Functions and Modules in Python

Key Concepts Reference Table

Term	Definition
Function	Named block of reusable code that performs a specific task when called
Module	Python file containing functions, classes, and variables that can be imported
Parameter	Variable listed in function definition that accepts input values
Argument	Actual value passed to a function when it is called
Scope	Region of code where a variable can be accessed (local vs global)
Return Statement	Keyword that sends a result back from function to caller
Lambda Function	Anonymous function defined using lambda keyword for simple operations
Import Statement	Command used to bring modules or specific functions into current namespace
Built-in Function	Pre-defined Python functions like len(), print(), type()

SECTION 1: INTRODUCTION TO FUNCTIONS

What Are Functions?

Definition

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

Real-World Analogy

Functions are like kitchen appliances:

- **Blender**: You put in ingredients (input), it processes them, and gives you a smoothie (output)
- **Microwave**: You put in food (input), set the time, and get heated food (output)
- **Coffee Maker**: You add water and coffee (inputs), it processes, and gives you coffee (output)

Each appliance has a specific job, can be used multiple times, and you don't need to know how it works internally - just how to use it!

Why Do We Need Functions?

Without Functions (Repetitive Code):

```
# Calculate area of rectangle 1
length1 = 10
width1 = 5
area1 = length1 * width1
print(f"Area 1: {area1}")

# Calculate area of rectangle 2
length2 = 8
width2 = 6
area2 = length2 * width2
print(f"Area 2: {area2}")

# Calculate area of rectangle 3
length3 = 12
width3 = 4
area3 = length3 * width3
print(f"Area 3: {area3}")

# Repetitive and hard to maintain!
```

With Functions (Clean and Reusable):

```
def calculate_area(length, width):
    """Calculate area of a rectangle"""
    return length * width

# Use the function multiple times
area1 = calculate_area(10, 5)
area2 = calculate_area(8, 6)
area3 = calculate_area(12, 4)

print(f"Area 1: {area1}")
print(f"Area 2: {area2}")
print(f"Area 3: {area3}")

# Clean, maintainable, and reusable!
```

Benefits of Functions

1. **Code Reusability** - Write once, use many times
2. **Organization** - Break complex problems into smaller pieces
3. **Maintainability** - Fix bugs in one place
4. **Readability** - Code is easier to understand
5. **Testing** - Test individual components separately
6. **Collaboration** - Team members can work on different functions

---

Built-in vs User-Defined Functions

Built-in Functions (Already Available in Python)

```
# Python provides many functions out of the box

# len() - get length
numbers = [1, 2, 3, 4, 5]
print(len(numbers))  # Output: 5

# type() - check data type
print(type(10))  # Output: <class 'int'>

# max() and min() - find maximum and minimum
print(max(numbers))  # Output: 5
print(min(numbers))  # Output: 1

# sum() - calculate sum
print(sum(numbers))  # Output: 15

# sorted() - sort items
unsorted = [5, 2, 8, 1, 9]
print(sorted(unsorted))  # Output: [1, 2, 5, 8, 9]

# input() - get user input
# name = input("Enter your name: ")

# print() - display output
print("Hello, World!")
```

User-Defined Functions (Functions You Create)

```
# You create these based on your needs

def greet_user(name):
    """Function to greet a user"""
    print(f"Hello, {name}! Welcome to Python!")

def calculate_total(price, quantity):
    """Calculate total cost"""
    return price * quantity

def is_even(number):
    """Check if a number is even"""
    return number % 2 == 0
```

Basic Function Syntax

Function Structure

```
def function_name(parameters):
    """Docstring: describes what function does"""
    # Function body - code that executes
    # Indentation is mandatory
    return result  # Optional
```

Components:

1. **`def`** - Keyword to define a function
2. **`function_name`** - Descriptive name (use snake_case)
3. **`parameters`** - Input variables (optional)
4. **`:`** - Colon starts the function body
5. **Docstring** - Documentation (optional but recommended)
6. **Function body** - Indented code that executes
7. **`return`** - Sends result back (optional)

