Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

```
def my_function():
  print("Hello from a function")
my_function()
```

Arguments

- Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
def my_function(fname):
print(fname + "Kumar")
my_function("Sandeep")
my_function("Venkat")
my_function("Bhaskar")
```

- Functions are blocks of code that can be called multiple times from different parts of a program.

Example:

```
def greet(name):
    print("Hello, " + name + "!")
greet("Kluians")
```

Output: "Hello, Kluians!"

Return Statement:

The return statement is used to exit a function and return a value to the caller.

Example 1:

```
def add(a, b):
    return a + b
result = add(5, 10)
print(result)  # Output: 15
```

Example 2: Returning multiple values

```
def calculate_area(length, width):
    area = length * width
    perimeter = 2 * (length + width)
    return area, perimeter

area, perimeter = calculate_area(10, 20)
```

```
print("Area:", area)  # Output: Area: 200
print("Perimeter:", perimeter)  # Output: Perimeter: 60
```

Example 3: Returning a list

```
def get_even_numbers(numbers):
    even_numbers = [num for num in numbers if num % 2 == 0]
    return even_numbers
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = get_even_numbers(numbers)
print(even_numbers)  # Output: [2, 4, 6]
```

Example 4: Returning a dictionary

```
def get_person_info(name, age):
    person_info = {"name": name, "age": age}
    return person_info

person_info = get_person_info("Alice", 30)
print(person_info)  # Output: {'name': 'Alice', 'age': 30}
```

Example 5: Returning None

```
def print_hello():
    print("Hello!")
result = print_hello()
print(result)  # Output: None
```

In this example, the print_hello function does not explicitly return a value, so it returns None by default.

Note that the return statement can be used to exit a function early, without executing the rest of the code in the function. For example:

```
def check_age(age):
    if age < 18:
        return "You are not eligible to vote."
    return "You are eligible to vote."
print(check_age(17))  # Output: You are not eligible to vote.
```

Types of Arguments in Functions:

1. Positional Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable-Length Arguments

1. Positional Arguments:

Positional arguments are passed to a function in the order they are defined.

Example:

```
def greet(name, age):
    print("Hello, " + name + "! You are " + str(age) + " years old.")
greet("Alice", 30)
```

2. Keyword Arguments:

Keyword arguments are passed to a function using the argument name.

Example:

```
def greet(name, age):
    print("Hello, " + name + "! You are " + str(age) + " years old.")
greet(age=30, name="Alice")
```

3. Default Arguments:

Default arguments have a default value if not passed to the function.

Example:

```
def greet(name, age=30):
    print("Hello, " + name + "! You are " + str(age) + " years old.")
greet("Alice")
```

4. Variable-Length Arguments:

Variable-length arguments can be passed to a function using *args or **kwargs.

Example:

```
def local_x(*args):
    print(args)
local_x(50, 60)
```

Key Differences:

Feature	`*args`	`**kwargs`
Purpose	For passing **positional arguments**	For passing **keyword arguments**
Type	Stored as a **tuple**	Stored as a **dictionary**
When to use	When you want to handle a variable number of positional arguments.	When you want to handle a variable number of keyword (named) arguments.
Example Call	`function(1, 2, 3)`	`function(a=1, b=2, c=3)`

Global Variables and Local Variables:

Functions can have global and local variables, positional arguments, keyword arguments, default arguments, and variable-length arguments.

Example:

- x = 10 # Global variable

```
def change_x(**kwargs):
    global x  # Access global variable
    x = kwargs.get('new_x', 20)
def print_x():
    print(x)  # Access global variable
change_x(new_x=30)
print_x()  # Output: 30
def local_x(*args):
    x = 40  # Local variable
    print(x)  # Output: 40
    print(args)  # Output: (50, 60)
local_x(50, 60)
print_x()  # Output: 30
```

Solution:

The function change_x changes the global variable x using keyword arguments. The function print_x prints the global variable x. The function local_x creates a local variable x and prints it, along with variable-length arguments.

Lambda Function:

A lambda function is a small anonymous function that can take any number of arguments, but can only have one expression.

It's a shorthand way to create small, one-time use functions. Lambda functions are defined using the lambda keyword followed by a list of arguments, a colon, and an expression.

Basic Python Programming Example:

```
# Define a lambda function to square a number

square = lambda x: x ** 2

# Use the lambda function

print(square(5))  # Output: 25
```

In this example, the lambda function takes one argument x and returns its square x ** 2. The lambda function is assigned to the variable square, which can then be used to call the function.

Breakdown:

- lambda keyword indicates the start of the lambda function
- x is the argument passed to the lambda function
- : separates the arguments from the expression
- x ** 2 is the expression that gets executed when the lambda function is called

Benefits of Lambda Functions:

- Concise code
- Anonymous, no need to declare a named function
- Can be used as higher-order functions (passed as arguments to other functions)
- Can be used as event handlers or callbacks

Common Use Cases:

- Data processing and transformation
- Event handling and callbacks
- Higher-order functions and functional programming
- Quick, one-time use functions

A **lambda function** in Python is an anonymous function (a function without a name) that is defined using the lambda keyword.

It can take any number of arguments but has only one expression, which is evaluated and returned.

Basic Syntax:

lambda arguments: expression

- arguments: Input values to the lambda function.
- expression: A single expression whose result will be returned.

Characteristics of Lambda Functions:

1. **Anonymous**: Lambda functions do not require a name.
2. **Single Expression**: They can only have one expression (no multiple statements).
3. **Used in Short-term Scenarios**: Typically used where a simple function is required temporarily, like inside higher-order functions (e.g., map(), filter()).

Example 1: A Simple Lambda Function

```
# Regular function to add 10 to a number
def add_10(x):
    return x + 10

# Lambda equivalent
add_10_lambda = lambda x: x + 10
print(add_10_lambda(5))  # Output: 15
```

Example 2: Lambda Function with Multiple Arguments

```
# Regular function to multiply two numbers
def multiply(x, y):
    return x * y
```

 # Lambda equivalent

```
multiply_lambda = lambda x, y: x * y
print(multiply_lambda(4, 5))  # Output: 20
```

Example 3: Using Lambda with map()

The map() function applies a function to all items in an iterable (like a list).

