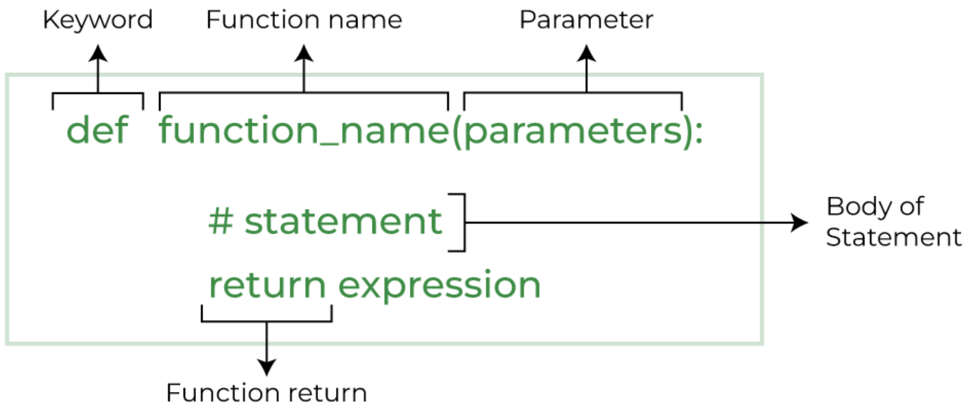
UNIT III

Python Functions

Python Functions are a block of statements that does a specific task. The idea is to put some commonly or repeatedly done task 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.

Function Declaration

The syntax to declare a function is:



Syntax of Python Function Declaration

Defining a Function

We can define a function in Python, using the **def keyword**. We can add any type of functionalities and properties to it as we require.

The def keyword stands for define. It is used to create a **user-defined function**. It marks the beginning of a function block and allows you to group a set of statements so they can be reused when the function is called.

Syntax:

def function_name(parameters):

# function body

Explanation:

- **def:** Starts the function definition.
- **function_name:** Name of the function.
- **parameters:** Inputs passed to the function (inside ()), optional.
- **Indented code:** The function body that runs when called.

Here, we define a function using def that prints a welcome message when called.

```
def fun():
    print("Welcome to GFG")
```

Calling a Function

After creating a function in Python we can call it by using the name of the functions followed by parenthesis containing parameters of that particular function.

```
def fun():
    print("Welcome to GFG")

fun() # Driver code to call a function
```

Output

```
Welcome to GFG
```

Variable Scope & Lifetime

Variable scope and lifetime are important concepts in programming languages, including Python. They determine where and for how long a variable can be accessed and used in your code. Let's explore these concepts in detail:

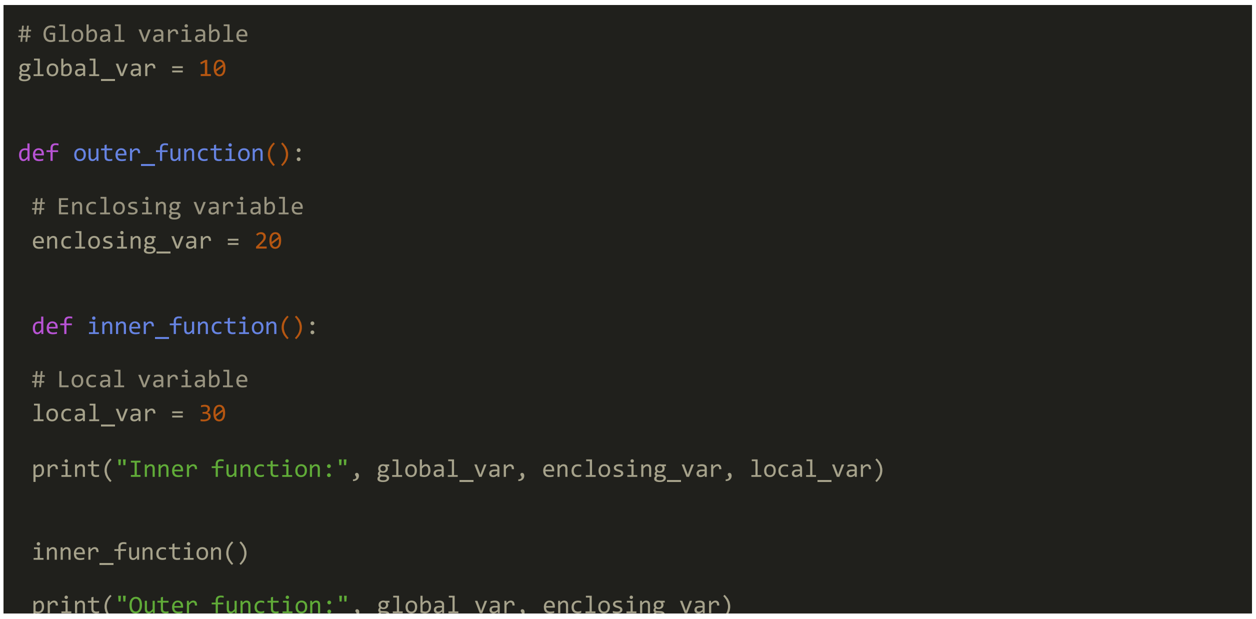
Variable Scope:

- Variable scope refers to the region of code where a variable is visible and accessible.
- In Python, variables can have one of the following scopes:
    - Global Scope: Variables defined outside of any function or class have global scope. They can be accessed from anywhere in the code.
    - Local Scope: Variables defined within a function have local scope. They can only be accessed within that function.
    - Enclosing (Nonlocal) Scope: Variables defined in an enclosing function have enclosing scope. They can be accessed by nested functions but not from outside the enclosing function.
- Python follows the LEGB (Local, Enclosing, Global, Built-in) rule to resolve variable names.

Lifetime of Variables:

- The lifetime of a variable is the duration for which it exists in memory during program execution.
- In Python, the lifetime of variables depends on their scope:
  - Global Variables: Exist until the program terminates or until explicitly deleted.
  - Local Variables: Exist only within the function where they are defined and are destroyed when the function returns.
  - Enclosing Variables: Exist as long as the enclosing function is executing and are destroyed when the function completes execution.
- Memory occupied by variables is automatically reclaimed by Python's garbage collector when they go out of scope.

Let's illustrate variable scope and lifetime with an example:



```
# Global variable
global_var = 10

def outer_function():
 # Enclosing variable
 enclosing_var = 20

 def inner_function():
 # Local variable
 local_var = 30
 print("Inner function:", global_var, enclosing_var, local_var)

 inner_function()
 print("Outer function:", global_var, enclosing_var)
```

In this example:

- global_varis a global variable accessible from all parts of the code.
- enclosing_varis an enclosing variable accessible within outer_functionand its nested functions.
- local_varis a local variable accessible only within inner_function.
- Each variable has a different scope and lifetime based on where it is defined.

Understanding variable scope and lifetime is essential for writing correct and

maintainable code in Python. It helps prevent naming conflicts, manage memory efficiently, and ensure proper encapsulation of data.

Python return statement

A **return statement** is used to end the execution of the function call and it "returns" the value of the expression following the return keyword to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned. A **return statement** is overall used to invoke a function so that the passed statements can be executed.

Example:

```
def add(a, b):

    # returning sum of a and b
    return a + b

def is_true(a):

    # returning boolean of a
    return bool(a)

# calling function
res = add(2, 3)
print(res)

res = is_true(2<5)
print(res)
```

Output

```
5

True
```

Explanation:

- **add(a, b) Function:** Takes two arguments a and b. Returns the sum of a and b.
- **is_true(a) Function:** Takes one argument a. Returns the boolean value of a.
- **Function Calls:** res = add(2, 3) computes the sum of 2 and 3, storing the result (5) in res. res = is_true(2 < 5) evaluates the expression 2 < 5 (which is True) and stores the boolean value True in res.

Types of Arguments in Python

Last Updated : 23 Jul, 2025

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma. There are many types of arguments in Python .

