100 Core Python Interview Questions

1. What are the *key features* of *Python*?

**Python** is a versatile and popular programming language known for its simplicity, **elegant syntax**, and a vast ecosystem of libraries. Let's look at some of the key features that make Python stand out.

Key Features of Python

1. Interpreted and Interactive

Python uses an interpreter, allowing developers to run code **line-by-line**, making it ideal for rapid prototyping and debugging.

2. Easy to Learn and Read

Python's **clean, readable syntax**, often resembling plain English, reduces the cognitive load for beginners and experienced developers alike.

3. Cross-Platform Compatibility

Python is versatile, running on various platforms, such as Windows, Linux, and macOS, without requiring platform-specific modifications.

4. Modular and Scalable

Developers can organize their code into modular packages and reusabale functions.

5. Rich Library Ecosystem

The Python Package Index (PyPI) hosts over 260,000 libraries, providing solutions for tasks ranging from web development to data analytics.

6. Exceptionally Versatile

From web applications to scientific computing, Python is equally proficient in diverse domains.

7. Memory Management

Python seamlessly allocates and manages memory, shielding developers from low-level tasks, such as memory deallocation.

8. Dynamically Typed

Python infers the data type of a variable during execution, easing the declartion and manipulation of variables.

9. Object-Oriented

Python supports object-oriented paradigms, where everything is an **object**, offering attributes and methods to manipulate data.

10. Extensible

With its C-language API, developers can integrate performance-critical tasks and existing C modules with Python.

2. How is *Python* executed?

**Python** source code is processed through various steps before it can be executed. Let's explore the key stages in this process.

Compilation & Interpretation

Python code goes through both **compilation** and **interpretation**.

* **Bytecode Compilation**: High-level Python code is transformed into low-level bytecode by the Python interpreter with the help of a compiler. Bytecode is a set of instructions that Python's virtual machine (PVM) can understand and execute.
* **On-the-fly Interpretation**: The PVM reads and executes bytecode instructions in a step-by-step manner.

This dual approach known as "compile and then interpret" is what sets Python (and certain other languages) apart.

Bytecode versus Machine Code Execution

While some programming languages compile directly to machine code, Python compiles to bytecode. This bytecode is then executed by the Python virtual machine. This extra step of bytecode execution **can make Python slower** in certain use-cases when compared to languages that compile directly to machine code.

The advantage, however, is that bytecode is platform-independent. A Python program can be run on any machine with a compatible PVM, ensuring cross-platform support.

Source Code to Bytecode: Compilation Steps

1. **Lexical Analysis**: The source code is broken down into tokens, identifying characters and symbols for Python to understand.
2. **Syntax Parsing**: Tokens are structured into a parse tree to establish the code's syntax and grammar.
3. **Semantic Analysis**: Code is analyzed for its meaning and context, ensuring it's logically sound.
4. **Bytecode Generation**: Based on the previous steps, bytecode instructions are created.

Just-In-Time (JIT) Compilation

While Python typically uses a combination of interpretation and compilation, **JIT** boosts efficiency by selectively compiling parts of the program that are frequently used or could benefit from optimization.

JIT compiles sections of the program to machine code on-the-fly. This direct machine code generation for frequently executed parts can significantly speed up those segments, blurring the line between traditional interpreters and compilers.

Code Example: Disassembly of Bytecode

import dis  
  
def example\_func():  
    return 15 \* 20  
  
# Disassemble to view bytecode instructions  
dis.dis(example\_func)

Disassembling code using Python's dis module can reveal the underlying bytecode instructions that the PVM executes. Here's the disassembled output for the above code:

  4           0 LOAD\_CONST               2 (300)  
              2 RETURN\_VALUE

3. What is *PEP 8* and why is it important?

**PEP 8** is a style guide for Python code that promotes code consistency, readability, and maintainability. It's named after Python Enhancement Proposal (PEP), the mechanism used to propose and standardize changes to the Python language.

PEP 8 is not a set-in-stone rule book, but it provides general guidelines that help developers across the Python community write code that's visually consistent and thus easier to understand.

Key Design Principles

PEP 8 emphasizes:

* **Readability**: Code should be easy to read and understand, even by someone who didn't write it.
* **Consistency**: Codebase should adhere to a predictable style so there's little cognitive load in reading or making changes.
* **One Way to Do It**: Instead of offering multiple ways to write the same construct, PEP 8 advocates for a single, idiomatic style.

