Python Modules and Imports

What Is a Module?

A **module** is a Python file containing code that can be reused in another Python file.

A module can contain:

- variables
- functions
- classes
- constants
- configuration
- other Python code

For example, suppose you create a file called:

`math_tools.py`

Inside it:

```
def add(a, b):
    return a + b


def subtract(a, b):
    return a - b
```

This file is a **module**.

Another Python file can import it and use its functions.

```
import math_tools

print(math_tools.add(10, 5))
```

Output:

```
15
```

The basic idea is:

```
math_tools.py
    □
  import
    □
main.py
    □
use its code
```

Why Do We Need Modules?

Imagine putting an entire application into one Python file.

```
app.py
□
□□□ user functions
□□□ payment functions
□□□ database functions
□□□ email functions
□□□ validation functions
□□□ API functions
□□□ utility functions
□□□ hundreds of other lines
```

As the project grows, the file becomes difficult to understand and maintain.

Modules allow you to split the application into logical pieces.

```
project/
□
□□□ users.py
□□□ payments.py
□□□ database.py
□□□ email.py
□□□ validation.py
□□□ main.py
```

Now each file has a clearer purpose.

What Is Importing?

Importing means bringing code from another module into the current Python file so you can use it.

For example:

```
import math

print(math.sqrt(25))
```

Here:

```
import math
```

tells Python:

> Load the `math` module so I can use it.

Your First Custom Module

Create a file:

`calculator.py`

Add:

```
def add(a, b):
    return a + b


def multiply(a, b):
    return a * b
```

Now create:

`main.py`

Import the module:

```
import calculator

print(calculator.add(5, 3))
print(calculator.multiply(4, 2))
```

Output:

```
8
8
```

The module's functions are accessed through the module name:

```
calculator.add()
calculator.multiply()
```

The `import` Statement

The simplest form is:

```
import module_name
```

Example:

```
import math
```

Then access something from the module:

```
math.sqrt(16)
```

Another example:

```
import random

number = random.randint(1, 10)
print(number)
```

Why Use `module_name.function()`?

Consider:

```
import math

print(math.sqrt(25))
```

The `math.` part tells us where `sqrt()` came from.

This improves readability.

Someone reading:

```
math.sqrt(25)
```

immediately knows that `sqrt()` belongs to the `math` module.

Importing Multiple Modules

You can import multiple modules:

```
import math
import random
import datetime
```

Then:

```
print(math.sqrt(25))

number = random.randint(1, 10)

today = datetime.date.today()
```

Generally, keep imports organized near the top of the file.

Import Specific Things

You do not always need to import the entire module name.

You can import a specific function:

```
from math import sqrt
```

Then:

```
print(sqrt(25))
```

You no longer need:

```
math.sqrt()
```

`import` vs `from ... import`

Compare:

Import the module

```
import math

math.sqrt(25)
math.floor(4.8)
```

Import specific functions

```
from math import sqrt, floor

sqrt(25)
floor(4.8)
```

Both are valid.

The first makes the source of the function more obvious.

The second can be convenient when only a few specific items are needed.

Importing Multiple Items

You can import several items:

```
from math import sqrt, ceil, floor
```

Then:

```
print(sqrt(25))
print(ceil(4.2))
print(floor(4.8))
```

Importing Everything With `*`

Python allows:

```
from math import *
```

This imports many names from the module into the current namespace.

However, this is generally discouraged.

Why?

Because it becomes difficult to know where a function came from.

For example:

```
sqrt(25)
```

Could be unclear when reading a large project.

Prefer:

```
import math

math.sqrt(25)
```

or:

```
from math import sqrt
```

Import Aliases

You can give a module another name using `as`.

```
import math as m
```

Then:

```
print(m.sqrt(25))
```

The alias can make long module names shorter.

Common Aliases

Some libraries have widely used aliases.

For example:

```
import datetime as dt
```

Then:

```
today = dt.date.today()
```

Another common example in data science is:

```
import numpy as np
```

and:

```
import pandas as pd
```

Aliases should be clear and follow common conventions.

Aliases for Specific Functions

You can also alias an imported function:

```
from math import sqrt as square_root
```

Then:

```
print(square_root(25))
```

Use aliases when they improve clarity or prevent naming conflicts.

