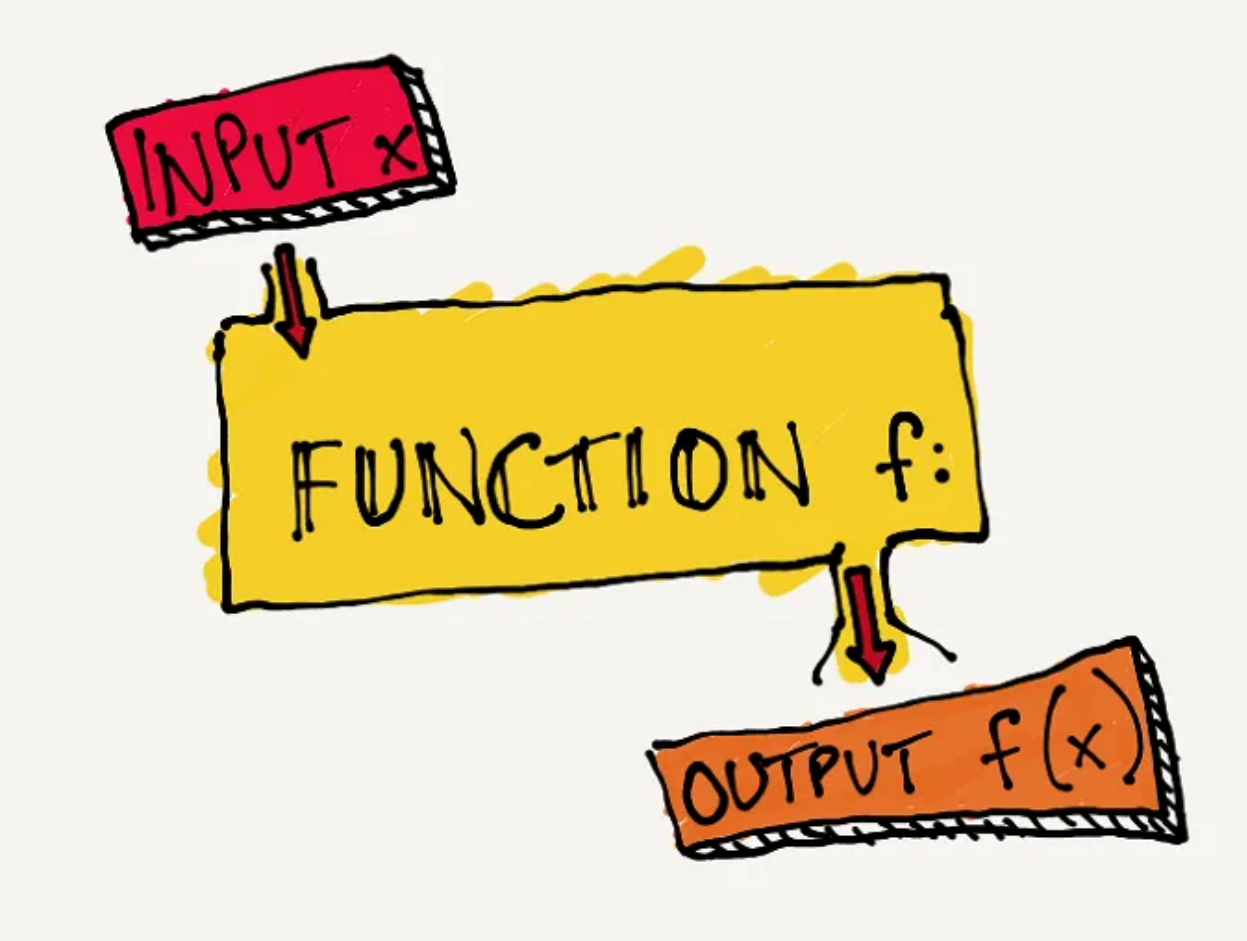
Functions:

Functions :



What is a Function?

A **function** is a reusable block of code designed to perform a specific task. Functions help in organ reducing redundancy, and enhancing readability and maintainability. They allow you to encapsulate logic and reuse it wherever needed, making your code modular and efficient.

How to Create a Function

In Python, functions are created using the `def` keyword followed by the function name and parentheses. Optionally, you can include parameters in the parentheses. The function body is indented and contains the code to be executed.

