User-Defined Functions in Python

User-defined functions in Python are blocks of reusable code designed to perform a specific task. They help in organizing code, making it modular, and enhancing its readability and maintainability.

You define a function using the def keyword, followed by the function name and parentheses.

Defining a Function

A function definition includes the following components:

1. **Function name**: A name to identify the function.
2. **Parameters (optional)**: Variables that accept input values.
3. **Function body**: A block of code that performs a specific task.
4. **Return statement (optional)**: Outputs the result of the function.

```
def function_name(parameters):
    """docstring"""
    # Function body
    return value
```

- def: The keyword used to define a function.
- function_name: The name of the function.
- parameters: (Optional) Variables passed to the function.
- docstring: (Optional) A string that describes the function's purpose.
- return: (Optional) The value returned by the function.

```
def greet():
    """This function returns a greeting message."""
    return "Hello, welcome to Python programming!"
# Calling the function
message = greet()
print(message)
```

Function Parameters

Function parameters are variables listed inside the parentheses in the function definition.

They allow you to pass information to the function, which can then be used within the function.

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

# calling function
greet(“Alice”)
```

Default Parameters

Default parameters allow you to define default values for function parameters. If an argument for a default parameter is not provided, the default value is used. Default parameters should always follow non-default parameters.

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

# Calling the function with and without an argument
greet("Alice")          # Output: Hello, Alice!
greet()         # Output: Hello, Guest!
```

In this example, the greet function has a default parameter name with a default value of "Guest".

Non-Default Parameters

Non-default parameters are parameters that do not have a default value and must be provided when calling the function.

```
def add_numbers(a, b):
    return a + b

# Calling the function with arguments
result = add_numbers(5, 3)
print("The sum is:", result)  # Output: The sum is: 8
```

Combining Default and Non-Default Parameters

When combining default and non-default parameters, ensure that the default parameters come after the non-default parameters.

```
def person(name, age=30, city="New York"):
    print(f"Name: {name}, Age: {age}, City: {city}")

# Calling the function with various arguments
person("Alice", 25, "Los Angeles")
# Output: Name: Alice, Age: 25, City: Los Angeles
person("Bob", 40)
# Output: Name: Bob, Age: 40, City: New York
person("Charlie")
# Output: Name: Charlie, Age: 30, City: New York
```

Keyword Arguments

You can call a function using keyword arguments, where you specify the parameter name along with its value.

This allows you to skip the order of parameters and only provide values for specific parameters.

```
def person(name, age=30, city="New York"):
    print(f"Name: {name}, Age: {age}, City: {city}")

# Using keyword arguments
person(name="David", city="Chicago")
# Output: Name: David, Age: 30, City: Chicago
person(city="Boston", name="Eva")
# Output: Name: Eva, Age: 30, City: Boston
```

Arbitrary arguments

Arbitrary arguments in Python allow you to create flexible functions that can accept an arbitrary number of arguments.

This is particularly useful when you don't know in advance how many arguments will be passed to the function.

There are two types of arbitrary arguments: ***args for non-keyword** arguments and ****kwargs for keyword arguments**.

*args: Non-Keyword Arguments

The *args syntax allows a function to accept any number of positional arguments. Inside the function, args is treated as a tuple of the arguments passed.

```
def function_name(*args):
    # Function body
```

Example:

```
def sum_all(*args):
    total = 0
    for num in args:
        total += num
    return total

# Calling the function with various numbers of arguments
print(sum_all(1, 2, 3))         # Output: 6
print(sum_all(4, 5, 6, 7, 8))   # Output: 30
```

Example:

```
def print_args(*args):
    for arg in args:
        print(arg)

# Calling the function with various arguments
print_args("Hello", "World", 123, [1, 2, 3])
# Output:
# Hello
# World
# 123
# [1, 2, 3]
```

****kwargs: Keyword Arguments**

The **kwargs syntax allows a function to accept any number of keyword arguments. Inside the function, kwargs is treated as a dictionary of the arguments passed.

