UNIT - 1

1. What is Python? Discuss the brief history of Python Programming Language.

Ans:

- Python is a high-level, interpreted programming language known for its simplicity and readability. Python supports multiple programming paradigms, including object-oriented, imperative, and functional programming styles, making it versatile for various applications. Python is widely used in many fields, including web development (with frameworks like Django and Flask), data science (using libraries like Pandas, NumPy, and Matplotlib), artificial intelligence (using TensorFlow, PyTorch), automation, and more.
- Python, one of the most popular programming languages today, has a rich history of development and evolution. From its inception in the late 1980s to its current status as a versatile and powerful language, Python's version history reflects the language's adaptability and the community's dedication to improvement. This article explores the significant milestones in Python's version history, highlighting the key features and enhancements introduced with each major release.

History of Python:

- Python is a widely used general-purpose, high-level programming language. It was initially designed by **Guido van Rossum** in **1991** and developed by Python Software Foundation. It was mainly developed to emphasize code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Who Invented Python?

- In the late 1980s, history was about to be written. It was that time when working on Python started. Soon after that, Guido Van Rossum began doing its application-based work in December of 1989 at Centrum Wiskunde & Informatica (CWI) which is situated in the Netherlands. It was started as a hobby project because he was looking for an interesting project to keep him occupied during Christmas.
- The programming language in which Python is said to have succeeded is ABC Programming Language, which had interfacing with the Amoeba Operating System and had the feature of exception handling. He had already helped create ABC earlier in his career and had seen some issues with ABC but liked most of the features. After that what he did was very clever.
- He had taken the syntax of ABC, and some of its good features. It came with a lot of complaints too, so he fixed those issues completely and created a good scripting language that had removed all the flaws.

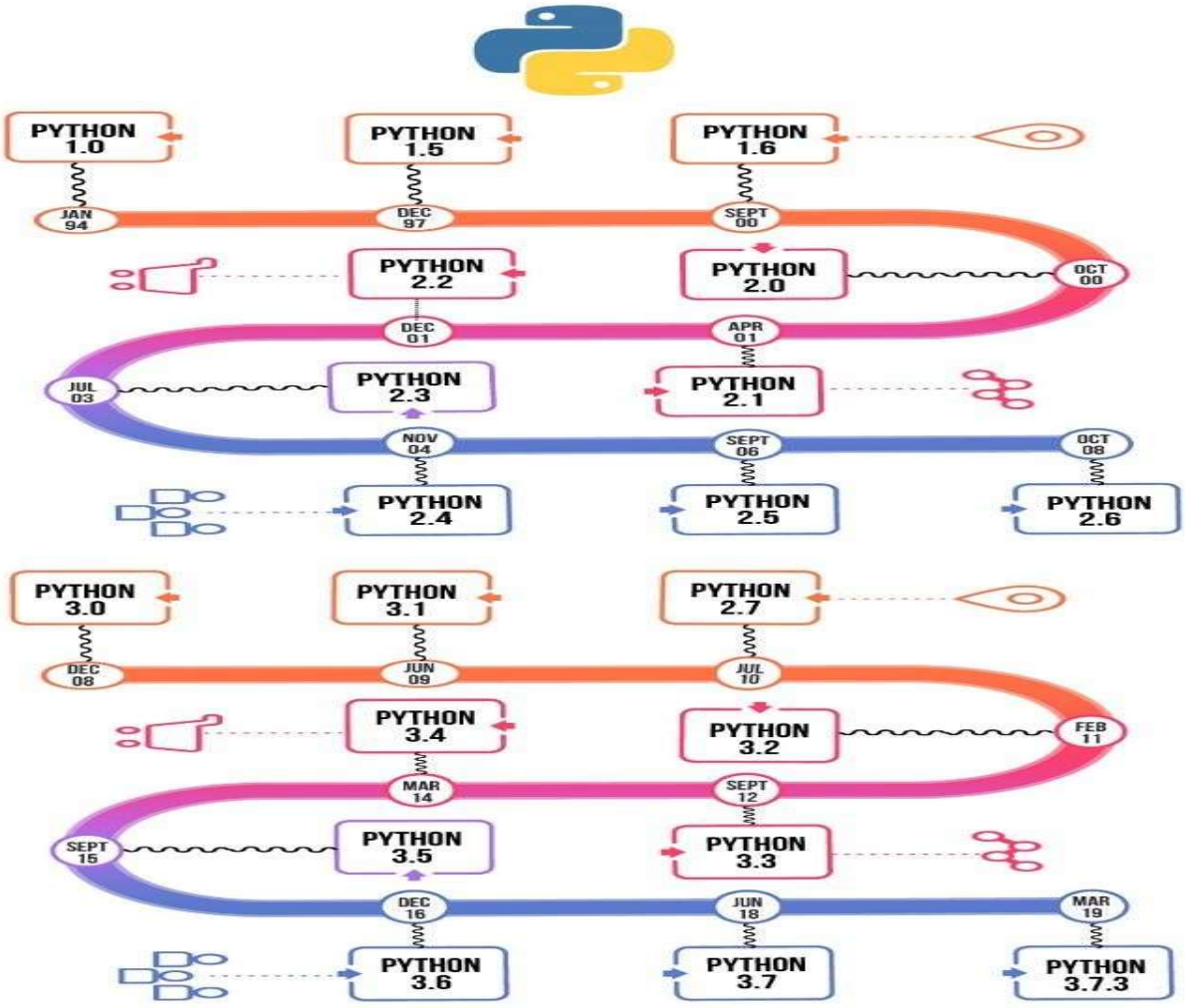
Why Python is called Python?

- The inspiration for the name came from the BBC's TV Show – ' **Monty Python's Flying Circus'**, as he was a big fan of the TV show and also he wanted a short, unique and slightly mysterious name for his invention and hence he named it Python! He was the "Benevolent dictator for life" (BDFL) until he stepped down from the position as the leader on 12th July 2018. For quite some time he used to work for Google, but currently, he is working at Dropbox.

Evolution of Python

- The language was finally released in 1991. When it was released, it used a lot fewer codes to express the concepts, when we compare it with Java, C++ & C.
- Its design philosophy was quite good too. Its main objective is to provide code readability and advanced developer productivity.
- When it was released, it had more than enough capability to provide classes with inheritance, several core data types of exception handling and functions.

Following are the illustrations of different versions of Python along with the timeline.



- The two of the most used versions has to Python **2.x & 3.x. Python 3.12.1 is the latest stable version.**
- For various purposes such as developing, scripting, generation, and software testing, this language is utilized. Due to its elegance and simplicity, top technology organizations like Dropbox, Google, Quora, Mozilla, Hewlett-Packard, Qualcomm, IBM, and Cisco have implemented Python.
- Python has come a long way to become the most popular coding language in the world. Python has just turned 30 and just recently at pycon22(python conference) a new feature was released by Anaconda foundation it's known as **pyscript** with this now python can be written and run in the browser like javascript which was previously not possible, but it still has that unknown charm & X factor.
- Python has been an inspiration for many other coding languages such as Ruby , Cobra, Boo, CoffeeScript ECMAScript, Groovy, Swift, Go, OCaml, Julia , etc.

Conclusion

- Python history shows how it has grown to become a key player in the programming world due to its ease of use and strong community support.

2. List out and Explain the features of Python Programming.

Ans :

- **Python** is a dynamic, high-level, free open source, and interpreted programming language.
- It supports object-oriented programming as well as procedural-oriented programming.

- In Python, we don’t need to declare the type of variable because it is a dynamically typed language.
- For example, x = 10 Here, x can be anything such as String, int, etc. In this article we will see what characteristics describe the python programming language

Features in Python

In this section we will see what are the features of Python programming language:

1. Free and Open Source

- Python is freely available for everyone. It is freely available on its official website www.python.org.
- It has a large community across the world that is dedicatedly working towards make new python modules and functions.
- Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any penny."

2. Easy to Read

- As you will see, learning Python is quite simple. As was already established, Python’s syntax is really straightforward.
- The code block is defined by the indentations rather than by semicolons or brackets.

3. Object-Oriented Language

- One of the key features of Python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, object encapsulation, etc.

4. High-Level Language

- Python is a high-level language. When we write programs in Python, we do not need to remember the system architecture, nor do we need to manage the memory.

5. Large Community Support

- Python has gained popularity over the years. Our questions are constantly answered by the enormous StackOverflow community. These websites have already provided answers to many questions about Python, so Python users can consult them as needed.

6. Python is a Portable language

- Python language is also a portable language. For example, if we have Python code for Windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.

7. Python is an Integrated language

- Python is also an Integrated language because we can easily integrate Python with other languages like C, C++, etc.

8. Interpreted Language:

- Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile Python code

this makes it easier to debug our code. The source code of Python is converted into an immediate form called **bytecode**.

9. Large Standard Library

- Python has a large standard library that provides a rich set of modules and functions so you do not have to write your own code for every single thing.
- There are many libraries present in Python such as regular expressions, unit-testing, web browsers, etc.

10. Dynamically Typed Language

- Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

3. Explain the difference between the different versions of Python. Ans :

Differences between Python 2.x and Python 3.x

Here, we will see the differences in the following libraries and modules:

1. Division operator
2. print function
3. Unicode
4. xrange
5. Error Handling

6. _future_ module

Python Division operator

If we are porting our code or executing python 3.x code in python 2.x, it can be dangerous if integer division changes go unnoticed (since it doesn't raise any error). It is preferred to use the floating value (like 7.0/5 or 7/5.0) to get the expected result when porting our code. For more information about this you can refer to Division Operator in Python.

Code :

```
print(7 / 5 )
print(-7 / 5)
'''
```

Output in Python 2.x

```
1
-2
```

Output in Python 3.x :

```
1.4
-1.4
'''
```

Print Function in Python

This is the most well-known change. In this, the print keyword in Python 2.x is replaced by the print() function in Python 3.x. However, parentheses work in Python 2 if space is added

after the print keyword because the interpreter evaluates it as an expression. You can refer to Print Single Multiple Variable Python for more info about this topic.

In this example, we can see, if we don't use parentheses in python 2.x then there is no issue but if we don't use parentheses in python 3.x, we get SyntaxError.

Code:

```
print 'Hello, Geeks'         # Python 3.x doesn't support

print('Hope You like these facts')

'''
```

Output in Python 2.x :

```
Hello, Geeks

Hope  You  like  these  facts
```

Output in Python 3.x :

```
File "a.py", line 1

    print 'Hello, Geeks'

                        ^

SyntaxError:  invalid  syntax
'''
```

Unicode In Python

In Python 2, an implicit str type is ASCII. But in Python 3.x implicit str type is Unicode. In this example, difference is shown between both the version of Python with the help of code and also the output in Python comments.

Code:

```
print(type('default string '))
print(type(b'string with b '))
'''
Output in Python 2.x (Bytes is same as str)
<type 'str'>
<type 'str'>
Output in Python 3.x (Bytes and str are different)
<class 'str'>
<class 'bytes'>
'''
```

Python 2.x also supports Unicode

```
print(type('default string '))
print(type(u'string with b '))
'''
Output in Python 2.x (Unicode and str are different)
<type 'str'>
<type 'unicode'>
Output in Python 3.x (Unicode and str are same)
<class 'str'>
<class 'str'>
'''
```

xrange vs range() in both versions

xrange() of Python 2.x doesn't exist in Python 3.x. In Python 2.x, range returns a list i.e. range(3) returns [0, 1, 2] while xrange returns a xrange object i. e., xrange(3) returns iterator object which works similar to Java iterator and generates number when needed.

