

Functions

Objectives:

By the end of this reading, you should be able to:

1. Describe the function concept and the importance of functions in programming
2. Write a function that takes inputs and performs tasks
3. Use built-in functions like `len()`, `sum()`, and others effectively
4. Define and use your functions in Python
5. Differentiate between global and local variable scopes
6. Use loops within the function
7. Modify data structures using functions

Introduction to functions

A function is a fundamental building block that encapsulates specific actions or computations. As in mathematics, where functions take inputs and produce outputs, programming functions perform similarly. They take inputs, execute predefined actions or calculations, and then return an output.

Purpose of functions

Functions promote code modularity and reusability. Imagine you have a task that needs to be performed multiple times within a program. Instead of duplicating the same code at various places, you can define a function once and call it whenever you need that task. This reduces redundancy and makes the code easier to manage and maintain.

Benefits of using functions

Modularity: Functions break down complex tasks into manageable components

Reusability: Functions can be used multiple times without rewriting code

Readability: Functions with meaningful names enhance code understanding

Debugging: Isolating functions eases troubleshooting and issue fixing

Abstraction: Functions simplify complex processes behind a user-friendly interface

Collaboration: Team members can work on different functions concurrently

Maintenance: Changes made in a function automatically apply wherever it's used

How functions take inputs, perform tasks, and produce outputs

Inputs (Parameters)

Functions operate on data, and they can receive data as input. These inputs are known as *parameters* or *arguments*. Parameters provide functions with the necessary information they need to perform their tasks. Consider parameters as values you pass to a function, allowing it to work with specific data.

Performing tasks

Once a function receives its input (parameters), it executes predefined actions or computations. These actions can include calculations, operations on data, or even more complex tasks. The purpose of a function determines the tasks it performs. For instance, a function could calculate the sum of numbers, sort a list, format text, or fetch data from a database.

Producing outputs

After performing its tasks, a function can produce an output. This output is the result of the operations carried out within the function. It's the value that the function "returns" to the code that called it. Think of the output as the end product of the function's work. You can use this output in your code, assign it to variables, pass it to other functions, or even print it out for display.

Example:

Consider a function named `calculate_total` that takes two numbers as input (parameters), adds them together, and then produces the sum as the output. Here's how it works:

```
1 def calculate_total(a, b): # Parameters: a and b
2     total = a + b         # Task: Addition
3     return total          # Output: Sum of a and b
4
5 result = calculate_total(5, 7) # Calling the function with inputs 5 and 7
6 print(result) # Output: 12
```

Python's built-in functions

Python has a rich set of built-in functions that provide a wide range of functionalities. These functions are readily available for you to use, and you don't need to be concerned about how they are implemented internally. Instead, you can focus on understanding what each function does and how to use it effectively.

Using built-in functions or Pre-defined functions

To use a built-in function, you simply call the function's name followed by parentheses. Any required arguments or parameters are passed into the function within these parentheses. The function then performs its predefined task and may return an output you can use in your code.

Here are a few examples of commonly used built-in functions:

len(): Calculates the length of a sequence or collection

```
string_length = len("Hello, World!") # Output: 13
list_length = len([1, 2, 3, 4, 5]) # Output: 5
```

sum(): Adds up the elements in an iterable (list, tuple, and so on)

```
total = sum([10, 20, 30, 40, 50]) # Output: 150
```

max(): Returns the maximum value in an iterable

```
highest = max([5, 12, 8, 23, 16]) # Output: 23
```

min(): Returns the minimum value in an iterable

```
lowest = min([5, 12, 8, 23, 16]) # Output: 5
```

Python's built-in functions offer a wide array of functionalities, from basic operations like `len()` and `sum()` to more specialized tasks.

Defining your functions

Defining a function is like creating your mini-program:

1. Use `def` followed by the function name and parentheses

Here is the syntax to define a function:

```
def function_name():  
    pass
```

A `"pass"` statement in a programming function is a placeholder or a no-op (no operation) statement. Use it when you want to define a function or a code block syntactically but do not want to specify any functionality or implementation at that moment.