In this example, we will create a simple function in Python to check whether the number passed as an argument to the function positive, negative or zero.

```python
def checkSign(a):
    if a > 0:
        print("positive")
    elif a < 0:
        print("negative")
    else:
        print("zero")

# call the function
checkSign(10)
checkSign(-5)
checkSign(0)
```

Output

```
positive

negative

zero
```

Types of Arguments in Python

Python provides various argument types to pass values to functions, making them more flexible and reusable. Understanding these types can simplify your code and improve readability. we have the following function argument types in Python:

- **Default argument**
- **Keyword arguments (named arguments)**
- **Positional arguments**
- **Arbitrary arguments** (variable-length arguments *args and **kwargs)

Default Arguments

Default Arguments is a parameter that have a predefined value if no value is passed during the function call. This following example illustrates Default arguments to write functions in Python.

```
def calculate_area(length, width=5):
    area = length * width
    print(f"Area of rectangle: {area}")
# Driver code (We call calculate_area() with only
# the length argument)
calculate_area(10)
# We can also pass a custom width
calculate_area(10, 8)
```

Output

```
Area of rectangle: 50

Area of rectangle: 80
```

Keyword arguments

Keyword arguments are passed by naming the parameters when calling the function. This lets us provide the arguments in any order, making the code more readable and flexible.

```
def fun(name, course):
    print(name,course)
# Positional arguments
fun(course="DSA",name="gfg")
fun(name="gfg",course="DSA")
```

Output

```
gfg DSA

gfg DSA
```

Positional arguments

Positional arguments in Python are values that we pass to a function in a specific order. The order in which we pass the arguments matters.

This following example illustrates Positional arguments to write functions in Python.

```
def productInfo(product, price):
    print("Product:", product)
    print("Price: $", price)

# Correct order of arguments
print("Case-1:")
productInfo("Laptop", 1200)

# Incorrect order of arguments
print("\nCase-2:")
productInfo(1200, "Laptop")
```

Output

```
Case-1:

Product: Laptop

Price: $ 1200



Case-2:

Product: 1200

Price: $ Laptop
```

Arbitrary arguments (variable-length arguments *args and **kwargs)

In Python Arbitrary arguments allow us to pass a number of arguments to a function. This is useful when we don't know in advance how many arguments we will need to pass. There are two types of arbitrary arguments:

- *args in Python (Non-Keyword Arguments): Collects extra positional arguments passed to a function into a tuple.

- **kwargs in Python (Keyword Arguments): Collects extra keyword arguments passed to a function into a dictionary.

Example 1 : Handling Variable Arguments in Functions

```
# Python program to illustrate
# *args for variable number of arguments
def myFun(*argv):
    for arg in argv:
        print(arg)
# Driver code with different arguments
myFun('Python', 'is', 'amazing')
```

Output

```
Python

is

amazing
```

Recursion in Python

Recursion is a programming technique where a function calls itself either directly or indirectly to solve a problem by breaking it into smaller, simpler subproblems.

In Python, recursion is especially useful for problems that can be divided into identical smaller tasks, such as mathematical calculations, tree traversals or divide-and-conquer algorithms.

Working of Recursion

A **recursive function** is just like any other Python function except that it calls itself in its body. Let's see basic structure of recursive function:

def recursive_function(parameters):
ifbase_case_condition:
return base_result
else:
return recursive_function(modified_parameters)

Recursive function contains two key parts:

- **Base Case:** The stopping condition that prevents infinite recursion.

- **Recursive Case:** The part of the function where it calls itself with modified parameters.

Examples of Recursion

Let's understand recursion better with the help of some examples.

Example 1: Factorial Calculation

This code defines a recursive function to calculate factorial of a number, where function repeatedly calls itself with smaller values until it reaches the base case.

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    else:       # Recursive case
        return n * factorial(n - 1)

print(factorial(5))
```

Output

```
120
```

Explanation:

- **Base Case:** When **n == 0**, recursion stops and returns **1**.
- **Recursive Case:** Multiplies **n** with the factorial of **n-1** until it reaches the base case.

Python Strings

1. Introduction to Strings

- A **string** in Python is a sequence of characters enclosed within single quotes (' '), double quotes (" "), or triple quotes (''' ''' or """ """).
- Strings are **immutable**, meaning once created, they cannot be changed.
- Strings can store letters, numbers, and special characters.

Example:

```
str1 = 'Hello'
str2 = "Python"
str3 = '''This is
```

a multiline string'''

2. String Creation

- Single Quotes: 'Hello'
- Double Quotes: "Hello"
- Triple Quotes: '''Hello''' (used for multi-line strings)

3. Accessing Characters in a String

- Strings are indexed, starting from **0**.
- Negative indexing starts from **-1** (last character).

Example:

```
s = "Python"
print(s[0])   # P
print(s[-1])  # n
```

4. String Slicing

- Extracts part of a string using [start:end:step].
- start → index to begin (inclusive).
- end → index to stop (exclusive).
- step → skip value.

Example:

```
s = "PythonProgramming"
print(s[0:6])     # Python
print(s[7:])      # Programming
print(s[:6])      # Python
print(s[::2])     # Pto rgamn
print(s[::-1])    # gnimmargorPnohtyP (reversed string)
```

5. String Concatenation and Repetition

- Concatenation: +
- Repetition: *

Example:

```
s1 = "Hello"
s2 = "World"
print(s1 + " " + s2)   # Hello World
print(s1 * 3)          # HelloHelloHello
```

6. String Comparison

- Strings can be compared using relational operators (==, !=, <, >, <=, >=).
- Comparison is **lexicographical** (alphabetical order, based on Unicode values).

Example:

```
print("apple" < "banana")   # True
print("abc" == "ABC")       # False
```

8. String Formatting

Python supports different formatting techniques:

Using % Operator

```
name = "John"
age = 25
print("My name is %s and I am %d years old" % (name, age))
```

Using str.format()

```
print("My name is {} and I am {} years old".format(name, age))
print("My name is {0} and I am {1}".format(name, age))
```

Using f-strings (Python 3.6+)

```
print(f"My name is {name} and I am {age} years old")
```

9. Escape Sequences

Used to include special characters inside strings.

Escape Code	Meaning
\n	Newline
\t	Tab
\\	Backslash
\'	Single quote
\"	Double quote
\b	Backspace

Example:

```
print("Hello\nWorld")
print("Python\tProgramming")
```

10. Iterating Through Strings

```
s = "Hello"
for ch in s:
    print(ch)
```

11. String Membership Operators

```
s = "Python Programming"
print("Python" in s)     # True
print("Java" not in s)   # True
```

Important Notes

- Strings are **immutable** → cannot be modified in place.
- To change, you must create a **new string**.

- Strings are widely used in text processing, data parsing, and natural language processing.

Immutable Strings

- In Python, **strings are immutable**, meaning **once a string is created, it cannot be modified**.
- Operations like concatenation, slicing, replacement, or case conversion **return a new string object** rather than modifying the original.

Example:

```
s = "Python"
print(id(s))          # memory address of original string
s = s + " Language"   # new string is created
print(id(s))          # memory address is different
```

Why Immutable?

- Ensures security (used in keys, database connections, etc.).
- Thread-safety (multiple processes can access the same string safely).
- Performance optimization (Python caches strings).

2. Built-in String Functions

These are general functions applicable to strings:

Function	**Description**	**Example**
len(str)	Returns the length of string	len("Python") → 6
max(str)	Returns character with max ASCII value	max("abc") → c
min(str)	Returns character with min ASCII value	min("abc") → a

Function	Description	Example
sorted(str)	Returns a sorted list of characters	sorted("bad") → ['a','b','d']
str()	Converts data type to string	str(123) → '123'

3. Built-in String Methods

Python provides a rich set of methods for string manipulation.

A. Case Conversion

```
s = "python Programming"
print(s.upper())      # PYTHON PROGRAMMING
print(s.lower())      # python programming
print(s.title())      # Python Programming
print(s.capitalize()) # Python programming
print(s.swapcase())   # PYTHON pROGRAMMING
```

B. Searching & Checking