```
numbers = [1, 2, 3, 4]
squares = map(lambda x: x**2, numbers)
print(list(squares))  # Output: [1, 4, 9, 16]
```

Example 4: Using Lambda with filter()

The filter() function filters elements of an iterable based on a condition.

```
numbers = [1, 2, 3, 4, 5, 6]
evens = filter(lambda x: x % 2 == 0, numbers)
print(list(evens))  # Output: [2, 4, 6]
```

Example 5: Lambda with sorted() for Custom Sorting

You can use a lambda function to customize sorting behavior.

```
# Sort a list of tuples by the second value in each tuple
tuples = [(1, 'one'), (3, 'three'), (2, 'two')]
sorted_tuples = sorted(tuples, key=lambda x: x[1])
print(sorted_tuples)  # Output: [(1, 'one'), (3, 'three'), (2, 'two')]
```

When to Use Lambda Functions:

- When you need a small function for a short period.
- Inside functions like map(), filter(), and sorted().
- When defining simple callbacks.

Limitations of Lambda Functions:

- Limited to a single expression.
- Less readable for complex operations compared to regular functions.
- Cannot contain multiple statements or assignments.

Lists in Python

- Lists are mutable, meaning they can be modified after creation.

- Lists can contain different data types, including strings, integers, floats, and other lists.

- Lists are indexed, meaning each element has a unique position (index) in the list.

- Lists have various methods for modifying and manipulating their contents.

There are 11 methods in Lists Datatype.

1. Append( )

Description: Adds an element to the end of the list.

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)          #Output: [1, 2, 3, 4]
```

2. Clear( )

Description: Removes all elements from the list.

```
my_list = [1, 2, 3]
my_list.clear()
print(my_list)          #Output: []
```

3. Copy( )

Description: Creates a shallow copy of the list.

my_list = [1, 2, 3]

new_list = my_list.copy()

print(new_list) **#Output:** [1, 2, 3]

4. Count( )

Description: Returns the number of occurrences of an element in the list.

my_list = [1, 2, 2, 3]

print(my_list.count(2)) **# Output:** 2

5. Extend( )

Description: Adds multiple elements to the end of the list.

my_list = [1, 2, 3]

my_list.extend([4, 5, 6])

print(my_list) **#Output:** [1, 2, 3, 4, 5, 6]

6. Index( )

Description: Returns the index of the first occurrence of an element in the list.

my_list = [1, 2, 3]

print(my_list.index(2)) **#Output:** 1

7. Insert( )

Description: Inserts an element at a specified position in the list.

my_list = [1, 2, 3]

my_list.insert(1, 4)

print(my_list) **#Output:** [1, 4, 2, 3]

8. Pop( )

Description: Removes and returns the last element from the list.

my_list = [1, 2, 3]

print(my_list.pop()) **#Output:** 3

print(my_list) [1, 2]

9. Remove( )

Description: Removes the first occurrence of an element from the list.

my_list = [1, 2, 3]

my_list.remove(2)

print(my_list) **#Output:** [1, 3]

10. Reverse( )

Description: Reverses the order of elements in the list.

my_list = [1, 2, 3]

my_list.reverse()

print(my_list) **#Output:** [3, 2, 1]

11. Sort( )

Description: Sorts the elements in the list in ascending order.

```
my_list = [3, 2, 1]
my_list.sort()
print(my_list)        #Output: [1, 2, 3]
```

Python – List Comprehension

Nested list:

```
matrix = []
for i in range(5):
 # Append an empty sublist inside the list
 matrix.append([])
 for j in range(5):
        matrix[i].append(j)
print(matrix)
```

o/p:[[0,1,2,3,4], [0,1,2,3,4], [0,1,2,3,4],,[0,1,2,3,4], [0,1,2,3,4]]

Python – List Comprehension

A Python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python list.
Example:

```
Num = [12, 13, 14]
doubls= [x *2  for x in numbers]
print(print(s)
```

Output
[24, 26, 28]

Python List Comprehension Syntax

Syntax: *newList =* ***[*** *expression(element)* ***for*** *element* ***in*** *oldList* ***if*** *condition* ***]***

Parameter:

- ***expression****: Represents the operation you want to execute on every item within the iterable.*
- ***element****: The term "variable" refers to each value taken from the iterable.*
- ***iterable****: specify the sequence of elements you want to iterate through.(e.g., a list, tuple, or string).*
- ***condition****: (Optional) A filter helps decide whether or not an element should be added to the new list.*

Return: *The return value of a list comprehension is a new list containing the modified elements that satisfy the given criteria.*

Python List comprehension provides a much more short syntax for creating a new list based on the values of an existing list.

List Comprehension in Python Example

Here is an example of using list comprehension to find the square of the number in Python.

- Python

```
numbers = [12, 13, 14]
    s= [x *2  for x in numbers]
    print(s)
```

Output

[24,26,28]

Iteration with List Comprehension

In this example, we are assigning 1, 2, and 3 to the list and we are printing the list using List Comprehension.

Using list comprehension to iterate through loop
List = [character **for** character **in** [1, 2, 3]]

# Displaying list
print(List)

Output

[1, 2, 3]

Even list using List Comprehension

In this example, we are printing the even numbers from 0 to 10 using List Comprehension.

```
        list = [i for i in range(11) if i % 2 == 0]
print(list)
```

Output

[0,2,4,6,8,10]

Matrix using List Comprehension

In this example, we are assigning integers 0 to 2 to 3 rows of the matrix and printing it using List Comprehension.

matrix = [[j **for** j **in** range(3)] **for** i **in** range(3)]

print(matrix)

Output

[[0, 1, 2], [0, 1, 2], [0, 1, 2]]

List Comprehensions vs For Loop

There are various ways to iterate through a list. However, the most common approach is to use the *for* loop. Let us look at the below example:

Empty list

```
List = []

# Traditional approach of iterating
for character in 'Geeks 4 Geeks!':
    List.append(character)

# Display list
print(List)
```

Output

```
['G', 'e', 'e', 'k', 's', ' ', '4', ' ', 'G', 'e', 'e', 'k', 's', '!']
```

Above is the implementation of the traditional approach to iterate through a list, string, tuple, etc. Now, list comprehension in Python does the same task and also makes the program more simple.

