PYTHON PROGRAMMING

BEGINNER'S GUIDE

13 PRACTICE SETS +
2 MEGA PROJECTS

TOPICS COVERED

CONTENT 0 - INTRODUCTION TO PYTHON

Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility. Created by Guido van Rossum and first released in 1991, Python has become one of the most popular programming languages in the world, used in various fields such as web development, data science, artificial intelligence, and automation.

Key features of Python include:

1. Easy to learn and read: Python uses indentation to define code blocks, making it visually clear and intuitive.

2. Interpreted language: Python code is executed line by line, making debugging easier.

3. Dynamically typed: Variables don't need explicit type declarations.

4. Large standard library: Python comes with a comprehensive set of built-in modules and functions.

5. Extensive third-party libraries: A vast ecosystem of packages for various purposes, easily installable via pip.

6. Cross-platform compatibility: Python runs on Windows, macOS, Linux, and many other platforms.

7. Object-oriented programming support: Python fully supports OOP concepts.

8. Functional programming features: Python allows for functional programming paradigms.

Python syntax is designed to be clear and concise. Here's a simple example:

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

print(greet("World"))
```

Python is widely used in:

- - Web development (Django, Flask)
- - Data analysis and visualization (Pandas, Matplotlib)
- - Machine learning and AI (TensorFlow, PyTorch)
- - Scientific computing (NumPy, SciPy)
- - Automation and scripting
- - Game development (Pygame)
- - Desktop applications (PyQt, Tkinter)

Python's philosophy emphasizes code readability and a clean, pragmatic approach to programming. This is encapsulated in **"The Zen of Python,"** a collection of guiding principles for writing computer programs that can be accessed by typing `import this` in a Python interpreter.

To get started with Python, you can download it from python.org. Python comes with **IDLE,** a basic integrated development environment, but many developers prefer more feature-rich IDEs like **PyCharm, Visual Studio Code, or Jupyter Notebooks** for data science work.

Python's community is large, active, and welcoming to newcomers. The Python Package Index (PyPI) hosts thousands of third-party modules, and there are numerous online resources, tutorials, and forums for learning and problem-solving.

Whether you're a beginner programmer or an experienced developer, Python offers a powerful and flexible toolset for a wide range of programming tasks and projects.

****

CONTENT 1 - MODULES , COMMENTS & PIP

Modules:

Modules are reusable Python files containing code, functions, and variables. They help organize and modularize programs, promoting code reuse and maintainability.

Example:

```
import random
print(random.randint(1, 10))
```

Pip:

Pip is Python's package manager for installing, upgrading, and managing third-party libraries and modules. It simplifies dependency management across projects.

Example:

```
pip install numpy
```

Types of modules:

1. Built-in modules (come with Python)
   Examples: os, sys
2. External modules (installed separately)
   Examples: numpy, matplotlib

Comments:

Comments are non-executable text in code used to explain functionality and improve readability. They are ignored by the interpreter and don't affect program execution.

Types of comments:

1. Single-line comment:

Start with '#' and extend to the end of the line. They're used for brief explanations or to comment out a single line of code.

```
# This is a single-line comment
```

2. Multi-line comment:

Enclosed in triple quotes (''' or """) and can span multiple lines. They're used for longer explanations, docstrings, or temporarily disabling blocks of code.

```
"""
This is a
multi-line com
"""
```

3. Inline comment:

Appear on the same line as code, following the '#' symbol. They provide brief explanations about specific parts of code on the same line.

```
x = 5  # This is an inline comment
```

FOLLOW UP QUESTIONS ON THE CONTENT 1

1. Write a single-line comment and a multi-line comment in Python.
2. Import the `random` module and use it to generate a random integer between 1 and 10.
3. What command would you use to install the `requests` library using pip?
4. Create a Python script that imports the `math` module, calculates the area of a circle with radius 5, and rounds the result to 2 decimal places. Then, write a function to calculate the volume of a cylinder using the circle's area and a given height. Use docstrings to document your function.

SOLUTIONS:

```
''' This is a multi-line comment '
   ' print("Comments example")
```

OUTPUT-1:

```
Comments example
```

```
import random

  random_number = random.randint(1, 10)
  print(f"Random number: {random_number}")
```

OUTPUT-2:

```
Random number: 7
```

```
pip install request
```

```
4.  import math

    def calculate_cylinder_volume(radius, height):
        """
        Calculate the volume of a cylinder.

        Args:
        radius (float): The radius of the cylinder's base
        height (float): The height of the cylinder

        Returns:
        float: The volume of the cylinder
        """
        circle_area = math.pi * radius ** 2
        volume = circle_area * height
        return round(volume, 2)

    circle_area = round(math.pi * 5 ** 2, 2)
    print(f"Area of circle with radius 5: {circle_area}")

    cylinder_volume = calculate_cylinder_volume(5, 10)
    print(f"Volume of cylinder with radius 5 and height 10: {cylinder_volume}")
```

OUTPUT-4:

```
Area of circle with radius 5: 78.54
Volume of cylinder with radius 5 and height 10: 785.
```

Python's f-strings:

(formatted string literals) provide a concise and readable way to embed expressions inside string literals. Introduced in Python 3.6, f-strings are prefixed with 'f' or 'F' and allow you to include expressions inside curly braces {}. These expressions are evaluated at runtime and their string representations are inserted into the string.

Example.