```
s = "Python Programming"
print(s.find("Pro"))      # 7
print(s.find("Java"))     # -1
print(s.index("Pro"))     # 7 (raises error if not found)
print(s.startswith("Py")) # True
print(s.endswith("ing"))  # True
```

C. Validation Methods

Return True or False:

```
print("123".isdigit())     # True
print("abc".isalpha())     # True
print("abc123".isalnum())  # True
print(" ".isspace())       # True
print("python".islower())  # True
print("PYTHON".isupper())  # True
print("Python".istitle())  # True
```

D. Trimming Whitespaces

```
s = "   Hello   "
print(s.strip())   # "Hello"
print(s.lstrip())  # "Hello   "
print(s.rstrip())  # "   Hello"
```

E. Replacing & Splitting

```
s = "I love Python"
print(s.replace("Python", "Java"))   # I love Java
print(s.split())   # ['I', 'love', 'Python']
print("one,two,three".split(","))  # ['one','two','three']
```

F. Joining

```
words = ['I', 'love', 'Python']
print(" ".join(words))   # I love Python
print("-".join(words))   # I-love-Python
```

G. Alignment Methods

```
s = "Python"
print(s.center(10, '*'))  # **Python**
print(s.ljust(10, '-'))   # Python----
print(s.rjust(10, '-'))   # ----Python
```

H. Counting

```
s = "Python Programming"
print(s.count("m"))   # 2
print(s.count("Pro")) # 1
```

I. Encoding & Decoding

```
s = "Python"
encoded = s.encode()
print(encoded)        # b'Python'
print(encoded.decode())  # Python
```

J. Formatting

```
name = "John"
age = 25

# Using format()
print("My name is {} and I am {} years old".format(name, age))

# Using f-strings
print(f"My name is {name} and I am {age} years old")

# Using %
print("My name is %s and I am %d years old" % (name, age))
```

Key Difference: Functions vs Methods

- **Functions** → Work on strings but are **not attached** to them (e.g., len(), max(), sorted()).
- **Methods** → Belong to string objects and are called using dot notation (e.g., "hello".upper()).

String Comparison

1) Operators used for string comparison

- == → Equal to
- != → Not equal to
- < → Less than
- > → Greater than

- <= → Less than or equal to
- >= → Greater than or equal to

2) Lexicographical comparison (dictionary order)

- Python compares strings character by character using **Unicode (ASCII) values**.

```
print("apple" < "banana")   # True  ('a' < 'b')
print("dog" > "cat")        # True  ('d' > 'c')
```

3) Equality and inequality

- == checks if both strings are identical.
- != checks if they are different.

```
print("hello" == "hello")   # True
print("hello" != "world")   # True
```

4) Case sensitivity

- Uppercase and lowercase letters have different Unicode values.
- Uppercase letters (A–Z) have smaller values than lowercase (a–z).

```
print("Zoo" < "apple")   # True  ('Z' < 'a')
print("abc" == "ABC")    # False
```

5) Case-insensitive comparison

- Convert both strings to the same case using .lower() or .casefold().

```
print("Python".lower() == "python".lower())   # True
print("Straße".casefold() == "strasse".casefold())  # True
```

6) Practical uses

- String comparison is useful in:
    - Sorting names or words
    - Searching and filtering text
    - Validating user input
    - Case-insensitive matching

So, Python compares strings **character by character** using Unicode values, and you can control case-sensitivity with .lower() or .casefold().

modules

Python modules are files containing Python code (functions, classes, variables, etc.) that can be imported and used in other Python programs. The import statement is the mechanism for accessing these modules and their contents.

Here's a breakdown of the import statement in Python modules:

1. Basic Import:

- **Syntax:** import module_name
- **Functionality:** This imports the entire module, making its contents accessible using the module_name.attribute syntax.

```
# my_module.py
def greet(name):
  return f"Hello, {name}!"

# main.py
import my_module
message = my_module.greet("Alice")
print(message)
```

2. Importing Specific Attributes:

- **Syntax:** from module_name import attribute1, attribute2
- **Functionality:** This imports only specified attributes (functions, classes, or variables) directly into the current namespace, allowing them to be used without the module_name. prefix.

```
# my_module.py
def greet(name):
   return f"Hello, {name}!"

class MyClass:
   pass

# main.py
from my_module import greet, MyClass
message = greet("Bob")
print(message)
obj = MyClass()
```

3. Importing All Attributes:

- **Syntax:** from module_name import *
- **Functionality:** This imports all public attributes from the module directly into the current namespace. While convenient, it can lead to name collisions and is generally discouraged in larger projects for clarity and maintainability.

4. Aliasing Imports:

- **Syntax:** import module_name as alias or from module_name import attribute as alias
- **Functionality:** This assigns an alias (alternative name) to the imported module or attribute, which can be useful for shorter names or resolving name conflicts.

```
import math as m
print(m.pi)
```

```
from collections import defaultdict as dd
my_dict = dd(int)
```

5. Module Search Path:

- When an import statement is encountered, Python searches for the module in a specific order:
    - The directory containing the input script (or current directory).
    - Directories listed in the PYTHONPATH environment variable.
    - Standard library directories.
    - The site-packages directory for third-party modules.

Python Modules

Last Updated : 23 Jul, 2025

Python Module is a file that contains built-in functions, classes,**its** and variables. There are many **Python modules**, each with its specific work.

In this article, we will cover all about Python modules, such as How to create our own simple module, Import Python modules, From statements in Python, we can use the alias to rename the module, etc.

What is Python Module

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.

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

Create a Python Module

To create a Python module, write the desired code and save that in a file with **.py** extension. Let's understand it better with an example:

Example:

Let's create a simple calc.py in which we define two functions, one **add** and another **subtract**.

```
# A simple module, calc.py
def add(x, y):
    return (x+y)

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

Import module in Python

We can import the functions, and classes defined in a module to another module using the **import statement** in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

Note*: A search path is a list of directories that the interpreter searches for importing a module.*

For example, to import the module calc.py, we need to put the following command at the top of the script.

Syntax to Import Module in Python

```
import module
```

Note: This does not import the functions or classes directly instead imports the module only. To access the functions inside the module the dot(.) operator is used.

Importing modules in Python Example

Now, we are importing the **calc** that we created earlier to perform add operation.

```
# importing  module calc.py
import calc

print(calc.add(10, 2))
```

Output:

```
12
```

Python Import From Module

Python's from statement lets you import specific attributes from a module without importing the module as a whole.

Import Specific Attributes from a Python module

Here, we are importing specific sqrt and factorial attributes from the math module.

```
# importing sqrt() and factorial from the
# module math
from math import sqrt, factorial

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

Output:

```
4.0
720
```

Import all Names

The * symbol used with the import statement is used to import all the names from a module to a current namespace.

Syntax:

```
from module_name import *
```

What does import * do in Python?

The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.

```
# importing sqrt() and factorial from the
# module math
from math import *

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

Output

```
4.0
720
```

Locating Python Modules

Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module, if not found then it looks for a list of directories defined in the sys.path. Python interpreter searches for the module in the following manner -

- First, it searches for the module in the current directory.
- If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH. The PYTHONPATH is an environment variable, consisting of a list of directories.
- If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.

Directories List for Modules

Here, sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

```
# importing sys module
import sys

# importing sys.path
print(sys.path)
```

Output:

```
['/home/nikhil/Desktop/gfg', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload', '', '/home/nikhil/.local/lib/python3.8/site-packages', '/usr/local/lib/python3.8/dist-packages', '/usr/lib/python3/dist-packages', '/usr/local/lib/python3.8/dist-packages/IPython/extensions', '/home/nikhil/.ipython']
```

dir() function in Python

The **dir()** function is a built-in Python tool used to list the attributes (like methods, variables, etc.) of an object. It helps inspect modules, classes, functions, and even user-defined objects during development and debugging.

Syntax

dir([object])

Parameters:

- **object (optional):** Any Python object (like list, dict, class, module, etc.)

Return Type: A list of names (strings) representing the attributes of the object or current scope.

Behavior

- **Without arguments:** Lists names in the current local scope.
- **With modules:** Lists all available functions, classes, and constants.
- **With user-defined objects:** Lists all attributes, including user-defined ones (if __dir__ is defined).
- **With built-in objects:** Lists all valid attributes and methods.