If we need to iterate over the same sequence multiple times, we prefer range() as range provides a static list. xrange() reconstructs the sequence every time. xrange() doesn't support slices and other list methods. The advantage of xrange() is, it saves memory when the task is to iterate over a large range.

In Python 3.x, the range function now does what xrange does in Python 2.x, so to keep our code portable, we might want to stick to using a range instead. So Python 3.x's range function *is* xrange from Python 2.x. You can refer to this article for more

understanding range() vs xrange() in Python.

Code:

```
for x in xrange(1, 5):
    print(x),
for x in range(1, 5):
    print(x),
'''
```

Output in Python 2.x

1 2 3 4 1 2 3 4

Output in Python 3.x

NameError: name 'xrange' is not defined

'''

Error Handling

There is a small change in error handling in both versions. In latest version of Python, ‘as’ keyword is optional.

Code : try:

    trying_to_check_error

except NameError, err:

Would not work in Python 3.x

    print err, 'Error Caused'

'''

Output in Python 2.x:

name 'trying_to_check_error' is not defined Error Caused

Output in Python 3.x :

File "a.py", line 3

    except NameError, err:

                      ^

SyntaxError: invalid syntax

'''

In this example, error handling is shown without the use of “as” keyword.

Code : try:

```
    trying_to_check_error
except NameError as err:
    print(err, 'Error Caused')
```

```
# code without using as keyword
try:
    trying_to_check_error
# 'as' is optional in Python 3.10
except NameError:
    print('Error Caused')
'''
```

Output in Python 2.x:

(NameError("name 'trying_to_check_error' is not defined",), 'Error Caused')

Output in Python 3.x :

name 'trying_to_check_error' is not defined Error Caused

'''

__future__module in Python

This is basically not a difference between the two versions, but a useful thing to mention here. The idea of the __future__module is to help migrate to Python 3.x.

If we are planning to have Python 3.x support in our 2.x code, we can use _future_ imports in our code.

For example, in the Python 2.x code below, we use Python 3.x's integer division behavior using the __future__module.

Code:

```
# In below python 2.x code, division works

# same as Python 3.x because we use __future__

from __future__import division

print(7 / 5)

print(-7 / 5)
```

Output:

1.4

-1.4

Feature	**Python 2.x**	**Python 3.x**
Print statement	print "Hello"	print("Hello")
Integer division	3 / 2 = 1	3 / 2 = 1.5

String handling	ASCII strings by default	Unicode strings by default
Range function	range() returns a list	range() returns an integer
End of life	No updates after 2020	Actively maintained and updated
Libraries	Some old libraries support it	Modern libraries focus on this

4. Write down the steps for instaling Python / Python IDLE.

Ans : To install Python on Windows, you need to download the Python installer from the official Python website and run it on your system. The installation process is straightforward and includes options to add Python to your system PATH.

Steps to Install Python 3 on Windows:

1. **Download the Installer:**

- Visit the official Python website: python.org.
- Go to the Downloads section and click on "Download Python 3.x.x" (the latest version).

2. **Run the Installer:**

- Locate the downloaded installer file (python-3.x.x.exe) and run it.

3. **Select Installation Options:**

- Check the box that says "Add Python to PATH" at the bottom of the installer window.
- Choose "Install Now" for a standard installation or "Customize Installation" to choose

specific features and installation location.

4. **Customize Installation (Optional):**

- If you chose "Customize Installation," select optional features like pip, tcl/tk, and documentation.
- Choose the installation location or accept the default.

5. **Complete the Installation:**

- The installer will copy the necessary files and set up Python on your system.
- Once the installation is complete, you can close the installer.

Steps to Verify Python Installation on Windows:

1. **Open Command Prompt:**

- Press Win + R, type cmd, and press Enter to open the Command Prompt.

2. **Check Python Version:**

- Type python --version and press Enter.
- You should see the installed Python version, e.g., Python 3.x.x.

3. **Check pip Version:**

- Type pip --version and press Enter.
- This verifies that pip, the Python package installer, is also installed correctly.

Environment Variables for Python on Windows:

Environment variables are used to configure the environment in which processes run. For Python, you often need to set the PATH environment variable so that

you can run Python and pip from the command line. This ensures that Python executables and scripts can be accessed from any command line prompt without specifying their full path.

Steps to Configure Python Path on Windows:

1. **Add Python to PATH During Installation:**

- When running the Python installer, ensure you check the box that says "Add Python to PATH."

2. **Manually Add Python to PATH:**

- Open the Start menu, search for "Environment Variables," and select "Edit the system environment variables."
- In the System Properties window, click on the "Environment Variables" button.
- Under "System variables," find the Path variable and click "Edit."
- Click "New" and add the path to the Python installation directory (e.g., C:\Python39) and the Scripts directory (e.g., C:\Python39\Scripts).
- Click "OK" to close all windows.

3. **Verify PATH Configuration:**

- Open Command Prompt.
- Type python and press Enter to start the Python interpreter. If Python starts, the PATH is configured correctly.
- Type exit() to exit the Python interpreter.
- Type pip and press Enter to verify that pip can be called from the command line.

5. Explain simple data types, complex data types in Python. OR List out all the data types of python programming langauge.

Ans : Python has a variety of built-in data types that are used to store different kinds of values. These data types can be categorized into simple (primitive) and complex data types.

Simple (Primitive) Data Types:

1. Numeric Types:

- `int` (Integer): Represents whole numbers, positive or negative, without a decimal point.

  - Example: `10`, `-5`, `0`

- `float` (Floating-Point Number): Represents real numbers with decimal points.

  - Example: `10.5`, `-3.14`, `2.0`

- `complex` (Complex Numbers): Represents complex numbers with real and imaginary parts.

  - Example: `2 + 3j`, `1 - 4j`

2. Text Type:

- `str` (String): Used to represent textual data. Strings are sequences of characters enclosed in single (`'`) or double quotes (`"`).

- Example: `'Hello'`, `"Python"`, `'123'`

3. Boolean Type:

- `bool` (Boolean): Represents `True` or `False` values, used for logical operations and conditions.

- Example: `True`, `False`

4. None Type:

- `NoneType`: Represents the absence of a value or a null value. Only one value, `None`, belongs to this type.

- Example: `None`

Complex Data Types:

1. Sequence Types:

- `list`: An ordered, mutable collection of items (can contain different data types). Lists are defined using square brackets `[]`.

- Example: `[1, 2, 3]`, `["apple", "banana", "cherry"]`

- `tuple`: An ordered, immutable collection of items. Tuples are defined using parentheses `()`.

- Example: `(1, 2, 3)`, `("apple", "banana")`

- `range`: Represents a sequence of numbers and is commonly used for

looping a specific number of times.

- Example: `range(0, 10)`

2. Mapping Type:

- `dict` (Dictionary): A collection of key-value pairs. It is unordered, mutable, and indexed by keys. Dictionaries are defined using curly braces `{}`.

- Example: `{"name": "John", "age": 30}`, `{1: "one", 2: "two"}`

3. Set Types:

- `set`: An unordered collection of unique elements. Sets are mutable and defined using curly braces `{}` or the `set()` function.

- Example: `{1, 2, 3}`, `set([1, 2, 3, 4])`

- `frozenset`: An immutable version of a set. It is defined using the `frozenset()` function.

- Example: `frozenset([1, 2, 3])`

4. Binary Types:

- `bytes`: Represents a sequence of bytes (immutable). It is used for binary data.

- Example: `b'hello'`

- `bytearray`: A mutable sequence of bytes. It is similar to `bytes` but allows modifications.

  - Example: `bytearray(b'hello')`

- `memoryview`: Provides a view of the memory of an existing `bytes` or `bytearray` object without copying it.

  - Example: `memoryview(bytearray(b'abc'))`

Summary Table of Python Data Types:

Category	Data Type	Description
Numeric	`int`	Integer (whole numbers)
	`float`	Floating-point numbers (decimals)
	`complex`	Complex numbers (real + imaginary parts)
Text	`str`	Strings (text data)
Boolean	`bool`	Boolean values (`True` or `False`)
None	`NoneType`	Represents a null or empty value
Sequence	`list`	Ordered, mutable collection of elements
	`tuple`	Ordered, immutable collection of elements
	`range`	Represents a range of numbers
Mapping	`dict`	Collection of key-value pairs
Set	`set`	Unordered collection of unique elements
	`frozenset`	Immutable set
Binary	`bytes`	Immutable sequence of bytes
	`bytearray`	Mutable sequence of bytes
	`memoryview`	Memory view object (access binary data without copying)

Notes:

- Simple data types like `int`, `float`, `str`, `bool`, and `None` represent single

values.

- Complex data types like `list`, `dict`, `set`, and `tuple` are collections of multiple values or objects.

These data types enable you to handle a wide range of programming scenarios efficiently in Python.

6. Explain String and its inbuilt functions (any 10).

Ans : Here are **10 built-in string functions in Python** with examples:

1. **len():** Returns the length of the string.

```
my_string = "Hello, World!"
print(len(my_string)) # Output: 13
```

2. **upper()**: Converts all characters in the string to uppercase.

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

3. **lower():** Converts all characters in the string to lowercase.

```
text = "HELLO"
print(text.lower()) # Output: hello
```

4. **strip()**: Removes leading and trailing whitespace (spaces, tabs, etc.) from the string.

```
text = "  Hello, World!  "
print(text.strip()) # Output: "Hello, World!"
```

5. **replace()**: Replaces occurrences of a substring with another substring.

```
text = "Hello, John!"
new_text = text.replace("John", "Jane")
print(new_text) # Output: Hello, Jane!
```

6. **split()**: Splits the string into a list of substrings based on a delimiter (by default, whitespace).

```
sentence = "Python is awesome"
words = sentence.split()
print(words) # Output: ['Python', 'is', 'awesome']
```

You can specify a custom delimiter:

```
text = "apple,banana,cherry"
fruits = text.split(',')
print(fruits) # Output: ['apple', 'banana', 'cherry']
```

7. **join()**: Joins elements of a list into a string, with a specified delimiter.

```
words = ["Python", "is", "awesome"]
sentence = " ".join(words)
print(sentence) # Output: Python is awesome
```

8. **startswith()**: Checks if the string starts with a specified prefix. Returns True if it does, otherwise False.

```
text = "Python programming"
print(text.startswith("Python")) # Output: True
print(text.startswith("Java")) # Output: False
```

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

```
text = "example.txt"
print(text.endswith(".txt"))     #     Output:     True
print(text.endswith(".pdf")) # Output: False
```

10. **find()**: Returns the lowest index of the substring if found, otherwise returns -1.

```
text = "Python programming"
index          =          text.find("gram")
print(index) # Output: 10


# If the substring is not found
index       =       text.find("Java")
print(index) # Output: -1
```

These built-in string functions are essential tools for manipulating text and data in Python. Python's string features offer a rich set of tools for developers.