- **Placeholder:** `"pass"` acts as a temporary placeholder for future code that you intend to write within a function or a code block.
- **Syntax Requirement:** In many programming languages like Python, using `"pass"` is necessary when you define a function or a conditional block. It ensures that the code remains syntactically correct, even if it doesn't do anything yet.
- **No Operation:** `"pass"` itself doesn't perform any meaningful action. When the interpreter encounters `"pass"`, it simply moves on to the next statement without executing any code.

Function Parameters:

- Parameters are like inputs for functions
- They go inside parentheses when defining the function
- Functions can have multiple parameters

Example:

```
1 def greet(name):
2     return "Hello, " + name
3
4 result = greet("Alice")
5 print(result) # Output: Hello, Alice
```

Docstrings (Documentation Strings)

- Docstrings explain what a function does
- Placed inside triple quotes under the function definition
- Helps other developers understand your function

Example:

```
1 def multiply(a, b):
2     """
3     This function multiplies two numbers.
4     Input: a (number), b (number)
5     Output: Product of a and b
6     """
7     print(a * b)
8 multiply(2,6)
```

Return statement

- Return gives back a value from a function
- Ends the function's execution and sends the result
- A function can return various types of data

Example:

```
1 def add(a, b):
2     return a + b
3
4 sum_result = add(3, 5) # sum_result gets the value 8
```

Understanding scopes and variables

Scope is where a variable can be seen and used:

- **Global Scope:** Variables defined outside functions; accessible everywhere
- **Local Scope:** Variables inside functions; only usable within that function

Example:

Part 1: Global variable declaration

```
1 global_variable = "I'm global"
```

This line initializes a global variable called `global_variable` and assigns it the value "I'm global".

Global variables are accessible throughout the entire program, both inside and outside functions.

Part 2: Function definition

```
1 def example_function():
2     local_variable = "I'm local"
3     print(global_variable) # Accessing global variable
4     print(local_variable)  # Accessing local variable
```

Here, you define a function called `example_function()`.

Within this function:

- A local variable named `local_variable` is declared and initialized with the string value "I'm local." This variable is local to the function and can only be accessed within the function's scope.

- The function then prints the values of both the **global variable (global_variable)** and the **local variable (local_variable)**. It demonstrates that you can access global and local variables within a function.

Part 3: Function call

```
example_function()
```

In this part, you call the `example_function()` by invoking it. This results in the function's code being executed.

As a result of this function call, it will print the values of the global and local variables within the function.

Part 4: Accessing global variable outside the function

```
print(global_variable) # Accessible outside the function
```

After calling the function, you print the value of the global variable `global_variable` outside the function. **This demonstrates that global variables are accessible inside and outside of functions.**

Part 5: Attempting to access local variable outside the function

```
# print(local_variable) # Error, local variable not visible here
```

In this part, you are attempting to print the value of the local variable `local_variable` outside of the function. However, this line would result in an error.

Local variables are only visible and accessible within the scope of the function where they are defined.

Attempting to access them outside of that scope would raise a `"NameError"`.

Using functions with loops

Functions and loops together

1. Functions can contain code with loops
2. This makes complex tasks more organized
3. The loop code becomes a repeatable function

Example:

```
1 def print_numbers(limit):  
2     for i in range(1, limit+1):  
3         print(i)  
4  
5 print_numbers(5) # Output: 1 2 3 4 5
```

Enhancing code organization and reusability

1. Functions group similar actions for easy understanding
2. Looping within functions keeps code clean
3. You can reuse a function to repeat actions

Example

```
1 def greet(name):  
2     return "Hello, " + name  
3  
4 for _ in range(3):  
5     print(greet("Alice"))
```

Modifying data structure using functions

You'll use Python and a list as the data structure for this illustration. In this example, you will create functions to add and remove elements from a list.