Examples dir() Function

Example 1: No Parameters Passed

In this example, we are using the dir() function to list object attributes and methods in Python. It provides a demonstration for exploring the available functions and objects in our Python environment.

```
print(dir())

import random
import math

print(dir())
```

Output:

```
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'traceback']
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'math', 'random', 'traceback']
```

Explanation:

- **dir()** lists names in the current local scope.
- Notice that after importing modules, the list includes **random**, **math**, etc, which were not present before importing them.

Modules and Namespace in Python

1) What is a Module?

- A **module** in Python is a file that contains Python definitions, functions, classes, and variables.

- Any .py file is considered a module.
- Purpose: to organize code, increase reusability, and reduce redundancy.

Example:

```
# math module usage
import math
print(math.sqrt(16))   # 4.0
```

2) Types of Modules

- **Built-in Modules** → Provided by Python (e.g., math, os, random).
- **User-defined Modules** → Created by the programmer (custom .py files).
- **Third-party Modules** → Installed using pip (e.g., numpy, pandas).

3) How Modules are Imported

- import module_name → imports whole module.
- import module_name as alias → short name.
- from module import function → imports specific function.
- from module import * → imports everything.

Example:

```
import random
print(random.randint(1,10))   # random number between 1–10
```

4) The dir() Function with Modules

- The dir() function shows all names (functions, classes, constants) defined in a module.

```
import math
print(dir(math))   # lists attributes inside math module
```

5) What is a Namespace?

- A **namespace** is a collection that maps **names (identifiers)** to **objects (values)**.

- Simply put, it’s where variables are stored.

Example:

```
x = 10    # 'x' is the name, 10 is the object
```

6) Types of Namespaces

1. **Built-in Namespace** → Reserved by Python (e.g., len(), id(), print()).
2. **Global Namespace** → Created when a program starts; includes variables defined at the top level.
3. **Local Namespace** → Created inside functions; includes variables defined within the function.

Example:

```
x = "global variable"   # global

def myfunc():
    y = "local variable"   # local
    print(y)

myfunc()
print(x)
```

7) Lifetime of Namespaces

- **Built-in Namespace** → exists as long as Python interpreter runs.
- **Global Namespace** → exists until the program ends.
- **Local Namespace** → exists only when the function is executing.

8) Relation between Modules and Namespace

- When you import a module, Python creates a **module namespace**.

- Each module has its **own separate namespace**, so names from one module don't clash with names in another.
- This is why you can have math.sqrt() and your own sqrt() without conflict.

summary:

- A **module** is a Python file containing code (functions, classes, variables) that can be reused.
- A **namespace** is a container that holds names mapped to objects.
- Python uses **built-in, global, and local namespaces** to manage variables.
- Modules always work in their own namespace to avoid conflicts.

Defining Our Own Modules

1) What is a Module?

- A **module** is simply a **Python file (.py)** that contains functions, classes, or variables.
- By creating your own modules, you can **organize code**, make it **reusable**, and keep your projects clean.
- Example: If you write a math_utils.py file with custom math functions, you can import and use it in multiple programs.

2) Steps to Define a Module

1. **Create a Python file** (.py).
2. **Write functions, classes, or variables** inside it.
3. **Save the file** with a meaningful name.
4. **Import the module** into another Python program.

3) Creating a Simple Module

File: mymodule.py

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


def add(a, b):
    return a + b


pi = 3.14159
```

File: main.py (using the module)

```
import mymodule


print(mymodule.greet("Vidhya"))    # Hello, Vidhya!
print(mymodule.add(5, 7))          # 12
print(mymodule.pi)                 # 3.14159
```

4) Different Ways to Import a Module

- **Normal import**

```
import mymodule
print(mymodule.greet("John"))
```

- **Import with alias**

```
import mymodule as mm
print(mm.greet("John"))
```

- **Import specific functions/variables**

```
from mymodule import greet, pi
print(greet("John"))
print(pi)
```

- **Import everything (not recommended)**

```
from mymodule import *
print(greet("John"))
```

5) The __name__ == "__main__" Concept

- When a Python file runs, a special variable __name__ is set.
- If the file is run directly → __name__ == "__main__".
- If the file is imported → __name__ == "filename".
- This allows a file to act as **both a script and a module**.

Example:

```
# mymodule.py
def greet(name):
    print(f"Hello, {name}")

if __name__ == "__main__":
    # Runs only when file is executed directly
    greet("Direct Run")
```

- If you run python mymodule.py → prints Hello, Direct Run.
- If you import it in another file → function is available but Direct Run won't print.

6) Module Search Path (sys.path)

- When you import a module, Python searches for it in:
  1. The **current directory**.
  2. The **Python standard library** (built-in modules).
  3. The **directories listed in sys.path**.

Example:

```
import sys
print(sys.path)   # shows list of paths where Python looks for modules
```

7) Organizing Modules into Packages

- A **package** is a collection of modules in a folder.
- A folder becomes a package if it contains an __init__.py file.
- Example:

```
mypackage/
   __init__.py
   math_utils.py
   string_utils.py
```

Usage:

```
from mypackage import math_utils
print(math_utils.add(2,3))
```

8) Advantages of Defining Our Own Modules

- **Code Reusability** → write once, use anywhere.
- **Better Organization** → split large projects into multiple files.
- **Maintainability** → easier to debug/update.
- **Namespace Management** → avoids name conflicts.

Here's the **flowchart showing how Python imports a user-defined module**:

- Starts with the import statement.
- Checks if the module is **built-in**.
- If not found, searches in the **current directory**.
- If still not found, searches in the **directories listed in sys.path**.
- If found → loads into the **namespace**.
- If not found → raises **ModuleNotFoundError**.

UNIT IV

Python Lists

In Python, a list is a built-in data structure that can hold an ordered collection of items. Unlike arrays in some languages, Python lists are very flexible:

- Can contain duplicate items
- **Mutable:** items can be modified, replaced, or removed
- **Ordered:** maintains the order in which items are added
- **Index-based:** items are accessed using their position (starting from 0)
- Can store mixed data types (integers, strings, booleans, even other lists)

Creating a List

Lists can be created in several ways, such as using square brackets, the list() constructor or by repeating elements. Let's look at each method one by one with example:

1. Using Square Brackets

We use square brackets [] to create a list directly.

```
a = [1, 2, 3, 4, 5] # List of integers
b = ['apple', 'banana', 'cherry'] # List of strings
c = [1, 'hello', 3.14, True] # Mixed data types

print(a)
print(b)
print(c)
```

Output

```
[1, 2, 3, 4, 5]
['apple', 'banana', 'cherry']
[1, 'hello', 3.14, True]
```

2. Using list() Constructor

We can also create a list by passing an iterable (like a tuple, string or another list) to the list() function.

```
a = list((1, 2, 3, 'apple', 4.5))
print(a)

b = list("GFG")
print(b)
```

Output

```
[1, 2, 3, 'apple', 4.5]
['G', 'F', 'G']
```

3. Creating List with Repeated Elements

We can use the multiplication operator * to create a list with repeated items.

```
a = [2] * 5
b = [0] * 7

print(a)
print(b)
```

Output

```
[2, 2, 2, 2, 2]

[0, 0, 0, 0, 0, 0, 0]
```

Accessing List Elements

Elements in a list are accessed using indexing. Python indexes start at 0, so a[0] gives the first element. Negative indexes allow access from the end (e.g., -1 gives the last element).

```
a = [10, 20, 30, 40, 50]
print(a[0])
print(a[-1])
print(a[1:4])   # elements from index 1 to 3
```

Output

```
10

50

[20, 30, 40]
```

Adding Elements into List

We can add elements to a list using the following methods:

- **append():** Adds an element at the end of the list.
- **extend():** Adds multiple elements to the end of the list.
- **insert():** Adds an element at a specific position.
- **clear():** removes all items. a = []

```
a.append(10)
print("After append(10):", a)

a.insert(0, 5)
print("After insert(0, 5):", a)

a.extend([15, 20, 25])
print("After extend([15, 20, 25]):", a)

a.clear()
print("After clear():", a)
```

Output

```
After append(10): [10]

After insert(0, 5): [5, 10]

After extend([15, 20, 25]): [5, 10, 15, 20, 25]

After clear(): []
```

Updating Elements into List

Since lists are mutable, we can update elements by accessing them via their index.