```
name = 'Alice'
age = 30
print(f"Hello, my name is {name} and I am {age} years old.")
```

```
Expected output: Hello, my name is Alice and I am 30 years old.
```

CONTENT 2 - VARIABLES AND DATATYPES

Variable:

A named storage location for data in a program.
Example:

```
`x = 5`
```

Keyword:

Reserved words in Python with special meanings.
Example:

```
`if`, `for`, `def`, `import`
```

Identifier: A name given to entities like variables, functions, or classes.
Example:

```
`my_variable = 10`
```

Data types:

Data types categorize and define the nature of data in programming. They determine how data is stored and manipulated in a program.

Most used data types:

1. Integer: `x = 5`
2. Float: `y = 3.14`
3. String: `name = "Alice"`
4. Boolean: `is_valid = True`
5. List: `numbers = [1, 2, 3]`
6. Tuple: `coordinates = (10, 20)`
7. Dictionary: `person = {"name": "Bob", "age": 30}`

Rules for choosing an identifier:

1. Start with a letter or underscore
2. Can contain letters, numbers, and underscores
3. Case-sensitive
4. Cannot be a Python keyword

Operators in Python:

Operators are symbols that perform operations on operands. They are used to manipulate data and perform calculations in Python.

Types of operators:

1. Arithmetic: `+`, `-`, `*`, `/`
2. Comparison: `==`, `!=`, `<`, `>`
3. Logical: `and`, `or`, `not`
4. Assignment: `=`, `+=`, `-=`
5. Bitwise: `&`, `|`, `^`

TYPE() Function:

Returns the type of an object.
Example:

```
`print(type(5))`
```

TypeCasting: Converting one data type to another.
Example:

```
`int("10")`
```

Integer to string conversion: `str(42)`
String to integer conversion: `int("42")`

INPUT() Function:

Reads a line of input from the user as a string.
Example:

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

FOLLOW UP QUESTIONS ON THE CONTENT 2

1. Create variables for a person's name, age, and height (in meters). Print out a sentence using these variables.
2. Create a list of three favorite colors and print the second color in the list.
3. Create a dictionary representing a book with keys for title, author, and year. Print out the author of the book.
4. Create a list of numbers. Use a list comprehension to create a new list with only the even numbers, then calculate and print the sum of this new list.

SOLUTIONS:

```
name = "Alice"
age = 30
height = 1.65

print(f"{name} is {age} years old and {height} meters tall.")
```

OUTPUT-1:

```
Alice is 30 years old and 1.65 meters tall.
```

```
favorite_colors = ["blue", "green", "purple"]
print(favorite_colors[1])
```

OUTPUT-2:

```
green
```

```
book = {
    "title": "To Kill a Mockingbird",
    "author": "Harper Lee",
    "year": 1960
}

print(book["author"])
```

OUTPUT-3:

```
Harper Lee
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [num for num in numbers if num % 2 == 0]
sum_even = sum(even_numbers)

print(f"Original list: {numbers}")
print(f"Even numbers: {even_numbers}")
print(f"Sum of even numbers: {sum_even}")
```

OUTPUT-4:

```
Original list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Even numbers: [2, 4, 6, 8, 10]
Sum of even numbers: 30
```

CONTENT 3 - STRINGS

Strings:

Strings in Python are sequences of characters, enclosed in single ('), double ("), or triple (''' ''' or """ """) quotes. They are immutable, meaning once created, their contents cannot be changed.

Types of strings with example code:

1. Single-quoted:

```
s1 = 'Hello'
```

2. Double-quoted:

```
s2 = "World"
```

3. Triple-quoted (multiline):

```
s3 = '''This is a
  multiline string'''
```

String Slicing:

String slicing allows you to extract a portion of a string using the syntax [start:end:step].
Example:

```
text = "Python";
print(text[1:4])  # Output: "yth"
```

Slicing with skip value:

Skip value in slicing determines the step between each character selected.
Example:

```
text = "Python";
print(text[::2])  # Output: "Pto"
```

String functions() :

Python provides numerous built-in functions and methods to manipulate strings.
These functions allow operations like searching, replacing, changing case, and more.

Example:

```
text = "hello world";
print(text.upper())  # Output: "HELLO WORLD"
```

Built-in Functions:

1. len(): Returns the length of a string.
Example:

```
print(len("Python"))  # Output: 6
```

2. string.endswith(): Checks if the string ends with a specified suffix.

Example:

```
print("Python".endswith("on"))  # Output: True
```

3. string.count(): Counts occurrences of a substring in the string.
Example:

```
print("Mississippi".count("s"))  # Output: 4
```

4. string.capitalize(): Returns a copy of the string with its first character capitalized.
Example:

```
print("python".capitalize())  # Output: "Python"
```

5. string.find(): Returns the lowest index of a substring in the string, or -1 if not found.
Example:

```
print("Hello".find("l"))  # Output: 2
```

6. string.replace(): Returns a copy of the string with all occurrences of a substring replaced.
Example:

```
print("Hello".replace("l", "w"))  # Output: "Hewwo"
```

ESCAPE SEQUENCE CHARACTERS;

Escape sequence characters are special characters in strings that begin with a backslash (). They are used to represent characters that are difficult or impossible to type directly, such as newlines or tabs.

Some common escape sequence examples:

1.\n - Newline:

```
print("Hello\nWorld") # Outputs "Hello" and "World" on separate lines
```

2,\t - Tab:

```
print("Name:\tJohn") # Outputs "Name: John" with a tab between
```

3.\ - Backslash:

```
print("C:\Users\John") # Outputs "C:\Users\John"
```

4.\b - Backspace:

```
print("Hello\bWorld") # Outputs "HellWorld"
```

FOLLOW UP QUESTIONS ON THE CONTENT 3

1. Create a string variable with your full name and print its length.
2. Create a string with a sentence and convert it to uppercase.
3. Create two string variables with your first name and last name. Concatenate them with a space in between and print the result.

SOLUTIONS:

```
full_name = "John Doe"
print(len(full_name))
```

OUTPUT-1:

```
8
```

```
sentence = "Python is a great programming language."
print(sentence.upper())
```

OUTPUT-2:

```
PYTHON IS A GREAT PROGRAMMING LANGUAGE.
```

```
first_name = "Jane"
last_name = "Smith"
full_name = first_name + " " + last_name
print(full_name)
```

OUTPUT-3:

```
Jane Smith
```

CONTENT 4 - LISTS AND TUPLES

Lists in Python:

Lists are ordered, mutable collections that can contain elements of different data types. They are defined using square brackets [] and elements are separated by commas.

Examples:

```
mixed_list = [1, "hello", 3.14, True, [1, 2, 3]]
empty_list = []
numbers = list(range(1, 6))  # Creates [1, 2, 3, 4, 5]
```

List indexing:

List elements can be accessed using zero-based indexing, with negative indices counting from the end.

Examples:

```
fruits = ["apple", "banana", "cherry", "date"]
print(fruits[0])    # Output: "apple"
print(fruits[-1])   # Output: "date"
print(fruits[1:3])  # Output: ["banana", "cherry"]
```

List methods:

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
```

- my_list.sort()   # Sorts the list in-place: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
- my_list.reverse()   # Reverses the list in-place: [9, 6, 5, 5, 4, 3, 3, 2, 1, 1]
- my_list.append(7)   # Adds 7 to the end: [9, 6, 5, 5, 4, 3, 3, 2, 1, 1, 7]
- my_list.insert(0, 0) # Inserts 0 at index 0: [0, 9, 6, 5, 5, 4, 3, 3, 2, 1, 1, 7]
- my_list.pop()   # Removes and returns the last element: 7
- my_list.remove(5)   # Removes the first occurrence of 5: [0, 9, 6, 5, 4, 3, 3, 2, 1, 1]

Tuples:

Tuples are ordered, immutable sequences, typically used to store collections of heterogeneous data.

Examples:

```
point = (3, 4)
person = ("John", 30, "New York")
singleton = (42,)  # Note the comma for a single-element tuple
```

Tuple methods:

Tuples have only two built-in methods:

- count()
- index()

Example:

```
my_tuple = (1, 2, 2, 3, 4, 2)
print(my_tuple.count(2))  # Output: 3
print(my_tuple.index(4))  # Output: 4
```

FOLLOW UP QUESTIONS ON THE CONTENT 4

1. Create a list of fruits and add a new fruit to the end of the list. Then print the updated list.
2. Create a tuple of three primary colors . Try to modify one of the colors and observe the result.
3. Create a list of numbers. Use slicing to print the first three numbers and the last two numbers.
4. Create a list of numbers. Use slicing to print the first three numbers and the last two numbers.

SOLUTIONS:

```
fruits = ["apple", "banana", "orange"]
fruits.append("grape")
print(fruits)
```

OUTPUT-1:

```
['apple', 'banana', 'orange', 'grape']
```

```
primary_colors = ("red", "blue", "yellow")
try:
    primary_colors[0] = "green"
except TypeError as e:
    print(f"Error: {e}")
print(primary_colors)
```

OUTPUT-2:

```
Error: 'tuple' object does not support item assignment
('red', 'blue', 'yellow')
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("First three numbers:", numbers[:3])
print("Last two numbers:", numbers[-2:])
```

OUTPUT-3:

```
First three numbers: [1, 2, 3]
Last two numbers: [9, 10]
```

```
import sys

my_list = [1, 2, 3, 4, 5]
my_tuple = (1, 2, 3, 4, 5)

list_size = sys.getsizeof(my_list)
tuple_size = sys.getsizeof(my_tuple)

print(f"Size of list: {list_size} bytes")
print(f"Size of tuple: {tuple_size} bytes")
print(f"The tuple is {list_size - tuple_size} bytes smaller than the list")
```

OUTPUT-4:

```
Size of list: 104 bytes
Size of tuple: 80 bytes
The tuple is 24 bytes smaller than the list
```

CONTENT 5 - DICTIONARIES & SETS

Dictionaries:

Dictionaries are mutable, unordered collections of key-value pairs.

Syntax:

```
my_dict = {
            key1: value1,
            key2: value2, …
           }
```

Examples:

```
person = {"name": "John", "age": 30, "city": "New York"}
grades = {("Alice", "Math"): 95, ("Bob", "History"): 88}
```

Properties of dictionaries:

1. Keys must be unique and immutable
2. Values can be of any type
3. Dictionaries are mutable
4. Dictionaries are unordered (in Python 3.6+, they maintain insertion order)

Example dictionary and methods:

```
student =
{"name": "Emma", "age": 22, "major": "Computer Science"}
```

- student.get("name") : Returns "Emma"
- student.keys() : Returns dict_keys(['name', 'age', 'major'])
- student.values() : Returns dict_values(['Emma', 22, 'Computer Science'])
- student.items() : Returns dict_items([('name', 'Emma'), ('age', 22), ('major', 'computer Science')])

Sets:

Sets are unordered collections of unique, immutable elements.

