Parameters

In Python, parameters are variables that are used in a function definition to represent the input values that the function expects. These parameters act as placeholders for actual values that will be provided when the function is called. Understanding how to work with parameters is crucial for creating flexible and reusable functions. Here's an overview of different aspects related to parameters in Python:

Function Parameters:

Function parameters are specified in the parentheses following the function name. They are variables that will be used within the function to perform operations.

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

# Calling the function with an argument
greet("Alice")
```

In this example, `name` is a parameter for the `greet` function. When the function is called with `greet("Alice")`, "Alice" is the argument provided for the `name` parameter.

Default Parameters:

You can assign default values to parameters in a function. If an argument is not provided for a parameter during the function call, the default value is used.

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

greet()         # Output: Hello, Guest!
greet("Bob")    # Output: Hello, Bob!
```

In this example, if no argument is provided, the default value "Guest" is used for the `name` parameter.

Keyword Arguments:

When calling a function, you can use keyword arguments to explicitly specify which parameter each argument should be assigned to. This can enhance code readability, especially for functions with multiple parameters.

```python
def person_info(name, age, country):
    print(f"Name: {name}, Age: {age}, Country: {country}")

# Using keyword arguments
person_info(age=25, name="Alice", country="USA")
```

Variable-Length Argument Lists:

Python allows you to define functions that accept a variable number of arguments. This is achieved using `*args` and `**kwargs`.

- `*args` allows a function to accept any number of positional arguments.

```python
def sum_values(*args):
    return sum(args)

result = sum_values(1, 2, 3, 4)
print(result)  # Output: 10
```

- `**kwargs` allows a function to accept any number of keyword arguments.

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

print_info(name="Alice", age=30, city="Wonderland")
```

Unpacking with *:

The `*` operator can be used to unpack iterable objects like lists or tuples and pass the elements as

separate arguments to a function.

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

numbers = (2, 3)
result = add(*numbers)
print(result)  # Output: 5
```

Here, `*numbers` unpacks the tuple `(2, 3)` into separate arguments for the `add` function.

Understanding these concepts allows you to create versatile functions that can handle different types and numbers of input parameters.