```
a = [10, 20, 30, 40, 50]
a[1] = 25
print(a)
```

Output

```
[10, 25, 30, 40, 50]
```

Removing Elements from List

We can remove elements from a list using:

- **remove():** Removes the first occurrence of an element.
- **pop():** Removes the element at a specific index or the last element if no index is specified.
- **del statement:** Deletes an element at a specified index. a = [10, 20, 30, 40, 50]

```
a.remove(30)
print("After remove(30):", a)

popped_val = a.pop(1)
print("Popped element:", popped_val)
print("After pop(1):", a)

del a[0]
print("After del a[0]:", a)
```

Output

```
After remove(30): [10, 20, 40, 50]

Popped element: 20

After pop(1): [10, 40, 50]

After del a[0]: [40, 50]
```

Iterating Over Lists

We can iterate over lists using loops, which is useful for performing actions on each item.

```
a = ['apple', 'banana', 'cherry']
for item in a:
    print(item)
```

Output

```
apple

banana

cherry
```

To learn various other methods, please refer to iterating over lists.

Nested Lists

A nested list is a list within another list, which is useful for representing matrices or tables. We can access nested elements by chaining indexes.

```
matrix = [ [1, 2, 3],
           [4, 5, 6],
           [7, 8, 9] ]
print(matrix[1][2])
```

Output

```
6
```

To learn more, please refer to Multi-dimensional lists in Python

List Comprehension

List comprehension is a concise way to create lists using a single line of code. It is useful for applying an operation or filter to items in an iterable, such as a list or range.

```
squares = [x**2 for x in range(1, 6)]
print(squares)
```

Output

```
[1, 4, 9, 16, 25]
```

Explanation:

- **for x in range(1, 6):** loops through each number from **1 to 5** (excluding 6).
- **x**2:** squares each number x.
- **[ ]:** collects all the squared numbers into a new list.

How Python Stores List Elements?

In Python, a list doesn’t store actual values directly. Instead, it stores references (pointers) to objects in memory. This means numbers, strings

and booleans are separate objects in memory and the list just keeps their addresses.
That’s why modifying a mutable element (like another list or dictionary) can change the original object, while immutables remain unaffected.

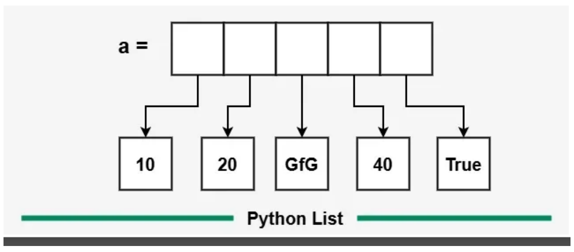
```
a = [10, 20, "GfG", 40, True]
print(a)
print(a[0])
print(a[1])
print(a[2])
```

Output

```
[10, 20, 'GfG', 40, True]

10

20

GfG
```

Explanation:

- The list a contains an integer (10, 20 and 40), a string ("GfG") and a boolean (True).
- Elements are accessed using indexing (a[0], a[1], etc.).
- Each element keeps its original type.



Python List methods

Python list methods are built-in functions that allow us to perform various operations on **lists**, such as a**dding, removing,** or **modifying** elements. In this article, we’ll explore all **Python list methods** with a simple example.

List Methods

Let's look at different list methods in Python:

- append(): Adds an element to the end of the list.

- copy(): Returns a shallow copy of the list.
- clear(): Removes all elements from the list.
- count(): Returns the number of times a specified element appears in the list.
- extend(): Adds elements from another list to the end of the current list.
- index(): Returns the index of the first occurrence of a specified element.
- insert(): Inserts an element at a specified position.
- pop(): Removes and returns the element at the specified position (or the last element if no index is specified).
- remove(): Removes the first occurrence of a specified element.
- reverse(): Reverses the order of the elements in the list.
- sort(): Sorts the list in ascending order (by default).

Examples of List Methods

append():

Syntax: *list_name.append(element)*

In the code below, we will add an element to the list.

```
a = [1, 2, 3]

# Add 4 to the end of the list
a.append(4)
print(a)
```

Output

```
[1, 2, 3, 4]
```

copy():

Syntax: *list_name.copy()*

In the code below, we will create a copy of a list.

```
a = [1, 2, 3]

# Create a copy of the list
b = a.copy()
```

```
print(b)
```

Output

```
[1, 2, 3]
```

clear():

Syntax: *list_name.clear()*

In the code below, we will clear all elements from the list.

```
a = [1, 2, 3]

# Remove all elements from the list
a.clear()
print(a)
```

Output

```
[]
```

count():

Syntax: *list_name.count(element)*

In the code below, we will count the occurrences of a specific element in the list.

```
a = [1, 2, 3, 2]

# Count occurrences of 2 in the list
print(a.count(2))
```

Output

```
2
```

extend():

Syntax: *list_name.extend(iterable)*

In the code below, we will extend the list by adding elements from another list.

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

```
# Extend list a by adding elements from list [3, 4]
a.extend([3, 4])
print(a)
```

Output

```
[1, 2, 3, 4]
```

index():

Syntax: *list_name.index(element)*

In the code below, we will find the index of a specific element in the list.

```
a = [1, 2, 3]

# Find the index of 2 in the list
print(a.index(2))
```

Output

```
1
```

insert():

Syntax: *list_name.insert(index, element)*

In the code below, we will insert an element at a specific position in the list.

```
a = [1, 3]

# Insert 2 at index 1
a.insert(1, 2)
print(a)
```

Output

```
[1, 2, 3]
```

pop():

Syntax: *list_name.pop(index)*

In the code below, we will remove the last element from the list.

```
a = [1, 2, 3]

# Remove and return the last element in the list
a.pop()
print(a)
```

Output

```
[1, 2]
```

remove():

Syntax: list_name.remove(element)

In the code below, we will remove the first occurrence of a specified element from the list.

```
a = [1, 2, 3]

# Remove the first occurrence of 2
a.remove(2)
print(a)
```

Output

```
[1, 3]
```

reverse():

Syntax: *list_name.reverse()*

In the code below, we will reverse the order of the elements in the list.

```
a = [1, 2, 3]

# Reverse the list order
a.reverse()
```

```
print(a)
```

Output

```
[3, 2, 1]
```

sort():

Syntax: *list_name.sort(key=None, reverse=False)*

In the code below, we will sort the elements of the list in ascending order

```
a = [3, 1, 2]

# Sort the list in ascending order
a.sort()
print(a)
```

Output

```
[1, 2, 3
```

Python Tuples

A tuple in Python is an immutable ordered collection of elements.

- Tuples are similar to lists, but unlike lists, they cannot be changed after their creation (i.e., they are immutable).
- Tuples can hold elements of different data types.
- The main characteristics of tuples are being ordered, heterogeneous and immutable.

Creating a Tuple

A tuple is created by placing all the items inside parentheses (), separated by commas. A tuple can have any number of items and they can be of different data types.

```
tup = ()
print(tup)

# Using String
tup = ('Geeks', 'For')
print(tup)

# Using List
li = [1, 2, 4, 5, 6]
print(tuple(li))

# Using Built-in Function
tup = tuple('Geeks')
print(tup)
```

Output

```
()

('Geeks', 'For')

(1, 2, 4, 5, 6)

('G', 'e', 'e', 'k', 's')
```

Let's understand tuple in detail:

Creating a Tuple with Mixed Datatypes.

Tuples can contain elements of various data types, including other tuples, lists, dictionaries and even functions.

