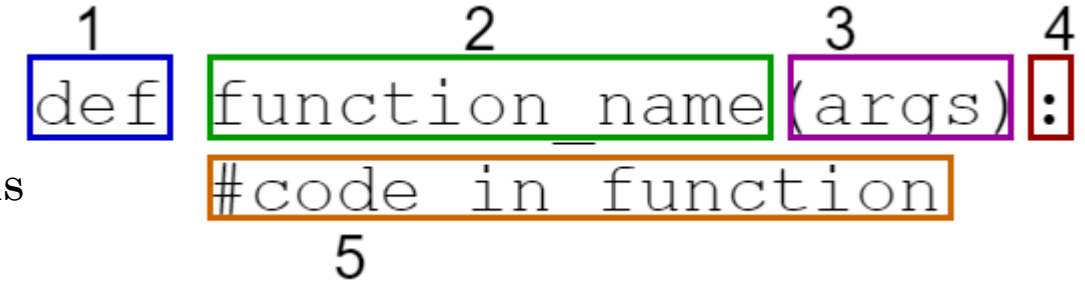
Introduction to Python Programming

What are Functions?

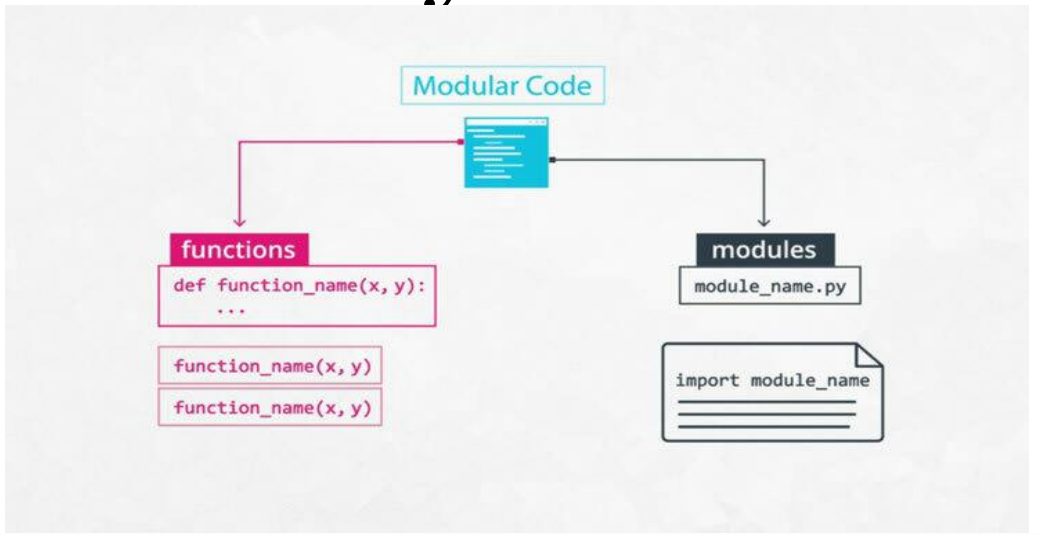
- **Definition:** Functions are named, reusable blocks of code designed to perform a specific task or calculation.
- **Purpose:**
  - They help in organizing code, allowing developers to break complex problems into simpler, manageable parts.
  - Functions can be called multiple times throughout a program, promoting code reusability and reducing redundancy.
  - Syntax: Functions are defined using the def keyword, followed by a function name, and parentheses containing any parameters.
- **Example:**
- def greet(name):
- print(f"Hello, {name}!")
- In this example, greet is a function that takes one parameter, name.



