Python Functions 101

Complete Guide to Python Functions

Table of Contents

---

1. Introduction to Functions

What is a Function?

A **function** is a reusable block of code that performs a specific task. Think of it as a mini-program within your program.

Why Do We Need Functions?

1. Code Reusability

PYTHON

```python
# Without functions - repetitive code
print("=" * 50)
print("Welcome to Section 1")
print("=" * 50)

print("=" * 50)
print("Welcome to Section 2")
print("=" * 50)

print("=" * 50)
print("Welcome to Section 3")
print("=" * 50)

# With functions - write once, use many times
def print_header(title):
    print("=" * 50)
    print(title)
    print("=" * 50)

print_header("Welcome to Section 1")
print_header("Welcome to Section 2")
print_header("Welcome to Section 3")
```

2. Code Organization Functions help break down complex problems into smaller, manageable pieces.

3. Maintainability If you need to fix a bug or update logic, you only need to change it in one place.

4. Abstraction You can use a function without knowing how it works internally.

---

2. Basic Syntax and Structure

Simple Function Definition

PYTHON

```python
def function_name():
    """This is a docstring - describes what the function does"""
    # Function body
    print("Hello, World!")

# Calling the function
function_name()  # Output: Hello, World!
```

Anatomy of a Function:

- `def` - keyword that tells Python we're defining a function
- `function_name` - the name you give your function (follow variable naming rules)
- `()` - parentheses that hold parameters (empty if no parameters)
- `:` - colon indicates the start of the function body
- Indented block - the code that runs when function is called
- Docstring (optional but recommended) - describes the function's purpose

3. Parameters vs Arguments

This is one of the most confused concepts in programming!

Parameters

Parameters are the variables listed in the function definition. They are placeholders.

```python
def greet(name, age):  # 'name' and 'age' are PARAMETERS
    print(f"Hello {name}, you are {age} years old")
```

Think of parameters as **empty boxes** waiting to receive values.

Arguments

Arguments are the actual values you pass to the function when calling it.

```python
greet("Alice", 25)  # "Alice" and 25 are ARGUMENTS
```

Think of arguments as the **actual items** you put into those boxes.

Analogy:

- Parameters = Form with blank fields
- Arguments = The information you fill in those fields

```python
def add_numbers(a, b):  # a, b are parameters (placeholders)
    return a + b

result = add_numbers(5, 3)  # 5, 3 are arguments (actual values)
print(result)  # Output: 8
```

4. The Return Statement

What Does Return Mean?

The `return` keyword sends a value back to the caller. It's how a function gives you a result.

Function Without Return

PYTHON

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

result = greet("Bob")
print(result)  # Output: None
```

Without `return`, the function performs an action but gives back `None`.

Function With Return

PYTHON

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

result = add(5, 3)
print(result)  # Output: 8
```

The function calculates and **returns** the result, which you can store or use.

Key Differences: Print vs Return

```python
# Using print - just displays, doesn't return
def calculate_print(x, y):
    print(x + y)

# Using return - gives back a value
def calculate_return(x, y):
    return x + y

# Compare:
a = calculate_print(5, 3)   # Prints: 8
print(a)                     # Prints: None (no return value)

b = calculate_return(5, 3)   # No output
print(b)                     # Prints: 8 (returned value)

# You can use returned values in expressions
c = calculate_return(5, 3) * 2
print(c)  # Output: 16
```

Multiple Return Values

```python
def get_user_info():
    name = "Alice"
    age = 30
    city = "New York"
    return name, age, city  # Returns a tuple

# Unpack the returned values
n, a, c = get_user_info()
print(n)  # Alice
print(a)  # 30
print(c)  # New York
```

Early Return

`return` also exits the function immediately.

PYTHON
```python
def check_age(age):
    if age < 18:
        return "Minor"  # Function ends here if age < 18
    return "Adult"

print(check_age(15))  # Output: Minor
print(check_age(25))  # Output: Adult
```

---

5. Types of Arguments

5.1 Positional Arguments

Arguments are matched to parameters by their **position** (order).

PYTHON
```python
def introduce(name, age, city):
    print(f"{name} is {age} years old and lives in {city}")

introduce("Alice", 25, "Paris")
# name="Alice", age=25, city="Paris"

introduce(25, "Alice", "Paris")  # WRONG ORDER!
# name=25, age="Alice", city="Paris" - Doesn't make sense!
```

Rule: Order matters! The first argument goes to the first parameter, second to second, etc.