Syntax

```
def function_name(**kwargs):
    # Function body
```

Example:

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Calling the function with various keyword arguments
print_kwargs(name="Alice", age=25, city="New York")
# Output:
# name: Alice
# age: 25
# city: New York
```

Combining *args and **kwargs

```
def print_args_kwargs(*args, **kwargs):
    print("Positional arguments:")
    for arg in args:
        print(arg)

    print("Keyword arguments:")
    for key, value in kwargs.items():
```

```
        print(f"{key}: {value}")

# Calling the function with various arguments
print_args_kwargs(1, 2, 3, name="Alice", age=25)
# Output:
# Positional arguments:
# 1
# 2
# 3
# Keyword arguments:
# name: Alice
# age: 25
```

Example:

```
def calculate_average_grade(student_name, *grades, **info):
    total = sum(grades)
    average = total / len(grades)
    print(f"Student: {student_name}")
    print(f"Average Grade: {average:.2f}")

    for key, value in info.items():
        print(f"{key}: {value}")

# Calling the function
calculate_average_grade("Bob", 85, 90, 78, 92, age=17, city="Boston")
```

```
# Output:
# Student: Bob
# Average Grade: 86.25
# age: 17
# city: Boston
```

Important

- **Order**: When combining *args and **kwargs, *args must come before **kwargs in the function definition.
- **Flexibility**: Using arbitrary arguments makes your functions more flexible and adaptable to different numbers and types of inputs.
- **Type**: *args is a tuple, while **kwargs is a dictionary.

Arbitrary arguments (*args and **kwargs) are powerful tools for creating versatile and reusable functions in Python.

return statements in functions

In Python, the return statement is used to exit a function and return a value to the caller. It can also be used to return multiple values.

```
def function_name(parameters):
    # Function body
    return value
```

return: The keyword used to exit a function and optionally pass a value back to the caller.

value: The value to be returned by the function.

Returning a Single Value

```
def square(x):
    return x * x


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

Returning Multiple Values

A function can return multiple values using tuples.

```
def calculate(a, b):
    sum_ = a + b
    diff = a - b
    prod = a * b
    quot = a / b
    return sum_, diff, prod, quot


result = calculate(10, 2)
print(result)  # Output: (12, 8, 20, 5.0)
```

In this, return sum_, diff, prod, quot returns a tuple containing all four values.

Early Exit from a Function

The return statement can be used to exit a function early, even before the function completes all its statements.

```
def check_number(n):
    if n > 0:
        return "Positive"
    elif n < 0:
        return "Negative"
    return "Zero"

result = check_number(-5)
print(result)  # Output: Negative
```

return Without a Value

Using return without a value exits the function and returns None.

```
def print_message():
    print("This is a message")
    return

result = print_message()
print(result)  # Output: This is a message, None
```

Quiz:

1. Which keyword is used to define a function in Python?

   A) func B) define C) def D) function

2. What will be the output of the following code?

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

   A) Hello, B) Hello, Alice! C) Hello, name! D) Error

3. What is the default return value of a function that does not have a return statement?

   A) 0 B) None C) An empty string D) Undefined

4. How do you define a function with a default parameter?

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

   What will be the output of greet("Bob")?

   A) Hello, ! B) Hello, Bob! C) , Bob! D) Error

5. What will be the output of the following code?

   ```
   python
   def add_numbers(a, b=10, c=5):
       return a + b + c
   print(add_numbers(3))
   ```

   A) 18 B) 20 C) 15 D) Error

6. Which of the following statements correctly returns multiple values from a function?

   ```
   def calculate(a, b):
       return a + b, a - b
   ```

   A) return (a + b) and (a - b) B) return [a + b, a - b]

C) return (a + b, a - b) D) return a + b, a - b

7. What is the correct way to call a function with arbitrary positional arguments?

```
def sum_all(*args):
    return sum(args)
```

A) sum_all(1, 2, 3) B) sum_all([1, 2, 3])

C) sum_all(a=1, b=2, c=3) D) sum_all(1, 2, 3, 4, 5)

8. Which of the following is a valid way to define a function with keyword arguments?

```
def print_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
print_details(name="Alice", age=25)
```

A) print_details(name="Alice", age=25)

B) print_details(name=Alice, age=25)

C) print_details("Alice", 25)

D) print_details(name="Alice"; age=25)

9. How can you document a function in Python?

```
def add(a, b):
    """This function returns the sum of two numbers."""
    return a + b
```

A) By using comments

B) By using docstrings

C) By using annotations

D) By using print statements

10. What will be the output of the following code?

```
def multiply(a, b):
    return a * b
print(multiply.__doc__)
```

A) This function multiplies two numbers.