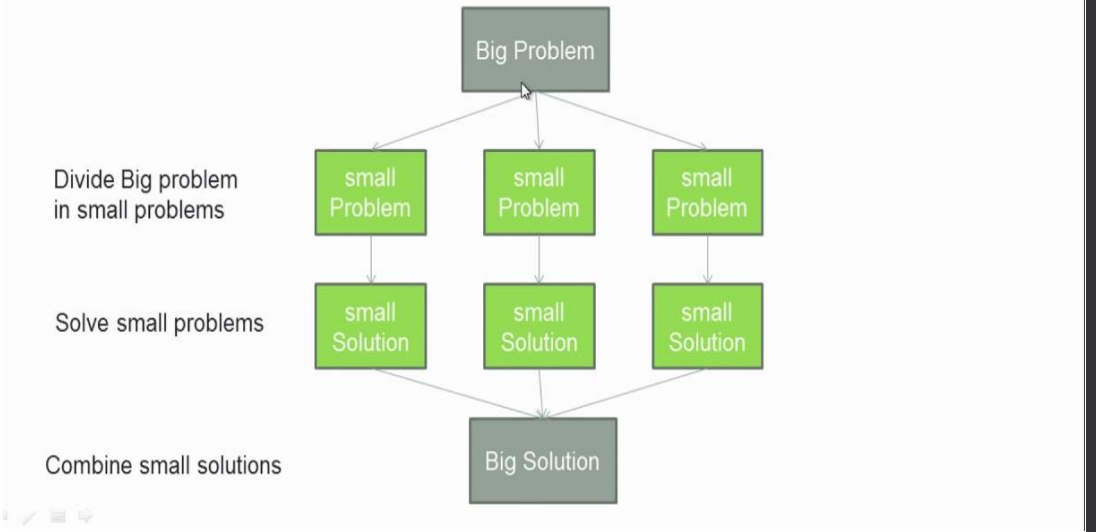
Built-in Functions in Python						
abs()	classmethod()	filter()	id()	max()	property()	str()
all()	compile()	float()	input()	memoryview()	range()	sum()
any()	complex()	format()	int()	min()	repr()	super()
ascii()	delattr()	frozenset()	isinstance()	next()	reversed()	tuple()
bin()	dict()	getattr()	issubclass()	object()	round()	type()
bool()	dir()	globals()	iter()	oct()	set()	vars()
bytearray()	divmod()	hasattr()	len()	open()	setattr()	zip()
bytes()	enumerate()	hash()	list()	ord()	slice()	__import__()
callable()	eval()	help()	locals()	pow()	sorted()	
chr()	exce()	hex()	map()	print()	staticmethod()	

Modularity and Code Reusability

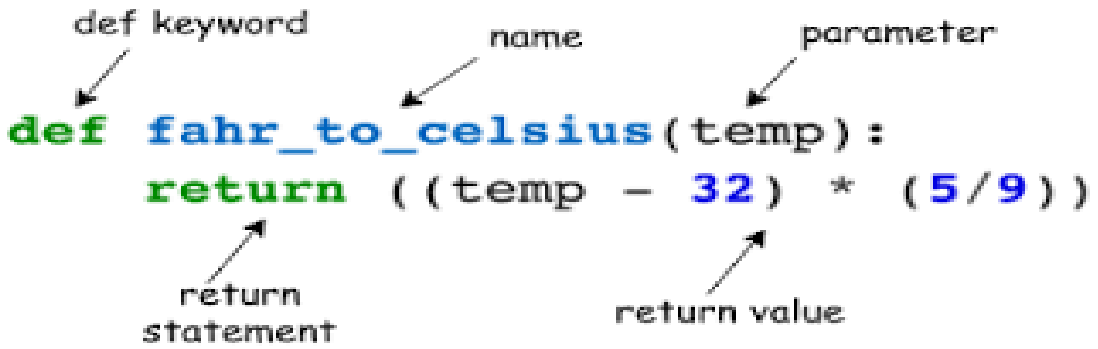
- **Modularity:**
  - Functions allow you to encapsulate functionality, making code more organized and easier to navigate.
  - Each function can be developed, tested, and debugged independently, improving overall software quality.



