Instead, they generate each item on-the-fly.
- Performance: They can make your code faster when working with large data sets because they don’t require all the data to be in memory at once.

Generator Example with Large Data:

```
def square_numbers(n):
    for i in range(n):
        yield i * i

squares = square_numbers(1000000)  # Creates a generator, not a list
```

The generator produces each square only when needed, so it won’t use up memory by storing all the squares at once.

Lesson 14: Advanced Topics

3. Exception Handling in Python

What is Exception Handling?

Exception handling is a mechanism to handle errors in Python. Instead of your program crashing when an error occurs, you can use exception handling to manage errors gracefully.

Basic Structure of Exception Handling

Python uses the `try`, `except`, and `finally` blocks to handle exceptions.

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Handle the exception
    print("Cannot divide by zero!")
finally:
    print("This block always runs.")
```

Output:

```
Cannot divide by zero!
This block always runs.
```

Explanation:

1. The code inside the `try` block may raise an exception.
2. The `except` block catches the exception and handles it.
3. The `finally` block will always run, regardless of whether an exception occurred or not.

Multiple Except Clauses:

You can catch different types of exceptions in separate `except` blocks.

```
try:
    x = int("abc")  # This will raise a ValueError
except ValueError:
    print("Invalid input!")
except Exception as e:
    print(f"Some other error occurred: {e}")
```

Raising Exceptions:

You can manually raise exceptions using the `raise` keyword.

```
def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero!")
    return x / y

try:
    divide(10, 0)
except ValueError as e:
    print(e)
```

Output:

```
Cannot divide by zero!
```

Lesson 15: Advanced Topics

4. Regular Expressions (Regex)

What is Regular Expression?

A regular expression (regex) is a powerful tool for matching patterns in strings. You can use regex to search for specific text, validate input, or manipulate strings in Python.

Using the `re` Module

Python's `re` module provides functions to work with regular expressions.

```
import re

# Search for a pattern in a string
pattern = r"\d+"  # Match one or more digits
text = "I have 2 apples and 12 oranges."

match = re.search(pattern, text)
if match:
    print("Found a match:", match.group())
else:
    print("No match found.")
```

Output:

```
Found a match: 2
```

Explanation:

- `r"\d+"` is the regular expression pattern. `\d` matches digits, and `+` means one or more.
- `re.search(pattern, text)` searches for the first occurrence of the pattern in the string.

Common Regex Patterns:

1. `\d` – Matches any digit.
2. `\w` – Matches any alphanumeric character (letters, digits, and underscores).
3. `\s` – Matches any whitespace character (spaces, tabs, newlines).
4. `.` – Matches any character except newline.
5. `^` – Matches the beginning of a string.

6. $ – Matches the end of a string.
7. * – Matches 0 or more repetitions of the preceding character.
8. + – Matches 1 or more repetitions.

Example: Matching Email Addresses

```
import re

email = "test@example.com"
pattern = r"^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"

if re.match(pattern, email):
    print("Valid email")
else:
    print("Invalid email")
```

Output:

```
Valid email
```

Other Useful Regex Functions:

1. `re.findall(pattern, string)`: Returns all matches as a list.
2. `text = "The price is 100 dollars and 200 euros."`
3. `numbers = re.findall(r"\d+", text)`
4. `print(numbers)  # ['100', '200']`
5. `re.sub(pattern, repl, string)`: Substitutes all matches of the pattern with a replacement string.
6. `text = "I have 2 apples"`
7. `new_text = re.sub(r"\d", "3", text)`
8. `print(new_text)  # I have 3 apples`

Summary of Advanced Topics:

1. Decorators: Functions that modify the behavior of other functions in a reusable way.
2. Generators: Functions that yield values one at a time, useful for handling large datasets efficiently.
3. Exception Handling: Mechanism to handle errors and maintain control over the program flow using `try`, `except`, and `finally`.
4. Regular Expressions: Tool for pattern matching and text manipulation within strings.

These advanced topics allow you to write cleaner, more efficient, and more maintainable Python code. They are essential for building more complex applications and solving real-world problems effectively.

Final Tips in Your Python Journey

1. Practice: The best way to improve is by practicing coding regularly. Try small projects, like a calculator or a to-do list app.
2. Explore Libraries: Once you’re comfortable with the basics, explore libraries like NumPy (for data science), Flask (for web development), and TensorFlow (for AI).