Python Functions

A function is a block of code that performs a specific task.

Suppose we need to create a program to make a circle and color it. We can create two functions to solve this problem:

1. function to create a circle
2. function to color the shape

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Create a Function

Let's create our first function.

```
def greet():
    print('Hello World!')
```

Here are the different parts of the program:

Create a Python Function

Here, we have created a simple function named `greet()` that prints **Hello World!**

Note: When writing a function, pay attention to indentation, which are the spaces at the start of a code line.

In the above code, the `print()` statement is indented to show it's part of the function body, distinguishing the function's definition from its body.

Calling a Function

In the above example, we have declared a function named `greet()`.

```
def greet():
    print('Hello World!')
```

If we run the above code, we won't get an output.

It's because creating a function doesn't mean we are executing the code inside it. It means the code is there for us to use if we want to.

To use this function, we need to call the function.

Function Call

```
greet()
```

Example: Python Function Call

```
def greet():
    print('Hello World!')

# call the function
greet()

print('Outside function')
```

Output

```
Hello World!
Outside function
```

In the above example, we have created a function named `greet()`. Here's how the control of the program flows:

Working of Python Function

Here,

1. When the function `greet()` is called, the program's control transfers to the function definition.
2. All the code inside the function is executed.
3. The control of the program jumps to the next statement after the function call.

Python Function Arguments

Arguments are inputs given to the function.

```
def greet(name):
    print("Hello", name)

# pass argument
greet("John")
```

Sample Output 1

```
Hello John
```

Here, we passed '`John`' as an argument to the `greet()` function.

We can pass different arguments in each call, making the function re-usable and dynamic.

Let's call the function with a different argument.

```
greet("David")
```

Sample Output 2

```
Hello David
```

Example: Function to Add Two Numbers

```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print("Sum: ", sum)

# function call with two values
add_numbers(5, 4)
```

Output

```
Sum: 9
```

In the above example, we have created a function named `add_numbers()` with arguments: *num1* and *num2*.

Python Function with Arguments

Parameters and Arguments

Parameters

Parameters are the variables listed inside the parentheses in the function definition. They act like placeholders for the data the function can accept

when we call them.

Think of parameters as the **blueprint** that outlines what kind of information the function expects to receive.

```
def print_age(age):  # age is a parameter
    print(age)
```

In this example, the `print_age()` function takes `age` as its input. However, at this stage, the actual value is not specified.

The `age` parameter is just a placeholder waiting for a specific value to be provided when the function is called.

Arguments

Arguments are the actual values that we pass to the function when we call it.

Arguments replace the parameters when the function executes.

```
print_age(25)  # 25 is an argument
```

Here, during the function call, the argument **25** is passed to the function.

The return Statement

We return a value from the function using the `return` statement.

```
# function definition
def find_square(num):
    result = num * num
    return result


# function call
square = find_square(3)

print('Square:', square)
```

Output

```
Square: 9
```

In the above example, we have created a function named `find_square()`. The function accepts a number and returns the square of the number.

Working of functions in Python

Note: The `return` statement also denotes that the function has ended. Any code after `return` is not executed.

The pass Statement

The `pass` statement serves as a placeholder for future code, preventing errors from empty code blocks.

It's typically used where code is planned but has yet to be written.

```
def future_function():
    pass

# this will execute without any action or error
future_function()
```

Note: To learn more, visit Python Pass Statement.

Python Library Functions

Python provides some built-in functions that can be directly used in our program.

We don't need to create the function, we just need to call them.

Some Python library functions are:

1. print() - prints the string inside the quotation marks

2. `sqrt()` - returns the square root of a number
3. pow() - returns the power of a number

These library functions are defined inside the module. And to use them, we must include the module inside our program.

For example, `sqrt()` is defined inside the math module.

Note: To learn more about library functions, please visit Python Library Functions.

Example: Python Library Function

```
import math

# sqrt computes the square root
square_root = math.sqrt(4)

print("Square Root of 4 is",square_root)

# pow() comptes the power
power = pow(2, 3)

print("2 to the power 3 is",power)
```

Output

```
Square Root of 4 is 2.0
2 to the power 3 is 8
```

Here, we imported a `math` module to use the library functions `sqrt()` and `pow()`.

More on Python Functions

User Defined Function Vs Standard Library Functions

In Python, functions are divided into two categories: user-defined functions and standard library functions. These two differ in several ways:

User-Defined Functions

These are the functions we create ourselves. They're like our custom tools, designed for specific tasks we have in mind.

They're not part of Python's standard toolbox, which means we have the freedom to tailor them exactly to our needs, adding a personal touch to our code.

Standard Library Functions

Think of these as Python's pre-packaged gifts. They come built-in with Python, ready to use.

These functions cover a wide range of common tasks such as mathematics, file operations, working with strings, etc.

They've been tried and tested by the Python community, ensuring efficiency and reliability.

Default Arguments in Functions

Python allows functions to have default argument values. Default arguments are used when no explicit values are passed to these parameters during a function call.

Let's look at an example.

```
def greet(name, message="Hello"):
    print(message, name)

# calling function with both arguments
greet("Alice", "Good Morning")

# calling function with only one argument
greet("Bob")
```

Output

```
Good Morning Alice
Hello Bob
```

Here, `message` has the default value of `Hello`. When `greet()` is called with only one argument, `message` uses its default value.

Note: To learn more about default arguments in a function, please visit Python Function Arguments.

Using `*args` and `**kwargs` in Functions

We can handle an arbitrary number of arguments using special symbols `*args` and `**kwargs`.

`*args` in Functions

Using `*args` allows a function to take any number of positional arguments.

```
# function to sum any number of arguments
def add_all(*numbers):
    return sum(numbers)

# pass any number of arguments
print(add_all(1, 2, 3, 4))
```

Output

```
10
```

`*kwargs` in Functions

Using `**kwargs` allows the function to accept any number of keyword arguments.

```
# function to print keyword arguments
def greet(**words):
    for key, value in words.items():
        print(f"{key}: {value}")

# pass any number of keyword arguments
greet(name="John", greeting="Hello")
```

Output

```
name: John
greeting: Hello
```

To learn more, visit Python *args and **kwargs.

Also Read:

- Python Recursive Function
- Python Modules
- Python Generators

Table of Contents