7. Write down a Python Program to perform different operations on string.

Ans : Here's a Python program that demonstrates different operations on strings, including concatenation, slicing, changing case, finding a substring, and more:

```
# String Operations in Python
#  1.  String  Concatenation
string1 = "Hello"
string2 = "World"
concatenated_string  =  string1  +  "  "  +  string2
```

```
print("Concatenated String:", concatenated_string)

# 2. String Slicing

sliced_string = concatenated_string[0:5]  # Slicing 'Hello'

print("Sliced String:", sliced_string)

# 3. Changing Case

uppercase_string = concatenated_string.upper()  # Convert to uppercase

lowercase_string = concatenated_string.lower()  # Convert to lowercase

print("Uppercase String:", uppercase_string)

print("Lowercase String:", lowercase_string)


# 4. Replacing a Substring

replaced_string = concatenated_string.replace("World", "Python")

print("Replaced String:", replaced_string)


# 5. Splitting a String

split_string = concatenated_string.split()  # Splits on whitespace

print("Split String:", split_string)


# 6. Joining Strings

joined_string = "-".join(split_string)  # Join with a hyphen

print("Joined String:", joined_string)
```

```
# 7. Checking Prefix and Suffix

starts_with_hello = concatenated_string.startswith("Hello")

ends_with_python = concatenated_string.endswith("Python")

print("Starts with 'Hello':", starts_with_hello)

print("Ends with 'Python':", ends_with_python)



# 8. Finding a Substring

index_of_world = concatenated_string.find("World")

print("Index of 'World':", index_of_world)


# 9. Length of String

string_length = len(concatenated_string)

print("Length of Concatenated String:", string_length)

# 10. Removing Whitespace

string_with_spaces = "  Hello, World!  "

trimmed_string = string_with_spaces.strip()

print("Trimmed String:", """ + trimmed_string + """)
```

```
Concatenated String: Hello World

Sliced String: Hello
```

Uppercase String: HELLO WORLD

Lowercase String: hello world

Replaced String: Hello Python

Split String: ['Hello', 'World']

Joined String: Hello-World

Starts with 'Hello': True

Ends with 'Python': False

Index of 'World': 6

Length of Concatenated String: 11

Trimmed String: 'Hello, World!'

This program covers a variety of useful string operations in Python.

8. What is 'Local', 'Global' and 'Nonlocal' variables in Python. Explain with an example.

Ans : Python Global variables are those which are not defined inside any function and have a global scope whereas Python local variables are those which are defined inside a function and their scope is limited to that function only. In other words, we can say that local variables are accessible only inside the function in which it was initialized whereas the global variables are accessible throughout the program and inside every function.

Python Local Variables

Local variables in Python are those which are initialized inside a function and belong only to that particular function. It cannot be accessed anywhere outside the function. Let's see how to create a local variable.

Creating local variables in Python

Defining and accessing local variables

```
def f():
    # local variable
    s = "I love Geeksforgeeks"
    print(s)
# Driver code
f()
```

Output

I love Geeksforgeeks

If we will try to use this local variable outside the function then let's see what will happen.

```
def f():
    # local variable
    s = "I love Geeksforgeeks"
    print("Inside Function:", s)
# Driver code
f()
print(s)
```

Output:

NameError: name 's' is not defined

Python Global Variables

These are those which are defined outside any function and which are accessible throughout the program, i.e., inside and outside of every function. Let's see how to

create a Python global variable.

Create a global variable in Python

Defining and accessing Python global variables.

```
# This function uses global variable s
def f():
    print("Inside Function", s)
# Global scope
s = "I love Geeksforgeeks"
f()
print("Outside Function", s)
```

Output

```
Inside Function I love Geeksforgeeks
Outside Function I love Geeksforgeeks
```

The variable s is defined as the global variable and is used both inside the function as well as outside the function.

3. Nonlocal Variables

A **nonlocal variable** is used in nested functions. It allows you to modify variables in the enclosing (non-global) scope. The nonlocal keyword is used to refer to variables in the nearest enclosing function scope that are neither global nor local to the current function.

Example of Nonlocal Variable:

```
def outer_function():
    outer_var = "I am outer variable"  # Variable in the outer function
    def inner_function():
```

```
    nonlocal outer_var  # Refers to the outer function's variable
    outer_var = "Modified by inner function"  # Modifying the outer variable print("Inner:",
    outer_var)
inner_function()
print("Outer:", outer_var)  # Check if outer variable is modified
outer_function()
```

Comparision Basis	Global Variable	Local Variable
Definition	declared outside the functions	declared within the functions
Lifetime	They are created the execution of the program begins and are lost when the program is ended	They are created when the function starts its execution and are lost when the function ends
Data Sharing	Offers Data Sharing	It doesn't offers Data Sharing
Scope	Can be access throughout the code	Can access only inside the function
Parameters needed	parameter passing is not necessary	parameter passing is necessary
Storage	A fixed location selected by the compiler	They are kept on the stack
Value	Once the value changes it is reflected throughout the code	once changed the variable don't affect other functions of the program

9. Explain conditional statements and loops in Python with examples. Ans :

Conditional Statements in Python

Conditional statements in Python allow you to execute different blocks of code based on certain conditions. These include if, elif, and else statements.

Conditional Statements are statements in Python that provide a choice for the control flow based on a condition. It means that the control flow of the Python program will be decided based on the outcome of the condition.

1. **if Statement**

If the simple code of block is to be performed if the condition holds then the if statement is used. Here the condition mentioned holds then the code of the block runs otherwise not.

Syntax of If Statement:

```
if condition:
    # Statements to execute if
    # condition is true
```

Example :

```
x = 10
if x > 5:
    print("x is greater than 5")
```

2. **if-else Statement**

In a conditional if Statement the additional block of code is merged as an else statement which is performed when if condition is false.

Syntax of Python If-Else:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

Example :

```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is less than or equal to 5")
```

3. **if-elif-else Statement**

The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final “else” statement will be executed.

Example :

```
x = 7
if x > 10:
    print("x is greater than 10")
elif x > 5:
    print("x is greater than 5 but less than or equal to 10")
else:
```

```
print("x is less than or equal to 5")
```

Loops in Python

Loops allow you to execute a block of code multiple times. Python has two types of loops: **for loops** and **while loops**.

1. **for Loop**

   The for loop iterates over a sequence (like a list, tuple, string, or range) and executes the block of code for each element.

   For Loop Syntax:

```
for iterator_var in sequence:
    statements(s)
```

   Example :

```
# Loop through a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

   You can also loop over a range of numbers using the range() function.

   Example:

```
# Loop through a range of numbers for
i in range(1, 6):
    print(i)
```

2. while Loop

A while loop repeatedly executes a block of code as long as the condition remains True.

Example:

```
count = 1
while count <= 5:
    print("Count:", count)
    count += 1  # Increment the counter
```

3. **break and continue Statements**

- **break**: Exits the loop prematurely when a specific condition is met.
- **continue**: Skips the current iteration and moves to the next one.

Example using break:

```
for i in range(1, 10):
    if i == 5:
        break  # Exit the loop when i is 5
    print(i)
```

Example using continue:

```
for i in range(1, 6):
    if i == 3:
        continue  # Skip the iteration when i is 3
    print(i)
```

4. **Nested Loops**

Loops can be nested, meaning you can place one loop inside another.

Example:

```
for i in range(1, 4):
    for j in range(1, 4):
        print(f"i = {i}, j = {j}")
```

Summary

1. **Conditional Statements**: Control the flow of the program by checking conditions (if, elif, else).
2. **Loops**: Repeat blocks of code (for and while loops) until a certain condition is met.
3. **break**: Exits the loop early.
4. **continue**: Skips the rest of the loop iteration and moves to the next one.
5. **Nested Loops**: One loop inside another to create complex iterations.

These concepts allow for decision-making and repetition in Python, enabling flexible and dynamic programming.

10. List out and discuss all the operators of Python Programming Language with example.

Ans : In Python programming, Operators in general are used to perform operations on values and variables. These are standard symbols used for logical and arithmetic operations. In this article, we will look into different types of **Python operators.**

- OPERATORS: These are the special symbols. Eg- + , * , /, etc.
- OPERAND: It is the value on which the operator is applied.

Python supports several types of operators:

1. **Arithmetic Operators**

These operators are used to perform basic mathematical operations.

Operator	Description	Example
`+`	Addition	`3 + 2 = 5`
`-`	Subtraction	`5 - 2 = 3`
`*`	Multiplication	`4 * 3 = 12`
`/`	Division	`10 / 2 = 5.0`
`%`	Modulus (Remainder)	`10 % 3 = 1`
`**`	Exponentiation (Power)	`2 ** 3 = 8`
`//`	Floor Division (Quotient)	`9 // 2 = 4`

Example:

```
a = 10

b = 3

print(a + b)   # 13

print(a - b)   #    7

print(a * b)   # 30

print(a / b)   # 3.333

print(a % b)   # 1

print(a ** b)  # 1000 (10 to the power of 3)

print(a // b) # 3 (Floor Division)
```

2. **Comparison (Relational) Operators**

These operators compare two values and return True or False.

Operator	Description	Example
`==`	Equal to	`5 == 5` → `True`
`!=`	Not equal to	`5 != 2` → `True`
`>`	Greater than	`7 > 3` → `True`
`<`	Less than	`5 < 8` → `True`
`>=`	Greater than or equal to	`5 >= 5` → `True`
`<=`	Less than or equal to	`3 <= 5` → `True`

Example:

```
x = 7

y = 5

print(x == y)   # False

print(x != y)   #  True

print(x > y)    #   True

print(x < y)    #  False

print(x >= y)   #  True

print(x <= y)   # False
```

3. **Logical Operators**

These operators are used to combine conditional statements.

Operator	Description	Example
`and`	True if both are True	`(5 > 3) and (8 > 5)` → `True`
`or`	True if at least one is True	`(5 > 3) or (3 > 5)` → `True`
`not`	True if the operand is False	`not(5 > 3)` → `False`

Example:

a = True

b = False

print(a and b) # False

print(a or b) # True

print(not a) # False

4. **Assignment Operators**

These operators are used to assign values to variables.

Operator	Description	Example
`=`	Assigns the right operand to the left operand	`a = 5`
`+=`	Adds and assigns	`a += 3` (i.e., `a = a + 3`)
`-=`	Subtracts and assigns	`a -= 2` (i.e., `a = a - 2`)
`*=`	Multiplies and assigns	`a *= 3` (i.e., `a = a * 3`)
`/=`	Divides and assigns	`a /= 2` (i.e., `a = a / 2`)
`%=`	Modulus and assigns	`a %= 3` (i.e., `a = a % 3`)
`**=`	Exponentiation and assigns	`a **= 2` (i.e., `a = a ** 2`)
`//=`	Floor division and assigns	`a //= 2` (i.e., `a = a // 2`)

Example:

x = 5

x += 3 # Equivalent to x = x + 3

print(x) # 8

5. **Bitwise Operators**

These operators work on bits and perform bit-level operations.

Operator	Description	Example
`&`	Bitwise AND	`5 & 3` → `1` (0101 & 0011 = 0001)
`	`	`	Bitwise OR
`^`	Bitwise XOR	`5 ^ 3` → `6` (0101 ^ 0011 = 0110)
`~`	Bitwise NOT (Inverts bits)	`~5` → `-6` (bitwise inversion)
`<<`	Left shift	`5 << 1` → `10` (0101 becomes 1010)
`>>`	Right shift	`5 >> 1` → `2` (0101 becomes 0010)

Example:

```
a = 5  # 0101 in binary

b = 3  # 0011 in binary

print(a & b)  # 1 (0001 in binary)

print(a | b)  # 7 (0111 in binary)

print(a ^ b)  # 6 (0110 in binary)

print(~a)     # -6 (inverts the bits)
```

6. **Membership Operators**

   These operators are used to check if a value is a member of a sequence (such as a list, tuple, or string).

Operator	Description	Example
`in`	Returns True if the value is found in the sequence	`"a" in "apple"` → `True`
`not in`	Returns True if the value is not found in the sequence	`"b" not in "apple"` → `True`

Example:

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

print("apple" in fruits)      # True

print("grape" not in fruits)  # True
```

7. **Identity Operators**

   These operators are used to compare the memory locations of two objects.