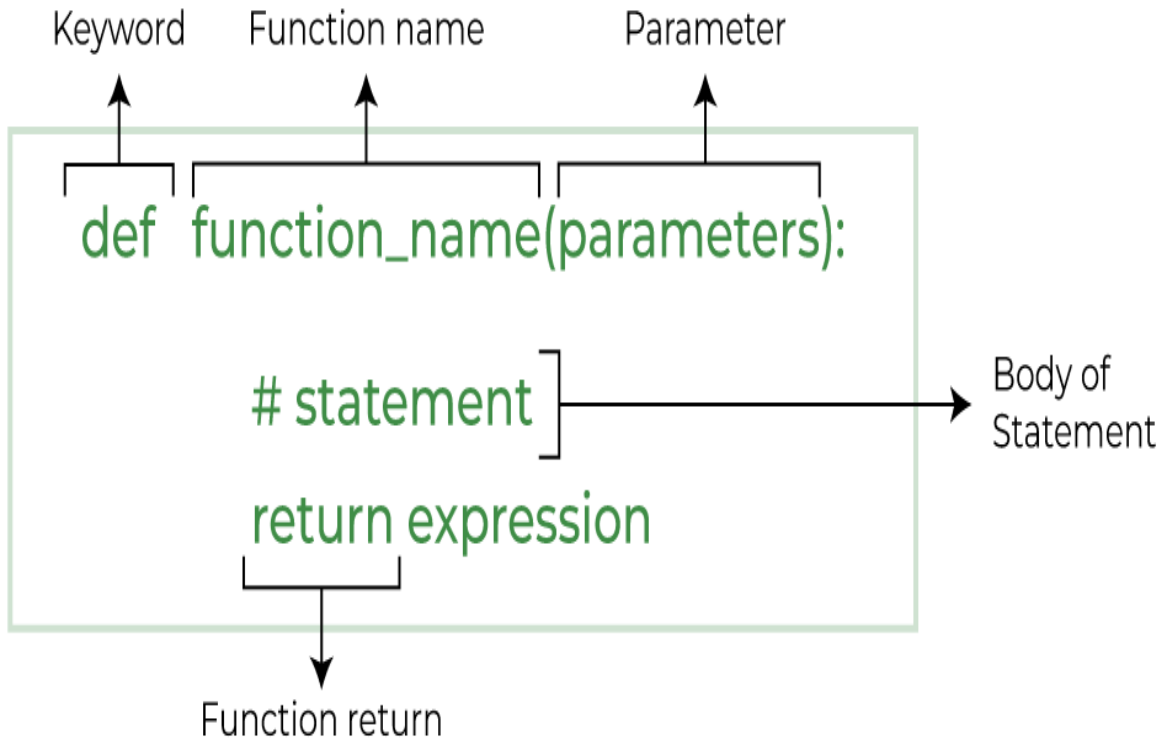
- **Code Reusability:**
  - Once a function is defined, it can be reused in different parts of the program or even in other programs.
  - This reduces the need to write the same code multiple times, making the codebase cleaner and easier to maintain.