Syntax:

```
def function_name(parameters):
    # Function body
    # Code to execute
    return value  # Optional
```

Example:

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

Here, `greet` is the function name, and `name` is a parameter. The function returns a greeting message.

Features of Functions

1. **Encapsulation**: Functions encapsulate a specific task or logic, making code easier to manage and understand.
2. **Reusability**: Once defined, a function can be called multiple times from different parts of the code, reducing code duplication.
3. **Modularity**: Functions allow you to break down complex problems into smaller, manageable pieces.
4. **Abstraction**: Functions provide a way to abstract and hide the implementation details from the user, exposing only the necessary interface.
5. **Parameters and Arguments**: Functions can take parameters and arguments, allowing them to work with different inputs.

6. **Return Values**: Functions can return values to the caller, which can be used in further computations or outputs.

Use Case of Functions

Functions are versatile and can be used in various scenarios. Here are a few common use cases:

1. **Data Processing**: Functions can process data, perform calculations, and return results.
2. **Code Reusability**: Functions help in reusing code by defining common tasks once and calling them as needed.
3. **Modular Design**: Functions help in breaking down large programs into smaller, manageable parts, making development and debugging easier.
4. **Encapsulation**: Functions encapsulate specific tasks or logic, which helps in organizing code and improving readability.
5. **Event Handling**: Functions can be used as callbacks or event handlers in graphical user interfaces or web applications.

1. Function Arguments vs. Parameters

What is the Difference Between Argument and Parameter?

Aspect	Parameter	Argument
Definition	A parameter is a variable named in the function or method definition. It acts as a placeholder for the data the function will use.	An argument is the actual value that is passed to the function or method when it is called.

Role	Parameters define the type and number of inputs a function can accept. They are part of the function's signature.	Arguments are the specific values provided to the function when it is executed.
Placement	Parameters appear in function declarations/definitions.	Arguments are used in function calls.
Example in Code	In def add(num1, num2):, num1 and num2 are parameters.	In add(5, 3), 5 and 3 are arguments.
Default Values	Parameters can have default values.	Arguments do not have default values; they are the actual values passed.
Mutability	Parameters themselves do not change. The value they hold can change as different arguments are passed to the function.	Arguments can be mutable or immutable depending on their data type.
Scope	Parameters are local to the function where they are defined.	Arguments can be any scope - they can be variables that exist outside the function or literals.
Flexibility	Parameters define the expected input for a function, providing a template.	Arguments provide the flexibility to pass different values to a function, allowing for dynamic execution.

Parameters and **arguments** are related concepts but refer to different aspects of functions:

- **Parameters**: These are the names used in a function definition to specify what kind of arguments the function expects. They act as placeholders for the actual values that will be passed into the function when it is called.
- **Arguments**: These are the actual values or expressions passed to a function when it is called. They replace the parameters in the function definition.

Example:

```
def greet(name):  # 'name' is a parameter
    print(f"Hello, {name}!")

greet("prince")    # 'Alice' is an argument
```

- In the function definition `greet(name)`, `name` is a parameter.
- When calling `greet("Alice")`, `"Alice"` is an argument.

2. Types of Function Arguments

Python supports several types of function arguments:

a. Positional Arguments

These arguments are passed to functions based on their position. The order of arguments matters.

Example:

```
def subtract(a, b):
    return a - b

print(subtract(10, 5))  # Output: 5
```

b. Keyword Arguments

These arguments are passed by explicitly naming each parameter and providing a value. The order of these arguments does not matter.

Example:

```
def greet(name, message):
    print(f"{message}, {name}!")
```

```
greet(message="Good morning", name="princ")  # Output: Good m
orning, Bob!
```

c. Default Arguments

These arguments have default values defined in the function. If the caller does not provide a value, the default is used.