Operator	Description	Example
`is`	Returns True if both variables refer to the same object	`x is y`
`is not`	Returns True if both variables refer to different objects	`x is not y`

Example:

```
x = ["apple", "banana"]

y = ["apple", "banana"]

z = x

print(x is z)       # True, because z refers to the same object as x

print(x is y)       # False, because x and y are different objects

print(x == y)       # True, because x and y have the same content
```

8. **Ternary (Conditional) Operator**

   This operator allows for a shorter version of the if-else statement.

   Syntax:

```
[on_true] if [condition] else [on_false]
```

Example:

```
a = 10
b = 20
min_value = a if a < b else b
print(min_value) # 10
```

Summary of Python Operators :

Type of Operator	Example Operators	Use
Arithmetic	`+`, `-`, `*`, `/`, `%`, `//`, `**`	Mathematical operations
Comparison (Relational)	`==`, `!=`, `>`, `<`, `>=`, `<=`	Compare values and return `True/False`
Logical	`and`, `or`, `not`	Combine multiple conditions
Assignment	`=`, `+=`, `-=`, `*=`	Assign and modify variable values
Bitwise	`&`, ``	`, `^`, `~`, `<<`, `>>` ``
Membership	`in`, `not in`	Check membership in sequences
Identity	`is`, `is not`	Compare memory locations of objects
Ternary	`a if condition else b`	Short `if-else` condition

These operators are the building blocks for performing calculations, logical checks, and managing data in Python.

11. What is the difference between 'break', 'continue' and 'pass' statements. Explain with example.

Ans : In Python, break, continue, and pass are control flow statements that are used within loops and conditional statements. Each serves a different purpose. Here's a detailed explanation of each, along with examples:

1. Break Statement in Python :

The break statement is used to exit a loop prematurely. When a break statement is encountered, the loop terminates immediately, and control is transferred to the first statement following the loop.

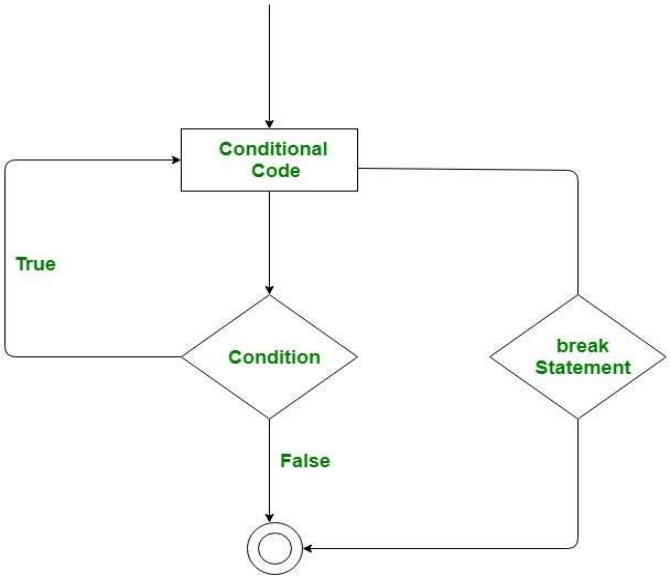
Syntax of Break Statement

```
for / while loop:
    # statement(s)
    if condition:
        break

    # statement(s)
# loop end
```

Working on Python Break Statement

The working of the break statement in Python is depicted in the following flowchart:



Example of break:

```
for i in range(10):
    if i == 5:
        break # Exit the loop when i equals 5
    print(i)
print("Loop exited.")
```

Output:

0

1

2

3

4

Loop exited.

In this example, the loop runs from 0 to 9, but when i equals 5, the break statement is executed, causing the loop to terminate early.

2. **continue Statement**

The continue statement is used to skip the current iteration of a loop and move to the next iteration. When continue is encountered, the remaining code in the loop for that iteration is skipped.

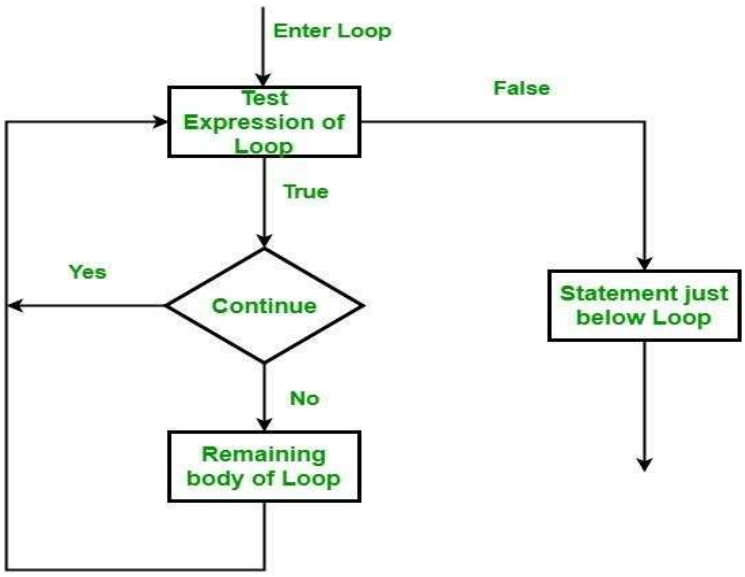
Syntax of Continue Statement

The continue statement in Python has the following syntax: for / while loop:

```
    #   statement(s)
    if condition:
continue
 # statement(s)
```

Working of Python Continue Statement

The working of the continue statement in Python is depicted in the following flowchart:



Example of continue:

```
for i in range(10):
    if i % 2 == 0:
        continue  # Skip the rest of the loop body for even numbers
    print(i)
print("Loop completed.")
```

Output:

1

3

5

7

9

Loop completed.

In this example, the loop iterates through numbers 0 to 9, but whenever i is even, the continue statement is executed, skipping the print(i) statement and moving to the next iteration.

3. pass Statement

The pass statement is a null operation. It is a placeholder that does nothing when executed. pass is typically used in situations where syntactically some code is required but no action is desired. It can be useful in function definitions, loops, or conditionals where you plan to implement logic later.

Syntax of Pass Statement

The pass statement in Python has the following syntax:

```
function/ condition / loop:
  pass
```

Example of pass:

```
for i in range(5):
    if i < 3:
        pass  # Do nothing for i less than 3
    else:
        print(i)
print("Loop  finished.")
```

Output:

3

4

Loop finished.

In this example, for values of i less than 3, the pass statement is executed, which does nothing. The loop continues to execute, and only when i is 3 or greater does the print(i) statement execute.

Summary of Differences :

Statement	Functionality	Usage
`break`	Exits the loop entirely	To terminate a loop early
`continue`	Skips the current iteration and continues with the next iteration	To skip specific cases in a loop
`pass`	Does nothing; a placeholder for future code	When syntactically required, but no action needed

These statements are essential for controlling the flow of loops in Python, allowing for flexible program logic.

UNIT – 2

1. What is a Function in Python. Explain with example.

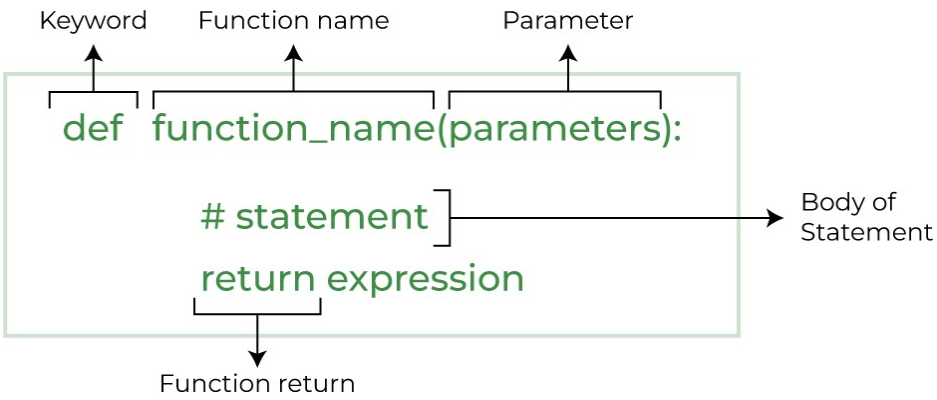
Ans : Python Functions is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

Some Benefits of Using Functions

- Increase Code Readability
- Increase Code Reusability

Python Function Declaration

The syntax to declare a function is:



Example:

```
# Define a function to add two numbers
def add_numbers(a, b):
```

```
    result = a + b
    return result
# Call the function
sum_result = add_numbers(5, 3)
print(sum_result)
```

Explanation:

1. The function add_numbers is defined with two parameters a and b.
2. Inside the function, it adds the values of a and b and stores the result in result.
3. The function then returns the result.
4. When the function is called using add_numbers(5, 3), it returns the sum of 5 and 3, which is 8.

Output:

8

Benefits of Using Functions:

- **Code Reusability**: Once a function is defined, it can be used multiple times.
- **Modularity**: Functions make code easier to manage and understand by splitting tasks into smaller parts.
- **Readability**: By using functions, the code becomes cleaner and easier to read.

In Python, functions can be classified into various types based on how they are defined and used. Here are the primary types of functions:

1. Built-in Functions

These are the functions that are provided by Python itself. You don't need to define these functions; you can use them directly.

Examples:

- print(): Prints the output.
- len(): Returns the length of an object.
- type(): Returns the type of the object.

```
print("Hello, World!")  # Output: Hello, World!
print(len([1, 2, 3]))   # Output: 3
print(type(123))        # Output: <class 'int'>
```

2. User-defined Functions

These are functions that are defined by the user to perform specific tasks.

Example:

```
def greet(name):
    return f"Hello, {name}!"
print(greet("Alice")) # Output: Hello, Alice!
```

3. Lambda (Anonymous) Functions

These are small, unnamed functions that are defined using the lambda keyword. Lambda functions are generally used for short, simple operations.

Syntax:

```
lambda arguments: expression
```

Example:

```
# Lambda function to add two numbers
add = lambda x, y: x + y
print(add(5, 3))  # Output: 8
```

4. Recursive Functions

A recursive function is a function that calls itself to solve a problem. It typically has a base condition to stop the recursion.

Example (Factorial using recursion):

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
print(factorial(5))  # Output: 120
```

Summary of Function Types:

- **Built-in Functions**: Predefined in Python.
- **User-defined Functions**: Created by the user.
- **Lambda Functions**: Anonymous, single-expression functions.
- **Recursive Functions**: Functions that call themselves.

2. Explain the following properties in the reference of Functions with example.

1. **Function as a Object.**
2. **Function as an argument.**
3. **Function as a variable.**
4. **Function as a reference for another function.**

Ans : In Python, functions are first-class objects, meaning they can be treated like any other object (such as integers, strings, lists, etc.). This allows for many flexible and powerful programming paradigms.

1. Function as an Object

In Python, a function is an object, which means it can be assigned to a variable, stored in data structures (like lists or dictionaries), and even passed around like any other object.

Example:

```
def greet():
    return "Hello!"
# Assigning the function object to a variable
greet_function = greet
# Calling the function using the new variable
print(greet_function()) # Output: Hello!
```

Explanation: Here, the function greet is assigned to the variable greet_function. Since functions are objects, the new variable can be used to call the function, just like the original

function name.

2. Function as an Argument

Functions can be passed as arguments to other functions. This is a common pattern in functional programming and allows you to create highly flexible code.

Example:

```
def apply_func(func, value):
    return func(value)
# Define a function to square a number
def square(x):
    return x * x

# Passing 'square' as an argument to 'apply_func' result
= apply_func(square, 5)
print(result)  # Output: 25
```

- **Explanation**: The apply_func function takes another function func as an argument and applies it to value. In this case, square is passed as an argument, and it computes the square of 5.

3. Function as a Variable

In Python, you can assign a function to a variable, which makes the variable behave like a function. This is very useful when dynamically deciding which function to execute or when building flexible systems.

Example:

```
# Define two simple functions
```

```
def add(x, y):
    return x + y def
subtract(x, y):
    return x - y

# Assign functions to variables
operation = add
print(operation(10, 5))  # Output: 15

# Reassign variable to another function
operation = subtract
print(operation(10, 5)) # Output: 5
```

Explanation: The operation variable first points to the add function, then it is reassigned to the subtract function. The function that operation refers to can be dynamically changed and used for computation.

4. Function as a Reference for Another Function

One function can return another function or be assigned as a reference to another function. This allows for creating **higher-order functions** and implementing things like function factories or decorators.

Example 1: Using One Function as a Reference for Another

```
def multiply(x, y):
    return x * y
def power(x, y):
    return x ** y
```

```
# Use function reference

operation = multiply  # Referencing multiply function

print(operation(2, 3)) # Output: 6

# Now assign the reference to another function operation

= power # Referencing power function

print(operation(2, 3)) # Output: 8
```

Explanation: The operation variable starts by referencing the multiply function and then later references the power function. It acts as a flexible reference, allowing for different operations.

Summary:

- **Function as an Object**: Functions are treated like objects, and you can assign them to variables, pass them around, etc.
- **Function as an Argument**: You can pass functions as arguments to other functions, enabling functional programming patterns.
- **Function as a Variable**: Functions can be assigned to variables, and those variables can be used to call the function.
- **Function as a Reference for Another Function**: One function can reference or return another function, making your code more modular and reusable.

These properties are powerful because they enable higher-order programming, flexibility, and modularity in code design.

3. What is recursion? Explain recursion function in Python with examples.

Ans : Recursion in Python is a method of solving a problem where a function calls itself as a subroutine. This allows the function to be repeated several times as it can call itself during its execution. Recursion is often used to solve problems that can be divided into smaller, similar sub-problems. The key to recursion is defining a base case, which stops the recursion from continuing indefinitely.

Key Concepts of Recursion:

1. **Base Case:** This is the condition that ends the recursion. Without a base case, the function would call itself indefinitely, causing a stack overflow.
2. **Recursive Case:** This is the part where the function calls itself with a modified argument to bring the problem closer to the base case.

Syntax:

```
def func(): <--
            |
            | (recursive call)
            |
   func() ----
```

Example of Recursion: Factorial Calculation

A common example to understand recursion is the calculation of a **factorial**. The factorial of a number nnn (written as n!n!n!) is the product of all positive integers from 1 to nnn.

Mathematically:

n!=n×(n−1)×(n−2)×⋯×1n! = n \times (n-1) \times (n-2) \times \dots \times 1n!=n×(n−1)×(n−2)×⋯×1

- Base Case: 1!=11! = 11!=1
- Recursive Case: n!=n×(n−1)!n! = n \times (n-1)!n!=n×(n−1)!

Python Code for Factorial Using Recursion:

```
def factorial(n):
    # Base case: if n is 0 or 1, return 1
    if n == 0 or n == 1:
        return 1
    # Recursive case: n * factorial(n - 1)
    else:
        return n * factorial(n - 1)
# Test the recursive factorial function
print(factorial(5)) # Output: 120
```

Explanation:

- **Base Case:** When n == 0 or n == 1, the recursion stops, and 1 is returned.
- **Recursive Case:** For any number greater than 1, the function calls itself with n - 1 until it reaches the base case.

For factorial(5), the recursion breaks down like this:

factorial(5) = 5 * factorial(4)

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

factorial(2) = 2 * factorial(1)

factorial(1) = 1 (Base case reached)

So the result is 5 * 4 * 3 * 2 * 1 = 120.

Example of Recursion: Sum of Digits

You can also use recursion to solve other problems like summing the digits of a number.

For instance, the sum of digits of 1234 is 1 + 2 + 3 + 4 = 10.

Python Code for Sum of Digits:

```
def sum_of_digits(n):
    # Base case: if n is 0, return 0
    if n == 0:
        return 0
    # Recursive case: sum the last digit and call for the remaining digits else:
        return n % 10 + sum_of_digits(n // 10)
# Test the sum of digits function
print(sum_of_digits(1234))  # Output: 10
```

Explanation:

- Base Case: If n == 0, the recursion stops and returns 0.
- Recursive Case: The last digit (n % 10) is added to the result of the recursive call on

the remaining digits (n // 10).

For sum_of_digits(1234), the recursion proceeds as:

sum_of_digits(1234) = 4 + sum_of_digits(123)

sum_of_digits(123) = 3 + sum_of_digits(12)

sum_of_digits(12) = 2 + sum_of_digits(1)

sum_of_digits(1) = 1 + sum_of_digits(0)

sum_of_digits(0) = 0 (Base case)

The final result is 4 + 3 + 2 + 1 = 10.

4. What is a Python Module. Explain with examples.

Ans : A **module** in Python is simply a file containing Python code (functions, classes, variables, etc.) that you can reuse in other Python programs. It allows you to logically organize your Python code by grouping related functionality into one file. This promotes code reuse, maintainability, and structure.

Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

In essence:

- A **module** is a file with a .py extension that can contain variables, functions, and classes.
- You can **import** a module into another script to use its functionality.
- Python provides many built-in modules (like math, random, os), and you can also create your own custom modules.

How to Use a Module

You can use a module in your code by importing it using the import statement. Once a module is imported, you can access its functions, variables, and classes.

1. **Using Built-in Python Modules**

Python comes with many built-in modules that provide a wide range of functionalities, such as mathematical operations, file handling, system operations, etc.

Example: Using the math Module

The math module provides various mathematical functions.

```
# Import the built-in math module
import math

# Use the sqrt function from the math module
result = math.sqrt(16)
print(result)  # Output: 4.0
```

In this example:

- The math module is imported using import math.
- The math.sqrt() function is used to calculate the square root of 16.

2. **Creating a Custom Module**

You can create your own module by writing Python code in a .py file and then importing it into other scripts.

Example: Creating a Custom Module mymodule.py

Create a Python file called mymodule.py with the following content:

```
# mymodule.py


# A simple function to add two numbers
def add(a, b):
    return a + b


# A function to subtract two numbers def
subtract(a, b):
    return a – b
```

Using the Custom Module

You can now import and use this module in another Python script:

```
# Import the custom module
import mymodule

# Use the functions defined in the module
result1 = mymodule.add(5, 3)
result2 = mymodule.subtract(10, 4)
print(result1)      #      Output:      8
print(result2) # Output: 6
```

In this example:

- The mymodule.py file defines two functions: add and subtract.
- These functions are imported and used in another Python script.

3. **Importing Specific Functions from a Module**

If you only need specific functions from a module, you can import them directly using the from ... import ... syntax.

Example:

```
# Importing only the sqrt and pow functions from the math module
from math import sqrt, pow
# Using the imported functions
result_sqrt = sqrt(25) result_pow
= pow(2, 3) print(result_sqrt)  #
Output: 5.0 print(result_pow) #
Output: 8.0
```

In this example, instead of importing the entire math module, only the sqrt and pow functions are imported.

Summary:

- A **module** in Python is a file that contains Python code, which can include variables, functions, and classes.
- Python provides a wide range of **built-in modules** (e.g., math, random, os) that you can use.

- You can also create your own **custom modules** by saving code in a .py file and importing it into other programs.
- You can **import entire modules** or just specific parts using the from ... import ... syntax.
- Modules help in organizing code, improving reusability, and keeping projects maintainable.

Modules are a fundamental building block of Python, allowing for code reuse, logical organization, and modular programming.

5. What is the difference between Python Modules and Python Packages. Explain with examples.

Ans : In Python, both modules and packages are mechanisms for organizing and structuring code, but they serve slightly different purposes.

Python Module :

A **module** is simply a single Python file (.py) that contains functions, classes, and variables that can be reused in other Python scripts. Modules allow you to logically organize your code by grouping related functionalities together.

Example of a Python Module:

Consider a Python file named mymodule.py that contains some functions:

```
# mymodule.py
def  add(a,  b):
```

```
    return a + b
def subtract(a, b):
    return a – b
```

You can now import and use this module in another script:

```
# Importing the custom module
import mymodule
# Using functions from the module
result1 = mymodule.add(5, 3)
result2 = mymodule.subtract(10, 4)
print(result1) # Output: 8
print(result2) # Output: 6
```

In this case:

- mymodule.py is a **module** that contains two functions, add and subtract.
- The module is imported using import mymodule in another Python file, and its functions are used.

Python Package :

A **package** is a collection of one or more modules organized in directories, with a special _____init__.py file (which can be empty). The presence of the __init__.py file in a directory tells Python that the directory should be treated as a package, allowing you to organize and distribute modules in a hierarchical way.

Packages enable better code organization, especially for large projects where you need multiple related modules. You can have **sub-packages** within a package as well.

Example of a Python Package:

Let's create a package called mypackage that contains two modules, math_operations.py and string_operations.py.

Directory Structure:

mypackage/

__init__.py

math_operations.py

string_operations.py

1. **math_operations.py:**

```
# math_operations.py
def add(a, b):
    return a + b def
subtract(a, b):
return a - b
```

2. **string_operations.py:**

```
# string_operations.py
def  concatenate(str1,  str2):
    return str1 + str2
def     to_uppercase(string):
    return string.upper()
```

3. **__init__.py:**

This file can be empty, or you can include initialization code to make the package easier to use.

__init__.py (empty or can contain initialization code)

Now, you can import the modules from the package:

```
# Importing the package modules

from mypackage import math_operations, string_operations



# Using functions from the math_operations module

result1 = math_operations.add(5, 3)

result2 = math_operations.subtract(10, 4)



# Using functions from the string_operations module result3

= string_operations.concatenate("Hello, ", "World!") result4

= string_operations.to_uppercase("python")



print(result1)      #      Output:      8

print(result2)      #      Output:      6

print(result3)  # Output: Hello, World!

print(result4)  # Output: PYTHON
```

In this case:

- mypackage is a **package** that contains two **modules**: math_operations.py and string_operations.py.
- The package is structured as a directory with an __init__.py file.
- You can import specific modules from the package using from mypackage import ....

Key Differences Between Modules and Packages:

Feature	Module	Package
Definition	A single Python file (`.py`)	A directory with one or more modules
Structure	Contains functions, variables, classes	Contains multiple modules (which are `.py` files)
Usage	A basic unit of code reusability	A higher-level organizational unit for related modules
Example	`mymodule.py`	`mypackage/` with multiple modules
Importing	`import mymodule`	`from mypackage import math_operations`
Special File	No special file needed	Requires an `__init__.py` file in the directory
Use Case	Small, self-contained functionality	Larger projects with multiple modules

Conclusion:

- A **module** is a single Python file containing code like functions, classes, or variables that can be reused.
- A **package** is a directory containing multiple related modules, organized with an __init__.py file to mark the directory as a package.
- **Modules** are great for organizing small to medium amounts of code, while **packages** are useful for larger projects with multiple modules that need to be grouped logically.

By using both modules and packages, you can maintain clean, organized, and scalable Python code, which is crucial for building large applications.

6. **Write down the steps and examples for creating a package in Python. Ans :** We usually organize our files in different folders and subfolders based on some criteria, so that they can be managed easily and efficiently. For example, we keep all our games in a Games folder and we can even subcategorize according to the genre of the game or something like that. The same analogy is followed by the Python Packages.

Python Packages are a way to organize and structure your Python code into reusable components. Think of it like a folder that contains related Python files (modules) that work together to provide certain functionality. Packages help keep your code organized, make it easier to manage and maintain, and allow you to share your code with others.

They're like a toolbox where you can store and organize your tools (functions and classes) for easy access and reuse in different projects.

How to Create Package in Python?

Creating packages in Python allows you to organize your code into reusable and manageable modules. Here's a brief overview of how to create packages:

- **Create a Directory:** Start by creating a directory (folder) for your package. This directory will serve as the root of your package structure.
- **Add Modules:** Within the package directory, you can add Python files (modules) containing your code. Each module should represent a distinct functionality or component of your package.
- **Init File:** Include an __init__.py file in the package directory. This file can be empty or can contain an initialization code for your package. It signals to

Python that the directory should be treated as a package.

- **Subpackages:** You can create sub-packages within your package by adding additional directories containing modules, along with their own __init__.py files.
- **Importing:** To use modules from your package, import them into your Python scripts using dot notation. For example, if you have a module named module1.py inside a package named mypackage, you would import its function like this: from mypackage.module1 import greet.
- **Distribution:** If you want to distribute your package for others to use, you can create a setup.py file using Python's setuptools library. This file defines metadata about your package and specifies how it should be installed.

Code Example

Here's a basic code sample demonstrating how to create a simple Python package:

1. Create a directory named mypackage.
2. Inside mypackage, create two Python files: module1.py and module2.py.
3. Create an __init__.py file inside mypackage (it can be empty).
4. Add some code to the modules.
5. Finally, demonstrate how to import and use the modules from the package.