List Comprehensions translate the traditional iteration approach using for loop into a simple formula hence making them easy to use. Below is the approach to iterate through a list, string, tuple, etc. using list comprehension in Python.

```
# Using list comprehension to iterate through loop
List = [character for character in 'Geeks 4 Geeks!']
```

Output

```
['G', 'e', 'e', 'k', 's', ' ', '4', ' ', 'G', 'e', 'e', 'k', 's', '!']
```

Time Analysis in List Comprehensions and Loop

The list comprehensions in Python are more efficient both computationally and in terms of coding space and time than a for a loop. Typically, they are written in a single line of code. The below program depicts the difference between loops and list comprehension based on performance.

```
import time
# define function to implement for loop
def for_loop(n):
    result = []
    for i in range(n):
        result.append(i**2)
    return result


# define function to implement list comprehension
def list_comprehension(n):
    return [i**2 for i in range(n)]


# Driver Code

# Calculate time taken by for_loop()
begin = time.time()
for_loop(10**6)
end = time.time()

# Display time taken by for_loop()
print('Time taken for_loop:', round(end-begin, 2))

# Calculate time takens by list_comprehension()
begin = time.time()
list_comprehension(10**6)
end = time.time()

# Display time taken by for_loop()
```

```
print('Time taken for list_comprehension:', round(end-begin, 2))
```

Output

Time taken for_loop: 0.39

Time taken for list_comprehension: 0.35

From the above program, we can see list comprehensions are quite faster than for loop.

Nested List Comprehensions

Nested List Comprehensions are nothing but a list comprehension within another list comprehension which is quite similar to nested for loops. Below is the program which implements nested loop:

```
matrix = []

for i in range(3):

    # Append an empty sublist inside the list
    matrix.append([])

    for j in range(5):
        matrix[i].append(j)

print(matrix)
```

Output

[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]

Now by using nested list comprehensions, the same output can be generated in fewer lines of code.

```
# Nested list comprehension
matrix = [[j for j in range(5)] for i in range(3)]
```

Output

[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]

List Comprehensions and Lambda

Lambda Expressions are nothing but shorthand representations of Python functions. Using list comprehensions with lambda creates an efficient combination. Let us look at the below examples:
In this example, we are inserting numbers from 10 to 50 in the list and printing it.

```
# using lambda to print table of 10
numbers = []

for i in range(1, 6):
    numbers.append(i*10)

print(numbers)
```

Output

```
[10, 20, 30, 40, 50]
```

Here, we have used for loop to print a table of 10.

```
numbers = [i*10 for i in range(1, 6)]

print(numbers)
```

Output

```
[10, 20, 30, 40, 50]
```

Now here, we have used only list comprehension to display a table of 10.

Output

```
[10, 20, 30, 40, 50]
```

Finally, we use lambda + list comprehension to display the table of 10. This combination is very useful to get efficient solutions in fewer lines of code for complex problems.

```
# using lambda to print table of 10
numbers = list(map(lambda i: i*10, [i for i in range(1, 6)]))

print(numbers)
```

Conditionals in List Comprehension

We can also add conditional statements to the list comprehension. We can create a list using range(), operators, etc. and cal also apply some conditions to the list using the if statement.

Key Points

- Comprehension of the list is an effective means of describing and constructing lists based on current lists.
- Generally, list comprehension is lightweight and simpler than standard list formation functions and loops.
- We should not write long codes for list comprehensions in order to ensure user-friendly code.
- Every comprehension of the list can be rewritten in for loop, but in the context of list interpretation, every for loop can not be rewritten.

Below are some examples which depict the use of list comprehensions rather than the traditional approach to iterate through iterable:

Python List Comprehension using If-else.

In the example, we are checking that from 0 to 7 if the number is even then insert **Even Number** to the list else insert **Odd Number** to the list.

```
Lis=["Even number" if i % 2 == 0
     else "Odd number" for i in range(8)]
print(lis)
```

Output

['Even number', 'Odd number', 'Even number', 'Odd number',
'Even number', 'Odd number', 'Even number', 'Odd number']

Nested IF with List Comprehension

In this example, we are inserting numbers in the list which is a multiple of 10 to 100, and printing it.

```
lis = [num for num in range(100)
     if num % 5 == 0 if num % 10 == 0]
print(lis)
```

Output

[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

Reverse each string in a Tuple

In this example, we are reversing strings in for loop and inserting them into the list, and printing the list.

verse each string in tuple

```
List = [string[::-1] for string in ('Geeks', 'for', 'Geeks')]

# Display list
print(List)
```

Output

['skeeG', 'rof', 'skeeG']

Creating a list of Tuples from two separate Lists

In this example, we have created two lists of **names and ages.** We are using **zip()** in list comprehension and we are inserting the name and age as a tuple to list. Finally, we are printing the list of tuples.

```
names = ["G", "G", "g"]
ages = [25, 30, 35]
person_tuples = [(name, age) for name, age in zip(names, ages)]
print(person_tuples)
```

Output:

[('G', 25), ('G', 30), ('g', 35)]

Display the sum of digits of all the odd elements in a list.

In this example, We have created a list and we are finding the digit sum of every odd element in the list.

```
# Explicit function
def digitSum(n):
    dsum = 0
    for ele in str(n):
        dsum += int(ele)
    return dsum



# Initializing list
List = [367, 111, 562, 945, 6726, 873]
```

```
# Using the function on odd elements of the list
newList = [digitSum(i) for i in List if i & 1]

# Displaying new list
print(newList)
```

Output

```
[16, 3, 18, 18]
```

Advantages of List Comprehension

- More time-efficient and space-efficient than loops.
- Require fewer lines of code.
- Transforms iterative statement into a formula.

The list() constructor converts an iterable (dictionary, tuple, string etc.) to a list and returns it.