Example 1: Function Without Parameters or Return

```
def greet():
    """Simple greeting function"""
    print("Hello, Data Engineer!")
    print("Welcome to Python Functions!")

# Call the function
greet()

# Output:
# Hello, Data Engineer!
# Welcome to Python Functions!
```

Example 2: Function With Parameters

```
def greet_person(name):
    """Greet a specific person"""
    print(f"Hello, {name}!")
    print("Nice to meet you!")

# Call with argument
greet_person("Ada")
greet_person("John")

# Output:
# Hello, Ada!
# Nice to meet you!
# Hello, John!
# Nice to meet you!
```

Example 3: Function With Return Value

```
def add_numbers(a, b):
    """Add two numbers and return the result"""
    total = a + b
```

```
    return total

# Capture returned value
result = add_numbers(5, 3)
print(f"5 + 3 = {result}")

# Use directly in expressions
print(f"10 + 20 = {add_numbers(10, 20)}")

# Output:
# 5 + 3 = 8
# 10 + 20 = 30
```

Example 4: Function With Multiple Parameters

```
def calculate_revenue(price, quantity, tax_rate):
    """Calculate total revenue with tax"""
    subtotal = price * quantity
    tax = subtotal * tax_rate
    total = subtotal + tax
    return total

# Call with multiple arguments
revenue = calculate_revenue(25.99, 100, 0.08)
print(f"Total revenue: ${revenue:.2f}")

# Output:
# Total revenue: $2806.92
```

Calling Functions

Basic Function Call

```
def say_hello():
    print("Hello!")

# Call the function - MUST include parentheses
say_hello()  # ✓ Correct - executes the function

# Common mistake - forgetting parentheses
print(say_hello)  # ✗ Prints function object, doesn't execute
# Output: <function say_hello at 0x...>
```

Function Call with Arguments

```
def multiply(x, y):
    return x * y

# Positional arguments - order matters
result = multiply(5, 3)  # x=5, y=3
print(result)  # Output: 15
```

```
# Wrong order gives different result
result = multiply(3, 5)  # x=3, y=5
print(result)  # Output: 15 (same in this case, but order matters!)

# Example where order matters
def divide(a, b):
    return a / b

print(divide(10, 2))  # Output: 5.0
print(divide(2, 10))  # Output: 0.2 (different!)
```

Storing and Reusing Function Results

```
def calculate_discount(price, discount_percent):
    """Calculate discounted price"""
    discount = price * (discount_percent / 100)
    final_price = price - discount
    return final_price

# Store result in variable
original_price = 100
discounted_price = calculate_discount(original_price, 20)

print(f"Original: ${original_price}")
print(f"Discounted: ${discounted_price}")

# Use result in calculations
savings = original_price - discounted_price
print(f"You saved: ${savings}")

# Output:
# Original: $100
# Discounted: $80.0
# You saved: $20.0
```

SECTION 2: PARAMETERS, ARGUMENTS, AND RETURN VALUES

Understanding Parameters vs Arguments

The Difference

```
def greet(name):  # 'name' is a PARAMETER
    print(f"Hello, {name}!")

greet("Alice")  # "Alice" is an ARGUMENT
```

Parameter = Variable in function definition (placeholder)
Argument = Actual value passed when calling function (concrete value)

Terminology Analogy

Think of a parameter as a **cup holder** in a car (the slot), and an argument as the **actual cup** you put in it.