```
mypackage/
│
├── __init__.py
├── module1.py
└── module2.py
```

Example: Now, let's create a Python script outside the mypackage directory to

import and use these modules:

```
#      module1.py
def greet(name):
    print(f"Hello,   {name}!")
# module2.py
def    add(a,    b):
    return a + b


from mypackage import module1, module2
#    Using    functions    from    module1
module1.greet("Alice")
#  Using  functions  from  module2
result = module2.add(3, 5)
print("The result of addition is:", result)
```

When you run the script, you should see the following output:

```
Hello, Alice!
The result of addition is: 8
```

In conclusion, Python packages are a powerful tool for organizing, managing, and sharing your code. By grouping related modules together, packages provide a structured way to build complex applications, enhance code reusability, and foster collaboration among developers. Whether you're working on small scripts or large- scale projects, mastering the art of creating and utilizing Python packages will undoubtedly streamline your development process and contribute to writing cleaner, more maintainable code.

7. Discuss 'math', 'datetime' modules with examples. (Using any 5 function)

Ans : The math and datetime modules in Python provide a wide range of built-in functions that simplify mathematical calculations and time/date manipulations, respectively.

Python DateTime module

Python Datetime module supplies classes to work with date and time. These classes provide several functions to deal with dates, times, and time intervals. Date and DateTime are an object in Python, so when you manipulate them, you are manipulating objects and not strings or timestamps.

The DateTime module is categorized into 6 main classes –

- date – An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Its attributes are year, month, and day.
- time – An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds. Its attributes are hour, minute, second, microsecond, and tzinfo.
- date-time – It is a combination of date and time along with the attributes year, month, day, hour, minute, second, microsecond, and tzinfo.
- timedelta – A duration expressing the difference between two date, time, or

datetime instances to microsecond resolution.

- tzinfo – It provides time zone information objects.
- timezone – A class that implements the tzinfo abstract base class as a fixed offset from the UTC (New in version 3.2).

How to import:

```
import datetime
```

Common Functions in the datetime Module:

1. datetime.datetime.now(): Returns the current date and time.

```
import datetime
current_time = datetime.datetime.now()
print(current_time)  # Output: e.g., 2024-09-29 10:45:12.345678
```

2. datetime.datetime.strptime(date_string, format) : Converts a string representing a date to a datetime object using the specified format.

```
date_string = "2023-09-29"
date_obj = datetime.datetime.strptime(date_string, "%Y-%m-%d")
print(date_obj) # Output: 2023-09-29 00:00:00
```

3. datetime.datetime.strftime(format) : Formats a datetime object into a string using the specified format.

```
current_time = datetime.datetime.now()
formatted_time = current_time.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_time) # Output: e.g., "2024-09-29 10:45:12"
```

4. datetime.timedelta(days, seconds, ...) : Represents a duration (difference between

two dates or times). You can specify days, seconds, hours, etc.

```
delta = datetime.timedelta(days=5)
future_date = datetime.datetime.now() + delta
print(future_date)  # Output: Current date + 5 days
```

5. **datetime.datetime.today() :** Returns the current local date without the time component (equivalent to calling now().date()).

```
today = datetime.datetime.today()
print(today.date())  # Output: e.g., 2024-09-29
```

Python math Module

Math Module consists of mathematical functions and constants. It is a built-in module made for mathematical tasks.

The math module provides the math functions to deal with basic operations such as addition(+), subtraction(-), multiplication(*), division(/), and advanced operations like trigonometric, logarithmic, and exponential functions.

How to import:

```
import math
```

Common Functions in the math Module:

1. math.sqrt(x) : Returns the square root of a number x.

```
import math
print(math.sqrt(16))  # Output: 4.0
print(math.sqrt(25))  # Output: 5.0
```

2. math.pow(x, y) : Raises x to the power of y (i.e., x^y).

print(math.pow(2, 3)) # Output: 8.0

print(math.pow(5, 2)) # Output: 25.0

3. math.sin(x) : Returns the sine of x (where x is in radians).

angle = math.pi / 2 # 90 degrees in radians

print(math.sin(angle)) # Output: 1.0

4. math.factorial(x) : Returns the factorial of a non-negative integer x.

print(math.factorial(5)) # Output: 120 (5! = 5 × 4 × 3 × 2 × 1)

print(math.factorial(0)) # Output: 1

5. math.log(x, base) : Returns the logarithm of x to the given base. If no base is provided, it returns the natural logarithm (base e).

print(math.log(10)) # Output: 2.302585 (Natural log of 10)

print(math.log(100, 10)) # Output: 2.0 (Log base 10 of 100)

Summary of Functions Used:

Module	Function	Description
`math`	`math.sqrt(x)`	Returns the square root of `x`.
	`math.pow(x, y)`	Raises `x` to the power of `y`.
	`math.sin(x)`	Returns the sine of `x` (in radians).
	`math.factorial(x)`	Returns the factorial of `x`.
	`math.log(x, base)`	Returns the logarithm of `x` to the specified base.
`datetime`	`datetime.datetime.now()`	Returns the current date and time.
	`datetime.datetime.strptime()`	Converts a string to a `datetime` object.
	`datetime.datetime.strftime()`	Converts a `datetime` object to a formatted string.
	`datetime.timedelta()`	Represents a time duration.
	`datetime.datetime.today()`	Returns the current date.

These functions from math and datetime modules are fundamental and widely used in Python programming for numerical and time-based operations.

8. Explain Python anonymous functions (lambda functions) with examples.

Ans : In Python, anonymous functions are functions that are defined without a name. While regular functions are defined using the def keyword, anonymous functions are created using the lambda keyword. These anonymous functions are also known as lambda functions.

Lambda functions are typically used for short, simple operations and are often passed as arguments to higher-order functions like map(), filter(), and sorted().

Syntax of a Lambda Function

The syntax of a lambda function is as follows:

lambda arguments: expression

- **lambda**: The keyword used to define the anonymous function.
- **arguments**: A comma-separated list of parameters (just like a normal function).
- **expression**: The single expression or operation that the function will return (lambda functions can only have one expression).

Lambda functions are limited to a single expression and cannot contain statements or multiple lines.

Example of a Lambda Function

1. Basic Lambda Function:

A simple lambda function that adds two numbers:

Lambda function to add two numbers

```
add = lambda x, y: x + y

# Using the lambda function

result = add(5, 3)

print(result) # Output: 8
```

Here, lambda x, y: x + y creates an anonymous function that takes two arguments x and y and returns their sum.

Common Use Cases of Lambda Functions

2. Lambda Functions with map()

The map() function applies a given function to each item of an iterable (such as a list) and returns a map object (which can be converted to a list).

Example:

```
# List of numbers

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

# Lambda function to square each number

squares = list(map(lambda x: x ** 2, numbers))

print(squares) # Output: [1, 4, 9, 16, 25]
```

In this example, lambda x: x ** 2 squares each element in the list numbers.

3. Lambda Functions with filter()

The filter() function filters items from an iterable based on a condition defined in the function.

Example:

```
# List of numbers

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Lambda function to filter out even numbers evens

= list(filter(lambda x: x % 2 == 0, numbers))

print(evens) # Output: [2, 4, 6, 8]

In this case, the lambda function lambda x: x % 2 == 0 returns True for even numbers, and filter() keeps only the even numbers.

4. Lambda Functions with sorted()

The sorted() function can take a key argument, which allows you to pass a function that determines the sorting order.

Example:

List of tuples

students = [("Alice", 25), ("Bob", 20), ("Charlie", 23)]

Sort by the second element (age) using a lambda function sorted_students

= sorted(students, key=lambda x: x[1]) print(sorted_students) # Output:

[('Bob', 20), ('Charlie', 23), ('Alice', 25)]

In this case, lambda x: x[1] specifies that sorting should be done based on the second element (age) of each tuple.

With lambda function	Without lambda function
Supports single-line sometimes statements that return some value.	Supports any number of lines inside a function block
Good for performing short operations/data manipulations.	Good for any cases that require multiple lines of code.
Using the lambda function can sometime reduce the readability of code.	We can use comments and function descriptions for easy readability.

Conclusion

Lambda functions in Python are useful for short, concise operations that require one- time, simple functionality. They are often used in combination with functions like map(), filter(), reduce(), and sorted(). However, for more complex logic, regular functions using the def keyword are preferred because they offer better readability and flexibility.

9. Explain python decorators with examples.

Ans : A decorator is a higher-order function that takes another function as an argument, adds some functionality, and returns a new function. It allows modifying or extending behavior of functions or methods.

Python decorators are a powerful feature that allows you to modify the behavior of a function or method without changing its code. Decorators are often used to add functionality like logging, access control, timing, etc., to existing functions. They are implemented as higher-order functions that take a function as an argument and return a new function that usually wraps the original one.

Syntax :

```
@decorator_name
def function_to_decorate():
    # Function body
    pass
```

How Decorators Work

A decorator is applied using the @decorator_name syntax just before the function

definition. It modifies or enhances the function to which it is applied.

Example 1: A Basic Decorator

Here's a simple example that shows how a decorator works: def

```
my_decorator(func):
    def wrapper():
        print("Something before the function is called.")
        func()
        print("Something after the function is called.")
    return wrapper
@my_decorator
def say_hello():
    print("Hello!")



say_hello()
```

Output:

Something before the function is called.

Hello!

Something after the function is called.

Explanation:

- my_decorator is a function that takes say_hello as its argument (func).
- Inside my_decorator, a new function wrapper is defined that wraps the original

function.

- The wrapper function adds behavior before and after calling the original function (func).
- The @my_decorator syntax applies the decorator to say_hello.

Example 2: Decorators with Arguments

Decorators can also take arguments, allowing them to be more flexible. Here's an example of a decorator that takes an argument:

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator
@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

Output:

```
Hello, Alice!
```

Hello, Alice!

Hello, Alice!

Example 3: Class-Based Decorators

Decorators can also be implemented as classes. A class-based decorator must define the __call__method, which allows the instance of the class to be used as a function.

```
class MyDecorator:

    def __init__(self, func):

        self.func = func



    def __call__(self,      *args,      **kwargs):

        print("Something before the function.")

        result   =   self.func(*args,   **kwargs)

        print("Something   after   the   function.")

        return result



@MyDecorator

def say_hello():

    print("Hello!")



say_hello()
```

Output:

Something before the function.

Hello!

Something after the function.