Base Rules

* **Indentation**: Use 4 spaces for each level of logical indentation.
* **Line Length**: Keep lines of code limited to 79 characters. This number is a guideline; longer lines are acceptable in certain contexts.
* **Blank Lines**: Use them to separate logical sections but not excessively.

Naming Styles

* **Class Names**: Prefer CamelCase.
* **Function and Variable Names**: Use lowercase\_with\_underscores.
* **Module Names**: Keep them short and in lowercase.

Documentation

* Use triple quotes for documentation strings.
* Comments should be on their own line and explain the reason for the following code block.

Whitespace Usage

* **Operators**: Surround them with a single space.
* **Commas**: Follow them with a space.

Example: Directory Walker

Here is the PEP8 compliant code:

import os  
  
def walk\_directory(path):  
    for dirpath, dirnames, filenames in os.walk(path):  
        for filename in filenames:  
            file\_path = os.path.join(dirpath, filename)  
            print(file\_path)  
  
walk\_directory('[/path/to/directory](https://colab.research.google.com/drive/16naJzp-CCOsHIP2lVkihsxxyLMzL2uhy)')

4. How is memory allocation and garbage collection handled in *Python*?

In Python, **both memory allocation** and **garbage collection** are handled discretely.

Memory Allocation

* The "heap" is the pool of memory for storing objects. The Python memory manager allocates and deallocates this space as needed.
* In latest Python versions, the obmalloc system is responsible for small object allocations. This system preallocates small and medium-sized memory blocks to manage frequently created small objects.
* The allocator abstracts the system-level memory management, employing memory management libraries like Glibc to interact with the operating system.
* Larger blocks of memory are primarily obtained directly from the operating system.
* **Stack** and **Heap** separation is joined by "Pool Allocator" for internal use.

Garbage Collection

Python employs a method called **reference counting** along with a **cycle-detecting garbage collector**.

Reference Counting

* Every object has a reference count. When an object's count drops to zero, it is immediately deallocated.
* This mechanism is swift, often releasing objects instantly without the need for garbage collection.
* However, it can be insufficient in handling **circular references**.

Cycle-Detecting Garbage Collector

* Python has a separate garbage collector that periodically identifies and deals with circular references.
* This is, however, a more time-consuming process and is invoked less frequently than reference counting.

Memory Management in Python vs. C

Python handles memory management quite differently from languages like C or C++:

* In Python, the developer isn't directly responsible for memory allocations or deallocations, reducing the likelihood of memory-related bugs.
* The memory manager in Python is what's known as a **"general-purpose memory manager"** that can be slower than the dedicated memory managers of C or C++ in certain contexts.
* Python, especially due to the existence of a garbage collector, might have memory overhead compared to C or C++ where manual memory management often results in minimal overhead is one of the factors that might contribute to Python's sometimes slower performance.
* The level of memory efficiency isn't as high as that of C or C++. This is because Python is designed to be convenient and easy to use, often at the expense of some performance optimization.

5. What are the *built-in data types* in *Python*?

Python offers numerous **built-in data types** that provide varying functionalities and utilities.

Immutable Data Types

1. int

Represents a whole number, such as 42 or -10.

2. float

Represents a decimal number, like 3.14 or -0.01.

3. complex

Comprises a real and an imaginary part, like 3 + 4j.

4. bool

Represents a boolean value, True or False.

5. str

A sequence of unicode characters enclosed within quotes.

6. tuple

An ordered collection of items, often heterogeneous, enclosed within parentheses.

7. frozenset

A set of unique, immutable objects, similar to sets, enclosed within curly braces.

8. bytes

Represents a group of 8-bit bytes, often used with binary data, enclosed within brackets.

9. bytearray

Resembles the 'bytes' type but allows mutable changes.

10. NoneType

Indicates the absence of a value.

Mutable Data Types

1. list

A versatile ordered collection that can contain different data types and offers dynamic sizing, enclosed within square brackets.

2. set

Represents a unique set of objects and is characterized by curly braces.

3. dict

A versatile key-value paired collection enclosed within braces.

4. memoryview

Points to the memory used by another object, aiding efficient viewing and manipulation of data.

5. array

Offers storage for a specified type of data, similar to lists but with dedicated built-in functionalities.

6. deque

A double-ended queue distinguished by optimized insertion and removal operations from both its ends.

7. object

The base object from which all classes inherit.

8. types.SimpleNamespace

Grants the capability to assign attributes to it.

9. types.ModuleType

Represents a module body containing attributes.

10. types.FunctionType

Defines a particular kind of function.