Creating Functions

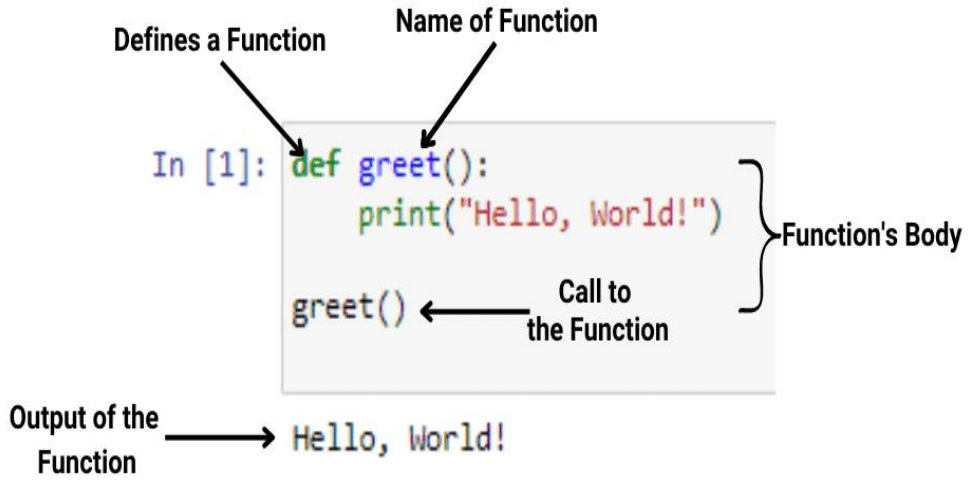


- Function Definition:
- To create a function, use the def keyword followed by the function name and parentheses.
- You can also include parameters inside the parentheses to accept inputs.
- **Example:**
- def add(a, b):
- return a + b
- In this example, add is a function that takes two parameters, a and b, and returns their sum.



Calling Functions

- Execution of Functions: A function is executed when i is called, which means the interpreter runs the code inside the function body.
