Creating Your Own Modules in Python

Introduction

As Python programs become larger, keeping everything inside one file becomes difficult.

A better approach is to split your program into smaller, organized files called **modules**.

Python allows you to create your own modules and reuse their functions, variables, and classes in other Python files.

For example, instead of putting all of these into one file:

```
calculator
user authentication
file handling
resource filtering
database operations
```

you can organize them:

```
calculator.py
auth.py
files.py
resources.py
database.py
```

Each `.py` file can act as a module.

---

1. What Is a Module?

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

For example:

`math_tools.py`

can contain:

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


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

Another Python file can import and use these functions.

```
import math_tools

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

Output:

```
15
5
```

The important idea is:

> **One Python file can provide reusable functionality to another Python file.**

---

2. Why Create Your Own Modules?

Creating your own modules helps with:

Organization

Large programs can be divided into logical files.

Reusability

Write a function once and use it in multiple files.

Maintainability

Changing one feature becomes easier because its code is kept in one place.

Readability

Instead of having thousands of lines in one file, related code can be grouped together.

Collaboration

Different developers can work on different modules.

---

3. A Simple Custom Module

Suppose we create:

`calculator.py`

Inside it:

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


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

Now create another file:

`main.py`

Import the module:

```
import calculator
```

Use its functions:

```
print(calculator.add(10, 5))
print(calculator.multiply(4, 3))
```

Output:

```
15
12
```

The syntax is:

```
module_name.function_name()
```

---

4. How Python Finds Your Module

When you write:

```
import calculator
```

Python searches for a module named:

```
calculator
```

If your file is:

```
calculator.py
```

Python can load it when it is available in the appropriate module search path.

A simple project might look like:

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

Because both files are inside the same project directory, `main.py` can normally import `calculator`.

---

5. Importing a Module

Use:

```
import calculator
```

Then access its contents through the module name.

```
calculator.add(5, 3)
```

This approach makes it clear where the function came from.

For example:

```
import calculator

result = calculator.add(10, 20)

print(result)
```

---

6. Importing Specific Functions

You do not always need to import the entire module.

You can import a specific function:

```
from calculator import add
```

Now you can call:

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

Instead of:

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

You can import multiple functions:

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

Then:

```
print(add(5, 2))
print(multiply(5, 2))
```

---

7. Using Aliases

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

```
import calculator as calc
```

Now:

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

This can be useful when a module has a long name.

For example:

```
import resource_processing as rp
```

Then:

```
rp.filter_resources()
```

---

8. Modules Can Contain More Than Functions

A module can contain:

- functions
- variables

- classes
- constants
- executable statements

Example:

```
APP_NAME = "haas.dev"


def get_message():
    return "Welcome to haas.dev"
```

Another file can use both:

```
import settings

print(settings.APP_NAME)
print(settings.get_message())
```

---

9. Creating a Useful Module

Suppose you are building a resource website.

You might have:

```
resource_utils.py
```

Inside:

```
def filter_free_resources(resources):
    return [
```

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


def count_resources(resources):
    return len(resources)
```

Then:

main.py

can use it:

```
import resource_utils

resources = [
    {"title": "Python Guide", "free": True},
    {"title": "Advanced JavaScript", "free": False},
    {"title": "Git Guide", "free": True}
]

free_resources = resource_utils.filter_free_resources(resources)

print(free_resources)
print(resource_utils.count_resources(resources))
```

The resource-related logic stays inside its own module.

---

10. Modules Create Separation of Responsibilities

Imagine this project:

```
my_app/
□
□□□ main.py
□□□ users.py
□□□ payments.py
□□□ resources.py
□□□ notifications.py
```

Each module has a responsibility.

users.py

Handles users.

```
def create_user(name):
    ...
```

payments.py

Handles payments.

```
def process_payment(amount):
    ...
```

resources.py

Handles resources.

```
def search_resources(query):
    ...
```

notifications.py

Handles notifications.

```
def send_notification(message):
    ...
```

main.py

Coordinates the application.

This is much easier to understand than putting everything inside `main.py`.

---

11. Modules as Building Blocks

Think of your application like a building.

Instead of constructing everything as one giant structure, you create separate sections.

```
Application
□
□□□ User System
□□□ Authentication
□□□ Payments
□□□ Resources
□□□ Notifications
□□□ Reports
```

Each section can have its own module.

The application then combines these modules.

This is one of the foundations of writing scalable software.

---

12. Creating a Module Step by Step

Suppose we want a module for temperature conversion.

