Lab 05: Functions

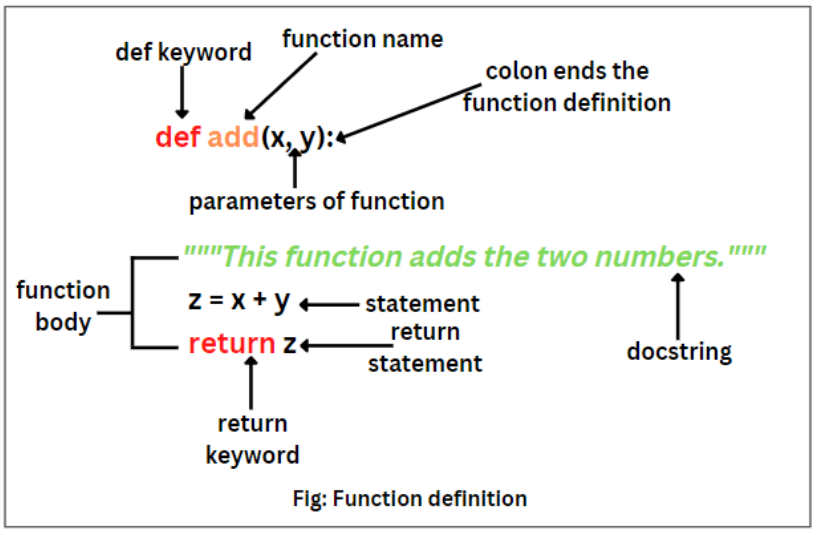

Fig: Function definition

Why Use Functions?

Imagine you need to convert temperatures from Fahrenheit to Celsius several times in your program. Without functions, you would have to write the same calculation code repeatedly:

Example

Without functions - repetitive code:

```
temp1 = 77
celsius1 = (temp1 - 32) * 5 / 9
print(celsius1)

temp2 = 95
celsius2 = (temp2 - 32) * 5 / 9
print(celsius2)

temp3 = 50
celsius3 = (temp3 - 32) * 5 / 9
print(celsius3)
```

With functions, you write the code once and reuse it:

Example

With functions - reusable code:

```
def fahrenheit_to_celsius(fahrenheit):
  return (fahrenheit - 32) * 5 / 9

print(fahrenheit_to_celsius(77))
print(fahrenheit_to_celsius(95))
print(fahrenheit_to_celsius(50))
```

Parameters vs Arguments

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

From a function's perspective:

A **parameter** is the variable listed inside the parentheses in the function definition.

An **argument** is the actual value that is sent to the function when it is called.

Example

```
def my_function(name): # name is a parameter
  print("Hello", name)

my_function("Emil") # "Emil" is an argument
```

```
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)

print("Case-1:")
nameAge("Suraj", 27)

print("\nCase-2:")
nameAge(27, "Suraj")
```

Output

```
Case-1:
Hi, I am Suraj
My age is  27

Case-2:
Hi, I am 27
My age is  Suraj
```

Lambda Functions

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

```
lambda arguments : expression
```

The expression is executed and the result is returned:

Example

Add 10 to argument `a`, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

Anonymous Functions

In Python, an anonymous function means that a function is without a name. As we already know the def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.

```
def cube(x): return x*x*x   # without lambda
cube_l = lambda x : x*x*x  # with lambda

print(cube(7))
print(cube_l(7))
```

Output

```
343
343
```

Using Lambda with filter()

The `filter()` function creates a list of items for which a function returns `True`:

Example

Filter out even numbers from a list:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
print(odd_numbers)
```

Recursive Functions

A recursive function is a function that calls itself to solve a problem. It is commonly used in mathematical and divide-and-conquer problems. Always include a base case to avoid infinite recursion.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(4))
```

Output

```
24
```

Fibonacci Sequence Generator

Generators can be used to create the Fibonacci sequence.

It can continue generating values indefinitely, without running out of memory:

Example

Generate 100 Fibonacci numbers:

```
def fibonacci():
  a, b = 0, 1
  while True:
    yield a
    a, b = b, a + b

# Get first 100 Fibonacci numbers
gen = fibonacci()
for _ in range(100):
  print(next(gen))
```

Exercise 01:

Write a function double(n) that returns the double of n.

Exercise 02:

Write a function is_even(n) that returns True if n is even.

Exercise 03:

Write a function that returns the sum of a list **without using sum()**.

Exercise 04:

Write a function that returns the number of even elements in a list.

Exercise 05:

Given two sorted lists, write a function that merges them **without using sorted()**