Part 1: Initialize an empty list

```
# Define an empty list as the initial data structure
```



```
my_list = []
```

In this part, you start by creating an empty list named `my_list`. This empty list serves as the data structure that you will modify throughout the code.

Part 2: Define a function to add elements

```
1 # Function to add an element to the list
2 def add_element(data_structure, element):
3     data_structure.append(element)
```

Here, you define a function called `add_element`. This function takes two parameters:

- `data_structure`: This parameter represents the list to which you want to add an element
- `element`: This parameter represents the element you want to add to the list

Inside the function, you use the `append` method to add the provided element to the `data_structure`, which is assumed to be a list.

Part 3: Define a function to remove elements

```
1 # Function to remove an element from the list
2 def remove_element(data_structure, element):
3     if element in data_structure:
4         data_structure.remove(element)
5     else:
6         print(f"{element} not found in the list.")
```

In this part, you define another function called `remove_element`. It also takes two parameters:

- `data_structure`: The list from which we want to remove an element
- `element`: The element we want to remove from the list

Inside the function, you use conditional statements to check if the element is present in the `data_structure`. If it is, you use the `remove` method to remove the first

occurrence of the element. If it's not found, you print a message indicating that the element was not found in the list.

Part 4: Add elements to the list

```
1 # Add elements to the list using the add_element function
2 add_element(my_list, 42)
3 add_element(my_list, 17)
4 add_element(my_list, 99)
```

Here, you use the `add_element` function to add three elements (42, 17, and 99) to the `my_list`. These are added one at a time using function calls.

Part 5: Print the current list

```
# Print the current list
print("Current list:", my_list)
```

This part simply prints the current state of the `my_list` to the console, allowing us to see the elements that have been added so far.

Part 6: Remove elements from the list

```
# Remove an element from the list using the remove_element function
remove_element(my_list, 17)
remove_element(my_list, 55) # This will print a message since 55 is not in the list
```

In this part, you use the `remove_element` function to remove elements from the `my_list`. First, you attempt to remove 17 (which is in the list), and then you try to remove 55 (which is not in the list). **The second call to `remove_element` will print a message indicating that 55 was not found.**

Part 7: Print the updated list

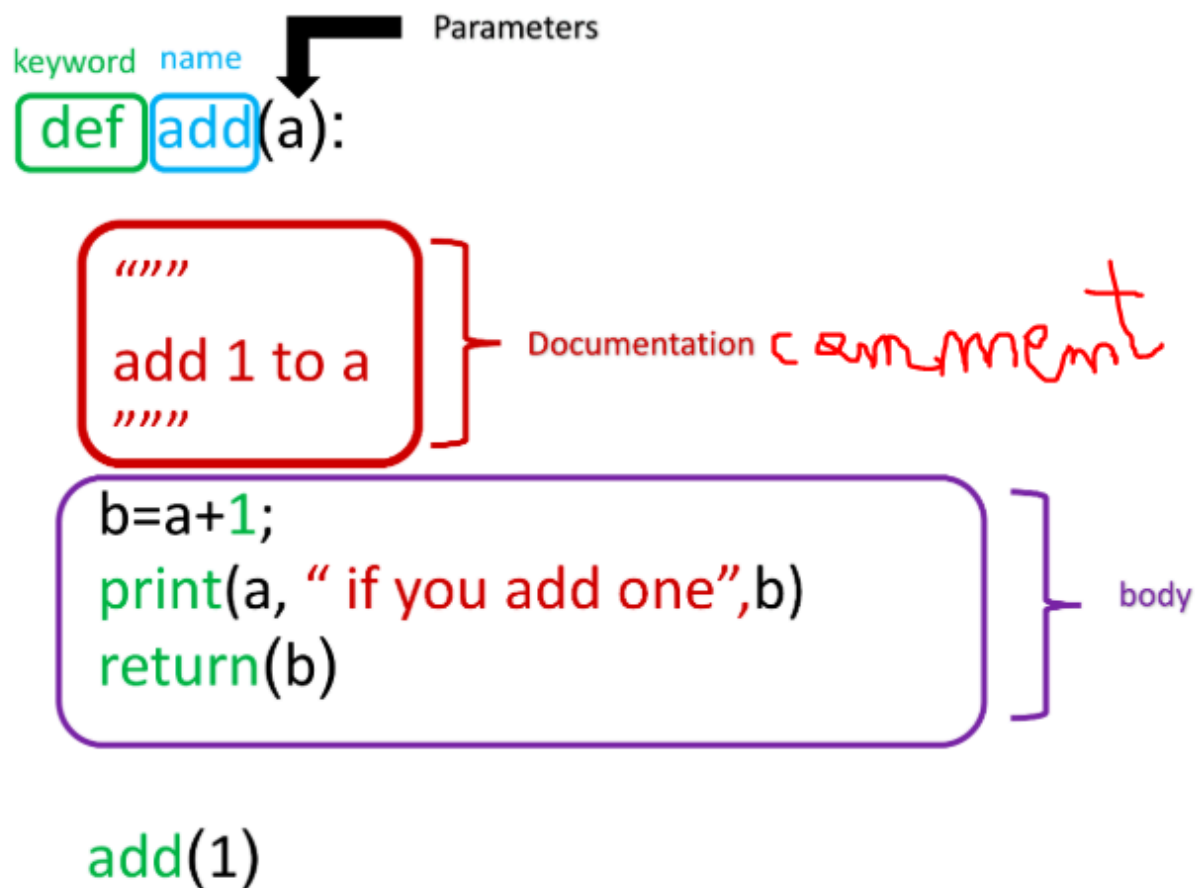
```
# Print the updated list
print("Updated list:", my_list)
```