Step 1: Create a file

`temperature.py`

Step 2: Add functionality

```
def celsius_to_fahrenheit(celsius):
    return (celsius * 9 / 5) + 32


def fahrenheit_to_celsius(fahrenheit):
    return (fahrenheit - 32) * 5 / 9
```

Step 3: Create another file

`main.py`

Step 4: Import the module

```
import temperature
```

Step 5: Use its functions

```
print(temperature.celsius_to_fahrenheit(25))
print(temperature.fahrenheit_to_celsius(77))
```

---

13. Module Names

A module filename should generally be:

- lowercase
- descriptive
- short
- related to its responsibility

Good:

```
users.py
database.py
payments.py
validators.py
file_utils.py
```

Avoid unnecessarily vague names:

```
stuff.py
things.py
random.py
mycode.py
```

A module name should help another developer understand what is inside it.

---

14. Avoid Naming Conflicts

Be careful when naming your own modules.

Do not create files such as:

```
math.py
```

```
json.py
random.py
typing.py
```

when you intend to use Python's standard-library modules with those names.

For example, if you create:

```
random.py
```

you can accidentally interfere with:

```
import random
```

Python may load your file instead of the standard-library module.

Prefer names such as:

```
random_utils.py
```

if that is what your file actually contains.

---

15. Importing Multiple Modules

You can import several modules:

```
import users
import payments
import notifications
```

Then:

```
users.create_user("Hafsa")
payments.process_payment(500)
notifications.send_notification("Payment successful")
```

Each module remains responsible for its own functionality.

---

16. Importing With `from`

You can write:

```
from users import create_user
from payments import process_payment
```

Then:

```
create_user("Hafsa")
process_payment(500)
```

This can make code shorter.

However, importing everything directly can sometimes make it less obvious where a function came from.

Compare:

```
users.create_user()
```

with:

```
create_user()
```

The first version immediately tells the reader that `create_user()` belongs to the `users` module.

---

17. Avoid `from module import *`

You may see:

`from calculator import *`

This imports everything from the module.

Although Python allows it, it is generally better to avoid this in normal application code.

Instead:

`from calculator import add, multiply`

This makes dependencies explicit.

---

18. Modules and `__name__`

Every Python module has a special variable:

`__name__`

When a file is executed directly, Python sets:

`__name__ == "__main__"`

When the file is imported, `__name__` usually contains the module's name.

Example:

```
def greet():
    print("Hello")


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

If you run:

```
python greeting.py
```

the function runs.

But if another file does:

```
import greeting
```

the code inside:

```
if __name__ == "__main__":
```

does not run automatically.

---

19. Why `if __name__ == "__main__"` Matters

Consider:

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

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

If another file imports this module:

```
import calculator
```

the `print()` statement also executes during import.

That may not be what you want.

Instead:

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


if __name__ == "__main__":
    print(add(10, 20))
```

Now the test code only runs when the file itself is executed.

This makes modules safer and more reusable.

---

20. Module-Level Variables

You can define constants in a module.

```
MAX_USERS = 100
APP_NAME = "My App"
```

Another file:

```
import settings

print(settings.APP_NAME)
print(settings.MAX_USERS)
```

Constants are commonly written in uppercase:

```
MAX_USERS = 100
API_TIMEOUT = 30
DEFAULT_LANGUAGE = "en"
```

---

21. A Practical Project Structure

A small application might look like:

```
my_app/
□
□□□ main.py
□□□ users.py
□□□ validators.py
□□□ resources.py
□□□ settings.py
```

users.py

```
def create_user(name):
    return {
        "name": name
    }
```

validators.py

```
def is_valid_name(name):
```

```
    return len(name.strip()) > 0
```

resources.py

```
def get_free_resources(resources):
    return [
        resource
        for resource in resources
        if resource["free"]
    ]
```

settings.py

```
APP_NAME = "Resource App"
```

main.py

```
import users
import validators
import resources
import settings

name = "Hafsa"

if validators.is_valid_name(name):
    user = users.create_user(name)
    print(user)

print(settings.APP_NAME)
```

This is already closer to how real applications are structured.

---

22. Module Dependencies

Modules can use other modules.

For example:

```
main.py
  
  ├── users.py
  
  ├── resources.py
  
  └── settings.py
```

`main.py` imports them.

A module can also import another module:

```
import validators
```

inside `users.py`.

This creates dependencies.

The more dependencies a project has, the more important good project structure becomes.

---

23. Avoid Circular Imports

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

For example:

```
users.py
   
payments.py
   
users.py
```

Example:

```
# users.py
import payments
```

and:

```
# payments.py
import users
```

This can create import problems.