Example

```
text = 'Python'

# convert string to list
text_list = list(text)
print(text_list)

# check type of text_list
print(type(text_list))

# Output: ['P', 'y', 't', 'h', 'o', 'n']
#         <class 'list'>
```

Run Code

list() Syntax

The syntax of list() is:

```
list([iterable])
```

list() Parameters

The list() constructor takes a single argument:

iterable (optional) - an object that could be a sequence (string, tuples) or collection (set, dictionary) or any iterator object

list() Return Value

The list() constructor returns a list.

If no parameters are passed, it returns an empty list

If an iterable is passed, it creates a list with the iterable's items.

Example 1: Create lists from string, tuple, and list

```
# empty list
print(list())

# vowel string
vowel_string = 'aeiou'
print(list(vowel_string))

# vowel tuple
vowel_tuple = ('a', 'e', 'i', 'o', 'u')
print(list(vowel_tuple))

# vowel list
vowel_list = ['a', 'e', 'i', 'o', 'u']
print(list(vowel_list))
```

Run Code

Output

```
[]
['a', 'e', 'i', 'o', 'u']
['a', 'e', 'i', 'o', 'u']
['a', 'e', 'i', 'o', 'u']
```

Example 2: Create lists from set and dictionary

```
# vowel set
vowel_set = {'a', 'e', 'i', 'o', 'u'}
print(list(vowel_set))
```

```
# vowel dictionary
vowel_dictionary = {'a': 1, 'e': 2, 'i': 3, 'o':4, 'u':5}
print(list(vowel_dictionary))
```

Run Code

Output

```
['a', 'o', 'u', 'e', 'i']
['o', 'e', 'a', 'u', 'i']
```

Note: In the case of dictionaries, the keys of the dictionary will be the items of the list. Also, the order of the elements will be random.

Example 3: Create a list from an iterator object

```
# objects of this class are iterators
class PowTwo:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.num = 0
        return self

    def __next__(self):
        if(self.num >= self.max):
            raise StopIteration
        result = 2 ** self.num
        self.num += 1
        return result

pow_two = PowTwo(5)
pow_two_iter = iter(pow_two)

print(list(pow_two_iter))
```

Run Code

Output

```
[1, 2, 4, 8, 16]
```

List Comprehension

#List Comprehension

```
list=[12,13,14]
s=[x*2 for x in list]
print(s)
```

#Using list comprehension to iterate through loop

```
List = [character for character in [1, 2, 3]]
print(List)
```

#printing the even numbers from 0 to 10 using List Comprehension

```
list=[i for i in range(11) if i%2==0]
print(list)
```

#Matrix using List Comprehension

```
matrix=[[j for j in range(3)] for i in range(3)]
print(matrix)
```

#Traditional approach of iterating

```
list=[]
for character in "Geeks 4 Geeks!":
    list.append(character)
print(list)
```

#Using list comprehension to iterate through loop

```
list=[character for character in "Geeks 4 Geeks!"]
print(list)
```

#Time Analysis in List Comprehensions and for Loop

```
import time
# define function to implement for loop
def for_loop(n):
    result = []
    for i in range(n):
        result.append(i**2)
    return result
# define function to implement list comprehension
def list_comprehension(n):
    return [i**2 for i in range(n)]
# Driver Code
# Calculate time taken by for_loop()
begin = time.time()
for_loop(10**6)
end = time.time()
# Display time taken by for_loop()
print( Time taken for_loop: , round(end-begin, 2))
```

```
# Calculate time takens by list_comprehension()
begin = time.time()
list_comprehension(10**6)
end = time.time()
# Display time taken by list comprehension
print( Time taken for list_comprehension: , round(end-begin, 2))


#for loop to print matrix
matrix = []
for i in range(3):
  # Append an empty sublist inside the list
    matrix.append([])
    for j in range(5):
        matrix[i].append(j)
print(matrix)

#Nested list comprehension to print matrix
matrix = [[j for j in range(5)] for i in range(3)]
print(matrix)
```

#for loop to print table of 10

```
numbers = []
for i in range(1, 6):
   numbers.append(i*10)
print(numbers)
```

#list comprehension to print table of 10

```
numbers = [i*10 for i in range(1,6)]
print(numbers)
```

#using lambda to print table of 10 (lambda + list comprehension)

```
numbers = list(map(lambda i: i*10, [i for i in range(1, 6)]))
print(numbers)
```

#Python List Comprehension using If-else.

```
lis=["Even number" if i % 2 == 0
      else "Odd number" for i in range(8)]
print(lis)
```

#Nested IF with List Comprehension

```
lis = [num for num in range(100)
      if num % 5 == 0 if num % 10 == 0]
print(lis)
```

#reverse each string in tuple

```
List = [string[::-1] for string in ( Geeks ,  for ,  Geeks )]
print(List)
```

Python Sets

- A set is an unordered collection of items. Every element is unique and must be immutable.
- Python sets are mutable; you can add and remove items from it.
- For example, integers, strings, and tuples are allowed, but lists or dictionaries (which are mutable) are not allowed as elements.
- Sets can be used to perform mathematical set operations like:
    - Union
    - Intersection
    - symmetric difference
- A set is created by placing all the elements inside curly braces {}, separated by comma or by using the built-in function set().
- The elements can be of different types (integer, float, tuple, string etc.).

```
#creating a set
numberSet = {1,2,3,4,3,2}
print(numberSet)

#creating an empty set
emptySet = {}   #This creates a dictionary
print(type(emptySet))

emptySet = set()   #This creates a empty set
print(type(emptySet))
```

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

<class 'dict'>

<class 'set'>
```

1. **Add( )**

Description: Adds an element to the set.

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)                    #Output: {1, 2, 3, 4}
```

2. **Clear( )**

Description: Removes all elements from the set.

```
my_set = {1, 2, 3}
my_set.clear()
print(my_set)                    # Output:  set()
```

3. **Copy( )**

Description: Creates a shallow copy of the set.

```
my_set = {1, 2, 3}
new_set = my_set.copy()
print(new_set)                   #Output:  {1, 2, 3}
```