Types of Parameters

1. Required Parameters

```
def create_profile(username, email):
    """Both parameters are required"""
    print(f"Username: {username}")
    print(f"Email: {email}")

# Must provide both arguments
create_profile("john_doe", "john@example.com")  # ✓ Works

# Missing argument causes error
# create_profile("john_doe")  # ✗ TypeError: missing 1 required argument
```

2. Default Parameters

Default parameters have preset values that are used when no argument is provided.

```
def process_data(data, format_type="csv"):
    """Process data with default format"""
    print(f"Processing {data} in {format_type} format")

# Use default format
process_data("sales_data")
# Output: Processing sales_data in csv format

# Override default
process_data("user_data", "json")
# Output: Processing user_data in json format

process_data("logs", "xml")
# Output: Processing logs in xml format
```

More Examples with Default Parameters

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

```
greet("Alice")              # Hello, Alice!
greet("Bob", "Hi")          # Hi, Bob!
greet("Charlie", "Good morning")  # Good morning, Charlie!
def calculate_total(price, quantity=1, tax_rate=0.0):
    """Calculate total with optional quantity and tax"""
    subtotal = price * quantity
    tax = subtotal * tax_rate
    total = subtotal + tax
    return total

print(calculate_total(100))              # 100.0 (1 item, no tax)
print(calculate_total(100, 3))           # 300.0 (3 items, no tax)
print(calculate_total(100, 3, 0.08))     # 324.0 (3 items, 8% tax)
```

Important Rule: Default Parameters Must Come Last

```
# ✗ WRONG - Default parameter before required parameter
def bad_function(name="Guest", age):
    print(f"{name} is {age} years old")
# SyntaxError!

# ✓ CORRECT - Default parameters after required ones
def good_function(age, name="Guest"):
    print(f"{name} is {age} years old")

good_function(25)              # Guest is 25 years old
good_function(30, "Alice")     # Alice is 30 years old
```

Return Statements

Functions Without Return (Return None Implicitly)

```
def print_message(text):
    """This function doesn't return anything"""
    print(text)

result = print_message("Hello")
print(f"Result: {result}")

# Output:
# Hello
# Result: None
```

Functions With Return

```
def calculate_sum(a, b):
    """This function returns a value"""
    total = a + b
    return total

result = calculate_sum(10, 5)
```

```
print(f"Sum: {result}")

# Output:
# Sum: 15
```

Return vs Print

```
# Using PRINT (displays but doesn't return)
def bad_calculator(x, y):
    total = x + y
    print(total)  # Only displays

result1 = bad_calculator(5, 3)
# Output: 8 (printed)
# result1 = None (nothing returned)

# Using RETURN (sends value back)
def good_calculator(x, y):
    total = x + y
    return total  # Returns value

result2 = good_calculator(5, 3)
# No output (nothing printed)
# result2 = 8 (value returned)

# Now we can use result2 in calculations
doubled = result2 * 2
print(f"Doubled: {doubled}")  # Output: Doubled: 16
```

Multiple Return Values (Using Tuples)

```
def calculate_stats(numbers):
    """Calculate min, max, and average"""
    minimum = min(numbers)
    maximum = max(numbers)
    average = sum(numbers) / len(numbers)


    return minimum, maximum, average  # Returns tuple

# Unpack returned values
data = [10, 20, 30, 40, 50]
min_val, max_val, avg_val = calculate_stats(data)

print(f"Minimum: {min_val}")
print(f"Maximum: {max_val}")
print(f"Average: {avg_val}")

# Output:
# Minimum: 10
# Maximum: 50
# Average: 30.0
```

Early Return (Exit Function Before End)

```
def validate_age(age):
    """Check if age is valid"""
    if age < 0:
        return "Error: Age cannot be negative"

    if age > 120:
        return "Error: Age seems unrealistic"

    return "Age is valid"

print(validate_age(-5))   # Error: Age cannot be negative
print(validate_age(150))  # Error: Age seems unrealistic
print(validate_age(25))   # Age is valid
```

Practical Examples

Example 1: Data Cleaning Function

```
def clean_text(text):
    """Remove extra whitespace and convert to title case"""
    cleaned = text.strip()  # Remove leading/trailing spaces
    cleaned = cleaned.title()  # Convert to title case
    return cleaned

# Test the function
messy_name = "  john doe  "
clean_name = clean_text(messy_name)
print(f"Original: '{messy_name}'")
print(f"Cleaned: '{clean_name}'")

# Output:
# Original: '  john doe  '
# Cleaned: 'John Doe'
```

SECTION 3: VARIABLE SCOPE AND ADVANCED PARAMETERS

Understanding Variable Scope

What is Scope?

