



PYTHON PROGRAMMING KEY

1 a. Explain features of Python which make it a more sought-after language and discuss the different application areas where Python is widely used.

Python is a highly sought-after programming language due to a combination of features that make it accessible, powerful, and versatile across a wide range of application areas.

Key Features That Make Python Popular:

- **Readable and Simple Syntax:** Python's syntax is clear and close to natural language, making it easy for beginners and experts alike to read, write, and maintain code. This reduces the learning curve and accelerates development.
- **Versatility:** Python supports multiple programming paradigms, including object-oriented, procedural, and functional programming, allowing developers to choose the best approach for their projects.
- **Dynamically Typed and Interpreted:** Variables do not need explicit declaration, and code is executed line by line, which simplifies debugging and speeds up the development cycle.
- **Extensive Standard Library and Ecosystem:** Python offers a vast collection of libraries and frameworks for tasks ranging from web development (Django, Flask) to scientific computing (NumPy, Pandas), machine learning (TensorFlow, scikit-learn), and more.
- **Cross-Platform Compatibility:** Python runs seamlessly on major operating systems like Windows, macOS, and Linux, ensuring portability of code.
- **Open Source and Community Support:** Python is free to use and benefits from a robust, active community that contributes to its continuous improvement and offers extensive support resources.
- **GUI and Web Development Support:** Libraries like Tkinter, PyQt, and frameworks such as Django and Flask enable the development of desktop and web applications.

Popular Application Areas:

1. **Web Development** – Django, Flask
2. **Data Science & Analytics** – Pandas, NumPy, Matplotlib
3. **Machine Learning & AI** – TensorFlow, PyTorch
4. **Game Development** – Pygame
5. **Cybersecurity** – Scripting, scanning, penetration testing

6. DevOps/Cloud – Automation, deployments

7. IoT – Raspberry Pi, MicroPython

1 b. What are keywords in Python? List any seven Python keywords and briefly explain the purpose of each.

Keywords in Python are special reserved words that have predefined meanings and purposes in the language. They are fundamental to Python's syntax and structure, and cannot be used as identifiers (such as variable, function, or class names)

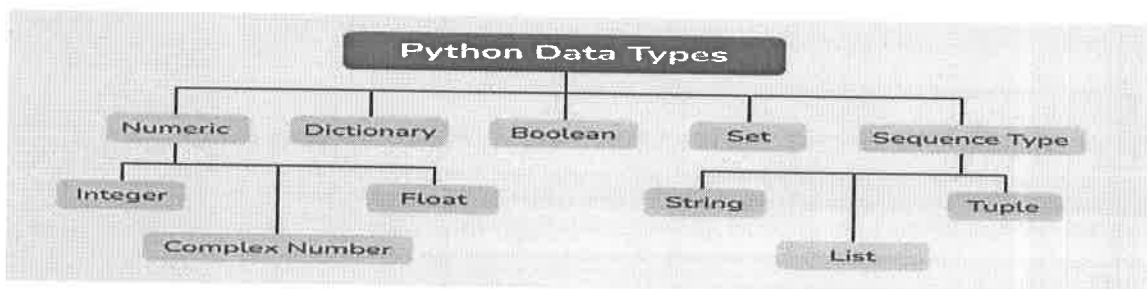
Keyword	Purpose
if	Used for conditional branching; executes a block of code if a specified condition is true.
for	Used to create loops that iterate over a sequence (like a list, tuple, or string).
def	Used to define a function.
class	Used to define a class, enabling object-oriented programming.
return	Exits a function and optionally passes back a value to the caller.
import	Used to include external modules and libraries into the current script.
break	Terminates the nearest enclosing loop prematurely.

2 a. Write in detail about the data types in Python?

The variables can hold values of different type called Data type. Data type is a set of values and allowable operations on those values.

Python has a great set of useful data types. Python's data types are built in the core of the language. They are easy to use and straightforward.

In Python, data types define the kind of value a variable can hold. Python has a rich set of built-in data types.



1. Numeric Types

int: Whole numbers $\rightarrow x = 10$

float: Decimal numbers $\rightarrow y = 3.14$

complex: Complex numbers $\rightarrow z = 2 + 3j$

2. Boolean Type

bool: Logical values \rightarrow True or False

3. Set Types

set: Unordered, unique items $\rightarrow \{1, 2, 3\}$

4. Sequence Types

list: An ordered, mutable collection of items (can be of mixed types).

Example: `[1, "apple", 3.5]`

tuple: An ordered, immutable collection of items.