4. **Difference( )**

   # Description: Returns a new set with elements in my_set but not in other_set.

```
my_set = {1, 2, 3}
other_set = {3, 4, 5}
print(my_set.difference(other_set))  # Output: {1, 2}
```

5. **Difference_Update( )**

   # Description: Removes elements in other_set from my_set.

```
my_set = {1, 2, 3}
other_set = {3, 4, 5}
my_set.difference_update(other_set)
print(my_set)                          #Output:  {1, 2}
```

6. **Discard( )**

```
# Description: Removes an element from the set if it exists.
my_set = {1, 2, 3}
my_set.discard(2)
print(my_set)                          #Output:  {1, 3}
```

7. **Intersection( )**

Description: Returns a new set with elements common to both sets.

my_set = {1, 2, 3}

other_set = {3, 4, 5}

print(my_set.intersection(other_set)) #**Output**: {3}

8. **Intersection_Update( )**

Description: Updates my_set to contain only elements common to both sets.

my_set = {1, 2, 3}

other_set = {3, 4, 5}

my_set.intersection_update(other_set)

print(my_set) #**Output**: {3}

9. **Isdisjoint( )**

Description: Returns True if both sets have no common elements.

my_set = {1, 2, 3}

other_set = {4, 5, 6}

print(my_set.isdisjoint(other_set)) #**Output**: True

10. Issubset( )

Description: Returns True if all elements of my_set are in other_set.

my_set = {1, 2, 3}

other_set = {1, 2, 3, 4, 5}

print(my_set.issubset(other_set)) #**Output**: True

11. Issuperset

Description: Returns True if all elements of other_set are in my_set.

my_set = {1, 2, 3, 4, 5}

other_set = {1, 2, 3}

print(my_set.issuperset(other_set)) #**Output**: True

12. Symmetric Difference

Description: Returns a new set with elements in either set but not both.

my_set = {1, 2, 3}

other_set = {3, 4, 5}

print(my_set.symmetric_difference(other_set)) #**Output**: {1, 2, 4, 5}

13. Symmetric Difference Update

Description: Updates my_set to contain elements in either set but not both.

my_set = {1, 2, 3}

other_set = {3, 4, 5}

my_set.symmetric_difference_update(other_set)

print(my_set) #**Output**: {1, 2, 4, 5}

14. Union

Description: Returns a new set with all elements from both sets.

my_set = {1, 2, 3}

other_set = {3, 4, 5}

print(my_set.union(other_set)) #**Output**: {1, 2, 3, 4, 5}

15. Update

Description: Updates my_set to contain all elements from both sets.

my_set = {1, 2, 3}

other_set = {3, 4, 5}

my_set.update(other_set)

print(my_set) #**Output**: {1, 2, 3, 4, 5}

16. Pop

Description: Removes and returns an arbitrary element from the set.

```
my_set = {1, 2, 3}
print(my_set.pop())                    #Output: 1
```

17. Remove( )

Description: The remove method deletes an element from the set. If the element is not present, it raises a KeyError.

```
# Define a set
my_set = {1, 2, 3, 4, 5}
# Use the remove method to delete an element
my_set.remove(3)
# Print the updated set
print(my_set)  # {1, 2, 4, 5}
```

- If you want to avoid the KeyError, you can use the discard method instead, which does nothing if the element is not present.

Example with discard method:

```
my_set = {1, 2, 3, 4, 5}
my_set.discard(6)  # No error raised
print(my_set)                          #Output: {1, 2, 3, 4, 5}
```

Tuple in Python

- A **tuple** is a collection in Python that is ordered and immutable (i.e., cannot be changed after its creation).
- Tuples are often used to store multiple items in a single variable.
- Tuples are similar to lists, but with a key difference that lists are mutable, while tuples are immutable.

1. Definition of Tuple:

A tuple is defined by placing items inside parentheses () and separated by commas.

Example:

```
my_tuple = (1, 2, 3)
```

2. Creation of Tuples:

Tuples can be created with or without parentheses and can hold elements of different data types.

```
# Tuple with parentheses
my_tuple = (1, 2, 3, 4)

# Tuple without parentheses (but not recommended)
my_tuple = 1, 2, 3, 4

# Tuple with mixed data types
my_tuple = (1, "apple", 3.14, True)
```

3. Tuple Length:

You can find the length of a tuple using the len() function.

```
my_tuple = (1, 2, 3, 4)
print(len(my_tuple))  # Output: 4
```

4. Create Tuple With One Item:

To create a tuple with only one item, you need to include a trailing comma after the item; otherwise, Python will not recognize it as a tuple.

```
# Correct way
single_item_tuple = (5,)
print(type(single_item_tuple))  # Output: <class 'tuple'>

# Incorrect way (without comma)
not_a_tuple = (5)
print(type(not_a_tuple))  # Output: <class 'int'>
```

5. Tuple Items - Data Types:

Tuples can contain elements of different data types.

```
my_tuple = (1, "Hello", 3.14, [1, 2, 3])
print(my_tuple)  # Output: (1, 'Hello', 3.14, [1, 2, 3])
```

6. type() Function:

You can use type() to check if a variable is a tuple.

```
my_tuple = (1, 2, 3)
print(type(my_tuple))  # Output: <class 'tuple'>
```

7. tuple() Constructor:

You can create a tuple using the tuple() constructor.

```
# From a list

my_tuple = tuple([1, 2, 3])

print(my_tuple)  # Output: (1, 2, 3)
```

```
# From a string
my_tuple = tuple("abc")
print(my_tuple)  # Output: ('a', 'b', 'c')
```

8. Access Tuple Items:

You can access tuple items by referring to their index. Indexing in Python starts from 0.

```
my_tuple = (1, 2, 3, 4)
print(my_tuple[0])  # Output: 1
print(my_tuple[3])  # Output: 4
```

9. Indexing:

You can use positive or negative indexing to access tuple items.

```
my_tuple = (10, 20, 30, 40)
print(my_tuple[1])  # Output: 20 (positive index)
print(my_tuple[-1])  # Output: 40 (negative index)
```

