Python Packages

What Is a Python Package?

A **package** is a way of organizing related Python modules into a directory structure.

Remember:

> **Module = usually one Python file**
> **Package = a collection of related modules**

For example:

```
project/
□
□□□ main.py
□
□□□ users/
   □□□ validation.py
   □□□ services.py
   □□□ database.py
```

Here:

```
validation.py
services.py
database.py
```

are modules.

The:

`users/`

directory organizes those modules into one package.

Why Do We Need Packages?

Small projects can work with a few Python files:

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

But as an application grows, you may have dozens or hundreds of modules.

For example:

```
project/
□□□ users/
□□□ payments/
□□□ authentication/
□□□ resources/
□□□ notifications/
□□□ database/
□□□ utilities/
```

Packages help organize these modules into meaningful groups.

Instead of having one huge collection of unrelated files, you create logical areas.

Module vs Package

This distinction is important.

Module

A Python file:

```
calculator.py
```

Example:

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

Package

A directory containing related Python modules:

```
calculator/
□□□ basic.py
□□□ scientific.py
□□□ statistics.py
```

Think of it like:

```
Module
  □
One Python file

Package
  □
```

Collection of related modules

A Simple Package Structure

Create:

```
my_project/
□
□□□ main.py
□
□□□ calculator/
   □□□ __init__.py
   □□□ basic.py
   □□□ advanced.py
```

Here:

`calculator/`

is the package.

Inside it:

`basic.py`
`advanced.py`

are modules.

What Is `__init__.py`?

Traditionally, a Python package contains a file named:

`__init__.py`

Example:

```
calculator/
□□□ __init__.py
□□□ basic.py
□□□ advanced.py
```

The file can be empty:

```
# __init__.py
```

or it can contain package-level code.

Modern Python also supports **namespace packages**, which can work without `__init__.py`, but for learning and many regular package structures, understanding `__init__.py` is still important.

Creating Your First Package

Suppose you want to create a package for mathematical operations.

Structure:

```
math_tools/
□□□ __init__.py
□□□ basic.py
□□□ advanced.py
```

The `basic.py` Module

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


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

The `advanced.py` Module

```
def square(number):
    return number * number


def cube(number):
    return number * number * number
```

Now you have a package containing two related modules.

Importing a Module From a Package

In `main.py`:

```
from math_tools import basic
```

Then:

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

```
print(basic.subtract(10, 5))
```

Output:

```
15
5
```

Importing Another Module

You can also import:

```
from math_tools import advanced
```

Then:

```
print(advanced.square(5))
print(advanced.cube(3))
```

Output:

```
25
27
```

Importing Specific Functions

Instead of importing the module:

```
from math_tools.basic import add
```

Now you can call:

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

You can also import multiple functions:

```
from math_tools.basic import add, subtract
```

Full Example

Project:

```
project/
□
□□□ main.py
□
□□□ math_tools/
   □□□ __init__.py
   □□□ basic.py
   □□□ advanced.py
```

basic.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
```

advanced.py:

```
def square(number):
    """Return the square of a number."""
    return number ** 2


def cube(number):
    """Return the cube of a number."""
    return number ** 3
```

main.py:

```
from math_tools.basic import add
from math_tools.advanced import square

print(add(10, 5))
print(square(4))
```

Output:

```
15
16
```

What Does `__init__.py` Do?

The `__init__.py` file can be used to:

- mark or organize a regular package
- expose selected package functionality
- define package-level variables
- run package initialization code

For example:

```
# math_tools/__init__.py

VERSION = "1.0"
```

Then:

```
from math_tools import VERSION

print(VERSION)
```

Output:

```
1.0
```

Re Exporting Functions Through `__init__.py`

Suppose:

```
math_tools/
□□□ __init__.py
□□□ basic.py
```

`basic.py`:

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

You could put this in `__init__.py`:

```
from .basic import add
```

Now users can write:

`from math_tools import add`

instead of:

`from math_tools.basic import add`

The package provides a simpler public interface.

What Does the Dot Mean?

Inside a package, you may see:

`from .basic import add`

The `.` means:

> Look inside the current package.

For example:

```
math_tools/
□□□ __init__.py
□□□ basic.py
```

Inside `__init__.py`:

`from .basic import add`

means:

from this package basic.py import add

Multiple Dots

In larger package structures, you may see relative imports such as:

```
from ..common import helper
```

The dots represent parent package levels.

For beginners, remember:

```
.  current package
..  parent package
```

Relative imports become more useful when working with larger applications.

Absolute Imports

An absolute import describes the package path from the project's import location.

Example:

```
from math_tools.basic import add
```

This clearly identifies where `add` comes from.

Absolute imports are often easier to understand in larger projects.

Relative vs Absolute Imports

Absolute