- Calling Syntax: You call a function by writing its nam followed by parentheses. If the function requires arguments, you provide them within the parentheses.
- **Example:**
- def greet(name):
- print(f"Hello, {name}!")
- # Calling the function
- greet("Alice") # Output: Hello, Alice!
- **Multiple Calls:** Functions can be called multiple times, allowing for code reuse.
- Example:
- greet("Bob") # Output: Hello, Bob!
- greet("Charlie") # Output: Hello, Charlie!



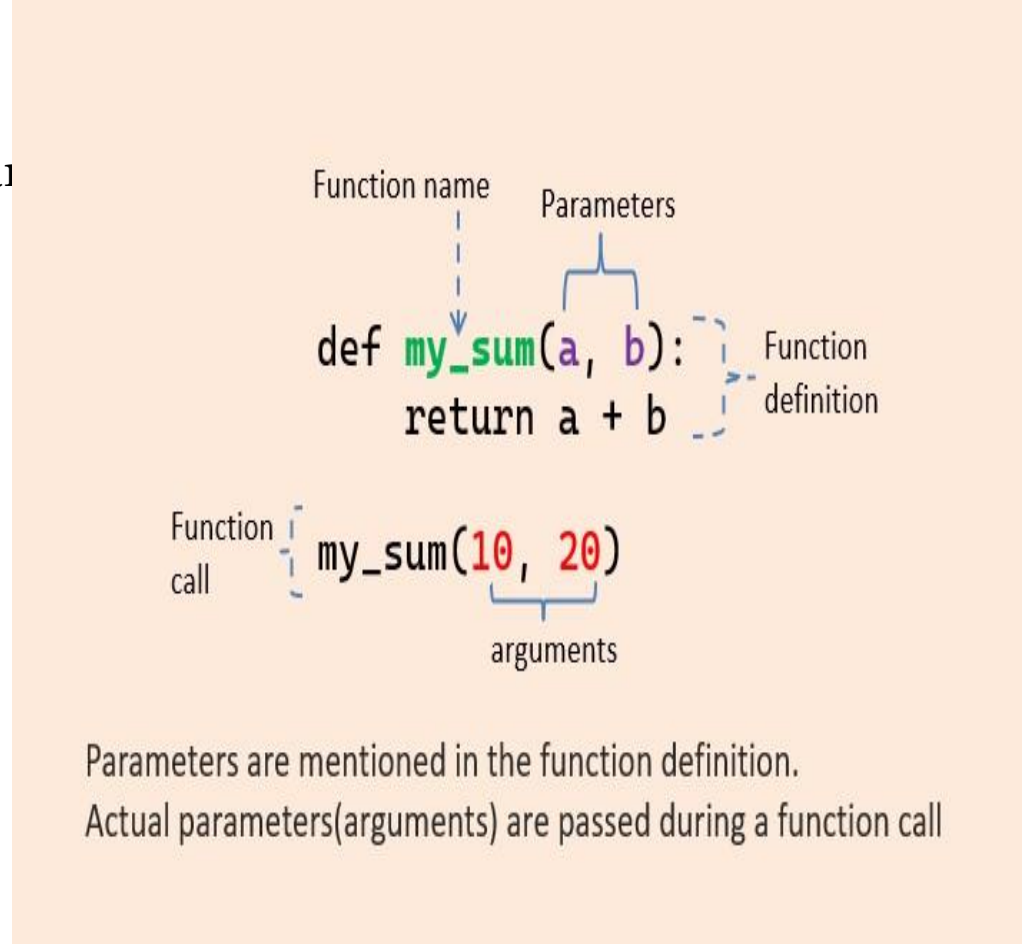

```
1  def multiply(x,y=0):
2      print("value of x=",x)
3      print("value of y=",y)
4
5      return x*y
6
7  print(multiply(y=2,x=4))
8
9
```

