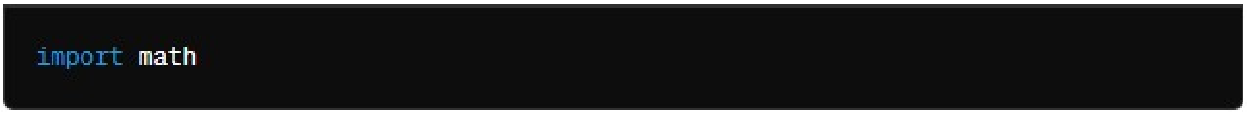
PD
PENTEST DIARIES
HTTPS://WWW.SAKSHAMDIXIT.COM

Import Works in Python

In Python, the `import` statement is used to bring modules and packages into your current script, allowing you to access functions, classes, and variables defined in those modules. Here's a detailed explanation of how the `import` system works in Python:

Basic Import

To import a module, you use the `import` statement followed by the module name:

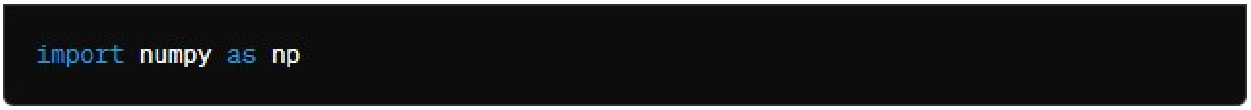


```
import math
```

This imports the `math` module, allowing you to use its functions and constants like `math.sqrt()`, `math.pi`, etc.

Import with Alias

You can give a module an alias using the `as` keyword:

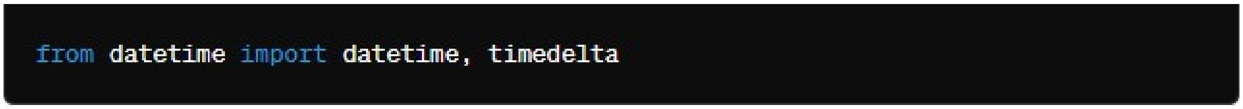


```
import numpy as np
```

This allows you to reference the `numpy` module with the shorter name `np`.

Import Specific Attributes

You can import specific functions, classes, or variables from a module:

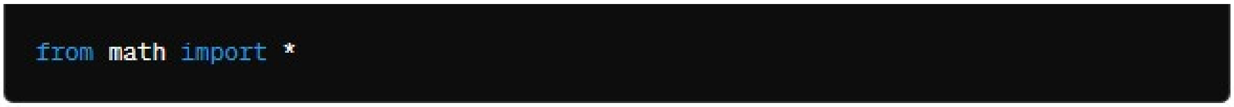


```
from datetime import datetime, timedelta
```

Now, you can use `datetime` and `timedelta` directly without the `datetime.` prefix.

Import All Attributes

To import everything from a module, use the `*` wildcard:



```
from math import *
```

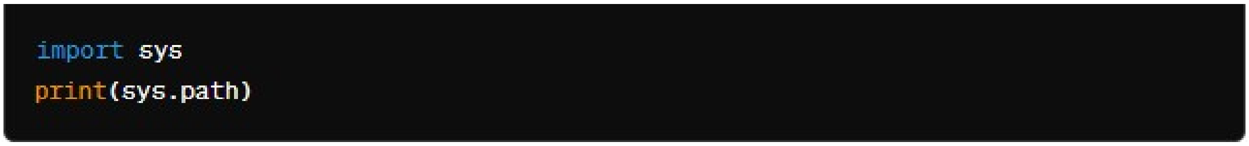
This imports all functions and constants from the `math` module into the current namespace. However, this practice is generally discouraged because it can lead to namespace pollution and make the code harder to read and debug.

Module Search Path

When you import a module, Python searches for it in the following order:

1. **Current directory**: The directory from which the input script was run.
2. **PYTHONPATH**: A list of directories specified by the environment variable `PYTHONPATH`.
3. **Standard library directories**: Directories where the standard library modules are installed.
4. **Site-packages directory**: Where third-party packages are installed.

