

Module 2

Chapter 2:

Functions Basics: Built-in Functions, Declaring and calling user defined functions, Parameters and default arguments, Fruitful functions and void functions, recursive functions.

Built-in Functions:

Python built-in functions are a set of predefined functions that are always available for use without needing to import any modules. These functions provide fundamental functionalities for various tasks, including data type manipulation, input/output operations, and working with iterables.

Here are some examples of commonly used built-in functions in Python:

Mathematical Functions:

- `abs()`: Returns the absolute value of a number.
- `divmod()`: Returns the quotient and remainder of a division.
- `max()`: Returns the largest item in an iterable or the largest of two or more arguments.
- `min()`: Returns the smallest item in an iterable or the smallest of two or more arguments.
- `pow()`: Returns the result of a number raised to a power.
- `round()`: Rounds a number to a specified number of decimal places.
- `sum()`: Returns the sum of all items in an iterable.

Type Conversion and Manipulation:

- `bool()`: Converts a value to its boolean equivalent.
- `chr()`: Returns the character corresponding to a given Unicode code point.
- `complex()`: Creates a complex number.
- `dict()`: Creates a dictionary.
- `float()`: Converts a value to a floating-point number.
- `int()`: Converts a value to an integer.
- `list()`: Creates a list.
- `ord()`: Returns the Unicode code point of a character.
- `set()`: Creates a set.
- `str()`: Converts a value to a string.
- `tuple()`: Creates a tuple.
- `type()`: Returns the type of an object.

Input/Output and Iterables:

- `dir()`: Returns a list of attributes and methods of an object.
 - `enumerate()`: Adds a counter to an iterable and returns it as an enumerate object.
 - `input()`: Reads input from the user.
 - `len()`: Returns the length of an object (e.g., string, list, tuple).
 - `print()`: Prints objects to the console.
 - `range()`: Generates a sequence of numbers.
 - `reversed()`: Returns a reverse iterator.
 - `sorted()`: Returns a new sorted list from the items in an iterable.
 - `zip()`: Combines multiple iterables into an iterator of tuples.
- This is not an exhaustive list, as Python includes over 60 built-in functions, each serving a specific purpose. These functions are fundamental to writing efficient and readable Python code.

Declaring and calling user defined functions:

Declaring and calling user-defined functions in Python involves two main steps: defining the function and then invoking it.

1. Declaring (Defining) a Function:

To define a function in Python, the `def` keyword is used, followed by the function's name and parentheses. Within these parentheses, any parameters the function will accept are listed, separated by commas. A colon `:` then follows, indicating the start of the function's body. The code block that constitutes the function's logic must be indented.

Python

```
def function_name(parameter1, parameter2):  
    """  
    Optional: This is a docstring describing what the function does.  
    It's good practice to include one.  
    """  
    # Function body - indented code block  
    result = parameter1 + parameter2  
    return result # Optional: return a value
```

- `def`: The keyword signaling the start of a function definition.
- `function_name`: A descriptive name for the function, adhering to Python's naming conventions (snake_case is common for functions).

- parameter1, parameter2: Optional parameters (also called arguments) that the function expects as input.
- :: Marks the end of the function signature and the beginning of the indented function body.
- **Indentation:** All lines belonging to the function's body must be consistently indented.
- return: Optional keyword to send a value back to the caller. If omitted, the function implicitly returns None.

2. Calling a Function:

To execute the code within a defined function, you "call" or "invoke" it. This is done by writing the function's name followed by parentheses, and providing any required arguments within those parentheses.

Python

```
# Calling the function defined above
value1 = 10
value2 = 20
output = function_name(value1, value2)
print(output) # This will print 30
```

- function_name(arguments): The function's name followed by parentheses containing the actual values (arguments) that correspond to the parameters defined in the function signature.

```
def greet_user(name):
    """
    Greets a user by their name.
    """
    print(f"Hello, {name}!")

# Calling the function
greet_user("Alice") # Output: Hello, Alice!
greet_user("Bob")   # Output: Hello, Bob!

def calculate_area(length, width):
    """
    Calculates the area of a rectangle.
    """
    area = length * width
    return area

# Calling the function and storing the returned value
rectangle_area = calculate_area(5, 8)
print(f"The area of the rectangle is: {rectangle_area}")
```

Output: The area of the rectangle is: 40

Parameters and default arguments:

1. Parameters in Python Functions

Definition:

Parameters are variables that are used inside the function definition to receive values (called *arguments*) when the function is called.