```
tup = (5, 'Welcome', 7, 'Geeks')
print(tup)

# Creating a Tuple with nested tuples
tup1 = (0, 1, 2, 3)
tup2 = ('python', 'geek')
tup3 = (tup1, tup2)
print(tup3)
```

```
# Creating a Tuple with repetition
tup1 = ('Geeks',) * 3
print(tup1)

# Creating a Tuple with the use of loop
tup = ('Geeks')
n = 5
for i in range(int(n)):
    tup = (tup,)
    print(tup)
```

Output

```
(5, 'Welcome', 7, 'Geeks')

((0, 1, 2, 3), ('python', 'geek'))

('Geeks', 'Geeks', 'Geeks')

('Geeks',)

(('Geeks',),)

((('Geeks',),),)

(((('Geeks',),),),)

((((('Geeks',),),),),)
```

Python Tuple Basic Operations

Below are the Python tuple operations.

- Accessing of Python Tuples
- Concatenation of Tuples
- Slicing of Tuple
- Deleting a Tuple

Accessing of Tuples

We can access the elements of a tuple by using indexing and slicing, similar to how we access elements in a list. Indexing starts at 0 for the first element and goes up to n-1, where n is the number of elements in the tuple. Negative indexing starts from -1 for the last element and goes backward.

```
# Accessing Tuple with Indexing
tup = tuple("Geeks")
print(tup[0])

# Accessing a range of elements using slicing
print(tup[1:4])
print(tup[:3])

# Tuple unpacking
tup = ("Geeks", "For", "Geeks")

# This line unpack values of Tuple1
a, b, c = tup
print(a)
print(b)
print(c)
```

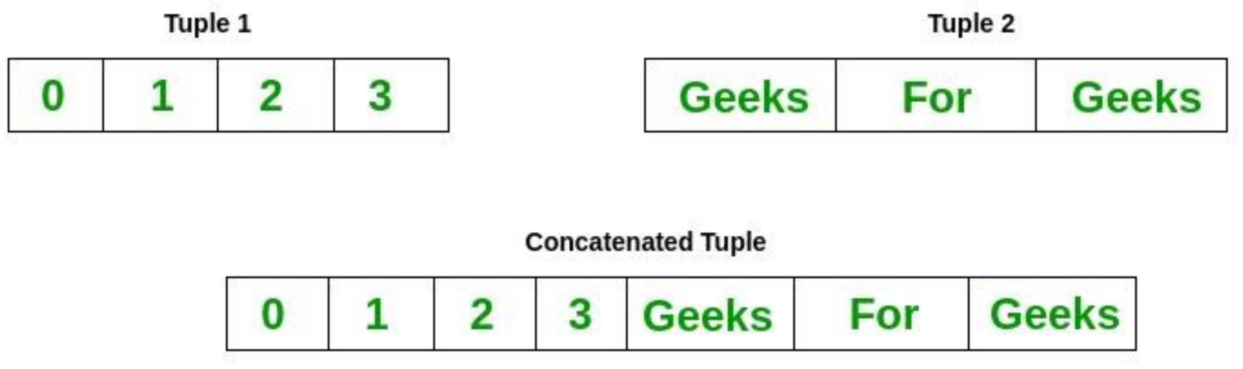
Output

```
G

('e', 'e', 'k')

('G', 'e', 'e')

Geeks

For

Geeks
```

Concatenation of Tuples

Tuples can be concatenated using the + operator. This operation combines two or more tuples to create a new tuple.

Note: *Only the same datatypes can be combined with concatenation, an error arises if a list and a tuple are combined.*



```
tup1 = (0, 1, 2, 3)
tup2 = ('Geeks', 'For', 'Geeks')

tup3 = tup1 + tup2
print(tup3)
```

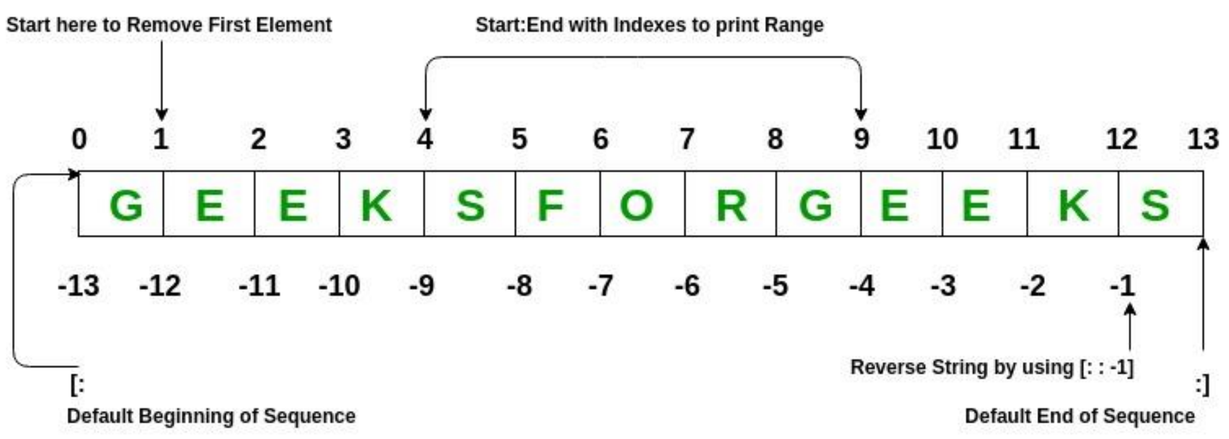
Output

```
(0, 1, 2, 3, 'Geeks', 'For', 'Geeks')
```

Slicing of Tuple

Slicing a tuple means creating a new tuple from a subset of elements of the original tuple. The slicing syntax is tuple[start:stop:step].

Note- *Negative Increment values can also be used to reverse the sequence of Tuples.*


Start here to Remove First Element
Start:End with Indexes to print Range
0 1 2 3 4 5 6 7 8 9 10 11 12 13
G E E K S F O R G E E K S
-13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
Reverse String by using [: : -1]
[:
Default Beginning of Sequence
:]
Default End of Sequence

```
tup = tuple('GEEKSFORGEEKS')

# Removing First element
print(tup[1:])

# Reversing the Tuple
print(tup[::-1])

# Printing elements of a Range
print(tup[4:9])
```

Output

```
('E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S')

('S', 'K', 'E', 'E', 'G', 'R', 'O', 'F', 'S', 'K', 'E', 'E', 'G')

('S', 'F', 'O', 'R', 'G')
```

Deleting a Tuple

Since tuples are immutable, we cannot delete individual elements of a tuple. However, we can delete an entire tuple using del statement.

Note: *Printing of Tuple after deletion results in an Error.*

```
tup = (0, 1, 2, 3, 4)
del tup
```

```
print(tup)
```

Output

```
ERROR!
Traceback (most recent call last):
File "<main.py>", line 6, in <module>
NameError: name 'tup' is not defined
```

Tuple Unpacking with Asterisk (*)

In Python, the **" * "** operator can be used in tuple unpacking to grab multiple items into a list. This is useful when you want to extract just a few specific elements and collect the rest together.

```
tup = (1, 2, 3, 4, 5)

a, *b, c = tup

print(a)
print(b)
print(c)
```

Output

```
1

[2, 3, 4]

5
```

Explanation:

- **a** gets the first item.
- **c** gets the last item.
- ***b** collects everything in between into a list.

Difference Between List and Tuple in Python

In Python, **lists** and **tuples** both store collections of data, but differ in mutability, performance and memory usage. Lists are mutable, allowing modifications, while tuples are immutable. Choosing between them depends on whether you need to modify the data or prioritize performance and memory efficiency.

Key Differences between List and Tuple

S.No	List	Tuple
1	Lists are mutable(can be modified).	Tuples are immutable(cannot be modified).
2	Iteration over lists is time-consuming.	Iterations over tuple is faster
3	Lists are better for performing operations, such as insertion and deletion.	Tuples are more suitable for accessing elements efficiently.
4	Lists consume more memory.	Tuples consumes less memory
5	Lists have several built-in methods.	Tuples have fewer built-in methods.
6	Lists are more prone to unexpected changes and errors.	Tuples, being immutable are less error prone.

Python Dictionary

Python dictionary is a data structure that stores the value in key: value pairs. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be immutable.

- Keys are case sensitive which means same name but different cases of Key will be treated distinctly.
- Keys must be immutable which means keys can be strings, numbers or tuples but not lists.
- Duplicate keys are not allowed and any duplicate key will overwrite the previous value.
- Internally uses hashing. Hence, operations like search, insert, delete can be performed in Constant Time.
- From Python 3.7 Version onward, Python dictionary are Ordered.

How to Create a Dictionary

Dictionary can be created by placing a sequence of elements within curly {} braces, separated by a 'comma'.