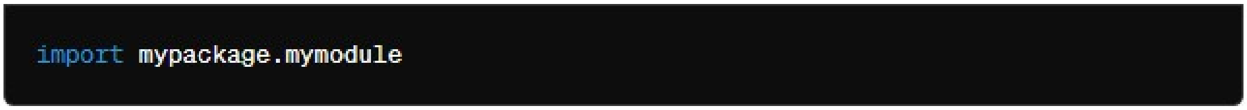
You can view the module search path by inspecting the `sys.path` list:



```
import sys
print(sys.path)
```

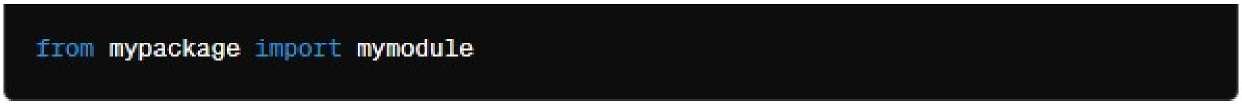
Importing from Packages

Packages are collections of modules in directories that include a special `__init__.py` file. You can import modules from a package like this:



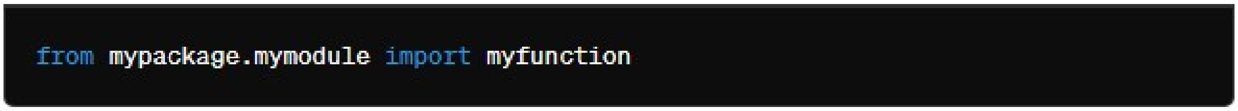
```
import mypackage.mymodule
```

Or using `from`:



```
from mypackage import mymodule
```

Or even specific attributes from a module within a package:



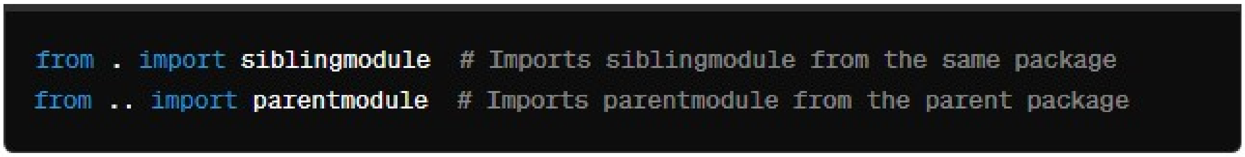
```
from mypackage.mymodule import myfunction
```

The `__init__.py` File

The `__init__.py` file makes a directory a package. It can be empty or can contain package initialization code. This file is executed when the package or its modules are imported.

Relative Imports

Within a package, you can use relative imports to import other modules from the same package:



```
from . import siblingmodule  # Imports siblingmodule from the same package
from .. import parentmodule  # Imports parentmodule from the parent package
```

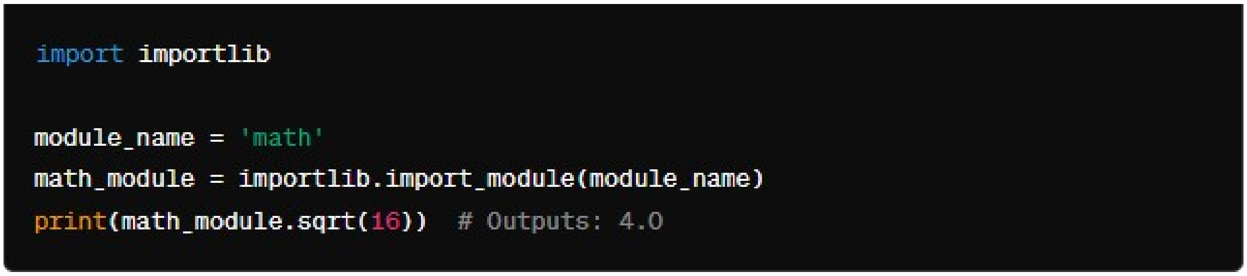
Relative imports use a leading dot (.) to indicate the current and parent packages.