Example:

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

greet()           # Output: Hello, Guest!
greet("Alice")    # Output: Hello, Alice!
```

d. Variable-Length Arguments

These arguments allow you to pass a variable number of arguments to a function. They are used with `*args` for positional arguments and `**kwargs` for keyword arguments.

Example:

- `args` **(Non-keyword variable-length arguments)**:

  ```
  pythonCopy code
  def add(*numbers):
      return sum(numbers)

  print(add(1, 2, 3, 4))  # Output: 10
  ```

- `*kwargs` **(Keyword variable-length arguments)**:

```
pythonCopy code
def print_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=30)  # Output: name: Alice \n
age: 30
```

1. Local Scope

- **Definition**: Variables created inside a function belong to the **local scope** of that function. They can only be accessed inside that function.
- **Example**: Variables defined inside a function.

Example:

```
pythonCopy code
def my_function():
    x = 10  # Local variable
    print(x)  # This works

my_function()  # Output: 10
# print(x)  # This will raise an error because x is not acces
sible outside the function
```

2. Enclosing (Nonlocal) Scope

- **Definition**: Variables in the **enclosing scope** refer to the scope of any enclosing functions for nested functions. These variables are not local to the inner function but are accessible within it.
- **Example**: Variables from an outer function in nested functions.

Example:

```
pythonCopy code
def outer_function():
    x = 20  # Enclosing variable

    def inner_function():
        nonlocal x  # Refers to the x in the outer function
        x = 30  # Modifying the enclosing variable
        print("Inner:", x)

    inner_function()
    print("Outer:", x)

outer_function()  # Output: Inner: 30, Outer: 30
```

3. Global Scope

- **Definition**: Variables defined at the top level of a script or module, outside of any function, belong to the **global scope**. These variables can be accessed throughout the code, including inside functions (unless shadowed by a local variable).
- **Example**: Global variables that can be accessed inside or outside of functions.

Example:

```
pythonCopy code
x = 50  # Global variable

def my_function():
    global x  # Refers to the global variable
    x = 60  # Modifying the global variable
    print("Inside function:", x)

my_function()  # Output: Inside function: 60
```

```
print("Outside function:", x)  # Output: Outside function: 60
```

4. Built-in Scope

- **Definition**: The **built-in scope** contains names that are pre-defined in Python, such as built-in functions (`print()`, `len()`, etc.) and exceptions. These are always available and accessible from any part of the code.
- **Example**: Built-in functions like `len()` and `print()` are always accessible.

Example:

```
pythonCopy code
def my_function():
    print("This is a built-in function.")  # Using a built-in
function

my_function()  # Output: This is a built-in function.
```

LEGB Rule Explained

The LEGB rule dictates how Python looks for variable names:

- **L (Local)**: Python first looks for a variable in the **local scope** (inside the current function).
- **E (Enclosing)**: If not found, it looks in the **enclosing scope** (inside any enclosing functions, in case of nested functions).
- **G (Global)**: If still not found, it checks the **global scope** (outside the function but in the current module).
- **B (Built-in)**: Finally, it checks the **built-in scope** for Python's built-in functions and constants.

Example Demonstrating LEGB:

```
pythonCopy code
x = "global"

def outer_function():
    x = "enclosing"

    def inner_function():
        x = "local"
        print(x)  # Output: local

    inner_function()
    print(x)  # Output: enclosing

outer_function()
print(x)  # Output: global
```

Global and Nonlocal Keywords

- `global` **keyword**: Allows modifying global variables from within a function.
- `nonlocal` **keyword**: Allows modifying a variable from the enclosing (nonlocal) scope in a nested function.

Global Keyword Example:

```
pythonCopy code
x = 5

def modify_global():
    global x  # Refers to the global variable
    x = 10

modify_global()
print(x)  # Output: 10
```

Nonlocal Keyword Example:

```
pythonCopy code
def outer():
    x = "outer"

    def inner():
        nonlocal x  # Refers to the outer function's x
        x = "inner"
        print(x)  # Output: inner

    inner()
    print(x)  # Output: inner

outer()
```

1. Purpose

- `print()`: Displays output to the console.
- `return`: Returns a value from a function to the caller.

Example:

```
pythonCopy code
def greet_with_print():
    print("Hello!")  # Display output to the console

def greet_with_return():
    return "Hello!"  # Return value to the caller

greet_with_print()   # Output: Hello!
result = greet_with_return()
print(result)  # Output: Hello!
```

2. Location

- `print()`: Can be used anywhere in the code.
- `return`: Ca

Example:

```
print("This is outside a function")  # Can use print anywhere

def inside_function():
    return "This is a return statement"

print(inside_function())  # Output: This is a return statemen
t
```

3. Effect on Execution

- `print()`: Does not affect the execution flow.
- `return`: Stops further execution of the function once it is called.

Example:

```
pythonCopy code
def print_example():
    print("First")
    print("Second")
    print("Third")

def return_example():
    print("First")
    return "Second"
    print("Third")  # This line will never execute

print_example()  # Output: First, Second, Third
```

```
print(return_example())  # Output: First, Second (Third is ne
ver printed)
```

4. Return Value

- `print()`: Always returns `None`.
- `return`: Returns a specific value.

Example:

```
pythonCopy code
def print_none():
    result = print("Displayed")  # Output: Displayed
    print(f"Return value from print: {result}")

def return_value():
    return 42  # Returns an integer

print_none()  # Output: Displayed \n Return value from print:
None
print(return_value())  # Output: 42
```

5. Visibility

- `print()`: Produces visible output on the console.
- `return`: The value must be explicitly printed to be seen.

Example:

```
def visible_print():
    print("Visible output")
```

```
def invisible_return():
    return "This is returned, not printed"

visible_print()  # Output: Visible output
print(invisible_return())  # Output: This is returned, not pr
inted
```

6. Multiple Values

- `print()`: Can print multiple values in a single call.
- `return`: Can return multiple values as a tuple.

Example:

```
def print_multiple():
    print("Hello", "World", 123)  # Output: Hello World 123

def return_multiple():
    return "Hello", "World", 123  # Returns a tuple

print_multiple()  # Output: Hello World 123
result = return_multiple()
print(result)  # Output: ('Hello', 'World', 123)
```

Example:

Feature	`print()`	`return`
Purpose	Displays output to the console	Returns value from a function
Location	Can be used anywhere	Only used inside functions

Effect on Execution	Does not affect flow	Stops further execution of the function
Return Value	Always returns `None`	Returns a specific value to the caller
Visibility	Visible output in console	No visible output unless printed
Multiple Values	Prints multiple values	Returns multiple values as a tuple
Use Cases	Used for debugging and displaying information	Used to return results from a function
Side Effects	Produces visible side effects	No side effects, purely functional

Example 1: Simple Function Execution

This example demonstrates the basic flow of a function call, including how the function executes its code and returns a result.

Code:

```
def add(a, b):
    result = a + b
    return result

# Function call
sum_result = add(3, 5)
print(sum_result)  # Output: 8
```

Example 2: Function with Conditional Statements

This example shows how functions handle conditional logic and how different code paths can be executed based on input values.

Code:

```
def classify_number(num):
    if num > 0:
        return "Positive"
    elif num < 0:
        return "Negative"
    else:
        return "Zero"

# Function calls
print(classify_number(10))   # Output: Positive
print(classify_number(-5))   # Output: Negative
print(classify_number(0))    # Output: Zero
```

Simple Level Code: Basic For Loop with Condition

T

```
pythonCopy code
def print_numbers():
    for i in range(1, 11):  # Loop from 1 to 10
        if i % 2 == 0:  # Skip even numbers
            continue
        if i > 7:  # Stop the loop if i is greater than 7
            break
        print(i)

# Example usage
print_numbers()
```

Example: Multiple `if-else` in a Function

```
def calculate_grade(math_score, science_score, english_scor
e):
    total_score = math_score + science_score + english_score
    average_score = total_score / 3

    if average_score >= 90:
        grade = 'A'
    elif average_score >= 80:
        grade = 'B'
    elif average_score >= 70:
        grade = 'C'
    elif average_score >= 60:
        grade = 'D'
    else:
        grade = 'F'

    if math_score < 35 or science_score < 35 or english_score
< 35:
        print("Fail due to low score in one or more subject
s.")
        return 'F'

    print(f"Total Score: {total_score}")
    print(f"Average Score: {average_score}")
    return grade

# Example usage
math = 85
science = 75
english = 90

grade = calculate_grade(math, science, english)
print(f"Final Grade: {grade}")
```

Example 3: Nested Function Calls

This example illustrates how functions can call other functions and how execution flows through multiple layers of function calls.

Code:

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

def sum_of_squares(a, b):
    return square(a) + square(b)

# Function call
result = sum_of_squares(3, 4)
print(result)  # Output: 25
```