```
d1 = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print(d1)

# create dictionary using dict() constructor
d2 = dict(a = "Geeks", b = "for", c = "Geeks")
print(d2)
```

Output

```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

{'a': 'Geeks', 'b': 'for', 'c': 'Geeks'}
```

Accessing Dictionary Items

We can access a value from a dictionary by using the **key** within square brackets or get() method.

```
d = { "name": "Prajjwal", 1: "Python", (1, 2): [1,2,4] }

# Access using key
print(d["name"])
```

```
# Access using get()
print(d.get("name"))
```

Output

```
Prajjwal

Prajjwal
```

Adding and Updating Dictionary Items

We can add new key-value pairs or update existing keys by using assignment.

```
d = {1: 'Geeks', 2: 'For', 3: 'Geeks'}

# Adding a new key-value pair
d["age"] = 22

# Updating an existing value
d[1] = "Python dict"

print(d)
```

Output

```
{1: 'Python dict', 2: 'For', 3: 'Geeks', 'age': 22}
```

Removing Dictionary Items

We can remove items from dictionary using the following methods:

- **del**: Removes an item by key.
- **pop():** Removes an item by key and returns its value.
- **clear():** Empties the dictionary.
- **popitem():** Removes and returns the last key-value pair. d = {1: 'Geeks', 2: 'For', 3: 'Geeks', 'age':22}

```
# Using del to remove an item
del d["age"]
```

```
print(d)

# Using pop() to remove an item and return the value
val = d.pop(1)
print(val)

# Using popitem to removes and returns
# the last key-value pair.
key, val = d.popitem()
print(f"Key: {key}, Value: {val}")

# Clear all items from the dictionary
d.clear()
print(d)
```

Output

```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Geeks

Key: 3, Value: Geeks

{}
```

Iterating Through a Dictionary

We can iterate over **keys** [using keys() method] , **values** [using values() method] or both [using item() method] with a for loop.

```
d = {1: 'Geeks', 2: 'For', 'age':22}

# Iterate over keys
for key in d:
    print(key)

# Iterate over values
for value in d.values():
    print(value)
```

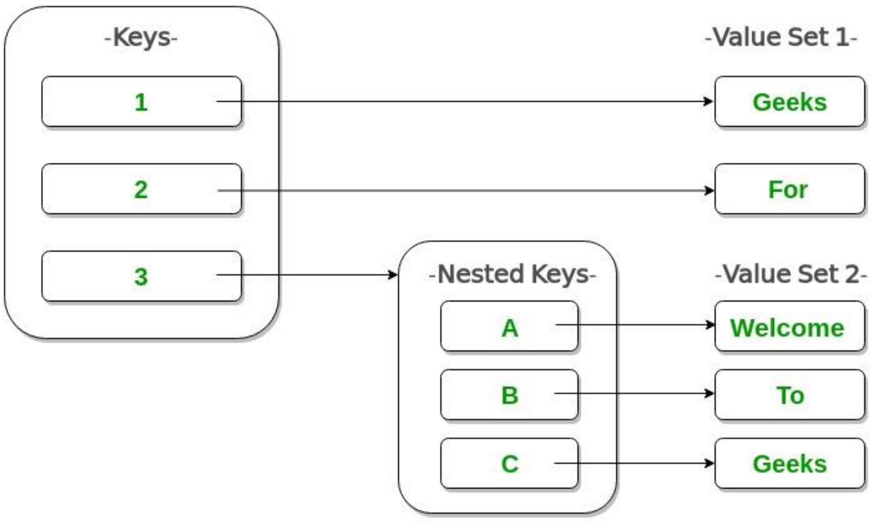
```
# Iterate over key-value pairs
for key, value in d.items():
    print(f"{key}: {value}")
```

Output

```
1

2

age

Geeks

For

22

1: Geeks

2: For

age: 22
```

Read in detail: Ways to Iterating Over a Dictionary

Nested Dictionaries



Example of Nested Dictionary:

```
d = {1: 'Geeks', 2: 'For',
    3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}

print(d)
```

Difference between a List and a Dictionary

The following table shows some differences between a list and a dictionary in Python:

List	**Dictionary**
The list is a collection of index value pairs like ArrayList in Java and Vectors in C++.	The dictionary is a hashed structure of the key and value pairs.
The list is created by placing elements in **[ ]** separated by commas ", "	The dictionary is created by placing elements in **{ }** as "key":"value", each key-value pair is separated by commas ", "
The indices of the list are integers starting from 0.	The keys of the dictionary can be of any immutable data type.
The elements are accessed via indices.	The elements are accessed via key.
The order of the elements entered is maintained.	They are unordered in python 3.6 and below and are ordered in python 3.7 and above.
Lists can duplicate values since each values have unique index.	Dictionaries cannot contain duplicate keys but can contain duplicate values since each value has unique key.

List	Dictionary
Average time taken to search a value in list takes O[n].	Average time taken to search a key in dictionary takes O[1].
Average time to delete a certain value from a list takes O[n].	Average time to delete a certain key from a dictionary takes O[1].

UNIT V

File Handling in Python

File handling refers to the process of performing operations on a file, such as creating, opening, reading, writing and closing it through a programming interface. It involves managing the data flow between the program and the file system on the storage device, ensuring that data is handled safely and efficiently.

Why do we need File Handling

- To store data permanently, even after the program ends.
- To access external files like .txt, .csv, .json, etc.
- To process large files efficiently without using much memory.
- To automate tasks like reading configs or saving outputs.
- To handle input/output in real-world applications and tools.

Opening a File

To open a file, we can use open() function, which requires file-path and mode as arguments:

Syntax:

file = open('filename.txt', 'mode')

- **filename.txt:** name (or path) of the file to be opened.
- **mode:** mode in which you want to open the file (read, write, append, etc.).

Note: *If you don't specify the mode, Python uses* ***'r'*** *(read mode) by default.*

Basic Example: Opening a File

```
f = open("geek.txt", "r")
print(f)
```

Explanation: This code opens file **geek.txt** in read mode. If the file exists, it returns a file object connected to that file; if the file does not exist, Python raises a FileNotFoundError.

Closing a File

The file.close() method closes the file and releases the system resources. If the file was opened in write or append mode, closing ensures that all changes are properly saved.

```
file = open("geek.txt", "r")
# Perform file operations
file.close()
```

We will also see later how closing can be handled automatically using the with statement and how to ensure files close properly using exception handling.

Checking File Properties

Once the file is open, we can check some of its properties:

```
f = open("geek.txt", "r")
print("Filename:", f.name)
print("Mode:", f.mode)
print("Is Closed?", f.closed)

f.close()
print("Is Closed?", f.closed)
```

Output:

```
Filename: geek.txt
Mode: r
Is Closed? False
Is Closed? True
```

Explanation:

- **f.name:** Returns the name of the file that was opened (in this case, "geek.txt").
- **f.mode:** Tells us the mode in which the file was opened. Here, it’s 'r' which means read mode.
- **f.closed:** Returns a boolean value- False when file is currently open otherwise True.

Reading and Writing to text files in Python

Python provides built-in functions for creating, writing and reading files. Two types of files can be handled in Python, normal text files and binary files (written in binary format, 0s and 1s).

- **Text files:** Each line of text is terminated with a special character called EOL (End of Line), which is new line character ('\n') in Python by default.
- **Binary files:** There is no terminator for a line and data is stored after converting it into machine-understandable binary format.

This article focuses on opening, closing, reading and writing data in a text file. Here, we will also see how to get Python output in a text file.

Open Text File

It is done using open() function. No module is required to be imported for this function.

File_object = open(r"File_Name","Access_Mode")

Example: Here, file1 is created as an object for MyFile1 and file2 as object for MyFile2.