6. Explain the difference between a *mutable* and *immutable* object.

Let's look at the difference between **mutable** and **immutable** objects.

Key Distinctions

* **Mutable Objects**: Can be modified after creation.
* **Immutable Objects**: Cannot be modified after creation.

Common Examples

* **Mutable**: Lists, Sets, Dictionaries
* **Immutable**: Tuples, Strings, Numbers

Code Example: Immutability in Python

Here is the Python code:

# Immutable objects (int, str, tuple)  
num = 42  
text = "Hello, World!"  
my\_tuple = (1, 2, 3)  
  
# Trying to modify will raise an error  
try:  
    num += 10  
    text[0] = 'M'  # This will raise a TypeError  
    my\_tuple[0] = 100  # This will also raise a TypeError  
except TypeError as e:  
    print(f"Error: {e}")  
  
# Mutable objects (list, set, dict)  
my\_list = [1, 2, 3]  
my\_dict = {'a': 1, 'b': 2}  
  
# Can be modified without issues  
my\_list.append(4)  
del my\_dict['a']  
  
# Checking the changes  
print(my\_list)  # Output: [1, 2, 3, 4]  
print(my\_dict)  # Output: {'b': 2}

Benefits & Trade-Offs

**Immutability** offers benefits such as **safety** in concurrent environments and facilitating **predictable behavior**.

**Mutability**, on the other hand, often improves **performance** by avoiding copy overhead and redundant computations.

Impact on Operations

* **Reading and Writing**: Immutable objects typically favor **reading** over **writing**, promoting a more straightforward and predictable code flow.
* **Memory and Performance**: Mutability can be more efficient in terms of memory usage and performance, especially concerning large datasets, thanks to in-place updates.

Choosing between the two depends on the program's needs, such as the required data integrity and the trade-offs between predictability and performance.

7. How do you *handle exceptions* in *Python*?

**Exception handling** is a fundamental aspect of Python, and it safeguards your code against unexpected errors or conditions. Key components of exception handling in Python include:

Components

* **Try**: The section of code where exceptions might occur is placed within a try block.
* **Except**: Any possible exceptions that are raised by the try block are caught and handled in the except block.
* **Finally**: This block ensures a piece of code always executes, regardless of whether an exception occurred. It's commonly used for cleanup operations, such as closing files or database connections.

Generic Exception Handling vs. Handling Specific Exceptions

It's good practice to **handle** specific exceptions. However, a more **general** approach can also be taken. When doing the latter, ensure the general exception handling is at the end of the chain, as shown here:

try:  
    risky\_operation()  
except IndexError:  # Handle specific exception types first.  
    handle\_index\_error()  
except Exception as e:  # More general exception must come last.  
    handle\_generic\_error()  
finally:  
    cleanup()

Raising Exceptions

Use this mechanism to **trigger and manage** exceptions under specific circumstances. This can be particularly useful when building custom classes or functions where specific conditions should be met.

**Raise** a specific exception:

def divide(a, b):  
    if b == 0:  
        raise ZeroDivisionError("Divisor cannot be zero")  
    return a / b  
  
try:  
    result = divide(4, 0)  
except ZeroDivisionError as e:  
    print(e)

**Raise a general exception**:

def some\_risky\_operation():  
    if condition:   
        raise Exception("Some generic error occurred")

Using with for Resource Management

The with keyword provides a more efficient and clean way to handle resources, like files, ensuring their proper closure when operations are complete or in case of any exceptions. The resource should implement a context manager, typically by having \_\_enter\_\_ and \_\_exit\_\_ methods.

Here's an example using a file:

with open("example.txt", "r") as file:  
    data = file.read()  
# File is automatically closed when the block is exited.

Silence with pass, continue, or else

There are times when not raising an exception is appropriate. You can use pass or continue in an exception block when you want to essentially ignore an exception and proceed with the rest of your code.

* **pass**: Simply does nothing. It acts as a placeholder.

try:  
    risky\_operation()  
except SomeSpecificException:  
    pass

* **continue**: This keyword is generally used in loops. It moves to the next iteration without executing the code that follows it within the block.

for item in my\_list:  
    try:  
        perform\_something(item)  
    except ExceptionType:  
        continue  
    ```

* **else with try-except blocks**: The else block after a try-except block will only be executed if no exceptions are raised within the try block

try:  
    some\_function()  
except SpecificException:  
    handle\_specific\_exception()  
else:  
    no\_exception\_raised()

Callback Function: ExceptionHook

Python 3 introduced the better handling of uncaught exceptions by providing an optional function for printing stack traces. The sys.excepthook can be set to match any exception in the module as long as it has a hook attribute.