Syntax and examples:

```
my_set = {1, 2, 3}
fruits = set(["apple", "banana", "cherry"])
```

Properties of sets:

1. Elements are unique
2. Elements must be immutable
3. Sets are unordered
4. Sets are mutable (can add or remove elements)

Operations on sets:

Sets support mathematical operations like union, intersection, difference, and symmetric difference.

```
fruits = {"apple", "banana", "cherry", "date"}
```

- **len(fruits):** Returns 4 (number of elements in the set).
- **fruits.remove("banana"):** Removes "banana" from the set, resulting in {"apple", "cherry", "date"}
- **fruits.pop():** Removes and returns a random element, e.g., "cherry", leaving {"apple", "date", "banana"}
- **fruits.clear():** Empties the set, resulting in set()
- **fruits.union({"grape", "kiwi"}):** Returns a new set {"apple", "banana", "cherry", "date", "grape", "kiwi"}
- **fruits.intersection({"banana", "kiwi", "apple"}):** Returns a new set {"apple", "banana"}

FOLLOW UP QUESTIONS ON THE CONTENT 5

1. Create a dictionary of three students with their names as keys and ages as values. Print the age of the second student.
2. Create a set of unique numbers. Add a new number to the set and try to add a duplicate number. Print the resulting set.
3. Create two sets of fruits. Find and print the intersection of these sets (common fruits).
4. Create a dictionary of book titles and their authors. Use a loop to print each book title and its author.

SOLUTIONS:

```
students = {
   "Alice": 20,
   "Bob": 22,
   "Charlie": 21
}
print("Bob's age:", students["Bob"])
```

OUTPUT-1:

```
Bob's age: 22
```

```
numbers = {1, 2, 3, 4, 5}
numbers.add(6)
numbers.add(3)  # This won't add a duplicate
print(numbers)
```

OUTPUT-2:

```
{1, 2, 3, 4, 5, 6}
```

```
fruits1 = {"apple", "banana", "cherry"}
fruits2 = {"banana", "orange", "kiwi"}
common_fruits = fruits1.intersection(fruits2)
print("Common fruits:", common_fruits)
```

OUTPUT-3:

```
Common fruits: {'banana'}
```

```
books = {
    "To Kill a Mockingbird": "Harper Lee",
    "1984": "George Orwell",
    "Pride and Prejudice": "Jane Austen"
}

for title, author in books.items():
    print(f"'{title}' by {author}")
```

OUTPUT-4:

```
'To Kill a Mockingbird' by Harper Lee
'1984' by George Orwell
'Pride and Prejudice' by Jane Austen
```

CONTENT 6 - CONDITIONAL EXPRESSIONS

Conditional expressions in Python allow you to execute different code based on specified conditions.
They use `if`, `else`, and `elif` statements to control program flow.

Basic if-else syntax:

```
if condition:
    # code if condition is True
else:
    # code if condition is False
```

Example code:

```
age = 18
if age >= 18:
    print("You can vote")
else:
    print("You cannot vote yet")
```

Relational operators

compare values and return boolean results.
Common relational operators include

- `==` (equal to)
- `!=` (not equal to)
- **and** `>` (greater than).

Logical operators

combine or modify boolean expressions.
Python's logical operators are

- `**and**`- true if both operends are true else false.
- `**or**`- true if atleast one operend is true else false.
- `**not**`- inverts true to false & false to true.

`**elif**` is used for multiple conditions and stands for "**else if**":

```
x = 5
if x > 10:
    print("Greater than 10")
elif x > 0:
    print("Positive")
else:
    print("Non-positive")
```

Important notes on `elif`:

1. You can have multiple `elif` statements to check various conditions.
2. `elif` is only executed if the previous conditions are False.

FOLLOW UP QUESTIONS ON THE CONTENT 6

1. Write a program that checks if a number is positive, negative, or zero.
2. Create a program that determines if a person is eligible to vote based on their age.
3. Write a program that checks if a number is even or odd
4. Create a program that determines the price of a movie ticket based on the person's age (children under 12 and seniors over 65 get a discount).

SOLUTIONS:

```
number = 5
result = "positive" if number > 0 else "negative" if number < 0 else "zero"
print(f"The number {number} is {result}")

number = -3
result = "positive" if number > 0 else "negative" if number < 0 else "zero"
print(f"The number {number} is {result}")

number = 0
result = "positive" if number > 0 else "negative" if number < 0 else "zero"
print(f"The number {number} is {result}")
```

OUTPUT-1:

```
The number 5 is positive
The number -3 is negative
The number 0 is zero
```

```
age = 18
eligibility = "eligible" if age >= 18 else "not eligible"
print(f"A person aged {age} is {eligibility} to vote")

age = 16
eligibility = "eligible" if age >= 18 else "not eligible"
print(f"A person aged {age} is {eligibility} to vote")
```

OUTPUT-2:

```
A person aged 18 is eligible to vote
A person aged 16 is not eligible to vote
```

```
number = 4
result = "even" if number % 2 == 0 else "odd"
print(f"The number {number} is {result}")

number = 7
result = "even" if number % 2 == 0 else "odd"
print(f"The number {number} is {result}")
```

OUTPUT-3:

```
The number 4 is even
The number 7 is odd
```

```
age = 25
ticket_price = 10 if 12 <= age <= 65 else 5
print(f"The ticket price for a {age}-year-old is ${ticket_price}")

age = 8
ticket_price = 10 if 12 <= age <= 65 else 5
print(f"The ticket price for a {age}-year-old is ${ticket_price}")

age = 70
```

```
ticket_price = 10 if 12 <= age <= 65 else 5
print(f"The ticket price for a {age}-year-old is ${ticket_price}")
```

OUTPUT-4:

```
The ticket price for a 25-year-old is $10
The ticket price for a 8-year-old is $5
The ticket price for a 70-year-old is $5

value_if_true if condition else value_if_false
```

CONTENT 7 - PYTHON LOOPS

Sometimes we want to repeat a set of statements in our program. For instance: Print 1 to 1000.