Finally, you print the updated `my_list` to the console. This allows us to observe the modifications made to the list by adding and removing elements using the defined functions.

Conclusion

Congratulations! You've completed the Reading Instruction Lab on Python functions. You've gained a solid understanding of functions, their significance, and how to create and use them effectively. These skills will empower you to write more organized, modular, and powerful code in your Python projects.

The figure below illustrates the terminology:



Variables

The input to a function is called a formal parameter.

A variable that is declared inside a function is called a local variable. The parameter only exists within the function (i.e. the point where the function starts and stops).

A variable that is declared outside a function definition is a global variable, and its value is accessible and modifiable throughout the program. We will discuss more :

```
# Function Definition

def square(a):

    # Local variable b
    b = 1
    c = a * a + b
    print(a, "if you square + 1", c)
    return(c)
```

The labels are displayed in the figure:

```
def square(a):  Formal parameter
    """
    Square the input add add 1
    """
    c=1  Local variable
    b=a*a+c;
    print(a, " if you square+1 ",b)
    return(b)  Function definition

x=2;
z= square(x)  Main program code
```

If there is no return statement, the function returns None. The following two functions are equivalent:

```
# Define functions, one with return value None and other without return value

def MJ():
    print('The BodyGuard')

def MJ1():
    print('The BodyGuard')
    return(None)
```

Printing the function after a call reveals a **None** is the default return statement:

```
# See what functions returns are
```

```
print(MJ())  
print(MJ1())
```

```
The BodyGuard  
None  
The BodyGuard  
None
```

The terms pre-defined functions and in-built functions are often used interchangeably, but they can have slightly different meanings depending on the context or programming language. Here's a clear distinction:

◆ 1. In-Built Functions

- **Definition:** These are functions that are part of the core language or standard library.
- **Purpose:** You can use them directly without defining them yourself.
- **Examples:** `print()`, `len()`, `type()` in Python.

✓ They come with the language itself.

python

Copy Edit

```
# Python example of in-built functions  
print("Hello")           # Displays text  
length = len("Apple")    # Returns length of string  
print(length)            # Output: 5
```

◆ 2. Pre-Defined Functions

- **Definition:** These are functions that are **already defined** in external libraries, modules, or by someone else, but **not part of the core language**.
- **Purpose:** You can **import and use** them, but they aren't built into the language by default.
- **Examples:** `sqrt()` from the `math` module in Python.