Use Cases for Decorators

- **Logging:** Automatically log information when a function is called.
- **Authentication:** Add access control checks before executing functions.
- **Memoization:** Cache results of expensive function calls to speed up subsequent calls.
- **Timing:** Measure the time a function takes to execute.

Decorators provide a clean, readable way to extend the behavior of functions and methods, making them a versatile tool in Python programming.

10. Explain Python generators with examples. Also discuss the difference between generator functions and normal functions.

Ans : A Generator in Python is a function that returns an iterator using the Yield keyword. Python generators are a special type of iterable that allow you to iterate through a sequence of values lazily—meaning that they generate the values on the fly as needed, rather than storing the entire sequence in memory at once. This makes them more memory-efficient, especially when dealing with large data sets.

Generators are defined using the yield keyword instead of return. When a function

contains the yield keyword, it becomes a generator function. Each call to yield produces a value, and the function's state is "paused" between yields, allowing it to be resumed later.

How Generators Work

1. **When the generator function is called**, it doesn't execute the function body immediately but returns an iterator object (the generator).
2. **When you call next() on the generator**, it runs the function until it hits a yield statement, which produces a value.
3. The generator's state is saved, and it can resume from where it left off the next time next() is called.

Syntax :

```
def function_name():
   yield statement
```

Example 1: A Simple Generator

```
def simple_generator():
   yield 1
   yield 2
   yield 3

gen = simple_generator()
print(next(gen)) # Output: 1
print(next(gen)) # Output: 2
print(next(gen)) # Output: 3
```

In this example:

- The generator simple_generator() yields three values, one at a time.
- Each call to next(gen) gets the next value in the sequence.

Advantages of Generators

Memory Efficiency: Generators generate values on the fly, reducing memory usage compared to storing the entire sequence in memory.

Lazy Evaluation: Values are computed only when needed, which can lead to performance improvements, especially with large datasets.

Maintain State: Generators automatically maintain their state between yield statements, making them suitable for iterative processes.

Convenience: Generators simplify code for iterating over large datasets or streams of data without needing to manage state explicitly.

Difference between Generator and Normal Function

Normal functions in Python are used for traditional computation tasks, with execution proceeding from start to finish, typically returning a single result. On the other hand, generator functions employ the `yield` statement to produce values lazily, preserving their state across multiple calls. This allows generators to efficiently handle large datasets or infinite sequences by yielding values one at a time and pausing execution when necessary, making them a valuable tool for memory-efficient and iterative tasks. In this article, we'll look into Python generator functions and normal function differences
i.e. how different are their syntax, how is data handled, and practical applications.

Python Generators

Yield statement

Yield statement allows a generator function to produce a value to the caller without losing the current state of the function. When the generator function is called again, execution resumes from where it left off, continuing to yield values one at a time.

Let us see its syntax

```
def generator_function():
    # Initialization or setup if needed
    while condition:
        # Calculate or generate a value
        yield value_to_yield
        # Logic to be applied if necessary
```

Example

```
# generator function
def Square():
    number = 2


    #Create infinite loop
    while True:
            # Yield the current value of 'number'
            yield number
            # Calculate the square of 'number' and update its value
            number *= number
```

```
# Create a generator object 'Sq' by calling the 'Square()' generator function
Sq = Square()
#           Function           call
print(next(Sq))  #  Output:  2
print(next(Sq))  #  Output:  4
```

Normal function in Python

Normal function performs a specific task and can be called from other parts of the program.

Also normal function return a single value and terminate the session.

```
# generator function named square()
def square():
    number = 2

    #  Create  infinite  loop
    while True:
        # Yield the current value of 'number'
        yield number


        # Calculate the square of 'number' and update its value
        number *= number
# Define a function to retrieve the next square from the generator.
def get_next_square():
    global     number_generator
    try:
        # Try to get the next square from the existing generator return
```

```
        next(number_generator)
    except NameError:
        # If 'number_generator' is not defined , initialize it
        number_generator = square()
        # Return the first square from the newly created generator.
        return next(number_generator)
# function call to retrieve the next square and print it.
print(get_next_square())        #        Output:        2
print(get_next_square()) # Output: 4
```

Difference :

The generator function in Python is normally defined like a normal function, using the @' keyword. If we need to generate value Generator functions make use of the yield keyword rather than using return. The major difference between the generator and normal function may be that the Generator function runs when the next() function is called and not by its name as in the case of normal functions.

Scope	Generator	Normal
Execution	They can be paused in the middle of execution and resumed	They runs to completion and returns a value
Return value	They can return multiple values through multiple iterations.	They returns a single value (or none)
Memory usage	They keeps the current value in memory,	They create a large amount of memory overhead
Usage	They are used generate values that can be iterated over.	They are used when to perform a task and return a result.

11. What is OOP in Python. List out all principles or features of OOP in Python.

Ans : Object Oriented Programming is a fundamental concept in Python, empowering developers to build modular, maintainable, and scalable applications. By understanding the core OOP principles—classes, objects, inheritance, encapsulation, polymorphism, and abstraction—programmers can leverage the full potential of Python's OOP capabilities to design elegant and efficient solutions to complex problems.

In Python object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of object-oriented Programming (OOPs) or oops concepts in Python is to bind the data and the functions that work together as a single unit so that no other part of the code can access this data.

Key Concepts of OOP in Python

1. **Class**: A blueprint or template for creating objects. It defines the attributes (data) and methods (behavior) that the objects created from the class will have.
2. **Object**: An instance of a class. An object is created using the class blueprint and can have its own unique data and behavior.
3. **Method**: A function defined inside a class that operates on the data (attributes) of objects of that class.
4. **Attribute**: A variable that belongs to an object or class. Objects can have instance attributes (unique to each object) and class attributes (shared across all objects of that class).

Core Principles or Features of OOP in Python

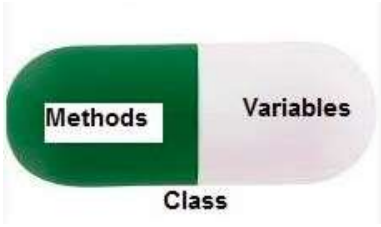
1. **Encapsulation**
2. **Abstraction**
3. **Inheritance**

4. **Polymorphism**

1. Encapsulation

Encapsulation refers to bundling data (attributes) and methods (functions) that operate on the data into a single unit (class). It also restricts direct access to some of the object's components, which is called information hiding.

In Python, you can use private attributes and methods (denoted by a leading underscore _ or double underscore __) to limit access and enforce encapsulation.



Example:

```
class Employee:

    def __init__(self,      name,      salary):
        self.name = name  # Public attribute
        self.__salary = salary  # Private attribute
    def get_salary(self):
        return self.__salary  # Public method to access private attribute
emp = Employee("Alice", 50000)
print(emp.name)          #          Accessible
```

```
print(emp.get_salary()) # Accessible
# print(emp.__salary)  # Error: Private attribute cannot be accessed directly
```

2. Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. It focuses on what the object does rather than how it does it.

In Python, abstraction can be implemented using abstract classes and interfaces (through the abc module).

Example:

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):

    def __init__(self, radius):
    self.radius = radius
def area(self):

        return 3.14 * self.radius * self.radius
```

```
circle = Circle(5)
print(circle.area())
```

In the example above, Shape is an abstract class that only defines an interface (the area method), and Circle provides the concrete implementation.

3. Inheritance

Inheritance allows a class to inherit attributes and methods from another class, promoting code reusability and establishing a relationship between parent (superclass) and child (subclass) classes. The subclass can inherit or override behavior from the parent class.

Example:

```
class Animal:
def __init__(self, name):
        self.name = name
    def speak(self):
        print(f"{self.name} makes a sound.")


class Dog(Animal): # Dog inherits from Animal def
    speak(self):
        print(f"{self.name} barks.")
dog = Dog("Buddy")
```

```
dog.speak()  # Output: Buddy barks.
```

4. Polymorphism

Polymorphism allows methods to have the same name but behave differently depending on the object that is calling them. This can be achieved via method overriding (inherited methods can be redefined in subclasses) and method overloading (in Python, this is typically achieved through default parameters or *args).

Example:

```
class  Cat(Animal):
   def speak(self):
      print(f"{self.name}          meows.")
animals = [Dog("Buddy"), Cat("Kitty")]
for animal in animals:
   animal.speak()  # Dog barks, Cat meows
```

In this example, both Dog and Cat inherit from Animal and have their own speak() method, which exhibits polymorphism as the same method behaves differently depending on the object.

Summary of OOP Principles and Features in Python

1. **Encapsulation**: Hides internal data and exposes only necessary functionality.
2. **Abstraction**: Simplifies complex systems by exposing only essential parts.
3. **Inheritance**: Enables a class to inherit attributes and methods from another

class.

4. **Polymorphism**: Allows methods to behave differently based on the object calling them.

Additional features like multiple inheritance, method overriding, static and class methods, and MRO make Python's OOP powerful and flexible.

12. Explain constructor, method overloading, operator overloading, overriding with examples.

Ans : 1. Constructor in Python

A constructor is a special type of method in Python that gets called when an object is instantiated. It is defined using the __init__() method in a class. The purpose of a constructor is to initialize the attributes of the class when an object is created.

Example:

```
class Person:

    def __init__(self, name, age): # Constructor self.name
        = name  # Initializing the 'name' attribute self.age =
        age # Initializing the 'age' attribute


    def display_info(self):

        print(f"Name: {self.name}, Age: {self.age}")
```

```
# Creating an object of the Person class
person1 = Person("Alice", 30)
person1.display_info()  # Output: Name: Alice, Age: 30
```

- **__init__()** is called when a new object (person1) is created.
- It initializes the object's attributes (name and age).

2. **Method Overloading in Python**

Method overloading refers to defining multiple methods with the same name but different parameters (number or type) to perform various tasks. However, Python does not support method overloading **directly** like some other programming languages. In Python, you can achieve method overloading by using default arguments or variable-length arguments (*args, **kwargs).

Example using Default Arguments:

```
class Calculator:
    def add(self, a, b=0):  # Method overloading using default argument
        return a + b
calc = Calculator()
print(calc.add(5))      # Output: 5 (uses only one argument)
print(calc.add(5, 10))  # Output: 15 (uses two arguments)
```

3. Operator Overloading in Python

Operator overloading allows the same operator to have different meanings depending on the context, specifically based on the operands it works with. In Python, you can

overload operators by defining special methods in your class, such as __add__(), __sub__(), __mul__(), etc.

Example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
    def __str__(self):
        return f"Point({self.x}, {self.y})"

# Creating two Point objects
p1 = Point(2, 3)
p2 = Point(1, 4)

# Using the + operator with Point objects
p3 = p1 + p2
```

```
print(p3)  # Output: Point(3, 7)
```

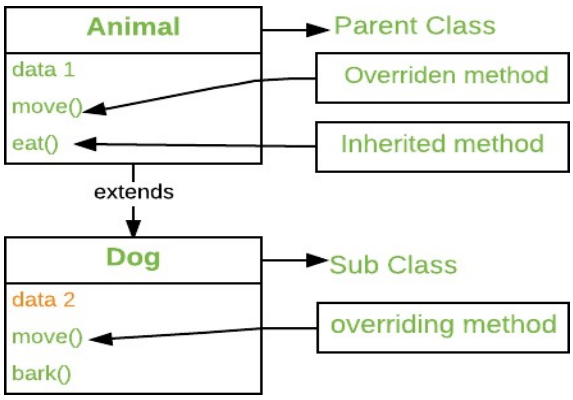
- **Explanation**: The _add_() method is used to overload the + operator for the Point class. This allows the + operator to add two Point objects by adding their respective x and y coordinates.