Loops make it easy for a programmer to tell the computer which set of instructions to repeat and how!

Types of loops in Python:

- while loop
- for loop.

While loop definition:

A while loop repeatedly executes a block of code as long as a condition is True.
It continues until the condition becomes False.

```
while condition:
    # code to be executed
```

Example code:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Important note: Ensure the loop condition eventually becomes False to avoid infinite loops.

For loop definition:

A for loop iterates over a sequence (e.g., list, tuple, string) or other iterable objects.

```
for item in sequence:
    # code to be executed
```

Example code:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Range function:

range() function is used to generate a sequence of number.

```
range(start, stop, step)
```

Example:

```
for i in range(0, 5, 1):
    print(i)
```

For loop with else:

Executes the else block when the loop completes normally (without break).

```
for i in range(3):
    print(i)
else:
    print("Loop finished")
```

Expected output:

```
0
1
2
Loop finished
```

Break statement:

Terminates the loop and transfers execution to the statement following the loop.

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Continue statement:

Skips the rest of the code inside the loop for the current iteration.

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Pass statement:

Acts as a placeholder and does nothing when executed

```
for i in range(5):
    if i == 2:
        pass
    else:
        print(i)
```

FOLLOW UP QUESTIONS ON THE CONTENT 7

1. Write a loop to print the first 5 even numbers.
2. Create a while loop to print the countdown from 5 to 1.
3. Use a for loop to calculate the sum of numbers from 1 to 5.
4. Write a loop to print the multiplication table of 3 up to 5.

SOLUTIONS:

```
for i in range(2, 11, 2):
  print(i)
```

OUTPUT-1:

```
2 4 6 8 10
```

```
count = 5
while count > 0:
    print(count)
    count -= 1
```

OUTPUT-2:

```
5 4 3 2 1
```

```
total = 0
for num in range(1, 6):
    total += num
print("Sum:", total)
```

OUTPUT-3:

```
Sum: 15
```

```
for i in range(1, 6):
    print(f"3 x {i} = {3 * i}")
```

OUTPUT-4:

```
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
```

CONTENT 8 - FUNCTIONS & RECURSIONS

A function is a reusable block of code that performs a specific task.
Functions are used to organize code, improve readability, and avoid repetition.

Function syntax:

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

Example of calling a function:

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

message = greet("Alice")
print(message)
```

Types of functions in Python:

1. Built-in functions
2. User-defined functions
3. Anonymous functions (lambda functions)

Examples of built-in functions:

- print()
- len()

Function with arguments:

Arguments allow you to pass data to a function for processing.

```
def calculate_area(length, width):
    return length * width

area = calculate_area(5, 3)
print(f"The area is: {area}")
```

Default parameter value:

Default values allow you to specify a default argument if one isn't provided.

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

greet()  # Uses default value
greet("Alice")  # Uses provided value
```

Recursion:

Recursion is when a function calls itself to solve a problem.
It typically has a base case to stop recursion and a recursive case.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print(f"Factorial of 5 is: {result}")
```

FOLLOW UP QUESTIONS ON THE CONTENT 8

1. Write a function to calculate the factorial of a number using recursion
2. Create a function that takes a list of numbers and returns the sum of all even numbers in the list.
3. Write a recursive function to calculate the nth Fibonacci number
4. Create a function that takes a string and returns True if it's a palindrome, False otherwise

SOLUTIONS:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

OUTPUT-1:

```
120
```

```
def sum_even(numbers):
    return sum(num for num in numbers if num % 2 == 0)

print(sum_even([1, 2, 3, 4, 5, 6]))
```

OUTPUT-2:

```
12
```

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(7))
```

OUTPUT-3:

```
13
```

```
def is_palindrome(s):
    s = s.lower().replace(" ", "")
    return s == s[::-1]

print(is_palindrome("A man a plan a canal Panama"))
print(is_palindrome("hello"))
```

OUTPUT-4:

```
True
False
```

CONTENT 9 - FILE I/O

File I/O (Input/Output) in Python refers to the process of working with files on a computer. It allows programs to read from and write to files, enabling data persistence and exchange.

File operations include opening, reading, writing, and closing files.
Real-life examples include saving user preferences, logging application events, or processing CSV data.

Types of files in Python:

1. Text files
2. Binary files

Opening a file:

The `open()` function is used to open a file, returning a file object.

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

Reading a file:

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

Another method to read a file:

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

File opening modes:

- 'r': Read (default)
- 'w': Write (overwrites existing content)
- 'a': Append
- 'x': Exclusive creation
- 'b': Binary mode
- '+': Read and write

Writing to a file:

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

The `with` statement ensures proper handling of resources, automatically closing the file after usage.