Here we have reversed the order of the value for x and y.

```
Run  Python10.2
"C:\Users\DK\Desktop\Python code\Python Test\Python 10\P
value of x= 4
value of y= 2
8
```

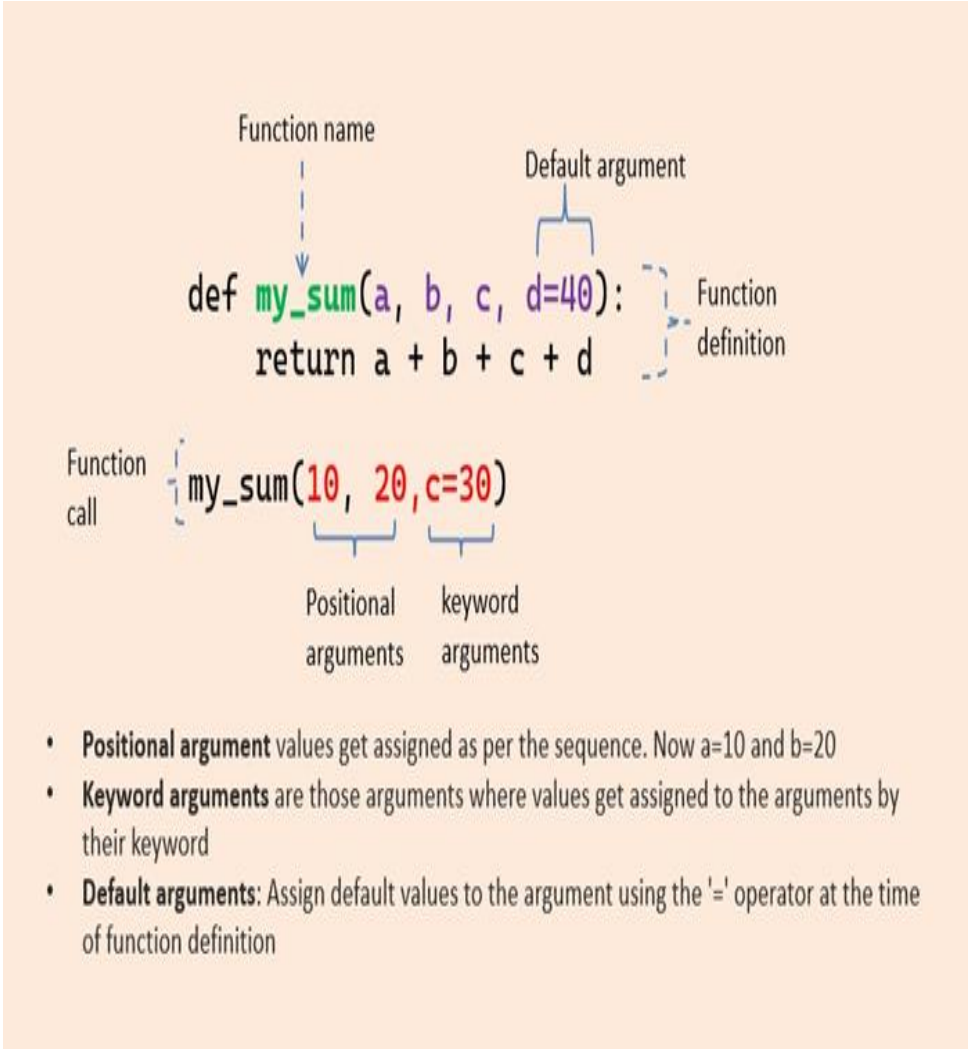
Passing Arguments

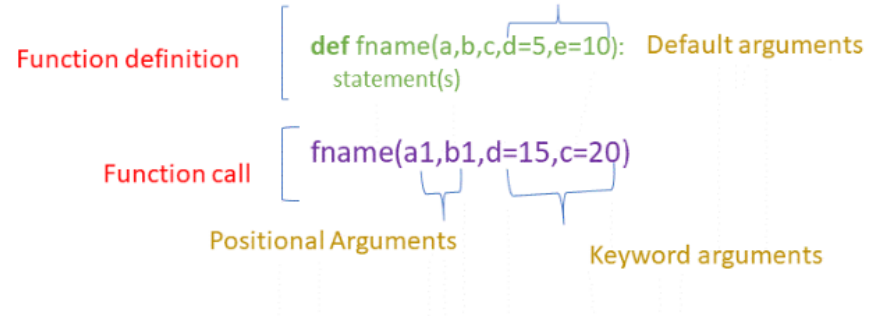
- **Definition:** When calling a function, you can pass values (arguments) that the function can use. These arguments can be used as inputs to perform calculations or manipulations within the function.
- Importance of Arguments: By using arguments, you can write more generic and flexible functions that operate on varying data without hardcoding values.
- **Example:**
- def multiply(a, b):
- return a * b
- # Passing arguments to the function
- result = multiply(4, 5)
- print(result) # Output: 20
- **Benefits:**Allows for dynamic functionality; the same function can work with different inputs.
- Enhances readability and maintainability of the code.



Positional Arguments

- **Definition: Definition:** Positional arguments are arguments that must be provided in the order that the function parameters are defined. The first argument corresponds to the first parameter, the second to the second, and so on.
- **Example:**
- def divide(numerator, denominator):
- if denominator == 0:
- return "Error: Cannot divide by zero."
- return numerator / denominator
- # Correct usage with positional arguments
- result = divide(10, 2) # numerator is 10, denominator is 2
- print(result) # Output: 5.0
- **Incorrect Order:** If the order of arguments is mixed up, it can lead to unexpected results.
- result = divide(2, 10) # numerator is 2, denominator is 10
- print(result) # Output: 0.2
- **Key Point:** Be mindful of the order when passing arguments to avoid logical errors.

