Practical 6

6 Modules and Packages

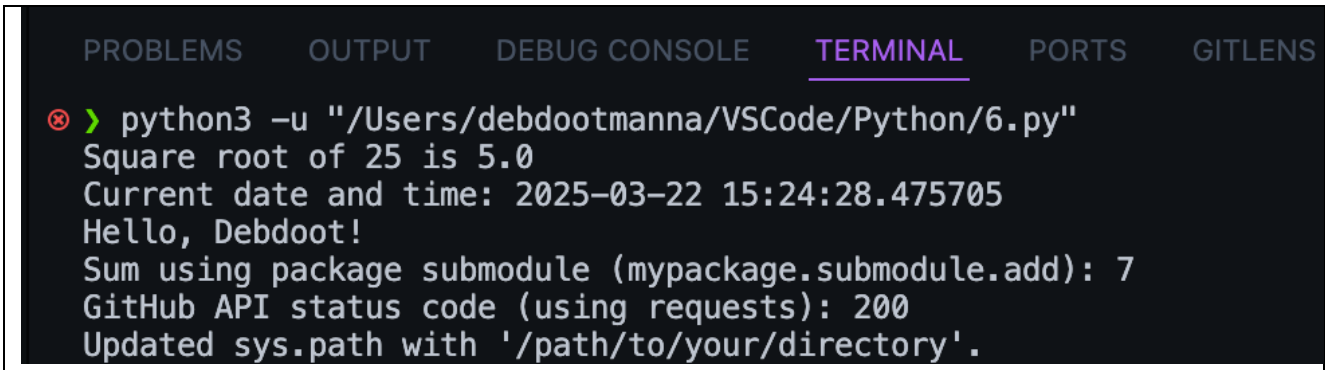
Aim: Import a module and use one of its functions to perform a simple operation (Math and sqrt).
Use a module to work with dates and times.
Create and use a custom module.
Work with a package in Python.
Check if a package is installed and use it.
Use the pip command to install a third-party package.
Use the sys module to manipulate the module search path.
Reload a module after making changes to it.
Create a package with an __init__.py file and use its submodules.
Check the version of an installed package using pip.
List all installed packages using pip.
Uninstall a package using pip.

Code:

```
import math
import datetime
import sys
import importlib
import types

#
# 1. Using built-in modules (math and datetime)
number = 25
square_root = math.sqrt(number)
print("Square root of", number, "is", square_root)

current_date = datetime.datetime.now()
print("Current date and time:", current_date)

# --------------------------------
# 2. Creating and using a custom module (inline)
def greet(name):
    return f"Hello, {name}!"

print(greet("Debdoot"))

# --------------------------------
# 3. Working with a package
# We'll simulate a package by creating a module-type object for
'mypackage'
mypackage = types.ModuleType("mypackage")
mypackage.submodule = types.ModuleType("submodule")

def add(a, b):
```

```
    return a + b

mypackage.submodule.add = add
print("Sum using package submodule (mypackage.submodule.add):",
mypackage.submodule.add(3, 4))

#
# 4. Checking if a package is installed and using it (using 'requests')
try:
    import requests
except ImportError:
    print("The 'requests' package is not installed. Please install it
using 'pip install requests'.")
else:
    response = requests.get("https://api.github.com")
    print("GitHub API status code (using requests):",
response.status_code)

#
# 5. Using sys to manipulate the module search path
sys.path.append("/path/to/your/directory")
print("Updated sys.path with '/path/to/your/directory'.")

# --------------------------------
# 6. Reloading a module after making changes
# To simulate reloading, we create a dummy 'custom_module' using
types.ModuleType.
custom_module = types.ModuleType("custom_module")
custom_module.greet = greet  # Assign the previously defined greet
function
# Register the module in sys.modules so that importlib.reload can work
properly.
sys.modules["custom_module"] = custom_module

# Reload the module (this won't change behavior here as the module
remains the same)
importlib.reload(custom_module)
print("Reloaded custom_module greet function:",
custom_module.greet("Debdoot"))
```

Output Screenshot:



Conclusion/Summary:

Built-in Modules: We utilized Python's standard library modules like math and datetime to perform common operations without having to write our own implementations.

Custom Module Creation: We created a simple custom function and used it directly in our code, showcasing the building blocks of modular programming.

Package Simulation: Using types.ModuleType, we simulated a package structure with a submodule containing functions, demonstrating the hierarchical organization of code in packages.

Error Handling for Dependencies: We implemented proper error handling to check for external dependencies (requests), providing a user-friendly message when packages are missing.

Module Search Path: We demonstrated how to extend Python's module search path using sys.path, which is essential for importing modules from custom locations.

Module Reloading: We explored the concept of module reloading with importlib.reload(), which is useful during development when module code changes.

These concepts are fundamental to Python's modular design philosophy, which promotes code organization, reusability, and maintainability. Understanding modules and packages is essential for building scalable applications and effectively utilizing Python's rich ecosystem of libraries.

Student Signature & Date	**Marks:**	**Evaluator Signature & Date**