A better solution is usually to reorganize shared functionality into another module.

For example:

```
users.py
payments.py
common.py
```

Both can use:

```
import common
```

instead of importing each other.

---

24. Modules vs Functions

A function is a reusable piece of behavior.

```
def calculate_total(items):
    ...
```

A module is a file that can contain multiple reusable pieces.

`cart.py`

might contain:

```
def add_item():
    ...


def remove_item():
    ...


def calculate_total():
    ...
```

So:

`Function → reusable behavior`

`Module → collection of related Python code`

---

25. Modules vs Packages

A **module** is typically one Python file:

```
users.py
```

A **package** is a directory that organizes multiple modules.

```
users/
□
□□□ __init__.py
□□□ authentication.py
□□□ profiles.py
□□□ permissions.py
```

Think:

```
Module = one room

Package = collection of related rooms
```

Packages become useful when a project grows beyond a few modules.

---

26. Common Mistakes

Mistake 1: Putting Everything in One File

Bad:

```
main.py
```

with thousands of lines.

Better:

```
main.py
users.py
payments.py
resources.py
database.py
```

Mistake 2: Creating Huge Modules

Modules should have a clear responsibility.

Avoid creating:

```
everything.py
```

with unrelated functionality.

Mistake 3: Using Vague Names

Avoid:

```
helpers.py
utils.py
stuff.py
```

unless the contents genuinely have a clear shared purpose.

Prefer meaningful names:

validators.py
resource_filters.py
date_utils.py

Mistake 4: Importing Everything

Avoid:

```
from module import *
```

Prefer explicit imports.

Mistake 5: Forgetting Module Dependencies

If `main.py` depends on:

```
import users
```

then `users.py` must be available in the appropriate project/module path.

Mistake 6: Running Code During Import

Avoid putting unnecessary execution at module level.

Instead of:

```
print("Starting application")
```

use:

```
if __name__ == "__main__":
    print("Starting application")
```

when that behavior is intended only for direct execution.

---

27. Best Practices

Give each module a clear responsibility

```
users.py  user operations
database.py  database operations
validators.py  validation
```

Keep modules reasonably focused

Do not put unrelated functionality into one module.

Use descriptive names

```
resource_filters.py
```

is better than:

```
stuff.py
```

Keep reusable logic separate from execution

Use:

```
if __name__ == "__main__":
```

for code intended to run only when the file is executed directly.

Avoid circular dependencies

Design your modules so dependencies flow in a clear direction.

Import only what you need

Prefer:

```
from calculator import add
```

over:

```
from calculator import *
```

when direct imports are appropriate.

---

28. A Better Way to Think About Modules

When creating a module, ask:

```
What responsibility does this file have?
```

Then ask:

```
What functionality belongs here?
```

Then:

```
What functionality does this module need from other modules?
```

Finally:

`Can another part of the application use this module without knowing its internal details?`

This way of thinking helps you move from simply writing Python code to designing software.

---

29. A Real World Example: haas.dev Resource System

Imagine a simplified haas.dev resource application.

Instead of putting everything inside one file:

`main.py`

we could create:

```
haas_resources/
□
□□□ main.py
□□□ resources.py
□□□ search.py
□□□ filters.py
□□□ categories.py
□□□ settings.py
```

resources.py

Responsible for resource data:

```
resources = [
  {
```

```
        "title": "Python Guide",
        "category": "Python",
        "free": True
    },
    {
        "title": "JavaScript Guide",
        "category": "JavaScript",
        "free": True
    }
]


def get_resources():
    return resources
```

filters.py

Responsible for filtering:

```
def get_free_resources(resources):
    return [
        resource
        for resource in resources
        if resource["free"]
    ]
```

search.py

Responsible for searching:

```
def search_resources(resources, keyword):
    keyword = keyword.lower()

    return [
        resource
```

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

main.py

Coordinates everything:

```
import resources
import filters
import search

all_resources = resources.get_resources()

free_resources = filters.get_free_resources(all_resources)

python_resources = search.search_resources(
    all_resources,
    "Python"
)

print(free_resources)
print(python_resources)
```

Now each module has a clear responsibility.

This is the basic idea behind modular software architecture.

---

30. Module Design Mindset

Before creating a new module, think:

Feature

Responsibility

Module

Functions / Classes

Used by other modules

For example:

User Authentication

authentication.py

login()
logout()
register()
validate_password()

This is much easier to maintain than putting all authentication logic inside the main program.