```
from math_tools.basic import add
```

Relative

```
from .basic import add
```

Both can be useful.

A simple rule:

> Use the style that makes your project's structure clear and consistent.

Nested Packages

Packages can contain other packages.

For example:

```
project/
□
□□□ main.py
□
□□□ shop/
   □□□ __init__.py
   □
```

```
   ├── products/
   │   ├── __init__.py
   │   └── catalog.py
   │
   └── orders/
       ├── __init__.py
       └── checkout.py
```

Here:

`shop`

is a package.

Inside it:

`products`
`orders`

are subpackages.

Importing From a Nested Package

You could write:

```
from shop.products.catalog import get_products
```

The path tells Python:

```
shop
 ├
```

```
products
 □
catalog.py
 □
get_products()
```

This is useful for large applications.

Real World Package Structure

Imagine an e-commerce application:

```
shop/
□□□ __init__.py
□
□□□ users/
□   □□□ __init__.py
□   □□□ models.py
□   □□□ services.py
□
□□□ products/
□   □□□ __init__.py
□   □□□ models.py
□   □□□ services.py
□
□□□ payments/
□   □□□ __init__.py
□   □□□ checkout.py
□   □□□ refunds.py
□
□□□ notifications/
    □□□ __init__.py
```

```
    email.py
```

This structure makes the application easier to navigate.

A developer looking for payment code knows to check:

```
payments/
```

A developer looking for user functionality checks:

```
users/
```

Packages and Separation of Responsibilities

Packages work especially well when each package represents a major area of the application.

For example:

```
users/
payments/
resources/
authentication/
notifications/
```

Each package can contain several modules related to that area.

This creates a hierarchy:

```
Application
  
Packages
```

Modules
Functions / Classes

Python Standard Library Packages

Python's standard library contains many modules and packages.

For example:

```
import email
```

Python also provides collections of related functionality such as:

```
import urllib
```

and:

```
import xml
```

You can explore standard library documentation to discover what is already available before writing functionality yourself.

Third Party Packages

Python's ecosystem becomes much larger when you use third-party packages.

Examples include:

`requests`
`numpy`
`pandas`
`flask`
`django`

These are not all part of Python itself.

They can be installed using package management tools such as `pip`.

What Is `pip`?

`pip` is Python's commonly used package installer.

For example:

```
pip install requests
```

This tells Python's package management system to install the `requests` package.

After installation:

```
import requests
```

can be used in your program.

Installing a Package

Suppose you need `requests`.

Run:

```
pip install requests
```

Then:

```
import requests
```

Now your application can use the package.

Installing a Specific Version

Sometimes a project needs a particular version.

For example:

```
pip install requests==2.32.3
```

This installs that specific version if available for the environment.

Version control becomes important in larger projects because different package versions can behave differently.

Upgrading a Package

You can upgrade a package:

```
pip install --upgrade requests
```

This asks `pip` to install a newer available version.

Before upgrading packages in a production project, developers should consider compatibility with the rest of the application.

Uninstalling a Package

To remove a package:

```
pip uninstall requests
```

Python will ask for confirmation before removing it.

Listing Installed Packages

You can see installed packages with:

```
pip list
```

This can show something similar to:

```
Package      Version
------------ -------
requests     2.x.x
pip          25.x
```

The exact versions depend on your environment.

Package Dependencies

A package may depend on other packages.

For example:

```
Your application
    
  Package A
    
  Package B
```

When installing Package A, its dependencies may also need to be installed.

This is why package management is important in real-world Python projects.

---

requirements.txt

A common way to record project dependencies is:

`requirements.txt`

Example:

```
requests==2.32.3
```

Another developer can install the listed dependencies using:

```
pip install -r requirements.txt
```

This helps recreate the project's environment.

Why Dependency Management Matters

Imagine your application uses:

```
requests
pandas
flask
```

You send the project to another developer.

If they only receive:

```
main.py
```

they may not know which external packages are required.

A dependency file communicates:

```
Install these packages before running the project.
```

Virtual Environments and Packages

Third-party packages should generally be installed inside a project's virtual environment rather than globally.

For example:

```
python -m venv .venv
```

Activate the environment and install packages there.

This keeps project dependencies isolated.

For example:

```
Project A
 □□□ requests 2.x

Project B
 □□□ requests 3.x
```

Each project can have its own environment and package versions.

Virtual environments are an important next step when working with real Python projects.

Package vs Library

These terms are sometimes used loosely, but they are not exactly identical.

Package

A distributable or importable collection of Python modules.

Library

A broader term for reusable code that developers can use in their programs.

A library can consist of one or more packages and modules.

For beginners:

`Module → Python file`
`Package → Organized collection of modules`
`Library → Reusable functionality`

Package vs Application

A package is generally reusable code.

An application is the complete software product that performs a specific job.

For example:

`requests`

is reusable software functionality.

An application using it might be:

`weather_app`

The application uses packages as building blocks.

Creating a Reusable Package

Suppose you want to create a package for text processing.

Structure:

```
text_tools/
□□□ __init__.py
□□□ cleaning.py
□□□ analysis.py
```

cleaning.py:

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

analysis.py:

```
def count_words(text):
    """Return the number of words."""
    return len(text.split())
```

__init__.py:

```
from .cleaning import clean_text
from .analysis import count_words
```

Now users can write:

```
from text_tools import clean_text, count_words
```

This creates a clean package interface.

Real World Example: haas.dev

Suppose the haas.dev application grows and has several resource-related features.

You could organize it like:

```
haas/
□
□□□ main.py
□
□□□ resources/
□   □□□ __init__.py
□   □□□ search.py
□   □□□ filtering.py
□   □□□ sorting.py
□
□□□ users/
□   □□□ __init__.py
□   □□□ validation.py
□   □□□ profiles.py
□
□□□ utilities/
    □□□ __init__.py
    □□□ formatting.py
    □□□ dates.py
```

For example, `resources/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 another part of the application can import:

```
from resources.search import search_resources
```

The package structure makes the codebase easier to navigate.

Designing Good Packages

A good package should group things that logically belong together.

Good:

```
payments/
□□□ checkout.py
□□□ refunds.py
□□□ invoices.py
```

These modules all relate to payments.

Less useful:

```
random/
□□□ payment.py
□□□ weather.py
□□□ user.py
□□□ calculator.py
```

The package name should communicate what its contents have in common.

Avoid Giant Packages

A package containing hundreds of unrelated modules can become difficult to navigate.

If a package grows too much, consider dividing it into meaningful subpackages.

For example:

```
application/
□□□ users/
□□□ payments/
□□□ products/
□□□ analytics/
```

instead of:

```
application/
□□□ everything/
```

Keep Package APIs Clear

A package can contain internal implementation details that users do not need to interact with directly.

For example:

```
text_tools/
□□□ __init__.py
□□□ cleaning.py
□□□ analysis.py
```

□□□ _internal.py

The package can expose only the functions that users need.

For example:

```
from .cleaning import clean_text
from .analysis import count_words
```

Then users do not need to know how the internal modules are organized.

Public vs Internal Code

A leading underscore often communicates that something is intended for internal use.

For example:

_internal.py

or:

```
def _helper():
    ...
```

This is a naming convention rather than an absolute security mechanism.

It tells other developers:

> This is implementation detail; avoid depending on it unless necessary.

Package Initialization

Avoid putting expensive or surprising operations inside `__init__.py`.

For example, this is usually a poor idea:

```
# __init__.py

connect_to_database()
download_large_file()
start_application()
```

Importing the package could unexpectedly perform these actions.

Prefer keeping initialization lightweight.

Package Structure and Maintainability

A well-structured package makes it easier to:

- find code
- reuse code
- test individual components
- collaborate with other developers
- replace individual modules
- understand application architecture

Package organization becomes increasingly important as applications grow.

Common Mistakes

1. Confusing Modules and Packages

Remember:

```
calculator.py  module
calculator/    package
```

---

2. Using Poor Package Names

Avoid vague names such as:

```
stuff/
things/
misc/
random/
```

Prefer meaningful names:

```
authentication/
payments/
resources/
users/
```

---

3. Putting Everything in `__init__.py`

`__init__.py` should not become another giant application file.

Use it primarily for package initialization and carefully selected exports.

---

4. Installing Packages Globally

Installing everything globally can cause dependency conflicts between projects.

Use a virtual environment for project-specific dependencies.

---

5. Ignoring Package Versions

Different package versions may have different APIs or behavior.

Record important dependencies and versions.

---

6. Creating Deeply Nested Packages Without a Reason

This:

```
app/
□□□ system/
   □□□ tools/
      □□□ data/
         □□□ processing/
            □□□ helpers/
```

may be unnecessarily complicated.

Create hierarchy when it improves organization, not simply because you can.

Mini Project: Resource Management Package

Create:

```
resource_app/
□
□□□ main.py
□
□□□ resources/
   □□□ __init__.py
   □□□ search.py
   □□□ filtering.py
   □□□ sorting.py
```

search.py

```
def search_resources(resources, keyword):
   """Return resources whose titles contain the keyword."""
   keyword = keyword.lower()

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

filtering.py

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

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

sorting.py

```
def sort_resources(resources):
    """Return resources sorted by title."""
    return sorted(
        resources,
        key=lambda resource: resource["title"]
    )
```

