**Python Functions — Complete Guide**

**1. What is a Function?**

A **function** in Python is a block of reusable code that performs a specific task.  
Instead of writing the same code multiple times, we define it once and call it whenever needed.

**2. Why Use Functions?**

✅ **Reusability** – Write once, use many times.  
✅ **Readability** – Code becomes cleaner and easier to understand.  
✅ **Maintainability** – Easy to update logic in one place.  
✅ **Organization** – Helps structure large programs into smaller parts.  
✅ **Avoid Repetition** – No copy-paste chaos.

**3. Types of Functions**

* **Built-in Functions** – Already provided by Python (print(), len(), type()).
* **User-defined Functions** – Functions you create yourself.
* **Lambda Functions** – Small anonymous functions.

**4. Defining a Function**

**Syntax:**

python

CopyEdit

def function\_name(parameters):

"""Optional docstring that explains the function"""

# code block

return value

**Example:**

python

CopyEdit

def greet(name):

"""This function greets the user."""

print(f"Hello, {name}!")

greet("Talha") # Output: Hello, Talha!

**5. Function Parameters & Arguments**

**5.1 Positional Arguments**

Passed in order.

python

CopyEdit

def add(a, b):

return a + b

print(add(5, 10)) # 15

**5.2 Keyword Arguments**

Passed with names, order doesn’t matter.

python

CopyEdit

print(add(b=10, a=5)) # 15

**5.3 Default Arguments**

Value is used if not provided.

python

CopyEdit

def greet(name="Guest"):

print(f"Hello, {name}!")

greet() # Hello, Guest!

**5.4 Variable-length Arguments**

* **\*args** → For multiple positional arguments.

python

CopyEdit

def total\_sum(\*args):

return sum(args)

print(total\_sum(1, 2, 3, 4)) # 10

* **\*\*kwargs** → For multiple keyword arguments.

python

CopyEdit

def print\_details(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print\_details(name="Talha", age=22)

**6. Return Statement**

* **return** sends a value back from the function.

python

CopyEdit

def square(x):

return x \*\* 2

result = square(5)

print(result) # 25

* If no return is given, Python returns None.

**7. Docstrings (Documentation Strings)**

Explain what the function does.

python

CopyEdit

def multiply(a, b):

"""This function returns the product of two numbers."""

return a \* b

print(multiply.\_\_doc\_\_)

**8. Scope in Functions**

* **Local Scope** → Inside the function.
* **Global Scope** → Outside all functions.
* **global keyword** → Access/modify global variables inside a function.

python

CopyEdit

x = 10 # Global

def test():

global x

x = 20 # Modify global x

test()

print(x) # 20

**9. Lambda Functions**

Short functions without def keyword.

square = lambda x: x \*\* 2

print(square(4)) # 16

**1. What is a Lambda Function in Python?**

A **lambda function** in Python is a **small, anonymous function** (no name) created using the lambda keyword.  
It can have **any number of arguments**, but **only one expression**.

**Syntax:**

python

CopyEdit

lambda arguments: expression

* lambda → keyword to define
* arguments → parameters (like in normal functions)
* expression → a single operation or value returned

**Example:**

python

CopyEdit

square = lambda x: x \*\* 2

print(square(5)) # Output: 25

Here:

* square is the variable holding the lambda function
* It takes x and returns x\*\*2

**2. Pure vs Impure Lambda Functions**

**A. Pure Lambda Functions**

A **pure** lambda function:

* Always produces the same output for the same input.
* Has **no side effects** (doesn’t change variables, print, modify lists, write files, etc.).
* It is **referentially transparent** (safe to replace with its result).

**Example (Pure):**

python

CopyEdit

add = lambda a, b: a + b

print(add(2, 3)) # Output: 5

print(add(2, 3)) # Always 5 → same input → same output

✅ **Advantages of pure:**

* Predictable
* Easy to test
* No unexpected changes in the program

**B. Impure Lambda Functions**

An **impure** lambda function:

* Produces **different output** for the same input OR
* Has **side effects** (modifies external variables, prints, changes lists, etc.)

