

Functions in Python

Table of contents

0.1	Functions	1
0.2	Defining a Function	2
0.3	Calling a Function	2
0.4	Default Parameter Values	2
0.5	Variable-Length Arguments	3
0.6	Function Annotation	3
0.7	Lambda (λ) Functions	4
0.8	Recursion	4
0.8.1	Understanding Recursion with Factorial	6
0.8.2	Recursive Factorial Function in Python	6
0.8.3	Advantages and Disadvantages of Recursion	7
0.9	Global vs Local Variables	7
1	Exercises	8
1.1	Exercise 1	8
1.2	Exercise 2	8
1.3	Exercise 3	9

0.1 Functions

- Functions are **reusable** blocks of code that perform a specific task
- They can take input parameters and return output values
- Functions are essential in modular programming, as they help organize code and make it more readable

0.2 Defining a Function

- To define a function in Python, use the keyword `def` followed by the function name and input parameters in parentheses
- The function body is **indented** below the header line
- Use the keyword `return` to specify the output value(s) of the function

```
1 def add_numbers(x, y):  
2     result = x + y  
3     return result  
4  
5 type(add_numbers)
```

function

0.3 Calling a Function

To call a function, use its name followed by input values in parentheses. The function returns the output value(s), which can be stored in a variable or used directly.

```
1 sum = add_numbers(2, 3)  
2 print(sum)
```

5

0.4 Default Parameter Values

- Functions can have default values for input parameters, which are used when no value is provided
- Default values are specified in the function header

```
1 def greet(name, greeting = "Hello"):  
2     print(greeting + ", " + name)  
3 greet("Alice")
```

Hello, Alice

```
1 greet("Bob", "Hi")
```

Hi, Bob

0.5 Variable-Length Arguments

- Variable-length arguments allow a function to accept any number of input arguments
- They are useful when the number of input arguments is unknown or can vary

```
1 def add_numbers(*args):
2     result = 0
3     for num in args:
4         result += num
5     return result
6
7 print(add_numbers(1, 2))
8 print(add_numbers(1, 2, 3))
```

3

6

0.6 Function Annotation

```
1 def add_numbers(*args):
2
3     """
4     Computes the sum of n numbers
5
6     Parameters:
7     args: A tuple of numbers
8
9     Returns:
10    int: The sum
11    """
12
13    result = 0
14    for num in args:
15        result += num
```

```
16     return result
17
18 help(add_numbers)
```

Help on function add_numbers in module __main__:

```
add_numbers(*args)
    Computes the sum of n numbers

    Parameters:
    args: A tuple of numbers

    Returns:
    int: The sum
```

0.7 Lambda (λ) Functions

- Lambda functions are **anonymous** functions that can be defined inline and used immediately
- They are useful for *simple* tasks that do not require a named function
- Lambda functions can only have **one** expression (i.e., no statements)

```
1 double = lambda x: x * 2
2 print(double(3))
```

6

0.8 Recursion

- Recursion is a technique where a function calls itself
- It is useful for solving problems that can be broken down into smaller subproblems
- Termination Condition is essential for preventing infinite loops in recursion
- Each recursive call consumes stack memory; deep recursion can lead to stack overflow if not properly managed



0.8.1 Understanding Recursion with Factorial

A factorial of a non-negative integer n is the product of all positive integers less than or equal to n . It is denoted by $n!$. Factorials are widely used in permutations and combinations.

The factorial of a number n is defined as:

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$$

For example:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

One of the elegant ways to compute factorials in programming is by using **recursion**.

0.8.2 Recursive Factorial Function in Python

```
1 def factorial(n):
2     """
3     This function calculates the factorial of a non-negative integer.
4
5     Args:
6         n: The non-negative integer whose factorial is to be calculated.
7
8     Returns:
9         The factorial of n.
10    """
11    if n == 0: # Base case
12        return 1
13    else:
14        return n * factorial(n-1) # Recursive case
15
16 # Test the function
17 factorial(5)
```

120

0.8.3 Advantages and Disadvantages of Recursion

0.8.3.1 Advantages

- **Simplicity:** Recursive solutions can be more readable and easier to understand.
- **Natural Fit:** Some problems, like tree traversals, are naturally recursive.

0.8.3.2 Disadvantages

- **Performance:** Recursive calls can be slower due to overhead.
- **Memory Usage:** Each recursive call consumes stack memory; deep recursion can lead to stack overflow.

0.9 Global vs Local Variables

- Global variables are defined outside of any function and can be accessed from anywhere in the program
- Local variables are defined inside a function and can only be accessed within that function

```
1 global_var = 10
2
3 def my_func():
4     local_var = 20
5     print(global_var)
6     print(local_var)
7
8 my_func()
```

```
10
20
```

```
1 print(global_var)
```

```
10
```

```
1 # print(local_var)
```

1 Exercises

1.1 Exercise 1

Write a Python function that takes a list of numbers and returns their product.

Sample:

```
1 multiply_numbers([2, 3, 4])
```

24

Answer:

```
1 def multiply_numbers(numbers):  
2     product = 1  
3     for num in numbers:  
4         product *= num  
5     return product  
6  
7 multiply_numbers([2, 3, 4])
```

24

1.2 Exercise 2

Write a Python function that finds the maximum and minimum numbers in a list.

Sample:

```
1 find_min_max ([2, 3, 4])
```

(2, 4)

Answer:


```

1 def find_min_max(numbers):
2     min_num = numbers[0] # Set the min to the first number in the list
3     max_num = numbers[0] # Set the max to the first number in the list
4     for num in numbers:
5         if num > max_num:
6             max_num = num
7         elif num < min_num:
8             min_num = num
9     return (min_num, max_num)
10
11 find_min_max ([2, 3, 4])

```

(2, 4)

1.3 Exercise 3

Write a Python function that takes a list of strings and returns the longest string.

Sample:

```

1 find_longest_string(["Vicky", "Cristina", "Barcelona"])

```

'Barcelona'

Answer:

```

1 def find_longest_string(strings):
2     # Set the longest string to first string in the list
3     longest_string = strings[0]
4     for string in strings:
5         # Compare the length of the current string to the longest string so far
6         if len(string) > len(longest_string):
7             longest_string = string
8     return longest_string
9
10 find_longest_string(["Vicky", "Cristina", "Barcelona"])

```

'Barcelona'