Example: `(1, "banana", 4.2)`

5. Mapping Type

dict: Stores key-value pairs. Keys must be unique and immutable.

Example: `{"name": "Alice", "age": 30}`

6. String: Represents a sequence of Unicode characters. Strings are immutable and used to store textual data.

Example: `"Hello, World!"`

2 b. Explain the usage of the input() function and the print() function with examples and also discuss how to handle different data types when taking input.

1. input() Function

The input() function is used to **take user input** from the keyboard.

Syntax:

```
variable = input("Enter something: ")
```

Example:

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

2. print() Function

The print() function is used to **display output** to the console.

Syntax:

```
print(value1, value2, ..., sep=' ', end='\n')
```

Example:

```
print("Python", "is", "user friendly language", sep="-", end="!!!\n")
```

Handling Different Data Types with input()

Expected Type	Conversion Required	Example
Integer	<code>int(input())</code>	<code>num = int(input("Enter number: "))</code>
Float	<code>float(input())</code>	<code>price = float(input("Enter price: "))</code>
Boolean	Custom logic	<code>val = input("Yes or No: ") == "Yes"</code>
List (basic)	<code>input().split()</code>	<code>items = input("Enter items: ").split()</code>

3 a. Describe four different types of operators and provide examples of expressions using each type

In Python, operators are special symbols that perform operations on variables and values. Operators are used to manipulate the value of operands.

Arithmetic Operators:

Arithmetic operators in Python are used to perform mathematical operations such as addition, subtraction, multiplication, and more.

1. Arithmetic Operators

Used to perform mathematical operations.

Operator	Description	Example	Result
+	Addition	<code>5 + 3</code>	8
-	Subtraction	<code>10 - 4</code>	6
*	Multiplication	<code>6 * 7</code>	42
/	Division	<code>8 / 2</code>	4.0

//	Floor Division	<code>9 // 2</code>	4
%	Modulus	<code>10 % 3</code>	1
**	Exponentiation	<code>2 ** 3</code>	8

```
a = 10
```

```
b = 3
```

```
print("Addition:", a + b)
```

```

print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Floor Division:", a // b)
print("Modulus:", a % b)
print("Exponentiation:", a ** b)

```

2. Relational (Comparison) Operators

Used to compare two values.

Operator	Description	Example	Result
<code>==</code>	Equal to	<code>5 == 5</code>	True
<code>!=</code>	Not equal to	<code>4 != 5</code>	True
<code>></code>	Greater than	<code>7 > 3</code>	True
<code><</code>	Less than	<code>2 < 6</code>	True
<code>>=</code>	Greater or equal to	<code>7 >= 7</code>	True
<code><=</code>	Less or equal to	<code>4 <= 9</code>	True

`a = 10`

`b = 5`

```

print("a == b:", a == b)
print("a != b:", a != b)
print("a > b:", a > b)
print("a < b:", a < b)
print("a >= b:", a >= b)
print("a <= b:", a <= b)

```

3. Logical Operators

Used to combine conditional statements.

Operator	Description	Example	Result
<code>and</code>	Logical AND	True and False	False
<code>or</code>	Logical OR	True or False	True

Operator	Description	Example	Result
not	Logical NOT	not True	False

x = 10

print(x > 5 and x < 20) # True

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example	Equivalent To
=	Assign	x = 5	x = 5
+=	Add and assign	x += 3	x = x + 3
-=	Subtract and assign	x -= 2	x = x - 2
*=	Multiply and assign	x *= 4	x = x * 4
/=	Divide and assign	x /= 2	x = x / 2
//=	Floor divide and assign	x //= 2	x = x // 2
%=	Modulus and assign	x %= 3	x = x % 3
**=	Power and assign	x **= 2	x = x ** 2

a = 10

a += 5 # a = a + 5

print("After += :", a)

a -= 3 # a = a - 3

print("After -= :", a)

a *= 2 # a = a * 2

print("After *= :", a)

a /= 4 # a = a / 4

print("After /= :", a)

a //= 2 # a = a // 2

print("After //= :", a)

a %= 2 # a = a % 2

print("After %= :", a)

```
a **= 3 # a = a ** 3  
print("After **= :", a)
```

5. Membership Operators

Used to test if a value is in a sequence.

Operator	Description	Example	Result
in	Value is present	'a' in 'apple'	True
not in	Value is not present	'x' not in 'apple'	True

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

```
print(3 in my_list) # Output: True
```

```
print(10 in my_list) # Output: False
```

```
print(6 not in my_list) # Output: True
```

6. Identity Operators

Used to compare object identities.