**Example (Impure):**

python

CopyEdit

total = 0

increment = lambda x: globals().\_\_setitem\_\_('total', total + x) # Changes global variable

increment(5)

print(total) # Output: 5 (changed global variable → side effect)

Another impure example:

python

CopyEdit

import random

rand\_num = lambda: random.randint(1, 10)

print(rand\_num()) # Output changes every time → not pure

⚠ **Disadvantages of impure:**

* Hard to debug
* Unpredictable behavior
* Harder to test

**3. Where Lambdas are Useful**

* **Sorting with custom keys:**

python

CopyEdit

items = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]

items.sort(key=lambda x: x[1])

print(items) # Sorted by names

* **Map, Filter, Reduce:**

python

CopyEdit

nums = [1, 2, 3, 4]

print(list(map(lambda x: x\*\*2, nums))) # Squares

print(list(filter(lambda x: x%2==0, nums))) # Evens

* **Quick throwaway functions** (instead of writing def)

**4. Key Takeaways**

* **Lambda** → small, single-expression anonymous function
* **Pure Lambda** → no side effects, same output for same input
* **Impure Lambda** → has side effects or unpredictable results
* **Best Practice** → Prefer **pure lambdas** for functional programming; use impure lambdas only when necessary (but often, a normal def is better for side effects).

**10. Function Annotations (Optional Typing)**

python

CopyEdit

def add(a: int, b: int) -> int:

return a + b

**11. Functions Inside Functions**

python

CopyEdit

def outer():

def inner():

print("Hello from inner!")

inner()

outer()

**12. Recursive Functions**

A function calling itself.

python

CopyEdit

def factorial(n):

if n == 1:

return 1

return n \* factorial(n - 1)

print(factorial(5)) # 120

**13. Higher-Order Functions**

Functions that accept other functions as arguments or return them.

python

CopyEdit

def shout(text):

return text.upper()

def process(func):

return func("hello")

print(process(shout)) # HELLO

**14. Advantages of Functions**

* **Modular** → Breaks program into small parts.
* **Reusable** → Save time & avoid duplication.
* **Easier Debugging** → Focus on one part at a time.

**15. Example — Complete Usage**

python

CopyEdit

def calculate\_bill(total, discount\_code=None):

discounts = {

"p10": (0.1, 0),

"p20": (0.2, 0),

"f10": (0, 10),

}

percent, fixed = discounts.get(discount\_code, (0, 0))

return total - (total \* percent + fixed)

print(calculate\_bill(100, "p10")) # 90.0

**Your function:**

python

CopyEdit

def special\_chai(\*ingridents, \*\*extras):

print(f"Ingridents: {ingridents}")

print(f"Extras: {extras}")

**1️⃣ \*ingredients → Collects positional arguments**

* The \* before a parameter means **“pack all extra positional arguments into a tuple”**.
* In your call:

python

CopyEdit

special\_chai("Cinamons", "Cardmom", sweetner="Honey", foam="yes")

The **positional arguments** are:

* + "Cinamons"
  + "Cardmom"

So inside the function:

python

CopyEdit

ingredients = ("Cinamons", "Cardmom") # tuple

**2️⃣ \*\*extras → Collects keyword arguments**

* The \*\* before a parameter means **“pack all extra keyword arguments into a dictionary”**.
* The **keyword arguments** in your call are:
  + sweetner="Honey"
  + foam="yes"

So inside the function:

python

CopyEdit

extras = {"sweetner": "Honey", "foam": "yes"} # dict

**3️⃣ How it prints**

When the function runs:

python

CopyEdit

print(f"Ingridents: {ingridents}")

# → Ingredients: ('Cinamons', 'Cardmom')

print(f"Extras: {extras}")

# → Extras: {'sweetner': 'Honey', 'foam': 'yes'}

**4️⃣ Why this is useful**

This lets your function accept:

* Any number of **positional arguments** (\*args style)
* Any number of **keyword arguments** (\*\*kwargs style)

So you can call:

python

CopyEdit

special\_chai("Tea Leaves", "Milk", "Sugar", topping="Whipped Cream", size="Large")

And it will handle both fixed and extra parameters without you changing the function definition.

💡 **Quick analogy:**

* \*args is like a bag for all extra unnamed items.
* \*\*kwargs is like a labeled box for named items.