```
# Open MyFile1.txt in append mode
file1 = open("MyFile1.txt", "a")
```

```
# Open MyFile2.txt in D:\Text with write+ mode
file2 = open(r"D:\Text\MyFile2.txt", "w+")
```

Also Read: File Mode in Python

Read Text File

There are three ways to read txt file. Let's understand it one by one:

1. Using read()

read(): Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

File_object.read([n])

2. Using readline()

readline(): Reads one line of the file and returns in form of a string. For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

File_object.readline([n])

3. Using readlines()

readlines(): Reads all the lines and return them as each line a string element in a list.

File_object.readlines()

Note: *'\n' is treated as a special character of two bytes.*

Example: In this example, a file myfile.txt is created in write mode (w) and data is added using write() and writelines(). The file is then reopened in read and append mode (r+) to demonstrate different read operations: read(), readline(), read(n), readline(n) and readlines(). Finally, file is closed.

```
file1 = open("myfile.txt", "w")
L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

file1.write("Hello \n")          # write single line
file1.writelines(L)             # write multiple lines
file1.close()                   # close file

file1 = open("myfile.txt", "r+") # reopen file in read+append mode
```

```
print("Output of read():")
print(file1.read())             # read whole file
print()

file1.seek(0)                   # move cursor to start
print("Output of readline():")
print(file1.readline())         # read first line
print()

file1.seek(0)
print("Output of read(9):")
print(file1.read(9))            # read first 9 chars
print()

file1.seek(0)
print("Output of readline(9):")
print(file1.readline(9))        # read 9 chars from line
print()

file1.seek(0)
print("Output of readlines():")
print(file1.readlines())        # read all lines as list
print()

file1.close()
```

Output

```
Output of read():
Hello
This is Delhi
This is Paris
This is London
Output of readline():
Hello
```

Output of read(9):

Hello

Th

Output of readline(9):

Hello

Output of readlines():

['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']

Write to Text File

There are two ways to write in a file:

1. Using write()

write(): Inserts the string str1 in a single line in the text file.

File_object.write(str1)

```
file = open("Employees.txt", "w")


for i in range(3):
name = input("Enter the name of the employee: ")
file.write(name)
file.write("\n")


file.close()
print("Data is written into the file.")
```

Output

Data is written into the file.

2. Using writelines()

writelines(): For a list of string elements, each string is inserted in text file. Used to insert multiple strings at a single time.

File_object.writelines(L) for L = [str1, str2, str3]

```
file1 = open("Employees.txt", "w")
lst = []
for i in range(3):
        name = input("Enter the name of the employee: ")
        lst.append(name + '\n')
```

```
file1.writelines(lst)
file1.close()
print("Data is written into the file.")
```

Output

Data is written into the file.

Append to a File

In this example, a file named "myfile.txt" is initially opened in write mode ("w") to write lines of text. The file is then reopened in append mode ("a") and "Today" is added to existing content. The output after appending is displayed using readlines. Subsequently, file is reopened in write mode, overwriting content with "Tomorrow". Final output after writing is displayed using readlines.

```
file1 = open("myfile.txt", "w")
L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]
file1.writelines(L)
file1.close()

# Append-adds at last
file1 = open("myfile.txt", "a")  # append mode
file1.write("Today \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after appending")
print(file1.readlines())
print()
file1.close()

# Write-Overwrites
file1 = open("myfile.txt", "w")  # write mode
file1.write("Tomorrow \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after writing")
print(file1.readlines())
print()
file1.close()
```

Output

Output of Readlines after appending
['This is Delhi \n', 'This is Paris \n', 'This is London \n', 'Today \n']
Output of Readlines after writing
['Tomorrow \n']

Related Article: File Objects in Python

Closing a Text File

Python close() function closes file and frees memory space acquired by that file. It is used at the time when file is no longer needed or if it is to be opened in a different file mode.

File_object.close()

```
file1 = open("MyFile.txt","a")
file1.close()
```

with statement in Python

The "**with**" statement in Python simplifies resource management by automatically handling setup and cleanup tasks. It's commonly used with files, network connections and databases to ensure resources are properly released even if errors occur making your code cleaner.

Why do we need "with" statement?

- **Simplifies Resource Management :** with statement ensures that resources are properly acquired and released, reducing the likelihood of resource leaks.
- **Replaces Try-Except-Finally Blocks:** Traditionally, resource management required try-except-finally blocks to handle exceptions and ensure proper cleanup. The with statement provides a more concise alternative.
- **Enhances Readability:** By reducing boilerplate code, the with statement improves code readability and maintainability.

Safe File Handling

When working with files, it’s important to open and close them properly to avoid issues like memory leaks or file corruption. Below are two simple examples using a file named **example.txt** that contains:

Hello, World!

Example 1 : Without "with" (Manual closing)

```
file = open("example.txt", "r")
try:
    content = file.read()
    print(content)
finally:
    file.close()  # Ensures the file is closed
```

Output

Hello, World!

Explanation: This code opens **"example.txt"** in read mode, reads its content, prints it and ensures file is closed using a finally block.

Example 2: Using "with" (Automatic closing)

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)  # File closes automatically
```

Output

Hello, World!

Explanation: with open(...) statement reads and prints file's content while automatically closing it, ensuring efficient resource management without a finally block.

Resource Management Using "with" Statement

Python’s with statement simplifies resource handling by managing setup and cleanup automatically. Let’s explore how it works and where it’s commonly used.

1. Using with statement for file handling

File handling is one of the most common use cases for with statement. When opening files using open(), the with statement ensures that the file is closed automatically after operations are completed.

Example 1 : Reading a file

```
with open("example.txt", "r") as file:
    contents = file.read()
    print(contents)  # Print file content
```

Output:

Hello, World!

Explanation: Opens example.txt in read mode ("r") and with ensures automatic file closure after reading and file.read() reads the entire file content into contents.

Example 2 : Writing to a file

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

Output:

Hello, Python with statement!

Explanation: The file is opened in write mode ("w"). After the with block, the file is automatically closed.

2. Replacing Try-Except finally with "with" statement

Without "with" statement, you need to explicitly manage resource closure:

Example 1 : Without using "with"

```
file = open("example.txt", "w")
try:
    file.write("Hello, Python!")
finally:
    file.close()  # Ensure file is closed
```

Output:

Hello, World!

Explanation: This code opens example.txt in write mode ("w"), creating or clearing it. The **try block** writes "Hello, Python!" and **finally** ensures the file closes, preventing resource leaks.

Example 2: Using "with"

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

Output

Hello, Python!

Explanation: This code opens **example.txt** in write mode ("w") using with, which ensures automatic file closure. It writes "Hello, Python!" to the file, replacing any existing content.

3. Context Managers and "with" statement

The with statement relies on context managers, which manage resource allocation and deallocation using two special methods:

- __enter__(): Acquires the resource and returns it.
- __exit__(): Releases the resource when the block exits

Example: Custom context manager for file writing

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()

# using the custom context manager
with FileManager('example.txt', 'w') as file:
    file.write('Hello, World!')
```

Output:

Hello, World!

Explanation:

- __init__() initializes the filename and mode, __enter__() opens the file, and __exit__() ensures it closes automatically.
- with FileManager('file.txt', 'w') as file: opens "file.txt" in write mode.
- file.write('Hello, World!') writes to the file, which closes upon exiting the block.

Python File Methods

Python has a set of methods available for the file object.

Method	Description
close()	Closes the file
detach()	Returns the separated raw stream from the buffer
fileno()	Returns a number that represents the stream, from the operating system's perspective
flush()	Flushes the internal buffer
isatty()	Returns whether the file stream is interactive or not

read()	Returns the file content
readable()	Returns whether the file stream can be read or not
readline()	Returns one line from the file
readlines()	Returns a list of lines from the file
seek()	Change the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writable()	Returns whether the file can be written to or not
write()	Writes the specified string to the file
writelines()	Writes a list of strings to the file