Here's an example for this test module:

# test.py  
import sys  
  
def excepthook(type, value, traceback):  
    print("Unhandled exception:", type, value)  
    # Call the default exception hook  
    sys.\_\_excepthook\_\_(type, value, traceback)  
  
sys.excepthook = excepthook  
  
def test\_exception\_hook():  
    throw\_some\_exception()

When run, calling test\_exception\_hook will print "Unhandled exception: ..."

*Note*: sys.excepthook will not capture exceptions raised as the result of interactive prompt commands, such as SyntaxError or KeyboardInterrupt.

8. What is the difference between *list* and *tuple*?

**Lists** and **Tuples** in Python share many similarities, such as being sequences and supporting indexing.

However, these data structures differ in key ways:

Key Distinctions

* **Mutability**: Lists are mutable, allowing you to add, remove, or modify elements after creation. Tuples, once created, are immutable.
* **Performance**: Lists are generally slower than tuples, most apparent in tasks like iteration and function calls.
* **Syntax**: Lists are defined with square brackets [], whereas tuples use parentheses ().

When to Use Each

* **Lists** are ideal for collections that may change in size and content. They are the preferred choice for storing data elements.
* **Tuples**, due to their immutability and enhanced performance, are a good choice for representing fixed sets of related data.

Syntax

List: Example

my\_list = ["apple", "banana", "cherry"]  
my\_list.append("date")  
my\_list[1] = "blackberry"

Tuple: Example

my\_tuple = (1, 2, 3, 4)  
# Unpacking a tuple  
a, b, c, d = my\_tuple

9. How do you create a *dictionary* in *Python*?

**Python dictionaries** are versatile data structures, offering key-based access for rapid lookups. Let's explore various data within dictionaries and techniques to create and manipulate them.

Key Concepts

* A **dictionary** in Python contains a collection of key:value pairs.
* **Keys** must be unique and are typically immutable, such as strings, numbers, or tuples.
* **Values** can be of any type, and they can be duplicated.

Creating a Dictionary

You can use several methods to create a dictionary:

1. **Literal Definition**: Define key-value pairs within curly braces { }.
2. **From Key-Value Pairs**: Use the dict() constructor or the {key: value} shorthand.
3. **Using the dict() Constructor**: This can accept another dictionary, a sequence of key-value pairs, or named arguments.
4. **Comprehensions**: This is a concise way to create dictionaries using a single line of code.
5. **zip() Function**: This creates a dictionary by zipping two lists, where the first list corresponds to the keys, and the second to the values.

Examples

Dictionary Literal Definition

Here is a Python code:

# Dictionary literal definition  
student = {  
    "name": "John Doe",  
    "age": 21,  
    "courses": ["Math", "Physics"]  
}

From Key-Value Pairs

Here is the Python code:

# Using the `dict()` constructor  
student\_dict = dict([  
    ("name", "John Doe"),  
    ("age", 21),  
    ("courses", ["Math", "Physics"])  
])  
  
# Using the shorthand syntax  
student\_dict\_short = {  
    "name": "John Doe",  
    "age": 21,  
    "courses": ["Math", "Physics"]  
}

Using zip()

Here is a Python code:

keys = ["a", "b", "c"]  
values = [1, 2, 3]  
  
zipped = zip(keys, values)  
dict\_from\_zip = dict(zipped) # Result: {"a": 1, "b": 2, "c": 3}

Using dict() Constructor

Here is a Python code:

# Sequence of key-value pairs  
student\_dict2 = dict(name="Jane Doe", age=22, courses=["Biology", "Chemistry"])  
  
# From another dictionary  
student\_dict\_combined = dict(student, \*\*student\_dict2)

10. What is the difference between *==* and *is operator* in *Python*?

Both the **==** and **is** operators in Python are used for comparison, but they function differently.

* The **==** operator checks for **value equality**.
* The **is** operator, on the other hand, validates **object identity**,

In Python, every object is unique, identifiable by its memory address. The **is** operator uses this memory address to check if two objects are the same, indicating they both point to the exact same instance in memory.

* **is**: Compares the memory address or identity of two objects.
* **==**: Compares the content or value of two objects.

While **is** is primarily used for **None** checks, it's generally advisable to use **==** for most other comparisons.