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

FOLLOW UP QUESTIONS ON THE CONTENT 9

1. Write a program to create a file named "sample.txt" with the content "Hello, File I/O!" and then read and print its contents
2. Create a program that appends a new line "This is a new line" to an existing file "sample.txt" and then prints all lines from the file.
3. Write a program that reads numbers from a file named "numbers.txt" (one number per line) and calculates their sum.
4. Create a program that reads a file named "names.txt", counts the number of lines (names) in it, and prints the count

SOLUTIONS:

```
# Writing to the file
with open("sample.txt", "w") as file:
    file.write("Hello, File I/O!")

# Reading from the file
with open("sample.txt", "r") as file:
    content = file.read()
    print(content)
```

OUTPUT-1:

```
Hello, File I/O!
```

```
# Appending to the file
with open("sample.txt", "a") as file:
    file.write("\nThis is a new line")

# Reading and printing all lines
with open("sample.txt", "r") as file:
    lines = file.readlines()
    for line in lines:
        print(line.strip())
```

OUTPUT-2:

```
Hello, File I/O!
This is a new line
```

```
# Assume "numbers.txt" contains:
# 10
# 20
# 30

total = 0
with open("numbers.txt", "r") as file:
    for line in file:
        total += int(line.strip())
print("Sum:", total)
```

OUTPUT-3:

```
Sum: 60
```

```
# Assume "names.txt" contains:
# Alice
# Bob
# Charlie

count = 0
with open("names.txt", "r") as file:
    for line in file:
        count += 1
print("Number of names:", count)
```

OUTPUT-4:

```
Number of names: 3
```

CONTENT 10 - OBJECT ORIENTED PROGRAMMING

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects, which are instances of classes.
OOP focuses on the concept of objects that contain data and code, emphasizing data rather than procedure.

Class definition:

```
class ClassName:
  # class body
```

Example:

```
class Car:
   def __init__(self, make, model):
      self.make = make
      self.model = model
```

Object:

An object is an instance of a class.

It bundles data and methods that operate on that data.
Objects can interact with one another through methods.

They are used to represent real-world entities or abstract concepts in code.

Modeling a problem in OOP involves identifying the relevant objects, their attributes, and their behaviors, then representing these as classes and methods.

Class attributes:

Variables shared by all instances of a class.

```
class Car:
   wheels = 4  # class attribute
```

Instance attributes:

Variables unique to each instance of a class.

```
class Car:
   def __init__(self, color):
      self.color = color  # instance attribute
```

Note: Instance attributes are defined inside methods, typically in the __init__ method.

Self parameter:

'self' refers to the instance of the class and is used to access instance variables and methods.

```
class Car:
    def start(self):
        print(f"Starting the {self.make}")
```

Static method:

A method that belongs to a class rather than an instance, defined using the @staticmethod decorator.

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y
```

__init__ constructor:

1. It's a special method called when creating a new instance of a class.
2. It's used to initialize the attributes of the new object.

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

person = Person("Alice", 30)
```

FOLLOW UP QUESTIONS ON THE CONTENT 10

1. Create a simple class called `Rectangle` with attributes for width and height. Add a method to calculate the area.
2. Create a simple class called `Rectangle` with attributes for width and height. Add a method to calculate the area.
3. Create a `BankAccount` class with methods for deposit and withdrawal. Include a balance attrib
4. Create a simple inheritance example with a `Person` class and a `Student` class that inherits from `Person`.

SOLUTIONS:

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

    def calculate_area(self):
        return self.width * self.height

rect = Rectangle(5, 3)
print(f"Area of rectangle: {rect.calculate_area()}")
```

OUTPUT-1:

```
Area of rectangle: 15
```

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display_info(self):
        return f"{self.brand} {self.model}"

my_car = Car("Toyota", "Corolla")
print(my_car.display_info())
```

OUTPUT-2:

```
Toyota Corolla
```

```
class BankAccount:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance

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

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
```

```
    print("Insufficient funds")

  def get_balance(self):
    return self.balance

account = BankAccount(100)
account.deposit(50)
account.withdraw(30)
print(f"Current balance: ${account.get_balance()}")
```

OUTPUT-3:

```
Current balance: $120
```

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

  def introduce(self):
    return f"My name is {self.name} and I'm {self.age} years old."

class Student(Person):
  def __init__(self, name, age, student_id):
    super().__init__(name, age)
    self.student_id = student_id

  def introduce(self):
    return f"{super().introduce()} My student ID is {self.student_id}."

person = Person("Alice", 30)
student = Student("Bob", 20, "S12345")

print(person.introduce())
print(student.introduce())
```

OUTPUT-4:

```
My name is Alice and I'm 30 years old.
My name is Bob and I'm 20 years old. My student ID is S12345.
```

CONTENT 11 - INHERITANCE

Inheritance allows a class to inherit attributes and methods from another class.

Inheritance syntax:

```
class ParentClass:
    # parent class code

class ChildClass(ParentClass):
    # child class code
```

Types of inheritance:

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Single inheritance:

A child class inherits from a single parent class.

ParentClass → ChildClass

Multiple inheritance:

A child class inherits from multiple parent classes.
It allows a class to inherit attributes and methods from more than one parent class.

ParentClass1 ↘
ParentClass2 → ChildClass
ParentClass3 ↗

Multilevel inheritance:

A class inherits from a child class, forming a parent-child-grandchild relationship.

GrandparentClass → ParentClass → ChildClass

Super() method:

It's used to call methods from the parent class in the child class.

```
class Child(Parent):
    def __init__(self):
        super().__init__()
```

Class method:

A method that belongs to the class rather than an instance, defined using the @classmethod decorator.

