Python Programming: Custom Modules

October 06, 2025

Creating a Basic Custom Module

Objective: To understand the structure of a Python module and learn how to create and import it to reuse code.

Theory: A **module** is a .py file that contains Python code, such as functions, classes, and variables. Modules are a fundamental way to organize code, promote reusability, and make programs more manageable. To use a module's contents, you must import it into your main script.

Sub-Topics:

- **Module Creation:** A module is simply a Python file. You can create a file (e.g., greetings.py) and add functions to it.
- **Standard Import:** The import module_name statement makes the entire module available. To call a function from the module, you must use the syntax module_name.function_name().

Program:

- **File 1: greetings.py**

```
# greetings.py
def say_hello(name):
    """Prints a simple greeting."""
    print(f"Hello, {name}!")


def say_goodbye(name):
    """Prints a farewell message."""
    print(f"Goodbye, {name}!")
```

- **File 2: main_script.py**

```
# main_script.py
# Import the entire 'greetings' module.
import greetings


# Use functions from the module by referencing the module name.
greetings.say_hello("Alice")
greetings.say_goodbye("Bob")
```

Output:

```
Hello, Alice!

Goodbye, Bob!
```

Different Ways to Import from a Module

Objective: To explore various import methods and understand their use cases.

Theory: Python offers multiple ways to import items from a module, each with its own advantages.

- **import module_name as alias:** Imports the module with a shorter name, which is useful for long module names.
- **from module_name import item1, item2, ...:** Imports specific functions or variables directly into the current namespace. You can use them without the module name prefix.
- **from module_name import * (Wildcard Import):** Imports all items from the module directly into the current namespace. **Caution:** This is generally discouraged as it can lead to naming conflicts and make the code's origin difficult to trace.

Program:

- **File 1: calculations.py**

```
# calculations.py

def add(a, b):

    return a + b


def multiply(a, b):

    return a * b
```

- **File 2: import_methods.py**

```
# import_methods.py

# Method 1: Import with an alias

import calculations as calc

print(f"Using alias: 5 * 3 = {calc.multiply(5, 3)}")
```

```
# Method 2: Import specific functions
from calculations import add
print(f"Using specific import: 10 + 7 = {add(10, 7)}")


# Method 3: Wildcard import
from calculations import *
print(f"Using wildcard: 4 * 4 = {multiply(4, 4)}")
```

Output:

```
Using alias: 5 * 3 = 15
Using specific import: 10 + 7 = 17
Using wildcard: 4 * 4 = 16
```

Creating a Custom Module with a Class

Objective: To demonstrate that a module can contain classes, which can be instantiated and used in other scripts.

Theory: Modules are not limited to functions. They can also contain classes, allowing you to organize your object-oriented code into separate, logical files. You can then import the class just like you would a function.

Program:

- **File 1: vehicle.py**

```
# vehicle.py
class Vehicle:
    """A class to represent a vehicle."""
    def __init__(self, make, model):
        self.make = make
        self.model = model
    def display_info(self):
        return f"Vehicle: {self.make} {self.model}"
```

- **File 2: car_shop.py**

```
# car_shop.py

# Import the Vehicle class from the 'vehicle' module.

from vehicle import Vehicle


# Create an object of the imported class.

my_car = Vehicle("Honda", "Civic")


# Use the object's method.

print(my_car.display_info())
```

Output:

```
Vehicle: Honda Civic
```