5.2 Keyword Arguments

Arguments are matched to parameters by **name**, not position.

PYTHON
```python
def introduce(name, age, city):
    print(f"{name} is {age} years old and lives in {city}")

# Using keyword arguments - order doesn't matter!
introduce(age=25, city="Paris", name="Alice")
introduce(city="Paris", name="Alice", age=25)

# Both produce: Alice is 25 years old and lives in Paris
```

Advantages:

- More readable
- Order doesn't matter
- Less prone to errors

5.3 Mixing Positional and Keyword Arguments

```python
def book_flight(passenger, destination, date, seat_class):
    print(f"{passenger} flying to {destination} on {date} in {seat_class}")

# Positional first, then keyword
book_flight("John", "London", date="2025-12-01", seat_class="Business")

# This is INVALID:
# book_flight(passenger="John", "London", date="2025-12-01", seat_class="Business")
# SyntaxError: positional argument follows keyword argument
```

Rule: Positional arguments must come before keyword arguments.

5.4 Default Arguments

Parameters can have default values.

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

greet("Alice")              # Output: Hello, Alice!
greet("Bob", "Hi")          # Output: Hi, Bob!
greet("Charlie", greeting="Hey")  # Output: Hey, Charlie!
```

Important Rules:

1. Default parameters must come after non-default parameters

```python
# CORRECT
def func(a, b, c=10, d=20):
    pass

# INCORRECT - SyntaxError
def func(a, c=10, b, d=20):
    pass
```

2. Mutable default arguments can be tricky!

```python
# DANGEROUS - Don't do this!
def add_item(item, my_list=[]):
    my_list.append(item)
    return my_list

print(add_item("apple"))   # ['apple']
print(add_item("banana"))  # ['apple', 'banana'] - Unexpected!

# SAFE - Do this instead
def add_item(item, my_list=None):
    if my_list is None:
        my_list = []
    my_list.append(item)
    return my_list

print(add_item("apple"))   # ['apple']
print(add_item("banana"))  # ['banana'] - Correct!
```

6. *args and **kwargs

These allow functions to accept a variable number of arguments.

6.1 *args (Arbitrary Positional Arguments)

`*args` collects extra positional arguments into a **tuple**.

```python
def sum_all(*args):
    print(f"args is a {type(args)}")
    print(f"Contents: {args}")
    total = 0
    for num in args:
        total += num
    return total

print(sum_all(1, 2, 3))          # Output: 6
print(sum_all(1, 2, 3, 4, 5))    # Output: 15
print(sum_all(10))                # Output: 10
```

How it works:

- The `*` collects all positional arguments into a tuple
- You can iterate over `args` like any tuple
- The name `args` is conventional, but you can use any name (e.g., `*numbers`)

Practical Example:

PYTHON
```python
def create_profile(name, *hobbies):
    print(f"Name: {name}")
    print("Hobbies:")
    for hobby in hobbies:
        print(f"  - {hobby}")

create_profile("Alice", "Reading", "Swimming", "Coding")
# Output:
# Name: Alice
# Hobbies:
#   - Reading
#   - Swimming
#   - Coding
```

6.2 **kwargs (Arbitrary Keyword Arguments)

`**kwargs` collects extra keyword arguments into a **dictionary**.

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

print_info(name="Alice", age=25, city="Paris")
# Output:
# kwargs is a <class 'dict'>
# Contents: {'name': 'Alice', 'age': 25, 'city': 'Paris'}
# name: Alice
# age: 25
# city: Paris
```

Practical Example:

```python
def create_user(username, email, **extra_info):
    user = {
        'username': username,
        'email': email
    }
    # Add all extra information
    user.update(extra_info)
    return user

user1 = create_user("alice", "alice@email.com", age=25, country="USA")
user2 = create_user("bob", "bob@email.com", age=30, country="UK", premium=True)

print(user1)
# {'username': 'alice', 'email': 'alice@email.com', 'age': 25, 'country': 'USA'}
```

6.3 Combining Everything

The order must be:

1. Regular positional parameters
2. *args
3. Keyword-only parameters
4. **kwargs

```python
def complex_function(a, b, *args, c=10, d=20, **kwargs):
    print(f"a: {a}")
    print(f"b: {b}")
    print(f"args: {args}")
    print(f"c: {c}")
    print(f"d: {d}")
    print(f"kwargs: {kwargs}")

complex_function(1, 2, 3, 4, 5, c=100, d=200, x=1000, y=2000)
# Output:
# a: 1
# b: 2
# args: (3, 4, 5)
# c: 100
# d: 200
# kwargs: {'x': 1000, 'y': 2000}
```

6.4 Unpacking with * and **

You can also use `*` and `**` to unpack arguments when calling functions.