```
class MyClass:
    @classmethod
    def class_method(cls):
        # method body
```

@property decorator:

It's used to define methods that act like attributes, allowing getter-like behavior.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def diameter(self):
        return 2 * self._radius
```

@getter and @setter:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value
```

Operator overloading:

Redefining how operators work for custom classes.
These methods are called when the corresponding operator is used with instances of the class.

Example methods (for a custom Vector class):

1. - __add__: +

2. - __sub__: -
3. - __mul__: *
4. - __truediv__: /
5. - __eq__: ==
6. - __lt__: <
7. - __gt__: >

Here are some two magic methods :

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"{self.title} by {self.author}"

    def __len__(self):
        return len(self.title)

book = Book("1984", "George Orwell")
print(str(book))  # Calls __str__
print(len(book))  # Calls __len__
```

FOLLOW UP QUESTIONS ON THE CONTENT 11

1. Create a base class `Animal` and a derived class `Dog`. Add a method to make sound in both classes.
2. Create a `Vehicle` class and a `Car` class that inherits from it. Add a method to start the engine in both classes.
3. Create a `Shape` class with an area method, then create `Rectangle` and `Circle` classes that inherit from it and implement their own area calculations.
4. Demonstrate multiple inheritance by creating a `Flying` class and a `Swimming` class, then create a `FlyingFish` class that inherits from both.

SOLUTIONS:

```
class Animal:
    def make_sound(self):
        return "Some generic animal sound"

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

```
animal = Animal()
dog = Dog()

print(animal.make_sound())
print(dog.make_sound())
```

OUTPUT-1:

```
Some generic animal sound
Woof!
```

```
class Vehicle:
    def start_engine(self):
        return "The vehicle's engine is starting..."

class Car(Vehicle):
    def start_engine(self):
        return super().start_engine() + " Vroom!"

vehicle = Vehicle()
car = Car()

print(vehicle.start_engine())
print(car.start_engine())
```

OUTPUT-2:

```
The vehicle's engine is starting...
The vehicle's engine is starting... Vroom!
```

```
import math

class Shape:
    def area(self):
        return "Area calculation not implemented"

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

    def area(self):
```

```
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

rectangle = Rectangle(5, 4)
circle = Circle(3)

print(f"Rectangle area: {rectangle.area()}")
print(f"Circle area: {circle.area():.2f}")
```

OUTPUT-3:

```
Rectangle area: 20
Circle area: 28.27
```

```
class Flying:
    def fly(self):
        return "I can fly!"

class Swimming:
    def swim(self):
        return "I can swim!"

class FlyingFish(Flying, Swimming):
    pass

flying_fish = FlyingFish()

print(flying_fish.fly())
print(flying_fish.swim())
```

OUTPUT-4:

```
I can fly!
I can swim!
```

CONTENT 12 - ADVANCED CONCEPTS OF PYTHON

1. Walrus Operator:

The walrus operator (:=) allows you to assign values to variables as part of an expression. It was introduced in Python 3.8.

Example:

```
if (n := len([1, 2, 3])) > 2:
    print(f"List has {n} items")
```

2. Type Definitions in Python:

Python 3.5+ introduced type hints to specify expected types of function parameters and return values.

Example:

```
from typing import List, Dict

def process_data(items: List[int], config: Dict[str, str]) -> str:
    return f"Processed {len(items)} items with {len(config)} config options"
```

3. Advanced Type Hints:

Python's typing module provides more complex type hints like Union, Optional, and Generic.

Example:

```
from typing import Union, Optional, TypeVar

T = TypeVar('T')

def maybe_process(value: Union[int, str]) -> Optional[T]:
    if isinstance(value, int):
        return value * 2
    elif isinstance(value, str):
        return value.upper()
    return None
```

4. Match Case:

The match case statement, introduced in Python 3.10, provides pattern matching capabilities.

Example:

```
def describe_type(value):
    match value:
        case int():
            return "It's an integer"
        case str():
            return "It's a string"
        case list():
            return f"It's a list with {len(value)} items"
        case _:
            return "It's something else"

print(describe_type(42))       # Output: It's an integer
print(describe_type("hello"))  # Output: It's a string
print(describe_type([1, 2, 3])) # Output: It's a list with 3 items
```

5. Dictionary Merge and Update Operations:

Python 3.9 introduced the | and |= operators for merging dictionaries.

Example:

```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}

merged = dict1 | dict2  # {'a': 1, 'b': 3, 'c': 4}
dict1 |= dict2  # dict1 is now {'a': 1, 'b': 3, 'c': 4}
```

6. Exception Handling:

Python uses try, except, else, and finally for exception handling.

Example:

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except ValueError:
    print("Invalid input. Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print(f"Result: {result}")
finally:
    print("Execution completed.")
```

7. Raising Exceptions in One Line:

You can raise exceptions using the `raise` keyword.

Example:

```
raise ValueError("This is an error message")
```

8. Try with Else and Finally:

```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found.")
else:
    print(f"File contents: {content}")
finally:
    file.close() if 'file' in locals() else None
```

Some More Concepts:

1. if __name__ == ' main ' in Python:

This condition checks if the script is being run directly (not imported).
It allows you to execute code only when the script is the main program.
Common use: run tests or examples when the script is executed directly.

2. Global keyword in one line:

The 'global' keyword in Python declares a variable as global, allowing it to be modified outside its local scope.

3. Enumerate function in Python:

Enumerate() adds a counter to an iterable and returns it as an enumerate object.

```
for index, value in enumerate(['a', 'b', 'c']):
    print(f"Index: {index}, Value: {value}")
```

4. List comprehensions

List comprehensions provide a concise way to create lists based on existing lists or iterables.

```
squares = [x**2 for x in range(10)]
```

FOLLOW UP QUESTIONS ON THE CONTENT 12

1. Takes a list of strings as input
2. Uses a list comprehension with the walrus operator to convert each string to an integer, skipping any that can't be converted
3. Uses a match case statement to categorize the resulting numbers
4. Handles potential exceptions