Types of Parameters:

1. **Positional Parameters** – Order matters.
2. **Keyword Parameters** – Arguments are passed by name.
3. **Default Parameters** – Have a default value if no argument is given.
4. **Variable-length Parameters** – Used when the number of arguments is unknown (*args, **kwargs).

Example:

```
python

def greet(name, message):
    print("Hello", name + ",", message)

# Calling the function
greet("Prashant", "Welcome to Python programming!")
```

Output:

```
css

Hello Prashant, Welcome to Python programming!
```

2. Default Arguments

Definition:

Default arguments are parameters that assume a default value if no argument is provided during the function call.

Example:

Example:

```
python

def greet(name, message="Good Morning"):
    print("Hello", name + ",", message)

# Calling with and without the second argument
greet("Prashant")
greet("Riya", "How are you?")
```

Output:

```
sql

Hello Prashant, Good Morning
Hello Riya, How are you?
```



Explanation:

- When only one argument is given, message takes the default value "Good Morning".
- When both are given, the default is overridden.

Fruitful functions and void functions:

Fruitful Functions

Definition:

A **fruitful function** is a function that **returns a value** using the return statement.

Purpose:

Used when we want to compute and get a result back.

Example:

```
def add(a, b):  
    result = a + b  
    return result  
  
# Calling the function and storing the return value  
sum_value = add(10, 20)  
print("Sum is:", sum_value)
```

Output:

csharp

Sum is: 30

**Explanation:**

The function add() returns the sum to the calling program — it produces a *fruit* (a value).

4. Void Functions**Definition:**

A **void function** is a function that **does not return any value**. It only performs some action (like printing, updating, etc.).

Example:

```
python

def display(name):
    print("Welcome", name)

display("Prashant")
```

Output:

```
nginx

Welcome Prashant
```

Recursion in Python:

Recursion is a programming technique where a function calls itself either directly or indirectly to solve a problem by breaking it into smaller, simpler subproblems.

In Python, recursion is especially useful for problems that can be divided into identical smaller tasks, such as mathematical calculations, tree traversals or divide-and-conquer algorithms.

Recursion is the process of defining something in terms of itself.

Working of Recursion:

A **recursive function** is just like any other Python function except that it calls itself in its body. Let's see basic structure of recursive function:

```
def recurse():
    ...
    recurse()
    ...

recurse()
```

Syntax:

```
def recursive_function(parameters):  
    if base_case_condition:  
        return base_result  
    else:  
        return recursive_function(modified_parameters)
```

Recursive function contains two key parts:

- **Base Case:** The stopping condition that prevents infinite recursion.
- **Recursive Case:** The part of the function where it calls itself with modified parameters.

Examples of Recursion

Let's understand recursion better with the help of some examples.

Example 1: Factorial Calculation

This code defines a recursive function to calculate factorial of a number, where function repeatedly calls itself with smaller values until it reaches the base case.

```
def factorial(n):  
    if n == 0: # Base case  
        return 1  
    else:      # Recursive case  
        return n * factorial(n - 1)  
  
print(factorial(5))
```

Output

120

Explanation:

- **Base Case:** When **n == 0**, recursion stops and returns **1**.
- **Recursive Case:** Multiplies **n** with the factorial of **n-1** until it reaches the base case.