Scope determines where in your code a variable can be accessed. Think of it like room access in a building:

- **Local scope** = Variables inside a specific room (function)

- **Global scope** = Variables in the hallway (accessible everywhere)

Local Scope

Variables created inside a function exist only within that function.

```
def calculate_total():
    # Local variables - exist only in this function
    price = 100
    quantity = 5
    total = price * quantity
    print(f"Inside function: total = {total}")

calculate_total()
# Output: Inside function: total = 500

# Try to access local variable outside
# print(total)  # ✗ NameError: name 'total' is not defined
```

Global Scope

Variables created outside functions are accessible everywhere.

```
# Global variable
company_name = "Tech Corp"

def print_company():
    # Can access global variable
    print(f"Company: {company_name}")

def print_welcome():
    # Can also access global variable
    print(f"Welcome to {company_name}!")

print_company()   # Output: Company: Tech Corp
print_welcome()   # Output: Welcome to Tech Corp!
print(company_name)  # Output: Tech Corp
```

Scope Example: Local vs Global

```
# Global variable
counter = 0

def show_counter():
    # Can READ global variable
    print(f"Counter: {counter}")

show_counter()  # Output: Counter: 0

def try_to_change():
    # This creates a NEW local variable
    counter = 10  # Local variable, not global
    print(f"Inside function: {counter}")
```

```
try_to_change()  # Output: Inside function: 10
print(f"Outside function: {counter}")  # Output: Outside function: 0
```

The `global` Keyword

Use `global` keyword to modify global variables inside functions.

```
# Global variable
total_sales = 0

def add_sale(amount):
    global total_sales  # Declare we're using global variable
    total_sales += amount
    print(f"Sale added: ${amount}")
    print(f"Total sales: ${total_sales}")

add_sale(100)
add_sale(200)
add_sale(150)

print(f"\nFinal total sales: ${total_sales}")

# Output:
# Sale added: $100
# Total sales: $100
# Sale added: $200
# Total sales: $300
# Sale added: $150
# Total sales: $450
#
# Final total sales: $450
```

Best Practice: Avoid Global Variables

```
# ❌ BAD - Using global variable
score = 0

def add_points(points):
    global score
    score += points

# ✅ GOOD - Use parameters and return values
def add_points_better(current_score, points):
    new_score = current_score + points
    return new_score

# Better approach
score = 0
score = add_points_better(score, 10)
score = add_points_better(score, 5)
print(f"Score: {score}")  # Output: Score: 15
```

Advanced Parameter Handling

Variable-Length Arguments (*args)

`*args` allows functions to accept any number of positional arguments.

```
def calculate_total(*values):
    """Sum any number of values"""
    total = sum(values)
    return total

# Can pass any number of arguments
print(calculate_total(10))                    # 10
print(calculate_total(10, 20))                # 30
print(calculate_total(10, 20, 30))            # 60
print(calculate_total(10, 20, 30, 40, 50))    # 150
```

**How args Works:*

```
def show_args(*args):
    """Display all arguments"""
    print(f"Type of args: {type(args)}")
    print(f"Arguments: {args}")

    for i, arg in enumerate(args):
        print(f"  Argument {i}: {arg}")

show_args(1, 2, 3, "hello", True)

# Output:
# Type of args: <class 'tuple'>
# Arguments: (1, 2, 3, 'hello', True)
#   Argument 0: 1
#   Argument 1: 2
#   Argument 2: 3
#   Argument 3: hello
#   Argument 4: True
```

Keyword Arguments (**kwargs)

`**kwargs` allows functions to accept any number of keyword arguments.

```
def create_user_profile(**details):
    """Create user profile with any number of details"""
    profile = {}
    for key, value in details.items():
        profile[key] = value
    return profile

# Can pass any number of keyword arguments
user1 = create_user_profile(name="John", age=28, city="Lagos")
print(user1)
# Output: {'name': 'John', 'age': 28, 'city': 'Lagos'}
```

```
user2 = create_user_profile(
    name="Alice",
    age=25,
    role="Data Engineer",
    location="Abuja",
    experience=3
)
print(user2)
# Output: {'name': 'Alice', 'age': 25, 'role': 'Data Engineer', 'location':
'Abuja', 'experience': 3}
```