---

31. Mini Project: Personal Utility Library

Create this structure:

utility_app/

 main.py
 calculator.py
 text_utils.py

□□□ validators.py

calculator.py

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


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


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

text_utils.py

```
def word_count(text):
    return len(text.split())


def make_uppercase(text):
    return text.upper()
```

validators.py

```
def is_positive(number):
    return number > 0
```

main.py

```
import calculator
import text_utils
import validators

number = 10
```

```
if validators.is_positive(number):
    print(calculator.multiply(number, 5))

text = "Python modules are useful"

print(text_utils.word_count(text))
print(text_utils.make_uppercase(text))
```

You have now created your own small Python library.

---

32. Five Minute Challenge

Create:

`student_tools.py`

Add:

```
def calculate_average(marks):
    return sum(marks) / len(marks)


def highest_mark(marks):
    return max(marks)
```

Then create:

`main.py`

Import the module:

```
import student_tools
```

Use:

```
marks = [80, 75, 92, 88]

print(student_tools.calculate_average(marks))
print(student_tools.highest_mark(marks))
```

Challenge

Add another function:

```
def lowest_mark(marks):
    ...
```

Then use it from `main.py`.

---

33. Exercises

Exercise 1

Create:

`geometry.py`

Add functions for:

```
rectangle_area()
square_area()
circle_area()
```

Import them into `main.py`.

Exercise 2

Create:

`string_tools.py`

Add:

`reverse_text()`
`count_vowels()`
`capitalize_words()`

Use the module from another Python file.

Exercise 3

Create:

`student.py`

Add:

```
def is_passing(mark):
    return mark >= 40
```

Import it into `main.py` and test several marks.

Exercise 4

Create:

`resource_tools.py`

Add:

```
search_resources()
filter_free_resources()
count_resources()
```

Use the module with a list of resource dictionaries.

---

34. Mini Quiz

Question 1

What is a Python module?

A. A Python installation

B. A Python file containing reusable code

C. A database

D. A loop

Answer: B

Question 2

How do you import a module named `calculator`?

A.

```
include calculator
```

B.

```
use calculator
```

C.

```
import calculator
```

D.

```
module calculator
```

Answer: C

Question 3

How do you access `add()` from the `calculator` module?

A.

```
calculator->add()
```

B.

`calculator.add()`

C.

`calculator:add()`

D.

`add.calculator()`

Answer: B

Question 4

What does this do?

```
from calculator import add
```

It imports the `add` function directly from the `calculator` module.

Question 5

Why is this useful?

```
if __name__ == "__main__":
```

It allows code to run when the file is executed directly without automatically running that code when the module is imported.

---

35. Interview Questions

1. What is a Python module?

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

2. How do you create your own module?

Create a `.py` file containing Python code and import it from another Python file.

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

`import calculator`

imports the module and requires:

`calculator.add()`

While:

`from calculator import add`

allows:

`add()`

4. Why should `from module import *` generally be avoided?

It can make it unclear where names came from and can create naming conflicts.

5. What is `__name__`?

`__name__` is a special variable that tells Python how the current module is being used.

6. What does `__name__ == "__main__"` mean?

It means the current file is being executed directly rather than imported as a module.

7. What is a circular import?

It occurs when modules depend on each other directly or indirectly.

8. What is the difference between a module and a package?

A module is generally a single Python file, while a package organizes multiple related modules into a directory.

---

36. Cheat Sheet

Create a module

`calculator.py`

Import module

`import calculator`

Use module function

`calculator.add(2, 3)`

Import specific function

`from calculator import add`

Import multiple functions

`from calculator import add, multiply`

Alias

```
import calculator as calc
```

Direct execution check

```
if __name__ == "__main__":
    ...
```

Good structure

```
project/
□
□□□ main.py
□□□ users.py
□□□ resources.py
□□□ settings.py
```

Core idea

```
One responsibility
      □
One focused module
      □
Reusable functionality
      □
Cleaner application
```

---

37. Key Takeaways

- A Python module is typically a `.py` file containing reusable code.
- You can create your own modules simply by creating Python files.
- Use `import` to use another module.

- Use `from ... import ...` to import specific functionality.
- Modules can contain functions, variables, classes, and constants.
- Good modules have a clear responsibility.
- Modular code is easier to read, reuse, test, and maintain.
- Avoid overly large modules and vague module names.
- Avoid unnecessary wildcard imports.
- Avoid circular dependencies.
- `if __name__ == "__main__":` separates direct execution from imported module behavior.
- Packages can organize multiple related modules.
- Creating modules is an important step toward designing larger Python applications.

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

Visit haas.dev for more resources and guides.