Example of a recursive function

```
def factorial(x):  
    """This is a recursive function  
    to find the factorial of an integer"""  
  
    if x == 1:  
        return 1  
    else:  
        return (x * factorial(x-1))  
  
num = 3  
print("The factorial of", num, "is", factorial(num))
```

Output

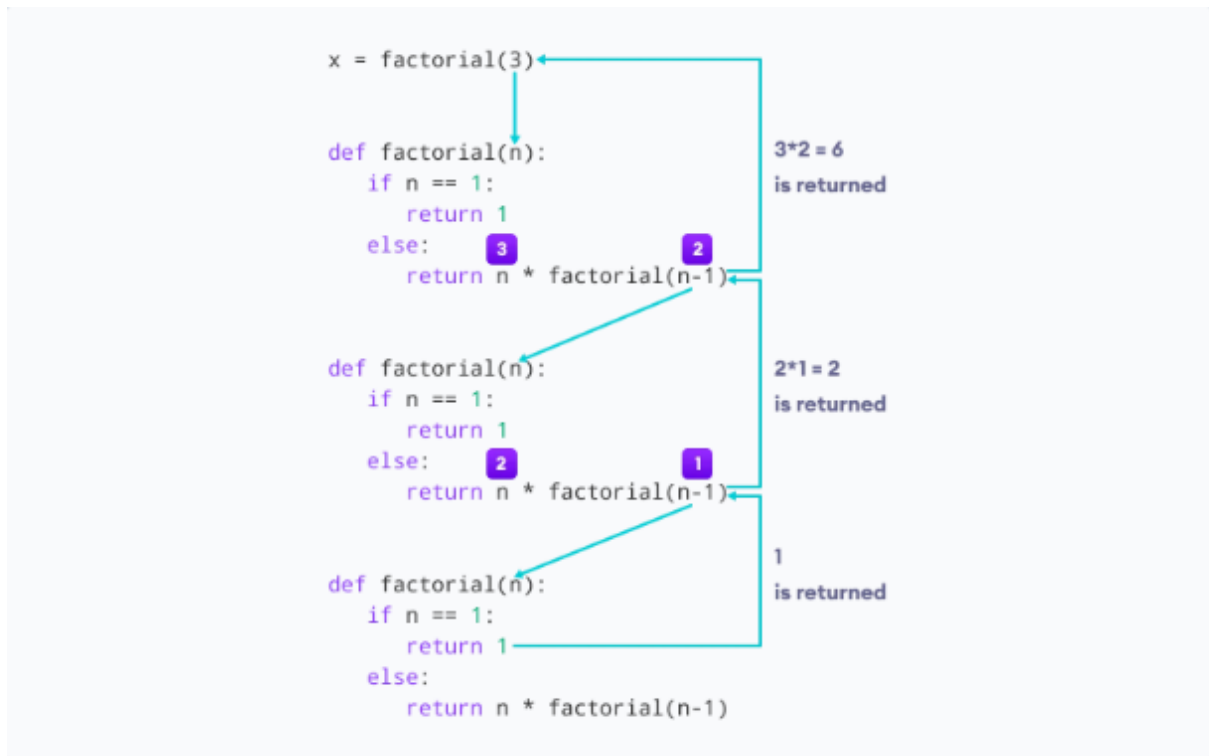
```
The factorial of 3 is 6
```

In the above example, `factorial()` is a recursive function as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial(3)      # 1st call with 3  
3 * factorial(2)  # 2nd call with 2  
3 * 2 * factorial(1) # 3rd call with 1  
3 * 2 * 1        # return from 3rd call as number=1  
3 * 2            # return from 2nd call  
6                # return from 1st call
```



Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

When to Avoid Recursion:

- When the problem can be solved easily with loops.
- When recursion depth is large enough to risk a stack overflow.
- When performance is critical and function call overhead matters.

Recursion vs Iteration

Recursion:

It is often more intuitive and easier to implement when the problem is naturally recursive, like tree traversals. It can lead to solutions that are easier to understand compared to iterative ones.

Iteration:

Iteration involves loops ([for](#), [while](#)) to repeat the execution of a block of code. It is generally more memory-efficient as it does not involve multiple stack frames like recursion.

When to Avoid Recursion

- When the problem can be solved easily with loops.
- When recursion depth is large enough to risk a stack overflow.
- When performance is critical and function call overhead matters.