. Simple Example with `if-else`, `for loop`, and `id()`

```
def check_numbers(numbers):
    for num in numbers:
        if num % 2 == 0:
            print(f"{num} is even. (ID: {id(num)})")
        else:
            print(f"{num} is odd. (ID: {id(num)})")
```

```
numbers = [1, 2, 3, 4, 5]
check_numbers(numbers)
```

```
pythonCopy code
def categorize_scores(scores):
    def categorize(score):
        if score >= 90:
            return "Excellent"
        elif score >= 75:
            return "Good"
        else:
            return "Needs Improvement"

    categories = {}
    for name, score in scores.items():
        category = categorize(score)
        categories[name] = category

    return categories

student_scores = {
    "Alice": 95,
    "Bob": 82,
    "Charlie": 70,
    "David": 60
}

result = categorize_scores(student_scores)
for student, category in result.items():
    print(f"{student}: {category} (ID: {id(category)})")
```

```
def factorial(num):
    if num == 1:
        return 1
    else:
        return num * factorial(num - 1)

def process_numbers(numbers):
    for num in numbers:
        fact = factorial(num)
        if fact % 2 == 0:
            print(f"Factorial of {num} is {fact}, and it's ev
en. (ID: {id(fact)})")
        else:
            print(f"Factorial of {num} is {fact}, and it's od
d. (ID: {id(fact)})")

numbers = [3, 4, 5]
process_numbers(numbers)
```

Nested Functions:

Definition and Purpose

A **nested function** (or inner function) is a function defined within another function (the outer function). This approach is often used to:

- Encapsulate functionality that is specific to the outer function.
- Simplify complex functions by breaking them into smaller, more manageable pieces.
- Access variables from the enclosing function’s scope (closures).

Syntax

```
def outer_function(outer_args):
    def inner_function(inner_args):
        # Code for inner function: Perform a simple operation
using inner_args
        result = outer_args + inner_args
        return result

    # Code for outer function: Call the inner function and ge
t its result
    inner_function_result = inner_function(5)
    return inner_function_result

# Function call
result = outer_function(10)
print(result)  # Output: 15
```

Features

1. **Access to Outer Scope**: Inner functions can access variables from their enclosing scope (the outer function).
2. **Encapsulation**: The inner function is not accessible from outside the outer function, which helps in keeping the inner function's scope private.
3. **Closure**: Inner functions can form closures, meaning they remember the environment in which they were created.

Examples

Example 1: Basic Nested Function

This example demonstrates a simple nested function.

```
def sum_of_squares(a, b):
    def multiply(x, y):
        """Multiplies two numbers."""
```

```
        return x * y

    def square(x):
        """Squares a number using the multiply function."""
        return multiply(x, x)

    # Calculate sum of squares using the nested square functi
on
    return square(a) + square(b)

# Function call
result = sum_of_squares(3, 4)
print(result)  # Output: 25
```

Example 2: Nested Functions with Loop and Conditional Statements (Prime Number Check)

```
def print_nested_loops():
    for i in range(1, 4):         # Outer loop
        for j in range(1, 4):     # Middle loop
            for k in range(1, 4): # Inner loop
                print(f"i={i}, j={j}, k={k}")

print_nested_loops()
```

```
pythonCopy code
def outer_function(x):
    def inner_function(y):
        return y * y

    return inner_function(x) + 10
```

```
result = outer_function(5)
print(result)  # Output: 35
```

1. Built-in Functions

Definition: Built-in functions are functions that are pre-defined in Python and are available for use without the need for imports or additional definitions.

Key Points:

- Always available in Python.
- Perform common tasks.
- Examples include `print()`, `len()`, `type()`, etc.

Examples:

```
pythonCopy code
# print() - prints objects to the console
print("Hello, World!")  # Output: Hello, World!

# len() - returns the length of an object
length = len("Python")
print(length)  # Output: 6

# type() - returns the type of an object
print(type(42))  # Output: <class 'int'>
```

What is an Iterable?

An **Iterable** is an object capable of returning its members one at a time, allowing you to iterate (loop) over its elements. Examples of iterables include lists, tuples,

dictionaries, strings, and sets. Any object with an `__iter__()` method is considered an iterable.