✓ You need to import the module or library first.

python

📄 Copy 🗑️ Edit

```
# Python example of pre-defined function
import math

square_root = math.sqrt(25)
print(square_root)    # Output: 5.0
```

✓ Summary Table

Feature	In-Built Functions	Pre-Defined Functions
Where it comes from	Core language	External modules or libraries
Need to import?	✗ No	✓ Yes
Example	<code>print()</code> , <code>len()</code>	<code>math.sqrt()</code> , <code>datetime.now()</code>
Defined by	Language developers	Library or module authors

⚠️ Note

In some books or contexts:

- "Pre-defined" may also refer **collectively to both** in-built and library functions — so it's important to check how the term is used.

Setting default argument values in your custom functions

You can set a default value for arguments in your function. For example, in the `isGoodRating()` function have a default rating of 4:

```
# Example for setting param with default value

def isGoodRating(rating=4):
    if(rating < 7):
        print("this album sucks it's rating is",rating)
    else:
        print("this album is good its rating is",rating)
```

```
# Test the value with default value and with input
```

```
isGoodRating()
isGoodRating(10)
```

```
this album sucks it's rating is 4
this album is good its rating is 10
```

Global variables

So far, we've been creating variables within functions, but we have not discussed variables outside the function. These are called global variables. Let's try to see what `printer1` returns:

```
# Example of global variable

album = "The BodyGuard"
def printer1(album):
    internal_var1 = "Thriller"
    print(album, "is an album")

printer1(album)
# try running the following code
#printer1(internal_var1)

The BodyGuard is an album
```

Example of global variable

```
album = "The BodyGuard"
def printer1(album):
    internal_var1 = "Thriller"
    print(album, "is an album")
```

try running the following code
printer1(internal_var1)

```
-----
NameError                                Traceback (most recent call last)
Cell In[54], line 10
      6 print(album, "is an album")
      9 # try running the following code
--> 10 printer1(internal_var1)

NameError: name 'internal_var1' is not defined
```

We got a Name Error: name 'internal_var' is not defined . Why?

It's because all the variables we create in the function is a local variable, meaning that the variable assignment does not persist outside the function.

But there is a way to create **global variables** from within a function as follows:

```
album = "The BodyGuard"

def printer(album):
    global internal_var
    internal_var = "Thriller"
    print(album, "is an album")

printer(album)
printer(internal_var)
```

The BodyGuard is an album
Thriller is an album

Scope of a Variable

The scope of a variable is the part of that program where that variable is accessible. Variables that are declared outside of all function definitions, such as the `myFavouriteBand` variable in the code shown here, are accessible from anywhere within the program. As a result, such variables are said to have global scope, and are known as global variables. `myFavouriteBand` is a global variable, so it is accessible from within the `getBandRating` function, and we can use it to determine a band's rating. We can also use it outside of the function, such as when we pass it to the print function to display it:

Example of global variable

```
myFavouriteBand = "AC/DC"

def getBandRating(bandname):
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is:", getBandRating("AC/DC"))
print("Deep Purple's rating is:", getBandRating("Deep Purple"))
print("My favourite band is:", myFavouriteBand)
```

```
AC/DC's rating is: 10.0
Deep Purple's rating is: 0.0
My favourite band is: AC/DC
```

Deleting the variable "myFavouriteBand" from the previous example to demonstrate an example of a local variable

```
del myFavouriteBand
```

