Chapter 11.1 – Python Modules

Teacher Edition with Full Notes and Solutions

---

1 Learning Objectives

- Define what a **module** is in Python.
- Explain how to **import** and use modules.
- Demonstrate the **importing process**.
- Distinguish between **scripts** and **modules**.
- Practice writing and importing custom modules.

2 Core Concepts and Vocabulary

Essential Terms

- **Script**: A file ending in `.py` that is executed directly to perform a task.
- **Module**: A Python file that defines reusable code (functions, variables, or classes) that can be imported by another script.
- **Import**: The command that brings a module's definitions into another program.
- **Namespace**: A "labeled space" that keeps identifiers from one module separate from another.
- **Dependency**: A module required by another program to run.
- **sys.modules**: Python's internal dictionary of all loaded modules.

Vocabulary Notes

Point out that modules are *namespaces*. The idea of "where does a name live?" will later connect to classes, packages, and scopes. Stress the value of modularity and how this supports collaboration across files.

3 Introducing Modules

When you use Python interactively, all variables and functions vanish when you close the interpreter. To reuse them, you save the code in a **script** file. But when multiple programs need the same code, it's better to put those definitions in a **module**.

Participation Activity – What Is a Module?

Step 1: Create a file `math_utils.py`

```
def double(x):
    return 2 * x

def triple(x):
    return 3 * x
```

Step 2: Create a second file `test_math.py`

```
import math_utils

print(math_utils.double(5))
print(math_utils.triple(7))
```

Expected Output:

```
10
21
```

Discussion: Why is this approach better than copying the same functions into every script?

Suggested Solution

Encourage students to say:

- "It's reusable."
- "If I fix a bug in one place, all scripts benefit."
- "It helps organize code by responsibility."

You can demo editing `math_utils.py` once and rerunning multiple importing scripts to show consistency.

4 How Import Works

When Python sees `import mymodule`, it executes the following:

1. Checks whether `mymodule` is already loaded in `sys.modules`.

2. If not, creates a **module object**.
3. Executes all the code inside `mymodule.py`.
4. Adds the module object to `sys.modules`.
5. Binds the module's name into the importer's namespace.

Demonstration – The Importing Process

File: `tools.py`

```
print("tools.py is running!")

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

File: `main.py`

```
import tools

print("main.py continues")
tools.greet("Jeremy")
```

Expected Output:

```
tools.py is running!
main.py continues
Hello, Jeremy!
```

Explanation and Solution Discussion

Explain that when a module is first imported, Python actually *runs* the file. Subsequent imports skip execution because the module now exists in `sys.modules`. Have students verify this by adding another `import tools` line—it won't print "tools.py is running!" twice.

5 Using Imported Modules

Once a module is imported, you can access its contents with dot notation:

```
import math_utils
print(math_utils.double(10))    # 20
```

Temporary overwriting is possible:

```
math_utils.double = lambda x: x * 10
print(math_utils.double(2))     # 20
```

Instructor Notes

Explain that reassignment like this affects only the in-memory object, not the source file. Restarting Python resets everything.

6 Guided Practice – Importing Multiple Modules

Try This!

File: `weather.py`

```
def forecast():
    return "Sunny and warm"
```

File: `mood.py`

```
def today():
    return "Feeling great!"
```

File: `main.py`

```
import weather
import mood

print(weather.forecast(), "and", mood.today())
```

Expected Output:

```
Sunny and warm and Feeling great!
```

Follow-up Tasks:

1. Add a `print()` statement at the top of each module to announce when it's imported.
2. Observe which import prints first.

Observation Discussion

If students import in this order:

```
import weather
import mood
```

then `weather` runs first. Reversing them flips the print order—proof that Python executes modules as it imports them. This becomes relevant later for circular imports.

7 Student Challenges with Solutions

Challenge 1 – Module Check

Create a module `stats_tools.py`:

```
def mean(nums):
    return sum(nums) / len(nums)

def median(nums):
    nums.sort()
    mid = len(nums) // 2
    return nums[mid]
```

Then use it in `analyze.py`:

```
import stats_tools

data = [4, 7, 2, 9, 6]
print(stats_tools.mean(data))
print(stats_tools.median(data))
```

Sample Output:

```
5.6
6
```

Teaching Focus

Discuss how sorting inside `median()` mutates the list; mention how you might copy it first. Encourage testing with both even- and odd-length lists to extend thinking.

Challenge 2 – Reimport Experiment

```
import stats_tools
import stats_tools
```

Expected Behavior: The module loads only once; Python reuses the object in `sys.modules`. You can verify by printing:

```
print(id(stats_tools))
```

twice and seeing the same memory address.

Discussion

Stress that this caching behavior is efficient and prevents redundant initialization. If you *do* need to reload (e.g., during development), you can use:

```
import importlib
```

```
importlib.reload(stats_tools)
```

Challenge 3 – Namespace Play

```
# cat.py
def speak():
    return "Meow!"

# dog.py
def speak():
    return "Woof!"

# zoo.py
import cat, dog
print(cat.speak(), dog.speak())
```

Output:

Meow! Woof!

Solution Key

Use this as a demonstration that each module provides its own namespace—no collisions even though both define `speak()`. Introduce the idea that the dot operator explicitly references the module's scope.

8 Reflection – Why Modules Matter

Key Takeaways

- Modules make code reusable and easier to maintain.
- They support collaboration—different people can maintain different files.
- Understanding module imports is essential for debugging and larger applications.

Discussion Prompt

Ask students: "If you were designing a large game or app, how would you break it into modules?" Encourage them to name example files (e.g., `player.py`, `inventory.py`, `battle.py`) to see real modular thinking.

9 Extension Prompts with Solutions

- **Task:** Create a module with a function `hello(name)` and import it using `from module import hello`. **Solution:** This imports only that function and allows calling `hello("World")` directly.
- **Task:** Investigate `importlib.reload()`. **Solution:** `importlib.reload()` re-runs a module's code, which is useful when editing modules interactively.
- **Task:** Try putting a module in a subfolder and import using dot notation. **Solution:** Example: `from utilities.math_tools import mean`. Discuss package imports next section.