10. Range of Indexes:

You can specify a range of indexes to access a subset of the tuple. This is known as slicing.

```
my_tuple = (10, 20, 30, 40, 50)
print(my_tuple[1:4])  # Output: (20, 30, 40)
```

11. Range of Negative Indexes:

You can also use negative indexes to specify a range.

```
my_tuple = (10, 20, 30, 40, 50)
print(my_tuple[-4:-1])  # Output: (20, 30, 40)
```

12. Check if Item Exists:

You can use the in keyword to check if an item exists in a tuple.

```
my_tuple = (10, 20, 30, 40)
if 20 in my_tuple:
    print("Yes, 20 is in the tuple")
# Output: Yes, 20 is in the tuple
```

Examples - 1, "apple", 3.14, True

```
# Define a tuple
my_tuple = (1, "apple", 3.14, True)

# Access tuple items
print(my_tuple[1])          # Output: apple
print(my_tuple[-1])         # Output: True

# Length of the tuple
print(len(my_tuple))  # Output: 4

# Create tuple with one item
single_item_tuple = (42,)
print(type(single_item_tuple))  # Output: <class 'tuple'>

# Tuple constructor from a list
list_to_tuple = tuple([1, 2, 3])
print(list_to_tuple)  # Output: (1, 2, 3)

# Slicing a tuple
print(my_tuple[1:3])  # Output: ('apple', 3.14)

# Check if item exists
if "apple" in my_tuple:
    print("Apple exists in the tuple")   # Output: Apple exists in the tuple
```

Dictionary in Python

- A **dictionary** in Python is a collection of key-value pairs.
- Each key is unique, and it maps to a corresponding value.
- Dictionaries are unordered, changeable (mutable), and indexed by keys.

Characteristics of a Dictionary:

1. **Unordered**: In Python, dictionaries do not keep the items in any particular order (up until Python 3.6, later they preserve the insertion order).
2. **Mutable**: Dictionaries can be changed, meaning you can update, add, or delete items after creation.
3. **Indexed by Keys**: Each item in a dictionary has a key, which acts as an identifier for accessing its corresponding value.
4. **Keys are Unique**: Duplicate keys are not allowed. If a duplicate key is added, the last inserted value for the key will overwrite the previous value.
5. **Heterogeneous Data**: Both keys and values can be of different data types (e.g., string, int, list, etc.).

1. Creation of a Dictionary:

Dictionaries are created by placing key-value pairs inside curly braces {}, with the key and value separated by a colon :.

```
#Simple dictionary
my_dict = {
   "name": "Alice",
   "age": 25,
   "city": "New York"
}
print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

2. Basic Operations:

a. Accessing Dictionary Items:

You can access the values in a dictionary using their keys.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Access value by key
print(my_dict["name"])  # Output: Alice
print(my_dict.get("age"))  # Output: 25
```

b. Copying a Dictionary:

You can copy a dictionary using the copy() method or by using the dict() constructor.

```
# Using copy() method
copy_dict = my_dict.copy()
print(copy_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

 # Using dict() constructor
copy_dict2 = dict(my_dict)
print(copy_dict2) **# Output**: {'name': 'Alice', 'age': 25, 'city': 'New York'}

c. Updating a Dictionary:

You can update an existing dictionary by adding new key-value pairs or modifying existing ones.

my_dict["age"] = 30 **# Modifying an existing value**
my_dict["country"] = "USA" **# Adding a new key-value pair**
print(my_dict)
 # Output: {'name': 'Alice', 'age': 30, 'city': 'New York', 'country': 'USA'}

d. Traversing a Dictionary:

You can loop through the keys, values, or key-value pairs of a dictionary using a for loop.

```
# Loop through keys
for key in my_dict:
    print(key, my_dict[key])
# Output:
# name Alice
# age 30
# city New York
# country USA

# Loop through values
for value in my_dict.values():
    print(value)
# Output: Alice, 30, New York, USA
```

```
# Loop through key-value pairs
for key, value in my_dict.items():
    print(f"{key}: {value}")
# Output:
# name: Alice
# age: 30
# city: New York
# country: USA
```

3. Dictionary Methods:

Python provides several useful methods for dictionaries. Here's a list with examples:

Method	Description	Example
clear()	Removes all elements from the dictionary.	my_dict.clear()
copy()	Returns a shallow copy of the dictionary.	my_dict.copy()
fromkeys()	Returns a dictionary with the specified keys and a default value.	dict.fromkeys(['a', 'b'], 0)
get()	Returns the value for a specified key.	my_dict.get("name")
items()	Returns a list of key-value tuple pairs.	my_dict.items()
keys()	Returns a list of dictionary keys.	my_dict.keys()
pop()	Removes the item with the specified key and returns its value.	my_dict.pop("age")
popitem()	Removes and returns the last inserted key-value pair.	my_dict.popitem()

Method	Description	Example
setdefault()	Returns the value of the key, and inserts the key with a default value if it does not exist.	my_dict.setdefault("name", "Unknown")
update()	Updates the dictionary with the key-value pairs from another dictionary.	my_dict.update({"age": 35})
values()	Returns a list of all the dictionary values.	my_dict.values()

Python Programming Examples with Expected Outputs:

Example 1: Accessing Items

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Access a value by key
print(my_dict["city"])  # Output: New York

# Using get() method
print(my_dict.get("age"))  # Output: 25
```

Example 2: Copying a Dictionary

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Copy the dictionary
copy_dict = my_dict.copy()
print(copy_dict)
# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

Example 3: Updating a Dictionary

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Update a value
my_dict["age"] = 30

# Add a new key-value pair
my_dict["country"] = "USA"

print(my_dict)
# Output: {'name': 'Alice', 'age': 30, 'city': 'New York', 'country': 'USA'}
```

Example 4: Looping through a Dictionary

```
my_dict = {"name": "Alice", "age": 30, "city": "New York", "country":
"USA"}

# Loop through keys and values
for key, value in my_dict.items():
    print(f"{key}: {value}")

# Output:
# name: Alice
# age: 30
# city: New York
# country: USA
```

Example 5: Using Methods

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Using pop() method to remove an item
age = my_dict.pop("age")
print(age)  # Output: 25
```