Example of local variable

```
def getBandRating(bandname):
    myFavouriteBand = "AC/DC"
    if bandname == myFavouriteBand:
        return 10.0
    else:
        return 0.0

print("AC/DC's rating is: ", getBandRating("AC/DC"))
print("Deep Purple's rating is: ", getBandRating("Deep Purple"))
print("My favourite band is", myFavouriteBand)
```

```
AC/DC's rating is: 10.0
Deep Purple's rating is: 0.0
```

```
-----
NameError                                Traceback (most recent call last)
Cell In[43], line 16
     14 print("AC/DC's rating is: ", getBandRating("AC/DC"))
     15 print("Deep Purple's rating is: ", getBandRating("Deep Purple"))
--> 16 print("My favourite band is", myFavouriteBand)

NameError: name 'myFavouriteBand' is not defined
```

`getBandRating` function will still work, because `myFavouriteBand` is still defined within the function. However, we can no longer print `myFavouriteBand` outside our function, because it is a local variable of our `getBandRating` function; it is only defined within the `getBandRating` function:

Example of global variable and local variable with the same name

myFavouriteBand = "AC/DC"

def getBandRating(bandname):

myFavouriteBand = "Deep Purple"

 if bandname == myFavouriteBand:

 return 10.0

 else:

 return 0.0

print("AC/DC's rating is:",getBandRating("AC/DC"))

print("Deep Purple's rating is: ",getBandRating("Deep Purple"))

print("My favourite band is:",myFavouriteBand) ?

AC/DC's rating is: 0.0

Deep Purple's rating is: 10.0

My favourite band is: AC/DC

✓ Part 1: `*args` → Packing into a Tuple

python

Copy Edit

```
def printAll(*args):  
    print("No of arguments:", len(args))  
    for argument in args:  
        print(argument)
```

💡 What does `*args` mean?

- The `*` before `args` means: "Take all **extra arguments** given to this function and **pack them into a tuple** named `args`."
- You can use **any name instead of** `args`, but `*args` is the common convention.

🔍 What does it do?

Let's walk through the call:

python

Copy Edit

```
printAll('Horsefeather', 'Adonis', 'Bone')
```

This call is passing **3 arguments** to `printAll`.

Python packs them into:


python

Copy Edit

```
args = ('Horsefeather', 'Adonis', 'Bone')
```

Then inside the function:

- `len(args)` returns 3
- The `for` loop prints each item in the tuple

 So:

python

Copy Edit

```
def printAll(*args):  
    print("No of arguments:", len(args)) # Prints how many were passed  
    for argument in args:                # Loop through the tuple  
        print(argument)                  # Print each one
```

This is useful when you don't know how many arguments will be passed.



Part 2: `**kwargs` → Packing into a Dictionary

python

Copy Edit

```
def printDictionary(**args):  
    for key in args:  
        print(key + " : " + args[key])
```



What does `**args` mean?

- The `**` before `args` means: "Take all extra named (keyword) arguments and pack them into a dictionary called `args`."
- Again, the name doesn't have to be `args`, but `**kwargs` (for "keyword arguments") is the common name.



What does it do?

Let's walk through this call:

python



Copy Edit

```
printDictionary(Country='Canada', Province='Ontario', City='Toronto')
```

Here you are passing 3 named arguments.

Python packs them into:



python

 Copy  Edit

```
args = {  
    'Country': 'Canada',  
    'Province': 'Ontario',  
    'City': 'Toronto'  
}
```

Then in the loop:



python

 Copy  Edit

```
for key in args:  
    print(key + " : " + args[key])
```

Each key-value pair is printed like:

yaml

 Copy  Edit

```
Country : Canada  
Province : Ontario  
City : Toronto
```



Summary Table

Syntax	Meaning	Packed As	Use Case
<code>*args</code>	Variable number of values	Tuple	Use when the number of arguments is unknown
<code>**kwargs</code>	Variable number of key=value	Dictionary	Use when passing named arguments

📌 Example using both:

python

Copy Edit

```
def mix(*args, **kwargs):
    print("Positional:", args)
    print("Named:", kwargs)

mix(1, 2, 3, name='Alice', age=25)
```

Output:

yml

Copy Edit

```
Positional: (1, 2, 3)
Named: {'name': 'Alice', 'age': 25}
```

Understanding Variable-Length Arguments in Python:

`*args` and `**kwargs`

Let me explain this concept in detail with step-by-step breakdowns.