B) None

C) multiply(a, b)

D) An error

1. Correct Answer: C) def
2. Correct Answer: B) Hello, Alice!
3. Correct Answer: B) None
4. Correct Answer: B) Hello, Bob!
5. Correct Answer: A) 18
6. Correct Answer: D) return a + b, a - b
7. Correct Answer: A) sum_all(1, 2, 3)
8. Correct Answer: A) print_details(name="Alice", age=25)
9. Correct Answer: B) By using docstrings
10. Correct Answer: B) None

MCQ

Question 1:
What will be the output of the following code?

```
def func(x, y=5, z=10):
    return x + y * z

result = func(2, z=4)
print(result)
```

A) 22 B) 42 C) 52 D) Error

Question 2:
Given the following function definition, what will be the output?
python

```
def func(a, b=1, *args, **kwargs):
    return a + b + sum(args) + sum(kwargs.values())

result = func(1, 2, 3, 4, x=5, y=6)
print(result)
```

A) 21 B) 22 C) 23 D) 24

Question 3:
What will be the output of the following code?

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

result = multiply(5, multiply(2, 3))
print(result)
```

A) 30 B) 15 C) 10 D) Error

Question 4:
Which of the following function definitions will raise an error?
A)

```
def func1(x, y, z=10): pass
```

B)

```
def func2(x, y=5, z=10): pass
```

C)
```
def func3(x=0, y, z): pass
```
D)
```
def func4(x, y=5, *args, **kwargs): pass
```

Question 5:
What will be the output of the following code?

```
def func(a, b):
    a = 10
    b[0] = 20

x = 5
y = [1, 2, 3]
func(x, y)
print(x, y)
```

A) 10, [1, 2, 3] B) 5, [1, 2, 3] C) 5, [20, 2, 3] D) 10, [20, 2, 3]

Question 6:

What will be the output of the following code?

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

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

def operate(func, x, y):
    return func(x, y)

print(operate(add, 5, 3))
print(operate(multiply, 5, 3))
```

A) 8, 15 B) 15, 8 C) 8, 8 D) 15, 15

Question 7:
Which of the following statements about functions in Python is true?
A) A function in Python can return only one value.
B) A function definition can contain multiple return statements.
C) Function parameters must always have default values.
D) A function cannot call another function within its body.

ANSWERS

1. Correct Answer: A) 22
2. Correct Answer: A) 21
3. Correct Answer: A) 30
4. Correct Answer: C) def func3(x=0, y, z): pass
5. Correct Answer: C) 5, [20, 2, 3]
6. Correct Answer: A) 8, 15
7. Correct Answer: B) A function definition can contain multiple return statements.

Concepts of global and local variables in Python

Local Variables

- **Definition:** Local variables are variables that are declared and used inside a function.
- They are not accessible outside the function in which they are declared.
- When a function is called, a new local scope is created for that function, and local variables are confined to this scope.

```
def example_function():
    local_var = 10  # local variable
    print("Inside the function, local_var:", local_var)

example_function()
print(local_var)  # This will generate error
```

```
NameError                                 Traceback (most recent call last)
<ipython-input-4-e8b8a2fe608d> in <cell line: 7>()
      5 example_function()
      6 # The following line will raise an error if uncommented, because local_var is not accessible outside the function
----> 7 print(local_var)
NameError: name 'local_var' is not defined
```

Global Variables

- Definition: Global variables are variables that are declared outside any function and are accessible throughout the entire program, including inside any functions.
- Global variables exist in the global scope.

```
global_var = 20  # global variable

def example_function():
    print("Inside the function, global_var:", global_var)

example_function()
print("Outside the function, global_var:", global_var)
```

Using Global Variables Inside Functions

- To modify a global variable inside a function, you must explicitly declare it as global within the function.
- Without this declaration, assigning a value to a variable inside a function will create a new local variable instead of modifying the global variable.

```
global_var = 20  # global variable
def modify_global_var():
    global global_var
    global_var = 30
    print("Inside the function, modified global_var:", global_var)

modify_global_var()
print("Outside the function, global_var:", global_var)
```

Output:

```
Inside the function, modified global_var: 30
Outside the function, global_var: 30
```