Import Hooks and Custom Importers

Advanced users can customize the import mechanism by using import hooks and writing custom importers. This involves modifying the `sys.meta_path` list, which Python uses to find modules.

Example: Using `importlib` for Dynamic Imports

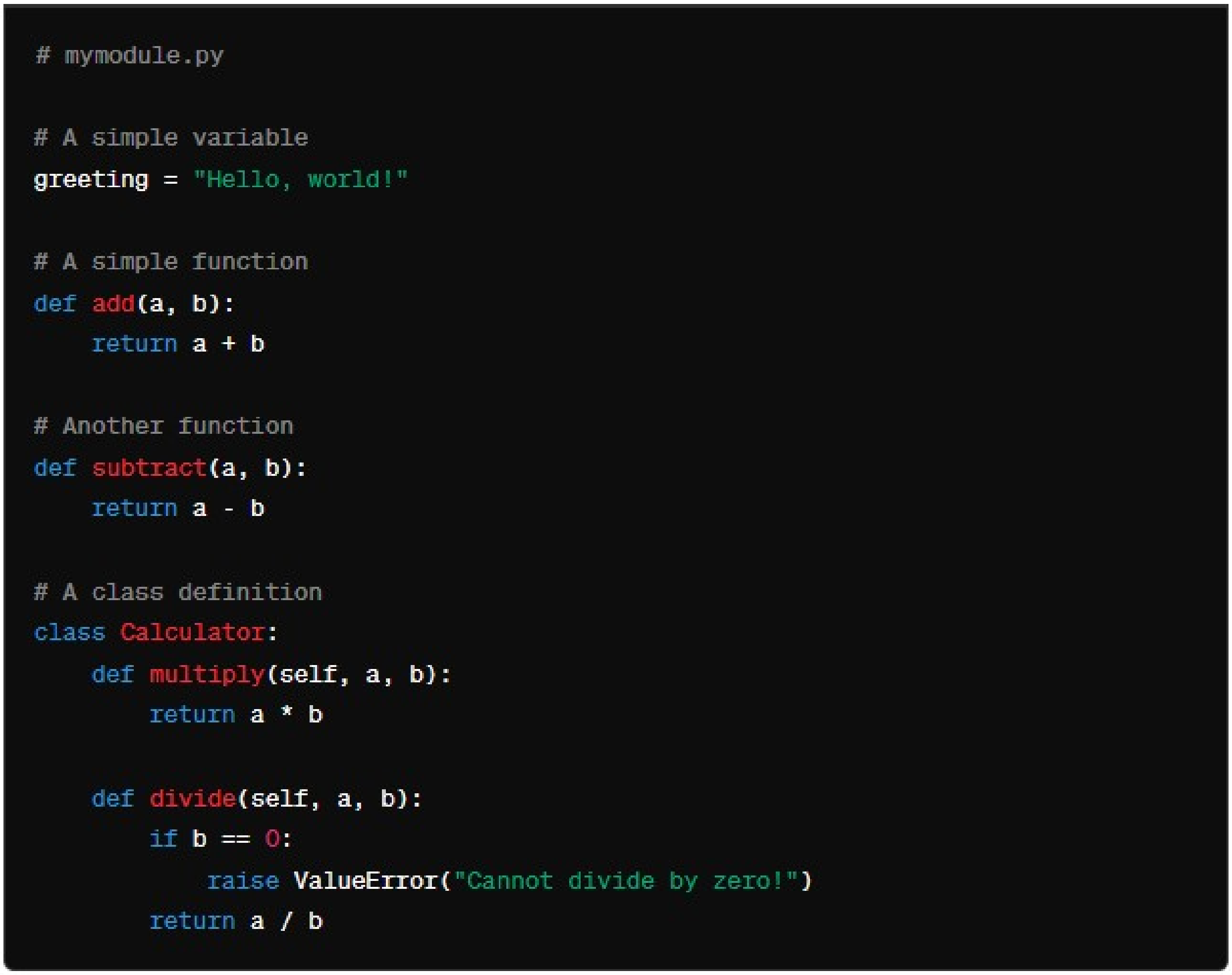
The `importlib` module allows for dynamic imports, where you can import a module whose name is only known at runtime:



```python
import importlib

module_name = 'math'
math_module = importlib.import_module(module_name)
print(math_module.sqrt(16))  # Outputs: 4.0
```