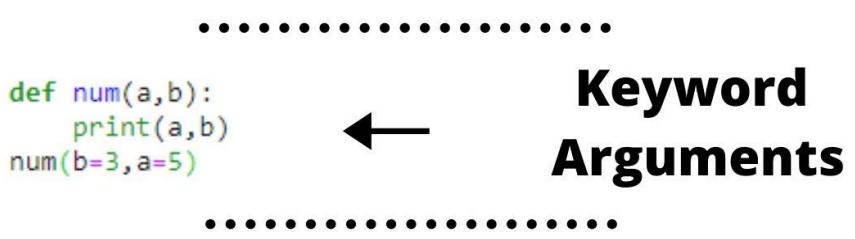


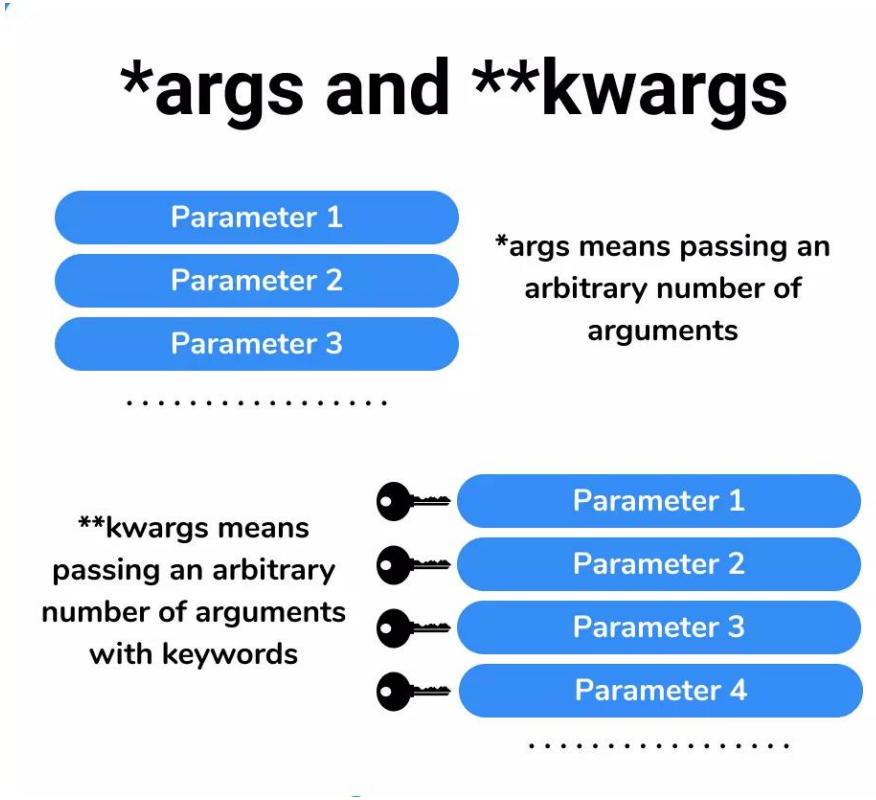


Keyword Arguments

- **Definition:** Keyword arguments allow you to specify the names of the parameters along with their values when calling a function. This means you can pass arguments in any order, improving code readability.
- **Example:**
- def profile(name, age, city):
- return f"{name} is {age} years old and lives in {city}."

- # Calling with keyword arguments
- info = profile(age=30, name="Alice", city="New York")
- print(info) # Output: Alice is 30 years old and lives in New York.
- **Benefits of Using Keyword Arguments:**
  - **Clarity:** Makes it clear what each argument represents, especially when a function has many parameters or when some parameters have default values.
  - **Flexibility:** You can provide arguments in any order, which can be particularly useful for functions with optional parameters.





Variable-length Arguments (*args)

- Definition: *args allows a function to accept any number of positional arguments, which are accessible as a tuple.
- Usage: This is useful when you don't know beforehand how many arguments will be passed to the function.
- Example:
- def concatenate_strings(*args):
- return " ".join(args)
- result = concatenate_strings("Hello", "world", "from", "Python!")
- print(result) # Output: Hello world from Python!
- Here, *args collects all provided arguments into a single tuple.

• Variable-length arguments:

```
def print_info(first, *rest):
    print(first, end='')
    for one in rest:
        print(' ', one, end='')
    print('')


print_info(1)
print_info(1, 2, 3)
```

```
1
1 2 3
```

Global and Local Variables

- **Global Variables**: Variables defined outside of any function are accessible anywhere in the code.
- Local Variables: Variables defined inside a function are only accessible within that function's scope.
- **Example:**
- global_var = 10 # Global variable
- def function():
-     local_var = 5 # Local variable
-     print(global_var) # Accessing global variable
- function()
- # print(local_var) # This would raise an error because local_var is not accessible outside the function.