Operator	Description	Example	Result
is	Same identity	x is y	True/False
is not	Not same identity	x is not y	True/False

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

```
b = a
```

```
print(a is b) # True
```

```
print(a == b) # True (values are equal)
```

3 b. What is the order of evaluation in Python expressions? Explain how parentheses can be used to alter the default order of evaluation.

In Python, expressions are evaluated based on operator precedence and associativity. Operators with higher precedence are evaluated before those with lower precedence. If operators have the same precedence, their associativity (left-to-right or right-to-left) determines the order.

Python Operator Precedence Table

Precedence Level	Operator(s)	Description	Associativity	Example	Evaluation Result
1 (Highest)	()	Parentheses — overrides all	—	$(2 + 3) * 4$	$5 * 4 = 20$
2	**	Exponentiation	Right to Left	$2^{3^{**}2}$	$2^{**}(3^{**}2) = 512$
3	+x, -x, ~x	Unary plus, minus, bitwise NOT	Right to Left	$-5 + 3$	-2
4	*, /, //, %	Multiplication, division, etc.	Left to Right	$10 + 6 / 2$	$10 + 3 = 13.0$
5	+, -	Addition, subtraction	Left to Right	$10 - 3 + 1$	8
6	<<, >>	Bitwise shift operators	Left to Right	$4 << 1$	8
7	&	Bitwise AND	Left to Right	$5 \& 3$	1
8	^	Bitwise XOR	Left to Right	$5 \wedge 3$	6
9	,		Bitwise OR	Left to Right	5
10	==, !=, >, <, >=, <=	Comparison operators	Left to Right	$5 > 3$ and $2 < 4$	True
11	not	Logical NOT	Right to Left	not True	False
12	and	Logical AND	Left to Right	True and False	False
13	or	Logical OR	Left to Right	False or True	True
14	if else	Conditional expressions	Right to Left	x if cond else y	Depends on condition
15 (Lowest)	=, +=, -= etc.	Assignment operators	Right to Left	$x = 5 + 3$	$x = 8$

4 a. Write a Python program that takes a list of numbers as input. Check whether each number is even or odd and print the result.

```
input_string = input("Enter numbers separated by commas: ")
numbers = [int(num.strip()) for num in input_string.split(",")]
results = []
```



```

for number in numbers:
    if number % 2 == 0:
        results.append(f"{number} is Even")
    else:
        results.append(f"{number} is Odd")
print("\nResults:")
for result in results:
    print(result)

```

4 b. Explain the purpose and usage of the break, continue and pass statements in Python. Provide separate code examples for each.

break Statement

Purpose: Used to exit a loop immediately, even if the loop condition is still True.

Example

```

for i in range(1, 11):
    if i == 6:
        break
    print(i)

```

continue Statement

Purpose: Used to skip the rest of the code in the current iteration and move to the next iteration of the loop.

Example

```

for i in range(1, 6):
    if i % 2 == 0:
        continue
    print(i)

```

pass Statement

Purpose: Used as a placeholder when a statement is syntactically required but you don't want to write any code yet. Prevents the program from throwing an error when the block is empty.

Example

```

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

```

5a. Describe how to create strings and access individual characters using positive and negative indexing. Provide code to check for string palindrome

Creating Strings:

In Python, strings can be created by enclosing characters in single ('), double ("), or triple quotes (''' or ''').

Examples of string creation

```
str1 = 'Hello'
```

```
str2 = "Python"
```

```
str3 = """Multi-line String"""
```

Accessing Characters in a String

Python strings are indexed — you can access individual characters using:

Positive indexing (from left to right, starting at 0)

Negative indexing (from right to left, starting at -1)

Example

```
text = "Python"
```

```
print(text[0])
```

```
print(text[2])
```

```
print(text[-1])
```

```
print(text[-3])
```

Python Code to Check Palindrome:

```
word = input("Enter a string: ")
```

```
word = word.lower()
```

```
if word == word[::-1]:
```

```
    print(f'{word}' is a palindrome.)
```

```
else:
```

```
    print(f'{word}' is not a palindrome.)
```

5 b. Write a Python program that:

Creates a dictionary to store the prices of three different fruits (e.g., "apple": 0.50, "banana": 0.25, "orange" 0.75)

Accesses and prints the price of one of the fruits using its key

Adds a new fruit and its price to the dictionary

Updates the price of an existing fruit in the dictionary.