Importing Your Own Functions

Suppose:

```
project/
□
□□□ calculator.py
□□□ main.py
```

calculator.py:

```
def add(a, b):
    return a + b
```

main.py:

```
from calculator import add

print(add(10, 20))
```

Output:

```
30
```

You imported only the `add` function.

Importing Multiple Functions From Your Module

`calculator.py`:

```
def add(a, b):
    return a + b


def subtract(a, b):
    return a - b


def multiply(a, b):
    return a * b
```

Then:

```
from calculator import add, subtract, multiply
```

Now:

```
print(add(10, 5))
print(subtract(10, 5))
print(multiply(10, 5))
```

Importing a Class

Modules can contain classes too.

user.py:

```
class User:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hello, {self.name}!"
```

Then:

```
from user import User

user = User("Hafsa")

print(user.greet())
```

Output:

```
Hello, Hafsa!
```

Modules Can Contain Variables

A module does not have to contain only functions.

config.py:

```
APP_NAME = "haas.dev"
VERSION = "1.0"
```

Another file can use them:

```
import config

print(config.APP_NAME)
print(config.VERSION)
```

Output:

```
haas.dev
1.0
```

A Module Can Contain Different Types of Code

For example:

```
APP_NAME = "haas.dev"


def greet_user(name):
    return f"Hello, {name}!"


class User:
    def __init__(self, name):
        self.name = name
```

All of these can exist inside one module.

Standard Library Modules

Python comes with a large collection of modules.

This collection is called the **Python Standard Library**.

You can use many useful modules without installing them separately.

Examples:

```
math
random
datetime
os
pathlib
json
re
statistics
collections
```

The `math` Module

The `math` module provides mathematical functions.

```
import math

print(math.sqrt(25))
print(math.ceil(4.2))
print(math.floor(4.8))
```

Output:

```
5.0
5
4
```

The `random` Module

The `random` module can generate random values.

```
import random

number = random.randint(1, 10)

print(number)
```

Every execution can produce a different number between 1 and 10.

The `datetime` Module

The `datetime` module works with dates and times.

```
from datetime import date

today = date.today()

print(today)
```

The `json` Module

The `json` module allows Python programs to work with JSON data.

```
import json

data = '{"name": "Hafsa", "role": "developer"}'

user = json.loads(data)

print(user["name"])
```

Output:

```
Hafsa
```

Modules become especially important when working with APIs, files, configuration, and external data.

Third Party Modules

Not every module comes with Python.

Developers can install third-party packages.

For example:

```
requests
numpy
pandas
flask
```

`django`

These are installed separately, usually with `pip`.

For example:

```
pip install requests
```

Then:

```
import requests
```

Third-party packages are covered more deeply when learning Python packages and virtual environments.

Module Search Path

When Python sees:

```
import calculator
```

it needs to find `calculator.py`.

Python searches locations available through its module search path.

You can inspect it using:

```
import sys

print(sys.path)
```

It contains directories where Python looks for modules.

For beginners, the important idea is:

> Python must be able to find the module you are trying to import.

Why Does Python Say "ModuleNotFoundError"?

Suppose you write:

```
import calculator
```

but Python cannot find the module.

You may see:

```
ModuleNotFoundError: No module named 'calculator'
```

Common causes include:

- the file does not exist
- the module name is misspelled
- the file is in the wrong location
- the required package has not been installed
- the virtual environment is incorrect

Example Project Structure

Suppose you have:

```
my_project/
□
□□□ main.py
□□□ calculator.py
□□□ helpers.py
```

`main.py` can import:

```
import calculator
import helpers
```

because the modules are part of the same project structure.

Organizing a Larger Project

As the project grows, you can organize related code into directories.

For example:

```
my_project/
□
□□□ main.py
□
□□□ users/
□   □□□ validation.py
□   □□□ services.py
□
□□□ payments/
□   □□□ checkout.py
□   □□□ refunds.py
□
```

```
□□□ utilities/
  □□□ formatting.py
  □□□ dates.py
```

This leads to the concept of **packages**, which will be covered separately.