__init__.py

```
from .search import search_resources
from .filtering import get_free_resources
from .sorting import sort_resources
```

main.py

```
from resources import (
    search_resources,
    get_free_resources,
    sort_resources
)

resources = [
    {
        "title": "Python Functions",
        "free": True
    },
    {
        "title": "React Fundamentals",
        "free": False
```

```
    },
    {
        "title": "Python Lists",
        "free": True
    }
]

python_resources = search_resources(resources, "python")

free_resources = get_free_resources(resources)

sorted_resources = sort_resources(resources)

print(python_resources)
print(free_resources)
print(sorted_resources)
```

This project demonstrates:

Package

Modules

Functions

Main application

---

5 Minute Challenge

Create a package called:

```
student_tools/
```

with:

```
student_tools/
□□□ __init__.py
□□□ grades.py
□□□ validation.py
```

In `grades.py`, create:

```
calculate_average()
calculate_grade()
```

In `validation.py`, create:

```
validate_marks()
```

Export the functions through `__init__.py`.

Then use them from:

```
main.py
```

The goal is to practice:

- creating packages
- creating modules
- importing from packages
- using `__init__.py`
- organizing related functionality

Exercises

Exercise 1

Create a package called:

`calculator/`

with:

`basic.py`
`advanced.py`

Put basic operations in one module and advanced operations in the other.

Exercise 2

Create:

`text_tools/`

with:

`cleaning.py`
`analysis.py`

Add useful text-processing functions.

Exercise 3

Create a package:

`user_tools/`

with:

`validation.py`
`formatting.py`

Create functions for validating and formatting user information.

Exercise 4

Use `__init__.py` to expose selected functions from your package.

Instead of:

```
from user_tools.validation import validate_email
```

allow:

```
from user_tools import validate_email
```

Exercise 5

Create a project with at least:

- 2 packages
- 2 modules per package
- 1 main application file

Draw the structure before writing the code.

Mini Quiz

1. What is a Python package?

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

Answer: A

---

2. What file is traditionally associated with a regular Python package?

A. `package.py`
B. `main.py`
C. `__init__.py`
D. `start.py`

Answer: C

---

3. How do you import `add` from `math_tools/basic.py`?

A.

```
import add
```

B.

`from math_tools.basic import add`

C.

`package add`

D.

`load math_tools.add`

Answer: B

---

4. What does this mean?

`from .basic import add`

A. Import from the current package
B. Import from Python's standard library
C. Delete `basic`
D. Create a new package

Answer: A

---

5. What is `pip` commonly used for?

A. Running loops
B. Installing Python packages
C. Creating variables
D. Formatting strings

Answer: B

---

6. Why are virtual environments useful?

A. They make Python syntax shorter
B. They isolate project dependencies
C. They replace modules
D. They automatically write code

Answer: B

Interview Questions

1. 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 structure.

2. What is `__init__.py`?

It is a special file traditionally used to define and initialize a regular Python package. It can also expose selected package functionality.

3. What is the difference between a package and a library?

A package is a specific way of organizing and distributing Python code. A library is a broader term for reusable functionality.

4. What is `pip`?

`pip` is a commonly used Python package installer and package management tool.

5. Why should developers use virtual environments?

They isolate project dependencies so different projects can use different package versions without interfering with each other.

6. What is a third-party package?

A package created outside Python's standard library that can be installed and used in Python projects.

7. What is a relative import?

An import that refers to modules relative to the current package.

Example:

```
from .basic import add
```

8. What is a package API?

The collection of functions, classes, and other functionality that a package intentionally exposes for other code to use.

Package Cheat Sheet

Package structure

```
my_package/
□□□ __init__.py
□□□ module_a.py
□□□ module_b.py
```

Import a module

```
from my_package import module_a
```

Import a function

```
from my_package.module_a import function
```

Relative import

```
from .module_a import function
```

Export through __init__.py

```
from .module_a import function
```

Then:

```
from my_package import function
```

Install a package

```
pip install package_name
```

Install a specific version

```
pip install package_name==1.2.3
```

Upgrade

```
pip install --upgrade package_name
```

Uninstall

```
pip uninstall package_name
```

List installed packages

```
pip list
```

Install dependencies

```
pip install -r requirements.txt
```

Key Takeaways

- A **package** organizes related Python modules.
- A **module** is generally one Python file.
- Packages make large projects easier to organize and maintain.
- `__init__.py` is traditionally used in regular Python packages and can expose selected functionality.
- Packages can contain multiple modules and even subpackages.
- You can import modules or specific functions from packages.
- Relative imports use `.` to refer to the current package.
- `pip` is commonly used to install and manage third-party packages.
- `requirements.txt` can record project dependencies.
- Virtual environments isolate dependencies between projects.
- Good packages group code according to meaningful responsibilities.
- Avoid giant packages, vague package names, unnecessary nesting, and heavy work during package import.
- A useful mental model is:

```
Application
  
Packages
  
Modules
  
Functions / Classes
```

Visit haas.dev for more resources and guides.

haas.dev

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