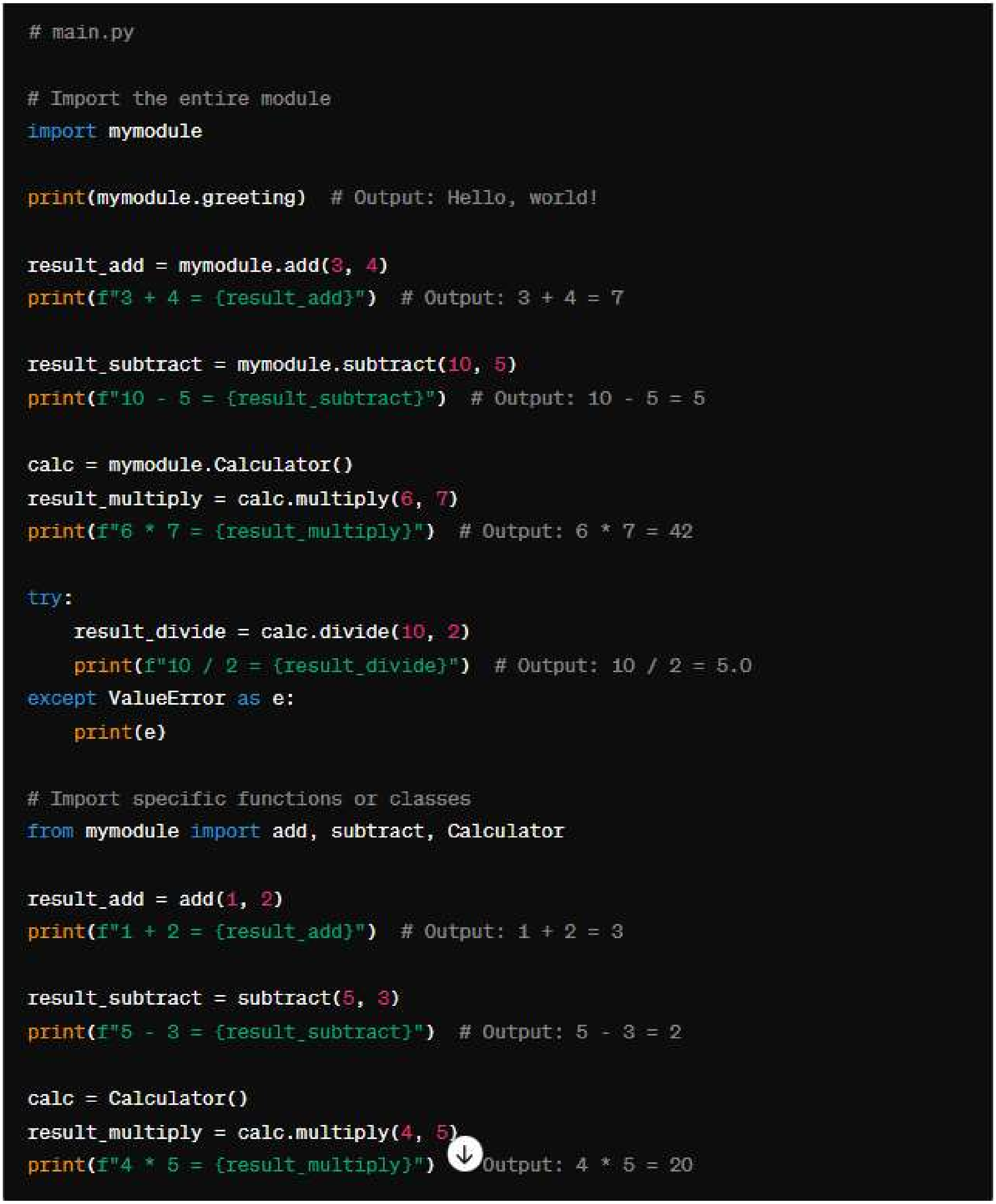
Code:

`mymodule.py`



```python
# mymodule.py

# A simple variable
greeting = "Hello, world!"

# A simple function
def add(a, b):
    return a + b

# Another function
def subtract(a, b):
    return a - b

# A class definition
class Calculator:
    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero!")
        return a / b
```

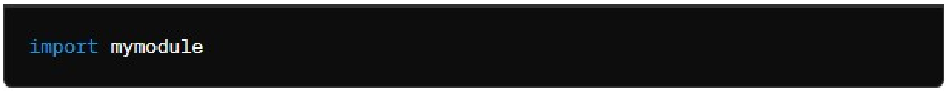
main.py



```python
# main.py

# Import the entire module
import mymodule

print(mymodule.greeting)  # Output: Hello, world!

result_add = mymodule.add(3, 4)
print(f"3 + 4 = {result_add}")  # Output: 3 + 4 = 7

result_subtract = mymodule.subtract(10, 5)
print(f"10 - 5 = {result_subtract}")  # Output: 10 - 5 = 5

calc = mymodule.Calculator()
result_multiply = calc.multiply(6, 7)
print(f"6 * 7 = {result_multiply}")  # Output: 6 * 7 = 42

try:
    result_divide = calc.divide(10, 2)
    print(f"10 / 2 = {result_divide}")  # Output: 10 / 2 = 5.0
except ValueError as e:
    print(e)

# Import specific functions or classes
from mymodule import add, subtract, Calculator

result_add = add(1, 2)
print(f"1 + 2 = {result_add}")  # Output: 1 + 2 = 3

result_subtract = subtract(5, 3)
print(f"5 - 3 = {result_subtract}")  # Output: 5 - 3 = 2

calc = Calculator()
result_multiply = calc.multiply(4, 5)
print(f"4 * 5 = {result_multiply}")    Output: 4 * 5 = 20
```

Explanation:

Import the Entire Module:



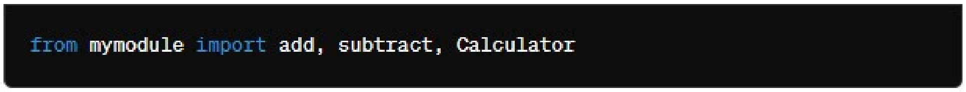
```
import mymodule
```

- This imports the entire module. You can access its contents using the module name as a prefix (e.g., `mymodule.add`).

Using Imported Module:

- Access variables: `mymodule.greeting`
- Call functions: `mymodule.add(3, 4)`
- Use classes: `calc = mymodule.Calculator()`

Import Specific Items:



```
from mymodule import add, subtract, Calculator
```

- This imports specific functions or classes from the module. You can use them directly without the module name prefix.

Key Points

- **Module Creation**: Any Python file can be a module. The file name (without `.py`) is used as the module name.
- **Import Syntax**:
- `import module_name`: Imports the entire module.
- `from module_name import item1, item2`: Imports specific items from the module.
- **Namespace**: Using `import module_name` keeps the module's contents within its namespace, avoiding naming conflicts.
- **Direct Access**: Using `from module_name import item` allows direct access to the item, but can lead to naming conflicts if not managed properly.