- **Common Iterable Objects**: Lists, tuples, strings, dictionaries, and sets are all iterable because you can loop over them.

What is an Iterator?

An **Iterator** is an object that represents a stream of data. It returns data, one element at a time, when you repeatedly call its `__next__()` method. Once all elements have been returned, it raises a `StopIteration` exception to signal the end of the iteration.

- **Iterators are "lazy"**: They only compute the next value when requested, making them memory efficient for large datasets.
- An iterator must implement two methods: `__iter__()` (returns the iterator object) and `__next__()` (returns the next element).

Difference Between Iterable and Iterator

- **Iterable**: Any object with an `__iter__()` method or supports iteration (e.g., lists, tuples). You cannot directly call `__next__()` on it.
- **Iterator**: An object that produces the next value when calling `__next__()` on it and keeps track of its state.

Example:

- **Iterable**: A list (`[1, 2, 3]`) is an iterable. You can loop through it using a `for` loop.
- **Iterator**: An iterator is created from the iterable, which gives you values one by one when you call `next()` on it.

How to Convert an Iterable into an Iterator?

You can convert an **iterable** into an **iterator** by passing it to the `iter()` function. The `iter()` function returns an iterator object, which can then be used to manually fetch elements using the `next()` function.

Example:

```
pythonCopy code
# Iterable (a list)
my_list = [1, 2, 3, 4]

# Convert iterable to iterator
my_iterator = iter(my_list)

# Fetch elements using next()
print(next(my_iterator))  # Output: 1
print(next(my_iterator))  # Output: 2
```

3. Lambda Functions

Definition: Lambda functions are small, anonymous functions defined using the `lambda` keyword. They are used for short, throwaway functions.

Key Points:

- Syntax: `lambda arguments: expression`
- Generally used for single-line operations.
- Often used with functions like `map()`, `filter()`, and `sorted()`.

Example:

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

# Lambda function used with map()
numbers = [1, 2, 3, 4]
```

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

1. Generators:

Generators are special functions in Python that allow you to yield a sequence of values over time, instead of returning all values at once. They are memory-efficient and allow lazy evaluation.

Key Points about Generators:

- Generators use the `yield` keyword instead of `return`.
- Each call to the generator function returns a generator object.
- The generator keeps its state in memory and resumes where it left off when you call `next()` on it.

Example 1: Simple Generator

```
pythonCopy code
def countdown(num):
    while num > 0:
        yield num
        num -= 1

# Using the generator
count_gen = countdown(5)
for number in count_gen:
    print(number)
```

Explanation:

- This generator function `countdown()` starts from the number you pass and decrements it until it reaches 0, yielding each number one by one.

- This example shows how to iterate over a generator using a `for` loop.

Example 2: Fibonacci Sequence Generator

```
pythonCopy code
def fibonacci(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b

# Using the generator
fib_gen = fibonacci(10)
for num in fib_gen:
    print(num)
```

Explanation:

- The generator `fibonacci()` yields values from the Fibonacci sequence up to a certain limit.
- This demonstrates how you can use generators to handle sequences like Fibonacci numbers lazily.

Higher-Order Functions

Definition: Higher-order functions take other functions as arguments or return functions as results.

Key Points:

- Can accept functions as arguments (`map()`, `filter()`, `sorted()`).
- Can return functions (`function factories`).

Example:

```
pythonCopy code
def apply_function(func, x):
    return func(x)

def square(n):
    return n * n

result = apply_function(square, 4)
print(result)  # Output: 16
```

7. Map, Filter, and Reduce Functions

Definition: These are functional programming tools for processing lists or other iterables.

Key Points:

- **Map**: Applies a function to all items in an iterable.
- **Filter**: Filters items based on a condition.
- **Reduce**: Applies a function cumulatively to items in an iterable.

Examples:

```
pythonCopy code
from functools import reduce

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

# filter() function
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]
```

```
# reduce() function
sum_all = reduce(lambda x, y: x + y, numbers)
print(sum_all)  # Output: 10
```

9. Decorator Functions

Definition: Decorators are functions that modify the behavior of other functions. They are applied using the `@` syntax.

Key Points:

- Used to extend the behavior of functions.
- Can be stacked to apply multiple decorators.

Example:

```
def decorator_function(func):
    def wrapper():
        print("Something is happening before the function is
called.")
        func()
        print("Something is happening after the function is c
alled.")
    return wrapper

@decorator_function
def say_hello():
    print("Hello!")

say_hello()
```

Example 1: Decorator to Calculate Execution Time

```
pythonCopy code
import time
```

```
def time_decorator(func):
    def wrapper():
        start_time = time.time()  # Record start time
        result = func()  # Call the original function
        end_time = time.time()  # Record end time
        print(f"Execution time: {end_time - start_time:.5f} s
econds")
        return result  # Return the result of the original fu
nction
    return wrapper

@time_decorator
def example_function():
    print("Executing the function...")
    time.sleep(2)  # Simulating a time-consuming task

# Using the decorated function
example_function()
```

Example 2: Decorator to Add Symbols Before and After Output

```
pythonCopy code
def symbol_decorator(func):
    def wrapper():
        print("### Start of Output ###")
        func()  # Call the original function
        print("### End of Output ###")
    return wrapper

@symbol_decorator
def greet():
    print("Hello, World!")
```

```
# Using the decorated function
greet()
```

`return` Features

1. **Terminates the Function**: When `return` is called, the function's execution ends, and a value is sent back to the caller.

```
pythonCopy code
def terminate_function(x):
    return x * 2  # Function ends here and returns the result

result = terminate_function(5)
print(result)  # Output: 10
```

1. **Single Value Return**: `return` typically returns a single value (or multiple values packed as a tuple).

```
pythonCopy code
def single_return(x):
    return x * 3  # Returns a single value

result = single_return(4)
print(result)  # Output: 12
```

1. **Normal Function**: Functions that use `return` are standard functions and do not retain their state between function calls.

```
pythonCopy code
def normal_function():
```

```
    x = 5
    return x

result1 = normal_function()  # First call
result2 = normal_function()  # Second call
print(result1, result2)  # Output: 5 5 (Function state is not
retained)
```

1. **Once Executed**: After `return` is executed, the function is done, and the function's state is not saved.

```
pythonCopy code
def once_executed(x):
    return x * 2

result = once_executed(6)
# Trying to access state here is not possible as the function
has ended
```

`yield` Features

1. **Pauses the Function**: When `yield` is called, the function's state is saved, and it pauses the execution, returning a value to the caller. The function can be resumed from where it left off.

```
pythonCopy code
def pause_function():
    yield 1  # Pauses and returns 1
    yield 2  # Resumes and returns 2
    yield 3  # Resumes and returns 3

gen = pause_function()
print(next(gen))  # Output: 1
```

```
print(next(gen))  # Output: 2
print(next(gen))  # Output: 3
```

1. **Multiple Values Over Time**: `yield` can produce multiple values over time in a sequence, one at a time.

```
pythonCopy code
def multiple_values():
    for i in range(4):
        yield i  # Yields values 0, 1, 2, 3 in sequence

gen = multiple_values()
for value in gen:
    print(value)  # Output: 0 1 2 3 (each on a new line)
```

1. **Generator Function**: Functions that use `yield` are called generators and can be iterated over.

```
pythonCopy code
def generator_function():
    yield 'a'
    yield 'b'
    yield 'c'

gen = generator_function()
for letter in gen:
    print(letter)  # Output: 'a' 'b' 'c' (each on a new line)
```

1. **State Retention**: Generators maintain their internal state and allow the function to resume where it left off after yielding a value.

```
pythonCopy code
def state_retention():
    x = 0
    while x < 3:
        yield x
        x += 1

gen = state_retention()
print(next(gen))  # Output: 0
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
# Function state is retained across yields
```

Tabular Difference Between `yield` and `return`

Feature	`return`	`yield`
Purpose	Terminates the function	Pauses the function and saves state
Type of Function	Normal function	Generator function
Return Type	Returns a single value (or tuple)	Returns a generator object
State Preservation	Does not preserve state	Preserves state between function calls
Multiple Values	Returns a single value	Can yield multiple values over time
Execution Flow	Ends the function completely	Pauses function and resumes on next call
Reusability	Needs to be called again for another execution	Resumes from the point it last yielded
Use Case	Used when you want to return a value and stop execution	Used when you want to generate a sequence of values
Example	`return 5`	`yield 5`