SOLUTIONS:

```
def process_strings(string_list):
    # Implementation will be added here
    pass

# Example input
input_list = ["10", "5", "abc", "  42  ", "-3", "200", "0"]
```

```
def convert_to_integers(string_list):
    return [int(x) for s in string_list if (x := s.strip()).isdigit()]

input_list = ["10", "5", "abc", "  42  ", "-3", "200", "0"]
result = convert_to_integers(input_list)
print(result)
```

OUTPUT-2:

```
[10, 5, 42, 200, 0]
```

```
def categorize_number(num):
    match num:
        case n if n < 0:
            return f"{n} is negative"
        case 0:
            return "Zero"
        case n if 0 < n <= 10:
            return f"{n} is between 1 and 10"
        case n if 10 < n <= 100:
            return f"{n} is between 11 and 100"
        case _:
            return f"{n} is greater than 100"
```

```
numbers = [10, 5, 42, 200, 0, -3]

for num in numbers:
    print(categorize_number(num))
```

OUTPUT-3:

```
10 is between 1 and 10
5 is between 1 and 10
42 is between 11 and 100
200 is greater than 100
Zero
-3 is negative
```

```
def safe_process(string_list):
    try:
        numbers = [int(x) for s in string_list if (x := s.strip()).isdigit()]
        for num in numbers:
            print(categorize_number(num))
    except Exception as e:
        print(f"An error occurred: {e}")

# Example with no exception
safe_process(["10", "5", "abc", "  42  ", "-3", "200", "0"])

# Example that would raise an exception without handling
safe_process(["10", "5", "1e10"])
```

OUTPUT-4A:

```
10 is between 1 and 10
5 is between 1 and 10
42 is between 11 and 100
200 is greater than 100
Zero
```

OUTPUT-4B:

```
10 is between 1 and 10
5 is between 1 and 10
An error occurred: invalid literal for int() with base 10: '1e10'
```

CONTENT 13 - ADVANCED CONCEPTS CONTINUED….

1. Virtual Environment:

A virtual environment is an isolated Python environment for a project.
It allows you to manage project-specific dependencies separately.

Installing and using a virtual environment:

```
python -m venv myenv
source myenv/bin/activate  # On Windows: myenv\Scripts\activate
```

2. pip freeze command:

pip freeze lists all installed packages in the current environment.
It's often used to create requirements.txt files for project dependencies.

Using pip freeze:

```
pip freeze > requirements.txt
```

3. Lambda functions:

Lambda functions are anonymous, single-expression functions defined using the lambda keyword.

Syntax and example:

```
lambda arguments: expression
# Example:
square = lambda x: x**2
```

4. JOIN METHOD (STRINGS):

The join() method concatenates a list of strings using a specified separator.

Example:

```
"-".join(["a", "b", "c"])  # Output: "a-b-c"
```

5. FORMAT METHOD (STRINGS):

The format() method formats specified values and inserts them into a string's placeholder.

Syntax:

```
"{}".format(value)
```

6. MAP, FILTER & REDUCE:

MAP: Applies a function to all items in an input list.
Syntax:

```
map(function, iterable)
```

FILTER: Constructs a list from elements of an iterable for which a function returns True.
Syntax:

```
filter(function, iterable)
```

REDUCE: Applies a function of two arguments cumulatively to the items of a sequence.
Syntax:

```
reduce(function, iterable[, initializer])
```

FOLLOW UP QUESTIONS ON THE CONTENT 13

1. Convert a list of temperatures from Celsius to Fahrenheit.
2. Square each number in a list.
3. Filter out even numbers from a list.
4. Filter strings that have length greater than 3.
5. Find the product of all numbers in a list.
6. Find the maximum number in a list.

SOLUTIONS:

```
from functools import partial

celsius = [0, 10, 20, 30, 40]
fahrenheit = list(map(lambda c: (c * 9/5) + 32, celsius))
print(fahrenheit)
```

OUTPUT-1:

```
Expected output: [32.0, 50.0, 68.0, 86.0, 104.0]
```

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)
```

OUTPUT-2:

```
Expected output: [1, 4, 9, 16, 25]
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
print(odd_numbers)
```

OUTPUT-3:

```
Expected output: [1, 3, 5, 7, 9]
```

```
words = ['cat', 'dog', 'elephant', 'fish', 'giraffe']
long_words = list(filter(lambda x: len(x) > 3, words))
print(long_words)
```

OUTPUT-4:

```
Expected output: ['elephant', 'fish', 'giraffe']
```

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)
```

OUTPUT-5:

```
Expected output: 120
```

```
from functools import reduce

numbers = [3, 7, 2, 9, 5, 1]
max_number = reduce(lambda x, y: x if x > y else y, numbers)
print(max_number)
```

OUTPUT-6:

```
Expected output: 9
```

THE CONCLUSION

As we conclude our comprehensive exploration of Python.

Conclusion: Reflecting on the Python Journey

become a cornerstone in the world of programming. From its intuitive syntax to its powerful capabilities, Python has proven to be an invaluable tool for a wide range of applications.

We've journeyed from the basics of variables and data types to more complex concepts like object-oriented programming and decorators. We've seen how Python's extensive standard library and third-party packages can simplify tasks that would be complex in other languages. The language's versatility shines through its applications in web development, data analysis, machine learning, and automation.

One of the most striking aspects of Python is its readability and emphasis on clean code. The PEP 8 style guide and the philosophy of "**There should be one-- and preferably only one --obvious way to do it**" have instilled good coding practices that will serve well in any programming endeavor.

We've also delved into Python's more advanced features, such as generators, context managers, and metaclasses, which offer sophisticated ways to handle complex programming challenges. The language's support for both procedural and object-oriented paradigms provides flexibility in approach to problem-solving.

****