Iterates through the modified dictionary and prints each fruit and its updated price.

```
fruit_prices = {  
    "apple": 0.50,  
    "banana": 0.25,  
    "orange": 0.75  
}
```

```
print("Price of apple:", fruit_prices["apple"])
```

```
fruit_prices["mango"] = 1.00
```

```
fruit_prices["banana"] = 0.30
print("\nUpdated Fruit Prices:")
for fruit, price in fruit_prices.items():
    print(f'{fruit.capitalize(): <10}: ${price:.2f}')
```

6 a. Explain the purpose and syntax of list comprehensions in Python. Write a program to generate a list of squares of numbers from 1 to 10 using list comprehension.

Purpose of List Comprehensions in Python

List Comprehension provides a concise, readable, and efficient way to create lists in Python. It allows you to generate a new list by applying an operation to each item in an existing iterable (like a list or range), all in a single line of code.

Syntax of List Comprehension:

```
new_list = [expression for item in iterable if condition]
```

Python Program: Generate Squares from 1 to 10

```
squares = [x**2 for x in range(1, 11)]
print("List of squares from 1 to 10:", squares)
```

6 b. How can slicing be used to extract a portion of a tuple? Provide examples of slicing with positive and negative indices, as well as specifying a Step.

Slicing a Tuple in Python

Slicing allows you to extract a portion (subtuple) from a tuple using the [start:stop:step] syntax, just like with lists and strings.

General Slicing Syntax:

```
tuple[start : stop : step]
```

Example

```
my_tuple = (10, 20, 30, 40, 50, 60, 70)
```

Slicing with Positive Indices:

```
print(my_tuple[1:4])
```

Slicing with Negative Indices:

```
print(my_tuple[-5:-2])
```

Slicing with a Step:

```
print(my_tuple[0:7:2])
```

7 a. Explain the different types of function arguments in Python with examples.

Types of Function Arguments in Python

In Python, functions can accept different types of arguments to give flexibility in how they are called and used.

Keyword Arguments

Passed using parameter names, so order doesn't matter.

Increases readability and avoids confusion.

Example

```
greet(age=25, name="Alice")
```

Default Arguments

Provide default values in the function definition.

If the caller doesn't provide a value, the default is used.

Example

```
def greet(name, age=18):  
    print(f"Hello {name}, you are {age} years old.")  
greet("Bob")      # Uses default age  
greet("Alice", 30) # Overrides default
```

Positional Arguments

Passed in the correct position (order matters).

Must match the order of parameters in the function definition.

Example

```
def greet(name, age):  
    print(f"Hello {name}, you are {age} years old.")  
greet("Alice", 25)
```

Variable-Length Arguments (*args)

Used when you don't know how many positional arguments will be passed.

Collected into a tuple.

Example

```
def total(*numbers):  
    print("Sum:", sum(numbers))  
total(10, 20, 30)
```

7 b. What is the lambda function in Python? How does it differ from a regular function?

A lambda function in Python is a small, anonymous (unnamed) function defined using the lambda keyword. It is often used for short, simple operations, especially when a full function definition is unnecessary.

Syntax of a Lambda Function:

lambda arguments: expression

Example

```
square = lambda x: x * x
```

```
print(square(5))
```

Lambda vs Regular Function

Feature	Lambda Function	Regular Function
Defined using	lambda keyword	def keyword
Name	Anonymous (can be assigned to a name)	Has a function name
Return	Implicit return	Must use return explicitly
Body	Single expression only	Can contain multiple lines & statements
Use Case	Short, simple tasks (often inline)	Complex logic and reuse

8 a. What are namespaces in Python? Provide an example illustrating how it prevents naming conflicts when working modules and functions.

A namespace in Python is a container that holds a mapping between names (identifiers) and objects (variables, functions, classes, etc.). It ensures that names are unique and helps prevent naming conflicts.

Types of Namespaces in Python:

Namespace Type	Scope
Built-in	Automatically created; includes functions like print(), len()
Global	Variables defined at the top level of a script or module
Local	Variables defined inside a function
Enclosed (nonlocal)	Variables in enclosing (outer) functions

Example: Avoiding Conflicts Using Namespaces with Modules

Module:

```
math.py  
  
import math  
  
print(math.sqrt(16))
```

function:

```
def sqrt(x):  
    return x ** 0.5
```

Example: Local vs Global Namespace

```
x = 10  
def example():  
    x = 5  
    print("Inside function:", x)  
example()  
print("Outside function:", x)
```

8 b. Create a Python package with two modules one for math operations and one for string operations. Demonstrate how to import and use these modules.