PYTHON

```python
def add_three(a, b, c):
    return a + b + c

# Unpacking a list with *
numbers = [1, 2, 3]
result = add_three(*numbers)  # Same as add_three(1, 2, 3)
print(result)  # Output: 6

# Unpacking a dictionary with **
def greet(name, age):
    print(f"Hello {name}, you are {age}")

person = {'name': 'Alice', 'age': 25}
greet(**person)  # Same as greet(name='Alice', age=25)
```

---

7. Scope and Lifetime of Variables

Local Scope

Variables created inside a function exist only within that function.

PYTHON

```python
def my_function():
    x = 10  # Local variable
    print(x)

my_function()  # Output: 10
print(x)  # NameError: name 'x' is not defined
```

Global Scope

Variables created outside functions are global.

PYTHON

```python
x = 10  # Global variable

def my_function():
    print(x)  # Can read global variable

my_function()  # Output: 10
print(x)  # Output: 10
```

Modifying Global Variables

```python
x = 10

def modify_wrong():
    x = 20  # This creates a NEW local variable!
    print(f"Inside: {x}")

def modify_correct():
    global x  # Now we're referring to the global x
    x = 20
    print(f"Inside: {x}")

modify_wrong()
print(f"Outside: {x}")  # Output: 10 (unchanged)

modify_correct()
print(f"Outside: {x}")  # Output: 20 (changed)
```

Best Practice: Avoid using `global`. Instead, pass values as parameters and return new values.

---

8. Best Practices

8.1 Naming Conventions

```python
# Good function names - descriptive and verb-based
def calculate_total_price():
    pass

def get_user_data():
    pass

def validate_email():
    pass

# Bad function names
def do_stuff():  # Too vague
    pass

def x():  # Not descriptive
    pass
```

8.2 Single Responsibility Principle

Each function should do ONE thing well.

```python
# BAD - doing too much
def process_user(name, email):
    # Validate email
    if '@' not in email:
        return False
    # Save to database
    save_to_db(name, email)
    # Send welcome email
    send_email(email)
    # Log activity
    log_action(name)

# GOOD - separate concerns
def validate_email(email):
    return '@' in email

def save_user(name, email):
    save_to_db(name, email)

def send_welcome_email(email):
    send_email(email)

def log_user_action(name):
    log_action(name)
```

8.3 Docstrings

Always document your functions!

```python
def calculate_circle_area(radius):
    """
    Calculate the area of a circle.

    Parameters:
        radius (float): The radius of the circle

    Returns:
        float: The area of the circle

    Example:
        >>> calculate_circle_area(5)
        78.53981633974483
    """
    return 3.14159 * radius ** 2
```

8.4 Keep Functions Short

If a function is too long, break it into smaller functions.

PYTHON

```python
# Instead of one long function
def process_order():
    # 100 lines of code ...
    pass

# Break it down
def validate_order():
    pass

def calculate_total():
    pass

def apply_discount():
    pass

def process_payment():
    pass

def send_confirmation():
    pass

def process_order():
    validate_order()
    total = calculate_total()
    total = apply_discount(total)
    process_payment(total)
    send_confirmation()
```

Summary Cheat Sheet

```python
# Basic function
def greet():
    print("Hello!")

# With parameters and return
def add(a, b):
    return a + b

# Default parameters
def power(base, exponent=2):
    return base ** exponent

# *args for variable positional arguments
def sum_all(*args):
    return sum(args)

# **kwargs for variable keyword arguments
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Everything combined
def complete(a, b=10, *args, c=20, **kwargs):
    pass

# Calling: complete(1, 2, 3, 4, c=30, x=100, y=200)
# a=1, b=2, args=(3,4), c=30, kwargs={'x':100, 'y':200}
```

This comprehensive guide should give you a solid understanding of Python functions from basics to advanced concepts. Practice writing your own functions to reinforce these concepts!