The important idea is:

```
Module  = usually one Python file

Package = collection of related Python modules
```

Module Namespace

When you write:

```
import calculator
```

Python creates a namespace for the module.

So:

```
calculator.add()
```

means:

> Find `add` inside the `calculator` module.

This prevents names from different modules from automatically colliding.

For example:

```
import math
import statistics
```

Both modules may contain functions with common names, but their namespaces keep them separate.

Name Conflicts

Suppose you have:

```
from module_a import calculate
from module_b import calculate
```

The second import replaces the name `calculate` in your current namespace.

This can cause confusion.

Instead:

```
import module_a
import module_b

module_a.calculate()
module_b.calculate()
```

Now the source is obvious.

Import Aliases Can Solve Conflicts

You can also do:

```
from module_a import calculate as calculate_a
from module_b import calculate as calculate_b
```

Then:

```
calculate_a()
calculate_b()
```

This makes both functions available without a naming conflict.

Importing a Module Executes Its Top Level Code

Consider:

```
# calculator.py

print("Calculator loaded")

def add(a, b):
    return a + b
```

Now:

```
import calculator
```

will execute the top-level `print()`.

Output:

```
Calculator loaded
```

This is important because module-level code runs when the module is imported.

Why This Can Be a Problem

Avoid putting unnecessary actions at the top level of reusable modules.

For example:

```
print("Starting application...")
connect_to_database()
send_email()
```

If another file imports the module, these actions may happen immediately.

A reusable module should generally define useful code rather than unexpectedly perform application actions when imported.

The `if __name__ == "__main__"` Pattern

A common solution is:

```
def add(a, b):
    return a + b


if __name__ == "__main__":
    print(add(5, 3))
```

Now the test code runs when the file itself is executed:

`python calculator.py`

But it does not run simply because another file imports the module.

Understanding `__name__`

Every Python module has a special variable:

`__name__`

When you run a file directly:

`python calculator.py`

Python sets:

`__name__ == "__main__"`

When another file imports it:

`import calculator`

then:

`__name__ == "calculator"`

This allows Python code to distinguish:

`Running this file directly`

vs
Importing this file

Practical Example

calculator.py:

```
def add(a, b):
    return a + b


def subtract(a, b):
    return a - b


if __name__ == "__main__":
    print(add(10, 5))
    print(subtract(10, 5))
```

If you run:

```
python calculator.py
```

you get:

```
15
5
```

But if main.py contains:

```
import calculator
```

the test prints do not execute.

You can simply use:

```
print(calculator.add(20, 10))
```

Avoid Circular Imports

A circular import happens when two modules depend on each other.

For example:

```
module_a
  
module_b
  
module_a
```

Example:

```
# a.py
from b import function_b
```

and:

```
# b.py
from a import function_a
```

This can cause import errors and confusing behavior.

A better design is usually to reorganize shared functionality into another module:

a.py □□□□□□□□
□
common.py
□
□
b.py □□□□□□□□

Import Best Practices

1. Put imports near the top

Usually:

```
import math
import random

from datetime import date
```

Then your own code follows.

---

2. Avoid wildcard imports

Avoid:

```
from math import *
```

Prefer:

```
import math
```

or:

```
from math import sqrt
```

---

3. Keep imports organized

Group related imports clearly.

For example:

```
import json
import math
import random

from datetime import date

import requests
```

In larger projects, standard library, third-party, and local imports are commonly separated.

---

4. Use meaningful aliases

Good:

```
import datetime as dt
```

Avoid confusing aliases:

```
import datetime as x
```

---

5. Avoid unnecessary imports

Do not import modules you never use.

Instead of:

```
import math
import random
import json
import datetime
```

when you only need:

```
import math
```

keep the imports minimal.

Modules Improve Code Organization

Imagine a web application.

Instead of:

```
main.py
```

containing everything, you might have:

```
project/
□
□□□ main.py
```

```
├── users.py
├── authentication.py
├── payments.py
├── resources.py
└── utilities.py
```

Each module represents a meaningful area of the application.

This makes the project easier to navigate.

Real World Example: haas.dev

Suppose haas.dev has resource-related functions.