Folder Structure:

```
my_package/  
|  
├── __init__.py  
├── math_operations.py  
└── string_operations.py
```

Create the Package Folder

Create a folder named my_package.

Create __init__.py

Inside my_package, add an empty file named __init__.py to make it a Python package.

Create math_operations.py

```
def add(a, b):  
    return a + b  
def subtract(a, b):  
    return a - b  
def multiply(a, b):  
    return a * b  
def divide(a, b):  
    if b != 0:
```

```
    return a / b
    return "Cannot divide by zero"
Create string_operations.py
```

```
def to_upper(s):
    return s.upper()
def to_lower(s):
    return s.lower()
def reverse_string(s):
    return s[::-1]
```

Using the Package in Another Script

```
from my_package import math_operations, string_operations
print("Addition:", math_operations.add(5, 3))
print("Division:", math_operations.divide(10, 2))
print("Uppercase:", string_operations.to_upper("hello"))
print("Reversed:", string_operations.reverse_string("Python"))
```

9 a. Explain the difference between class variables and object variables in Python How are they defined and accessed?

Key Differences:

Class Variables (Static Variables)

Shared among all instances of the class.

Defined inside the class, but outside any method.

Accessed using the class name or instance name.

Object Variables (Instance Variables)

Unique to each instance (object).

Defined using self inside methods, usually in `__init__()`.

Accessed using the object/instance name.

Example

```
class Student:
    school_name = "GVP College"
    def __init__(self, name, age)
        self.name = name
        self.age = age
s1 = Student("Alice", 20)
s2 = Student("Bob", 22)
print(s1.school_name)
print(s2.school_name)
```

```

print(s1.name, s1.age)
print(s2.name, s2.age)
Student.school_name = "ABC University"
print(s1.school_name)
print(s2.school_name)
s1.age = 21
print(s1.age)
print(s2.age)

```

9 b. What is method overriding in Python? Write a Python program to demonstrate method overriding using inheritance.

Method overriding in Python is a feature of inheritance where a subclass provides its own implementation of a method that is already defined in its parent class.

The method in the child class has the same name, parameters, and signature as the one in the parent class. When called on a child class object, the child's version overrides the parent's method.

Program to Demonstrate Method Overriding using inheritance

```

class Animal:
    def sound(self):
        print("Animals make sounds.")
class Dog(Animal):
    def sound(self):
        print("Dog barks.")
class Cat(Animal):
    def sound(self):
        print("Cat meows.")
a = Animal()
d = Dog()
c = Cat()
a.sound()
d.sound()
c.sound()

```

10 a. Explain the role of the try, except, and finally blocks in Python's exception handling mechanism with an example program.

Roles of Each Block:

Block	Purpose
try	Code that might raise an exception is placed here
except	Code that runs if an exception occurs in the try block
finally	Code that always runs, whether an exception occurred or not

Example Program:

```
def divide_numbers(a, b):
    try:
        result = a / b
        print("Result:", result)
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
    finally:
        print("Execution complete.")
divide_numbers(10, 2)
print()
divide_numbers(5, 0)
```

10 b. What are the different modes for opening files in Python. Write a Python program to reverse the content of the file.

Python's built-in open() function is used to work with files. It supports several modes:

Common File Modes

Mode	Description
'r'	Read (default) – file must exist
'w'	Write – creates new file or overwrites
'a'	Append – adds content to end of file
'x'	Create – creates new file, error if exists
'b'	Binary mode (e.g., 'rb', 'wb')
't'	Text mode (default)
'r+'	Read and write
'w+'	Write and read (overwrites)
'a+'	Append and read

Program:

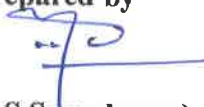
```
def reverse_file_content(input_file, output_file):
    try:
        with open(input_file, 'r') as infile:
            content = infile.read()
            reversed_content = content[::-1]
        with open(output_file, 'w') as outfile:
            outfile.write(reversed_content)
        print(f"Reversed content written to '{output_file}'")
```

```
except FileNotFoundError:
    print(f'Error: '{input_file}' not found.')
except Exception as e:
    print("An error occurred:", e)
reverse_file_content('input.txt', 'reversed_output.txt')
```

Verified by


(Mr.G.Appajee)

Prepared by


(Dr.S.Sumahasan)