 # Using values() method to get all values
print(list(my_dict.values())) # Output: ['Alice', 'New York']

Dictionary Methods

1: clear() Method

The clear() method removes all elements from the dictionary.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
my_dict.clear()

print(my_dict)
```
 # Output: {}

2: copy() Method

The copy() method creates a shallow copy of the dictionary.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
copy_dict = my_dict.copy()

print(copy_dict)
```
 # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}

3: fromkeys() Method

The fromkeys() method creates a new dictionary with keys from an iterable and a specified value.

```
keys = ['a', 'b', 'c']
value = 0
new_dict = dict.fromkeys(keys, value)

print(new_dict)
```
 # Output: {'a': 0, 'b': 0, 'c': 0}

4: get() Method

The get() method returns the value for the specified key. If the key does not exist, it returns None (or a default value if specified).

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Existing key
print(my_dict.get("age"))
# Output: 25

# Non-existing key with default value
print(my_dict.get("country", "Unknown"))
# Output: Unknown
```

5: items() Method

The items() method returns a view object containing the key-value pairs as tuples.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
items = my_dict.items()
print(items)
# Output: dict_items([('name', 'Alice'), ('age', 25), ('city', 'New York')])
```

6: keys() Method

The keys() method returns a view object containing the keys of the dictionary.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
keys = my_dict.keys()

print(keys)
# Output: dict_keys(['name', 'age', 'city'])
```

7: pop() Method

The pop() method removes the specified key from the dictionary and returns its value. If the key does not exist, it raises a KeyError.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
age = my_dict.pop("age")

print(age)
# Output: 25

print(my_dict)
# Output: {'name': 'Alice', 'city': 'New York'}
```

8: popitem() Method

The popitem() method removes and returns the last inserted key-value pair as a tuple.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
last_item = my_dict.popitem()
print(last_item)
# Output: ('city', 'New York')
print(my_dict)
# Output: {'name': 'Alice', 'age': 25}
```

9: setdefault() Method

The setdefault() method returns the value of the specified key. If the key does not exist, it inserts the key with the specified default value.

```
my_dict = {"name": "Alice", "age": 25}

# Existing key
print(my_dict.setdefault("name", "Unknown"))
# Output: Alice
```

```
# Non-existing key with default value
print(my_dict.setdefault("city", "Unknown"))
# Output: Unknown

print(my_dict)
# Output: {'name': 'Alice', 'age': 25, 'city': 'Unknown'}
```

10: update() Method

The update() method updates the dictionary with key-value pairs from another dictionary or an iterable of key-value pairs.

```
my_dict = {"name": "Alice", "age": 25}
update_dict = {"age": 30, "city": "New York"}

my_dict.update(update_dict)

print(my_dict)
# Output: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

11: values() Method

The values() method returns a view object containing the values of the dictionary.

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
values = my_dict.values()

print(values)
# Output: dict_values(['Alice', 25, 'New York'])
```

Here's a comparison between list, set, tuple, and dictionary in Python based on their characteristics and usage:

Feature	List	Set	Tuple	Dictionary
Definition	Ordered collection of items.	Unordered collection of unique items.	Ordered and immutable collection of items.	Unordered collection of key-value pairs.
Syntax	`[ ]`	`{ }`	`( )`	`{ }`
Ordered	Yes	No	Yes	No
Mutable	Yes (can be changed)	Yes (items can be added or removed)	No (immutable, cannot be changed)	Yes (values can be changed, but keys are immutable)
Duplicates Allowed	Yes	No	Yes	No (keys must be unique)
Indexing/Slicing	Yes (can access elements by index)	No (unordered, no indexing)	Yes (can access elements by index)	No (access via keys)
Heterogeneous	Yes	Yes	Yes	Yes
Use Cases	Storing ordered, changeable items	Storing unique, unordered items	Storing ordered, unchangeable items	Storing key-value pairs (mapping relationships)

1. **List**:

- **Mutable**: You can modify its contents (add, remove, or change elements).
- **Ordered**: The elements have a defined order, and they can be accessed by index.
- **Duplicates**: Lists can have duplicate elements.
- **Syntax**: `my_list = [1, 2, 3, 4]`

Example:

```python
my_list = [1, 2, 3, 4]
print(my_list[0])  # Output: 1
my_list.append(5)  # Add new element
```

2. **Set**:

- **Mutable**: Elements can be added or removed, but the elements themselves must be immutable.

- **Unordered**: Sets do not have a specific order, so you cannot access items by index.

- **No Duplicates**: A set automatically removes duplicate items.

- **Syntax**: `my_set = {1, 2, 3, 4}`

Example:

```python
my_set = {1, 2, 3, 4, 4}  # Duplicate values are removed automatically
print(my_set)  # Output: {1, 2, 3, 4}
my_set.add(5)  # Add new element
```

3. **Tuple**:

- **Immutable**: Once created, you cannot modify its contents (no addition, removal, or changing of elements).

- **Ordered**: Elements have a specific order and can be accessed by index.

- **Duplicates**: Tuples can contain duplicate elements.

- **Syntax**: `my_tuple = (1, 2, 3, 4)`

Example:

```python
my_tuple = (1, 2, 3, 4)
print(my_tuple[0])  # Output: 1
# my_tuple[0] = 10  # This will raise an error since tuples are immutable
```

4. **Dictionary**:

- **Mutable**: You can change, add, or remove key-value pairs.
- **Unordered**: The elements are stored as key-value pairs in an unordered manner.
- **No Duplicate Keys**: Keys must be unique, but values can be duplicated.
- **Access**: Items are accessed by key, not by index.
- **Syntax**: `my_dict = {"name": "Alice", "age": 25}`

Example:

```python
my_dict = {"name": "Alice", "age": 25}
print(my_dict["name"])  # Output: Alice
my_dict["age"] = 26  # Update value
```

Summary:

- **List**: Best for ordered, mutable collections where duplicates are allowed.
- **Set**: Best for unordered collections where uniqueness is required.
- **Tuple**: Best for ordered, immutable collections.
- **Dictionary**: Best for mapping keys to values, where key uniqueness is essential.

Class, Object, Methods, Constructors in Python

Class

A class is a blueprint or template that defines the properties and behavior of an object.

Example:

```
class Car:
```

Object

An object is an instance of a class.

Example:

```
my_car = Car()
```

Attributes

Attributes are data members of a class that describe the characteristics of an object.

Example:

```
class Car:
    def __init__(self):          #self is a reference to the current object.
        self.color = "Red"       #color and model are attributes.
        self.model = "Tesla"
my_car = Car()
print(my_car.color)  # Output: Red
```

```
print(my_car.model)  # Output: Tesla
```

Methods

Methods are functions that belong to a class and describe the actions an object can perform.

Example:

```
class Car:
    def __init__(self):
        self.color = "Red"
        self.model = "Tesla"
    def start_engine(self):       #start_engine is a method.
        print("Ready to drive!")
my_car = Car()
my_car.start_engine()  #Output: Ready to drive!
```

Constructors

Constructors are special methods that initialize objects when they're created.

Example:

```
class Car:
    def __init__(self, color, model):     #__init__ is a constructor method.
        self.color = color
```

```
        self.model = model
my_car = Car("Blue", "Toyota")
print(my_car.color)  # Output: Blue
print(my_car.model)  # Output: Toyota
```

In this example:

- __init__ is a constructor method.
- self is a reference to the current object.
- color and model are attributes.
- start_engine is a method.

Full Example

```
class Car:                  # Class definition
    def __init__(self, color, model):    #Constructor initialization
        self.color = color
        self.model = model
    def start_engine(self):
        print("Ready to drive!")
    def accelerate(self):
        print("Accelerating...")
my_car = Car("Red", "Tesla")     #Object creation
print(my_car.color)  # Output: Red   #Attribute access
print(my_car.model)  # Output: Tesla   #Attribute access
my_car.start_engine()  # Output: Ready to drive!       #Method calls
```

```
my_car.accelerate()  # Output: Accelerating...          #Method calls
```

Q. How can a class be instantiated in Python? Write a Python program to define a class Rectangle with attributes length and breadth, and include methods area(), perimeter() to calculate the area and perimeter of the rectangle. Instantiate an object 'rect' to demonstrate finding the area and perimeter. (6m)

Solution:

```
# Define a class Rectangle
class Rectangle:
    # Constructor method to initialize attributes
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth

    # Method to calculate area
    def area(self):
        return self.length * self.breadth

    # Method to calculate perimeter
    def perimeter(self):
        return 2 * (self.length + self.breadth)
```

```
# Instantiate an object 'rect'
rect = Rectangle(5, 3)

# Access attributes
print("Length:", rect.length)
print("Breadth:", rect.breadth)

# Call methods to calculate area and perimeter
print("Area:", rect.area())
print("Perimeter:", rect.perimeter())
```

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects and classes.

It's a way of designing and organizing software that simulates real-world objects and systems.

Key Features of OOP:

1. Encapsulation: Bundling data and methods that operate on that data within a single unit (class or object).

2. Abstraction: Hiding implementation details and showing only necessary information.

3. Inheritance: Creating new classes based on existing classes.

4. Polymorphism: Ability of objects to take on multiple forms.

5. Composition: Combining objects to form new objects.

Basic OOP Concepts:

1. Class: Blueprint or template for creating objects.

2. Object: Instance of a class.

3. Attributes: Data members of a class (e.g., variables).

4. Methods: Functions that belong to a class (e.g., procedures).

5. Inheritance: Parent-child relationships between classes.

Benefits of OOP:

1. Modularity: Easier maintenance and updates.
2. Reusability: Code can be reused across multiple projects.
3. Flexibility: Easier to adapt to changing requirements.
4. Scalability: Larger programs can be managed more efficiently.

Common OOP Terms:

1. Instantiation: Creating an object from a class.
2. Interface: Defines methods that must be implemented.
3. Override: Redefining a method in a subclass.
4. Overload: Multiple methods with the same name but different parameters.

Programming Languages that Support OOP:

1. Java
2. C++
3. Python
4. C#
5. JavaScript

6. Ruby
7. PHP
8. Swift

Real-World Applications of OOP:

1. Simulation software
2. Games
3. Graphical user interfaces (GUIs)
4. Database systems
5. Web applications
6. Mobile apps
7. Embedded systems

By using OOP principles, developers can create more organized, maintainable, and scalable software systems that accurately model real-world objects and interactions.

1. Encapsulation

Encapsulation is hiding the implementation details of an object from the outside world.

Example:

```
class BankAccount:
    def __init__(self):
        self.__balance = 0

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

    def get_balance(self):
        return self.__balance

account = BankAccount()
account.deposit(100)
print(account.get_balance())  # 100
```

In this example, the __balance attribute is encapsulated within the BankAccount class.

2. Abstraction

Abstraction is showing only the necessary information to the outside world.

Example:

```
class CoffeeMachine:
    def __init__(self):
        self.__water_temp = 0

    def make_coffee(self):
        self.__heat_water()
        print("Coffee is ready")

    def __heat_water(self):
        self.__water_temp = 100

machine = CoffeeMachine()
machine.make_coffee()  # Coffee is ready
```

In this example, the __heat_water method is abstracted from the outside world.

3. Inheritance

Inheritance is creating a new class based on an existing class.

Example:

```
class Animal:
    def __init__(self):
        self.name = ""

    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Woof"

dog = Dog()
dog.name = "Buddy"
print(dog.name)  # Buddy
print(dog.sound())  # Woof
```

In this example, the Dog class inherits from the Animal class.

4. Polymorphism

Polymorphism is the ability of an object to take on multiple forms.

Example:

```
class Shape:
    def area(self):
        pass

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

    def area(self):
        return 3.14 * self.radius ** 2

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

```
    def area(self):
        return self.length * self.width

shapes = [Circle(5), Rectangle(4, 5)]
for shape in shapes:
    print(shape.area())
```