How **kwargs Works:

```
def show_kwargs(**kwargs):
    """Display all keyword arguments"""
    print(f"Type of kwargs: {type(kwargs)}")
    print(f"Keywords: {kwargs}")

    for key, value in kwargs.items():
        print(f"  {key} = {value}")

show_kwargs(name="Alice", age=30, city="Lagos", role="Engineer")

# Output:
# Type of kwargs: <class 'dict'>
# Keywords: {'name': 'Alice', 'age': 30, 'city': 'Lagos', 'role': 'Engineer'}
#   name = Alice
#   age = 30
#   city = Lagos
#   role = Engineer
```

Combining Different Parameter Types

Order must be: regular parameters → *args → keyword parameters → **kwargs

```
def flexible_function(required, *args, default="hello", **kwargs):
    """Demonstrates all parameter types"""
    print(f"Required: {required}")
    print(f"Args: {args}")
    print(f"Default: {default}")
    print(f"Kwargs: {kwargs}")

flexible_function(
    "must_provide",
    1, 2, 3,
    default="custom",
    extra1="value1",
    extra2="value2"
)

# Output:
# Required: must_provide
# Args: (1, 2, 3)
# Default: custom
```

```
# Kwargs: {'extra1': 'value1', 'extra2': 'value2'}
```

Practical Example: Flexible Data Logger

```
def log_data(level, *messages, **metadata):
    """Flexible logging function"""
    print(f"[{level.upper()}]", end=" ")

    # Print all messages
    for msg in messages:
        print(msg, end=" ")
    print()  # New line

    # Print metadata
    if metadata:
        print("  Metadata:", metadata)

# Use with different numbers of arguments
log_data("info", "User logged in")
log_data("warning", "Low disk space", "Only 10GB remaining")
log_data("error", "Database connection failed", user="admin", attempt=3)

# Output:
# [INFO] User logged in
# [WARNING] Low disk space Only 10GB remaining
# [ERROR] Database connection failed
#   Metadata: {'user': 'admin', 'attempt': 3}
```

SECTION 4: MODULES AND CODE ORGANIZATION

What Are Modules?

A **module** is a Python file containing functions, variables, and classes that can be imported and reused in other programs.

Why Use Modules?

1. **Organization** - Group related functions together
2. **Reusability** - Use same code across multiple projects
3. **Maintainability** - Update code in one place
4. **Collaboration** - Share code with team members
5. **Namespace Management** - Avoid naming conflicts

Creating a Module

Step 1: Create a Python File

Create a file named `data_utils.py`:

```
"""
File: data_utils.py
Data engineering utility functions
"""

def clean_text(text):
    """Remove extra whitespace and convert to title case"""
    return text.strip().title()

def validate_email(email):
    """Basic email validation"""
    return "@" in email and "." in email

def calculate_percentage(part, total):
    """Calculate percentage with error handling"""
    if total == 0:
        return 0
    return round((part / total) * 100, 2)

def convert_to_uppercase(text):
    """Convert text to uppercase"""
    return text.upper()

def calculate_average(numbers):
    """Calculate average of a list of numbers"""
    if len(numbers) == 0:
        return 0
    return sum(numbers) / len(numbers)

# Module-level variable
MODULE_VERSION = "1.0.0"
```

Step 2: Use the Module

Create another file in the same directory:

```
# main.py
import data_utils

# Use module functions
name = "  john doe  "
cleaned = data_utils.clean_text(name)
print(f"Cleaned name: {cleaned}")

email = "user@example.com"
is_valid = data_utils.validate_email(email)
print(f"Valid email: {is_valid}")

percentage = data_utils.calculate_percentage(75, 100)
print(f"Percentage: {percentage}%")
```

```
# Access module variable
print(f"Module version: {data_utils.MODULE_VERSION}")

# Output:
# Cleaned name: John Doe
# Valid email: True
# Percentage: 75.0%
# Module version: 1.0.0
```