Instead of putting everything inside:

`main.py`

you could create:

```
project/
│
├── main.py
├── resources.py
├── search.py
└── formatting.py
```

`resources.py`:

```
def get_free_resources(resources):
    """Return resources that are free."""
```

```
    return [
        resource
        for resource in resources
        if resource["free"]
    ]
```

search.py:

```
def search_resources(resources, keyword):
    """Return resources matching a keyword."""
    keyword = keyword.lower()

    return [
        resource
        for resource in resources
        if keyword in resource["title"].lower()
    ]
```

Then main.py:

```
from resources import get_free_resources
from search import search_resources
```

Now the application is divided into meaningful modules.

Modules and Reusability

One of the biggest benefits of modules is reuse.

Suppose you create:

```
# text_tools.py

def clean_text(text):
    """Return cleaned text."""
    return text.strip().lower()
```

You can use it in multiple files:

```
from text_tools import clean_text
```

You do not need to rewrite the function.

Modules and Team Development

Modules also make collaboration easier.

Imagine a team where:

```
developer A → authentication.py
developer B → payments.py
developer C → resources.py
```

Each person can work on a logical part of the project.

A well-organized module structure reduces unnecessary conflicts and makes responsibilities clearer.

Module vs Function

These are different concepts.

Function

A reusable block of behavior:

```
def add(a, b):
    return a + b
```

Module

A Python file that can contain functions, classes, variables, and other code:

```
# calculator.py

def add(a, b):
    return a + b


def subtract(a, b):
    return a - b
```

Think:

```
Function  one reusable piece of behavior

Module  a container for related Python code
```

Module vs Package

A simple way to remember:

```
Function
  
```

Module (.py file)

Package (collection of modules)

Application

For example:

Function:
calculate_total()

Module:
orders.py

Package:
shop/

Application:
online_store/

A Practical Module Design Workflow

When creating a module, ask:

Step 1: What belongs together?

For example:

User-related functionality

Step 2: Create a module

`users.py`

Step 3: Put related code inside

```
def create_user(user):
    ...


def validate_user(user):
    ...


def delete_user(user_id):
    ...
```

Step 4: Import what you need

```
from users import create_user
```

Step 5: Keep unrelated functionality elsewhere

Do not put payment, email, and database code into `users.py just because it is convenient.

Common Mistakes

1. Wrong Module Name

You write:

```
import calculater
```

but the file is:

`calculator.py`

Python cannot find the misspelled module.

---

2. Importing From the Wrong Location

Your module may exist, but Python may not be searching the directory where it is located.

Check your project structure.

---

3. Wildcard Imports

Avoid:

```
from module import *
```

because it can make names difficult to track.

---

4. Circular Imports

Avoid:

```
A imports B
B imports A
```

Reorganize shared code when necessary.

---

5. Too Much Code at Module Level

Avoid putting actions such as database connections or application startup logic directly at module level unless that behavior is intentional.

---

6. Duplicate Utility Functions

Do not create the same helper function in multiple files.

Create a reusable module instead.

Mini Project: Calculator Modules

Create this structure:

```
calculator_project/
□
□□□ main.py
□□□ operations.py
□□□ display.py
```

operations.py

```
def add(a, b):
    """Return the sum of two numbers."""
    return a + b


def subtract(a, b):
    """Return the difference between two numbers."""
```

```
    return a - b


def multiply(a, b):
    """Return the product of two numbers."""
    return a * b
```

display.py

```
def show_result(result):
    """Display a calculation result."""
    print(f"Result: {result}")
```

main.py

```
from operations import add, subtract, multiply
from display import show_result

result = add(10, 5)
show_result(result)

result = subtract(10, 5)
show_result(result)

result = multiply(10, 5)
show_result(result)
```

Output:

```
Result: 15
Result: 5
Result: 50
```

This demonstrates how modules allow different parts of an application to work together.

---

5 Minute Challenge

Create two files:

`string_tools.py`
`main.py`

Inside `string_tools.py`, create:

```
def reverse_text(text):
    """Return the text reversed."""
    ...


def count_words(text):
    """Return the number of words in the text."""
    ...
```

Then import them in `main.py`:

```
from string_tools import reverse_text, count_words
```

Test:

```
text = "Python makes development easier"

print(reverse_text(text))
print(count_words(text))
```

The goal is to practice:

- creating a module

- defining functions
- importing functions
- reusing code

Exercises

Exercise 1

Create a module called:

`calculator.py`

Add:

- `add()`
- `subtract()`
- `multiply()`
- `divide()`

Import the functions into `main.py`.