Tips for Using Operators

* **==**: Use for equality comparisons, like when comparing numeric or string values.
* **is**: Use for comparing membership or when dealing with singletons like **None**.

11. How does a *Python function* work?

**Python functions** are the building blocks of code organization, often serving predefined tasks within modules and scripts. They enable reusability, modularity, and encapsulation.

Key Components

* **Function Signature**: Denoted by the def keyword, it includes the function name, parameters, and an optional return type.
* **Function Body**: This section carries the core logic, often comprising conditional checks, loops, and method invocations.
* **Return Statement**: The function's output is determined by this statement. When None is specified, the function returns by default.
* **Local Variables**: These variables are scoped to the function and are only accessible within it.

Execution Process

When a function is called:

1. **Stack Allocation**: A stack frame, also known as an activation record, is created to manage the function's execution. This frame contains details like the function's parameters, local variables, and **instruction pointer**.
2. **Parameter Binding**: The arguments passed during the function call are bound to the respective parameters defined in the function header.
3. **Function Execution**: Control is transferred to the function body. The statements in the body are executed in a sequential manner until the function hits a return statement or the end of the function body.
4. **Return**: If a return statement is encountered, the function evaluates the expression following the return and hands the value back to the caller. The stack frame of the function is then popped from the call stack.
5. **Post Execution**: If there's no return statement, or if the function ends without evaluating any return statement, None is implicitly returned.

Local Variable Scope

* **Function Parameters**: These are a precursor to local variables and are instantiated with the values passed during function invocation.
* **Local Variables**: Created using an assignment statement inside the function and cease to exist when the function execution ends.
* **Nested Scopes**: In functions within functions (closures), non-local variables - those defined in the enclosing function - are accessible but not modifiable by the inner function, without using the nonlocal keyword.

Global Visibility

If a variable is not defined within a function, the Python runtime will look for it in the global scope. This behavior enables functions to access and even modify global variables.

Avoiding Side Effects

Functions offer a level of encapsulation, potentially reducing side effects by ensuring that data and variables are managed within a controlled environment. Such containment can help enhance the robustness and predictability of a codebase. As a best practice, minimizing the reliance on global variables can lead to more maintainable, reusable, and testable code.

12. What is a *lambda function*, and where would you use it?

A **Lambda function**, or **lambda**, for short, is a small anonymous function defined using the lambda keyword in Python.

While you can certainly use named functions when you need a function for something in Python, there are places where a lambda expression is more suitable.

Distinctive Features

* **Anonymity**: Lambdas are not given a name in the traditional sense, making them suited for one-off uses in your codebase.
* **Single Expression Body**: Their body is limited to a single expression. This can be an advantage for brevity but a restriction for larger, more complex functions.
* **Implicit Return**: There's no need for an explicit return statement.
* **Conciseness**: Lambdas streamline the definition of straightforward functions.

Common Use Cases

* **Map, Filter, and Reduce**: Functions like map can take a lambda as a parameter, allowing you to define simple transformations on the fly. For example, doubling each element of a list can be achieved with list(map(lambda x: x\*2, my\_list)).
* **List Comprehensions**: They are a more Pythonic way of running the same map or filter operations, often seen as an alternative to lambdas and map.
* **Sorting**: Lambdas can serve as a custom key function, offering flexibility in sort orders.
* **Callbacks**: Often used in events where a function is needed to be executed when an action occurs (e.g., button click).
* **Simple Functions**: For functions that are so basic that giving them a name, especially in more procedural code, would be overkill.

Notable Limitations

* **Lack of Verbose Readability**: Named functions are generally preferred when their intended use is obvious from the name. Lambdas can make code harder to understand if they're complex or not used in a recognizable pattern.
* **No Formal Documentation**: While the function's purpose should be apparent from its content, a named function makes it easier to provide direct documentation. Lambdas would need a separate verbal explanation, typically in the code or comments.

13. Explain *\*args* and *\*\*kwargs* in *Python*.

In Python, \*args and \*\*kwargs are often used to pass a variable number of arguments to a function.

\*args collects a variable number of positional arguments into a **tuple**, while \*\*kwargs does the same for keyword arguments into a **dictionary**.

Here are the key features, use-cases, and their respective code examples.

**\*args**: Variable Number of Positional Arguments

* **How it Works**: The name \*args is a convention. The asterisk (\*) tells Python to put any remaining positional arguments it receives into a tuple.
* **Use-Case**: When the number of arguments needed is uncertain.