1. The Problem with Fixed Arguments

Normally, functions have a fixed number of parameters:

python

Copy Download

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

greet("Alice", "Hello") # Works
greet("Bob") # Error - missing 1 argument
```

But what if we want a function that can handle any number of arguments? That's where `*args` and `**kwargs` come in.

2. Packing Arguments into a Tuple (`*args`)

What is `*args` ?

- The `*` before `args` tells Python to pack all positional arguments into a tuple
- `args` is just a convention - you can use any name (like `*numbers`), but `*args` is standard

Your Example Explained:

python

 Copy  Download

```
def printAll(*args):  
    print("No of arguments:", len(args))  
    for argument in args:  
        print(argument)
```

1. When you call `printAll('Horsefeather', 'Adonis', 'Bone')`:

- Python packs these 3 arguments into a tuple: `args = ('Horsefeather', 'Adonis', 'Bone')`
- `len(args)` gives 3
- The loop prints each argument

2. When you call `printAll('Sidecar', 'Long Island', 'Mudslide', 'Carriage')`:

- Now `args = ('Sidecar', 'Long Island', 'Mudslide', 'Carriage')`
- `len(args)` gives 4
- The loop prints all four

More Examples:

python

 Copy  Download

```
printAll(1, 2, 3) # args = (1, 2, 3)  
printAll() # args = () - empty tuple
```


3. Packing Arguments into a Dictionary (`**kwargs`)

What is `**kwargs` ?

- The `**` before `kwargs` tells Python to pack all keyword arguments into a dictionary
- `kwargs` stands for "keyword arguments" (again, convention)

Your Example Explained:

python

Copy Download

```
def printDictionary(**kwargs):  
    for key in kwargs:  
        print(key + " : " + kwargs[key])  
  
printDictionary(Country='Canada',Province='Ontario',City='Toronto')
```

1. When you call with keyword arguments:

- Python packs them into a dictionary: `kwargs = {'Country': 'Canada', 'Province': 'Ontario', 'City': 'Toronto'}`
- The loop goes through each key-value pair and prints them



More Examples:

python

Copy Download

```
printDictionary(name="Alice", age=25)  
# kwargs = {'name': 'Alice', 'age': 25}  
  
printDictionary() # kwargs = {} - empty dictionary
```

4. Combining Both

You can use both in one function:

python

Copy Download

```
def combined_example(*args, **kwargs):  
    print("Positional arguments:", args)  
    print("Keyword arguments:", kwargs)  
  
combined_example(1, 2, 3, name="Alice", age=25)
```

Output:

text

 Copy  Download

```
Positional arguments: (1, 2, 3)
Keyword arguments: {'name': 'Alice', 'age': 25}
```

5. When to Use These

- `*args` is useful when:
 - You want to make a function that can handle any number of inputs
 - You're wrapping another function and need to pass through arguments
- `**kwargs` is useful when:
 - You want to handle named parameters flexibly
 - You're working with configuration options that might vary

6. Practical Example

python

 Copy  Download

```
def make_pizza(*toppings, **details):
    print("\nMaking a pizza with:")
    for topping in toppings:
        print(f"- {topping}")

    print("\nDetails:")
    for key, value in details.items():
        print(f"{key}: {value}")

make_pizza('pepperoni', 'mushrooms', size='large', crust='thin')
```

Output:

text

 Copy  Download

Making a pizza with:

- pepperoni
- mushrooms

Details:

size: large
crust: thin

Key Points to Remember

1. `*args` collects extra positional arguments into a tuple
2. `**kwargs` collects extra keyword arguments into a dictionary
3. The names `args` and `kwargs` are conventions, not requirements
4. Order matters in function definition: `def func(a, b, *args, **kwargs)`
5. These are often used when you need flexibility in function arguments