Exercise 2

Create:

`temperature.py`

Add:

`celsius_to_fahrenheit()`

`fahrenheit_to_celsius()`

Import and test both functions.

Exercise 3

Create:

`user_tools.py`

Add:

`clean_username()`
`validate_email()`

Use them from another file.

Exercise 4

Create a module with a class and import the class into another file.

Exercise 5

Create a small project containing at least three modules.

Example:

```
project/
```

□
□□□ main.py
□□□ calculations.py
□□□ validation.py
□□□ formatting.py

Give each module a clear responsibility.

Mini Quiz

1. What is a Python module?

A. A loop
B. A Python file containing reusable code
C. A variable
D. A data type

Answer: B

---

2. Which statement imports the `math` module?

A.

```
include math
```

B.

```
use math
```

C.

```
import math
```

D.

```
load math
```

Answer: C

---

3. How do you import only `sqrt` from `math`?

A.

```
import sqrt from math
```

B.

```
from math import sqrt
```

C.

```
math.import(sqrt)
```

D.

```
get math.sqrt
```

Answer: B

---

4. What does `as` allow you to do?

A. Create a class
B. Rename an imported module or object
C. Delete a module
D. Install a package

Answer: B

---

5. What does this do?

```
import calculator
```

A. Deletes the calculator
B. Creates a new calculator
C. Makes the calculator module available in the current file
D. Installs Python

Answer: C

---

6. What is a package?

A. A collection of related Python modules
B. A function parameter
C. A loop
D. A variable

Answer: A

Interview Questions

1. What is a module in Python?

A module is a Python file containing reusable code such as functions, classes, and variables.

2. Why are modules useful?

They help organize code, encourage reuse, reduce duplication, and make larger applications easier to maintain.

3. What is the difference between `import module` and `from module import function`?

With:

```
import module
```

you access the object through the module:

```
module.function()
```

With:

```
from module import function
```

you can use:

```
function()
```

directly.

4. What is an import alias?

An alias gives an imported module or object another name.

```
import datetime as dt
```

5. Why should wildcard imports generally be avoided?

They can introduce many names into the current namespace and make it difficult to determine where those names came from.

6. What is `__name__`?

`__name__` is a special variable that identifies the current module. When a file is executed directly, its value is usually `"__main__"`.

7. Why use `if __name__ == "__main__"`?

It allows code to run when a module is executed directly while preventing that code from running when the module is imported.

8. What is a circular import?

A circular import occurs when modules depend on each other, directly or indirectly.

Example:

```
A □ B
□   □
□□□□□
```

It can cause import problems and often indicates that the code structure needs to be reorganized.

Module Cheat Sheet

Import a module

```
import math
```

Use something from it

```
math.sqrt(25)
```

Import a specific function

```
from math import sqrt
```

Use it

```
sqrt(25)
```

Import multiple items

```
from math import sqrt, ceil, floor
```

Create an alias

```
import datetime as dt
```

Access documentation

```
help(math)
```

Inspect module search paths

```
import sys

print(sys.path)
```

Protect direct execution

```
if __name__ == "__main__":
    main()
```

Key Takeaways

- A **module** is usually a Python file containing reusable code.
- Modules help organize large programs into logical pieces.
- `import` makes another module available in your current file.
- You can import an entire module or specific functions/classes.
- `as` creates an alias for an imported name.
- Avoid wildcard imports because they make code harder to understand.
- Python includes a large **standard library** of ready-to-use modules.
- Third-party modules can be installed separately.
- Modules have their own namespaces.
- Module-level code executes when the module is imported.
- `if __name__ == "__main__"` prevents direct-execution code from running during imports.
- Circular imports should generally be avoided.
- A module should contain code that belongs together.
- Modules make applications easier to organize, reuse, test, and maintain.
- **Module = reusable Python file.**
- **Package = collection of related modules.**

Visit haas.dev for more resources and guides.

haas.dev

https://dev-roast-app.vercel.app