Code Example: "\*args"

def sum\_all(\*args):  
    result = 0  
    for num in args:  
        result += num  
    return result  
  
print(sum\_all(1, 2, 3, 4))  # Output: 10

**\*\*kwargs**: Variable Number of Keyword Arguments

* **How it Works**: The double asterisk (\*\*) is used to capture keyword arguments and their values into a dictionary.
* **Use-Case**: When a function should accept an arbitrary number of keyword arguments.

Code Example: "\*\*kwargs"

def print\_values(\*\*kwargs):  
    for key, value in kwargs.items():  
        print(f"{key}: {value}")  
  
# Keyword arguments are captured as a dictionary  
print\_values(name="John", age=30, city="New York")  
# Output:  
# name: John  
# age: 30  
# city: New York

14. What are *decorators* in *Python*?

In Python, a **decorator** is a design pattern and a feature that allows you to modify functions and methods dynamically. This is done primarily to keep the code clean, maintainable, and DRY (Don't Repeat Yourself).

How Decorators Work

* Decorators wrap a target function, allowing you to execute custom code before and after that function.
* They are typically **higher-order functions** that take a function as an argument and return a new function.
* This paradigm of "functions that modify functions" is often referred to as **metaprogramming**.

Common Use Cases

* **Authorization and Authentication**: Control user access.
* **Logging**: Record function calls and their parameters.
* **Caching**: Store previous function results for quick access.
* **Validation**: Verify input parameters or function output.
* **Task Scheduling**: Execute a function at a specific time or on an event.
* **Counting and Profiling**: Keep track of the number of function calls and their execution time.

Using Decorators in Code

Here is the Python code:

from functools import wraps  
  
# 1. Basic Decorator  
def my\_decorator(func):  
    @wraps(func)  # Ensures the original function's metadata is preserved  
    def wrapper(\*args, \*\*kwargs):  
        print('Something is happening before the function is called.')  
        result = func(\*args, \*\*kwargs)  
        print('Something is happening after the function is called.')  
        return result  
    return wrapper  
  
@my\_decorator  
def say\_hello():  
    print('Hello!')  
  
say\_hello()  
  
# 2. Decorators with Arguments  
def decorator\_with\_args(arg1, arg2):  
    def actual\_decorator(func):  
        @wraps(func)  
        def wrapper(\*args, \*\*kwargs):  
            print(f'Arguments passed to decorator: {arg1}, {arg2}')  
            result = func(\*args, \*\*kwargs)  
            return result  
        return wrapper  
    return actual\_decorator  
  
@decorator\_with\_args('arg1', 'arg2')  
def my\_function():  
    print('I am decorated!')  
  
my\_function()

Decorator Syntax in Python

The @decorator syntax is a convenient shortcut for:

def say\_hello():  
    print('Hello!')  
say\_hello = my\_decorator(say\_hello)

Role of **functools.wraps**

When defining decorators, particularly those that return functions, it is good practice to use @wraps(func) from the functools module. This ensures that the original function's metadata, such as its name and docstring, is preserved.

15. How can you create a *module* in *Python*?

You can **create** a Python module through one of two methods:

* **Define**: Begin with saving a Python file with .py extension. This file will automatically function as a module.
* **Create a Blank Module**: Start an empty file with no extension. Name the file using the accepted module syntax, e.g., \_\_init\_\_, for it to act as a module.

Next, use **import** to access the module and its functionality.

Code Example: Creating a math\_operations Module

Module Definition

Save the below math\_operations.py file :

def add(x, y):  
    return x + y  
  
def subtract(x, y):  
    return x - y  
  
def multiply(x, y):  
    return x \* y  
  
def divide(x, y):  
    return x / y

Module Usage

You can use math\_operations module by using import as shown below:

import math\_operations  
  
result = math\_operations.add(4, 5)  
print(result)  
  
result = math\_operations.divide(10, 5)  
print(result)

Even though it is not required in the later **versions of Python**, you can also use statement from math\_operations import \* to import all the members such as functions and classes at once:

from math\_operations import \*  # Not recommended generally due to name collisions and readability concerns  
  
result = add(3, 2)  
print(result)

Best Practice

Before submitting the code, let's make sure to follow the **Best Practice**:

* **Avoid Global Variables**: Use a main() function.
* **Guard Against Code Execution on Import**: To avoid unintended side effects, use:

if \_\_name\_\_ == "\_\_main\_\_":  
    main()

This makes sure that the block of code following if \_\_name\_\_ == "\_\_main\_\_": is only executed when the module is run directly and not when imported as a module in another program.