Common Operator Overloading Methods:

- __add__(self, other): Overloads the + operator.
- __sub__(self, other): Overloads the - operator.
- __mul__(self, other): Overloads the * operator.
- __truediv__(self, other): Overloads the / operator.
- __eq__(self, other): Overloads the == operator.

4. Method Overriding in Python

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its parent class. This allows the subclass to modify or extend the functionality of the method.



Example:

```
class Animal:

    def speak(self):

        print("Animal   makes   a   sound.")
class Dog(Animal):
    def                 speak(self):
        print("Dog barks.")
# Creating an object of the Dog class
dog = Dog()
dog.speak()  # Output: Dog barks
```

Explanation: The Dog class overrides the speak() method of the Animal class. When speak() is called on a Dog object, the overridden version is executed.

Summary

1. **Constructor (__init__)**: Initializes object attributes when an object is created.
    - Example: __init__(self, name, age)

2. **Method Overloading**: Not directly supported in Python, but can be achieved using default arguments or *args.
    - Example: def add(self, *args):
3. **Operator Overloading**: Allows using operators with user-defined objects by defining special methods (__add__(), __sub__(), etc.).
    - Example: def __add__(self, other):
4. **Method Overriding**: A subclass provides a specific implementation of a method that is already defined in its superclass.
    - Example: def speak(self): in both parent and child classes.

13. Discuss the difference between constructor and destructor in Python with example.

Ans : Constructor vs. Destructor in Python

Both constructor and destructor are special methods in Python that handle the initialization and cleanup of objects in classes, respectively. Here's a comparison of the two:

Feature	Constructor (`__init__`)	Destructor (`__del__`)
Purpose	Initializes an object and sets up its initial state (attributes, resources, etc.)	Cleans up when an object is about to be destroyed (releasing resources, closing files, etc.)
Method	`__init__(self, ...)`	`__del__(self)`
Call Time	Called automatically when an object is instantiated (created)	Called when an object is about to be destroyed (usually when no references to the object remain)
Parameters	Typically accepts parameters to initialize object attributes	Does not usually accept any parameters, other than `self`
Usage	Used to set up and initialize instance variables	Used to release resources like file handles, network connections, etc.
Manual Call	Should not be called manually, automatically invoked during object creation	Rarely called manually, Python's garbage collector handles the cleanup process

Constructor (__init__)

A **constructor** is used to initialize an object when it is created. It is defined using the __init__() method and is responsible for setting up the initial state of the object (such as defining its attributes).

Example:

```
class Employee:

    def __init__(self, name, salary):  # Constructor
        self.name = name
        self.salary = salary

        print(f"Employee {self.name} with salary {self.salary} created.")


# Creating an Employee object
emp1 = Employee("Alice", 50000)
```

Output:

Employee Alice with salary 50000 created.

- **Explanation**: The __init__() method is automatically called when the Employee object emp1 is created. It initializes the name and salary attributes and prints a message.

Destructor (__del__)

A **destructor** is called when an object is about to be destroyed (when it is no longer needed, i.e., when its reference count drops to zero). It is defined using the __del__() method and is typically used to free up resources (e.g., closing files or network

connections).

Example:

```
class Employee:

    def __init__(self, name, salary):  # Constructor

        self.name = name

        self.salary = salary

        print(f"Employee {self.name} created with salary {self.salary}.")



    def __del__(self):  # Destructor

        print(f"Employee {self.name} is being deleted.")
# Creating an Employee object

emp1 = Employee("Bob", 60000)



# Deleting the Employee object

del emp1
```

Output:

```
Employee Bob created with salary 60000.

Employee Bob is being deleted.
```

Explanation: The __del__() method is called automatically when the object emp1 is deleted using del emp1. In practice, the garbage collector will eventually delete objects

when they are no longer referenced.

Summary

- **Constructor (__init__)**: Used to initialize an object when it's created, setting initial attributes and resources.
- **Destructor (__del__)**: Used to clean up resources when the object is no longer needed and is about to be destroyed.

14. What is Inheritance. Explain its types with examples.

Ans : One of the core concepts in object-oriented programming (OOP) languages is inheritance. It is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class. Inheritance is the capability of one class to derive or inherit the properties from another class.

Benefits of inheritance are:

Inheritance allows you to inherit the properties of a class, i.e., base class to another, i.e., derived class. The benefits of Inheritance in Python are as follows:

- It represents real-world relationships well.
- It provides the **reusability** of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

- Inheritance offers a simple, understandable model structure.
- Less development and maintenance expenses result from an inheritance.

Python Inheritance Syntax

The syntax of simple inheritance in Python is as follows:

```
Class BaseClass:
    {Body}
Class DerivedClass(BaseClass):
    {Body}
```

Types of inheritance Python

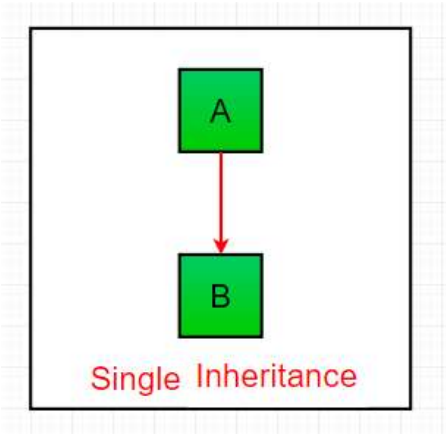
Inheritance is defined as the mechanism of inheriting the properties of the base class to the child class.

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

1. Single Inheritance

Single inheritance enables a derived class to inherit properties from a single parent class,

thus enabling code reusability and the addition of new features to existing code.



Example:

```
# Python program to demonstrate
# single inheritance
#   Base   class
class Parent:
    def func1(self):

            print("This function is in parent class.")
# Derived class
class   Child(Parent):
    def func2(self):
            print("This function is in child class.")
# Driver's code
object   =   Child()
object.func1()
object.func2()
```

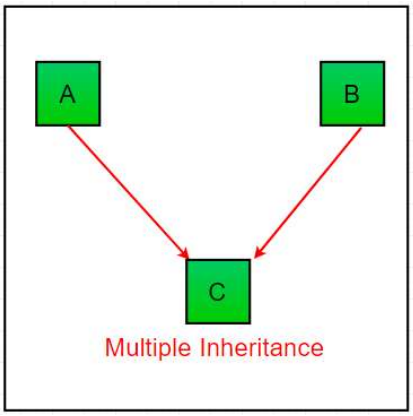
Output:

This function is in parent class.

This function is in child class.

2. Multiple Inheritance

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



Example:

```
# Python program to demonstrate
# multiple inheritance
#  Base  class1
class Mother:
    mothername = ""
    def mother(self):
            print(self.mothername)
```

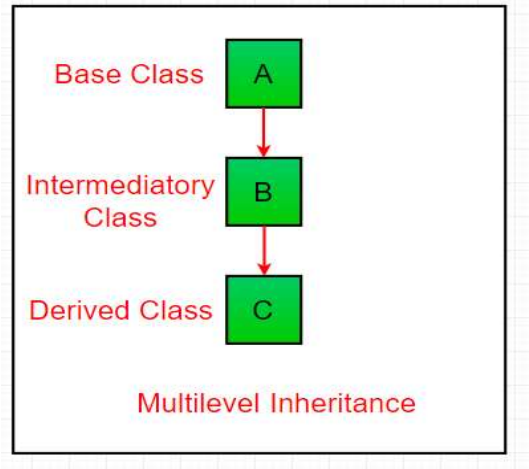
```
# Base class2
class Father:
    fathername = ""
    def father(self):
        print(self.fathername)
# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)
#  Driver's  code
s1 = Son()
s1.fathername  =  "RAM"
s1.mothername = "SITA"
s1.parents()
```

Output:

```
Father  :  RAM
Mother : SITA
```

3. Multilevel Inheritance

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.



Example:

```
# Python program to demonstrate

# multilevel inheritance

# Base class

class Grandfather:



 def __init__(self,                grandfathername):

        self.grandfathername = grandfathername



# Intermediate class

class Father(Grandfather):

 def __init__(self, fathername, grandfathername):

        self.fathername = fathername



        # invoking constructor of Grandfather class
```

```
        Grandfather.__init__(self, grandfathername)


#   Derived   class
class Son(Father):
 def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname


        # invoking constructor of Father class

        Father.__init__(self, fathername, grandfathername)
 def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father      name      :",      self.fathername)
        print("Son name :", self.sonname)
# Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
```

print(s1.grandfathername)

s1.print_name()

Output:

Lal mani

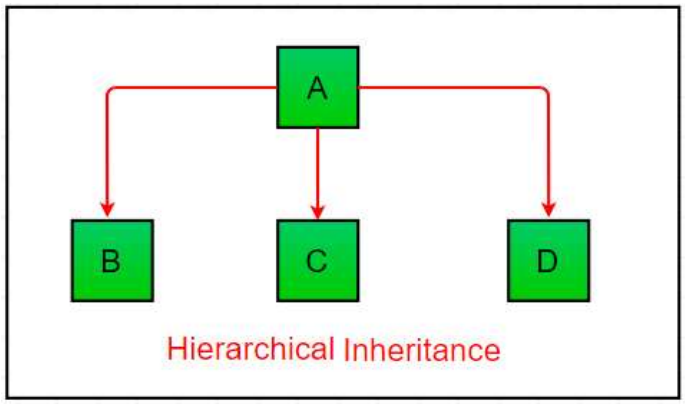
Grandfather name : Lal mani

Father name : Rampal

Son name : Prince

4. Hierarchical Inheritance:

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



Example:

Python program to demonstrate

Hierarchical inheritance

Base class

class Parent:

def func1(self):

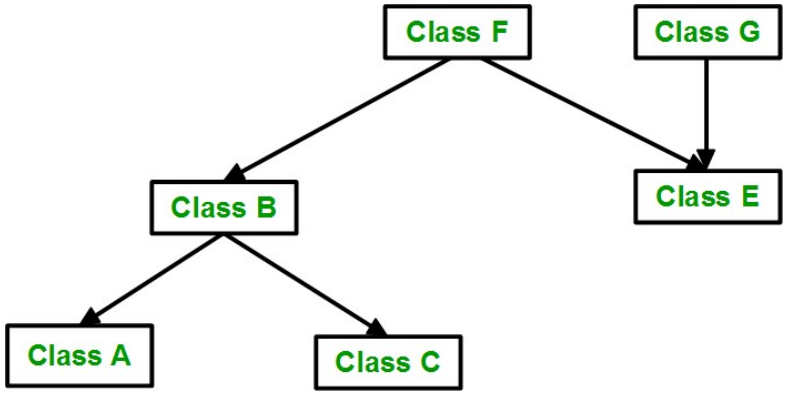
```
        print("This function is in parent class.")
# Derived class1
class Child1(Parent):
 def func2(self):
        print("This function is in child 1.")
#    Derivied    class2
class Child2(Parent):
 def func3(self):
        print("This function is in child 2.")
# Driver's code
object1    =    Child1()
object2    =    Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

5. Hybrid Inheritance

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.



Example:

```
# Python program to demonstrate
# hybrid inheritance
class School:
 def func1(self):
        print("This function is in school.")
class Student1(School):
 def func2(self):
        print("This function is in student 1. ")
class Student2(School):
 def func3(self):
        print("This function is in student 2.")
class Student3(Student1, School):
 def func4(self):
        print("This function is in student 3.")
# Driver's code
```

```
object = Student3()
object.func1()
object.func2()
```

Output:

```
This function is in school.
This function is in student 1.
```

By using inheritance in Python, you can create a logical structure of classes and promote code reuse, making your programs easier to maintain and extend.