Different Ways to Import

1. Import Entire Module

```
import data_utils

# Use with module name prefix
result = data_utils.clean_text("hello")
```

Pros: Clear where functions come from
Cons: More typing required

2. Import Specific Functions

```
from data_utils import clean_text, validate_email

# Use directly without prefix
result = clean_text("hello")
is_valid = validate_email("test@email.com")
```

Pros: Less typing
Cons: Can cause naming conflicts

3. Import with Alias

```
import data_utils as du

# Use shorter alias
result = du.clean_text("hello")
```

Pros: Shorter code, clear origin
Cons: Need to remember alias

4. Import Everything (Not Recommended)

```
from data_utils import *

# Use all functions without prefix
result = clean_text("hello")
```

```
average = calculate_average([1, 2, 3])
```

Pros: Very little typing
Cons:

- Unclear where functions come from
- Can overwrite existing functions
- Makes debugging harder
- Not recommended in production code

Module Best Practices

1. Use Docstrings

```
"""
Module: user_management.py
Description: Functions for managing user accounts
Author: Your Name
Date: 2024-01-15
"""

def create_user(username, email):
    """
    Create a new user account.

    Args:
        username (str): Unique username
        email (str): User's email address

    Returns:
        dict: User profile dictionary
    """
    return {"username": username, "email": email}
```

2. Organize Related Functions

```
# validators.py - All validation functions
def validate_email(email):
    pass

def validate_phone(phone):
    pass

def validate_age(age):
    pass


# transformers.py - All data transformation functions
def clean_text(text):
    pass
```

```
def normalize_date(date_string):
    pass

def format_currency(amount):
    pass
```

SECTION 5: LAMBDA FUNCTIONS

What Are Lambda Functions?

Lambda functions are small, anonymous functions defined in a single line. They're called "anonymous" because they don't need a name.

Regular Function vs Lambda

```
# Regular function
def square(x):
    return x ** 2

print(square(5))  # Output: 25

# Lambda function (equivalent)
square_lambda = lambda x: x ** 2

print(square_lambda(5))  # Output: 25
```

Lambda Syntax

```
lambda parameters: expression
```

Components:

- `lambda` - Keyword to define lambda function
- `parameters` - Input variables (can be multiple)
- `:` - Separates parameters from expression
- `expression` - Single expression that returns a value

Lambda Function Examples

Example 1: Simple Arithmetic

```
# Add two numbers
add = lambda a, b: a + b
```

```
print(add(10, 5))  # Output: 15

# Multiply
multiply = lambda x, y: x * y
print(multiply(4, 3))  # Output: 12

# Check if even
is_even = lambda n: n % 2 == 0
print(is_even(10))  # Output: True
print(is_even(7))   # Output: False
```

Example 2: String Operations

```
# Convert to uppercase
to_upper = lambda text: text.upper()
print(to_upper("hello"))  # Output: HELLO

# Get string length
string_length = lambda s: len(s)
print(string_length("Python"))  # Output: 6

# Reverse string
reverse = lambda s: s[::-1]
print(reverse("hello"))  # Output: olleh
```

When to Use Lambda vs Regular Functions

Use Lambda When:

- ✅ Function is simple (one line)
- ✅ Used only once
- ✅ Used with `filter()`, `map()`, `sorted()`
- ✅ Quick throwaway function needed

```
# Good use of lambda
sorted_data = sorted(items, key=lambda x: x['value'])
```

Use Regular Function When:

- ✅ Function is complex (multiple lines)
- ✅ Function is reused multiple times
- ✅ Function needs documentation
- ✅ Function has multiple steps

```
# Better as regular function
def process_data(data):
    """Process and validate data"""
    cleaned = data.strip()
    validated = validate(cleaned)
```

```
transformed = transform(validated)
return transformed
```