



PyCon  
Greece

29-30  
August  
2025

Technopolis City of Athens



# Consistent importing

Jan Bielecki





## Jan Bielecki



2005

Game maker  
enthusiast



2015

First Python code  
(internship)



2019

Using Python at  
work



Scan for full presentation  
`consistent_importing_all_slides.pdf`



## Ørsted[1]



>8000

All employees in  
19 countries



~1000

IT employees



50-100

Pythonistas



Scan for full presentation  
consistent\_importing\_all\_slides.pdf

# Agenda

- I. How importing in Python works?
- II. How to import in a consistent way?
- III. How internal dependencies impact a codebase architecture?
- IV. How we can enforce the selected importing strategy in CI?

# Consistent importing



**Why are you  
telling me this?**

## Re-exporting definitions through `__init__.py`

```
# model/geometry/node.py
class Node: ...

# model/geometry/__init__.py
from .node import Node, function_a
from .edge import Edge, function_b

# model/__init__.py
from .geometry import Node, Edge, function_a, function_b
from .other_module import SomeOtherClass

# gui/__init__.py
from model import Node, Edge, function_a, function_b, SomeOtherClass
```

### ✓ Pros:

- Encapsulation
- Single entry point
- Short imports

### ✗ Cons:

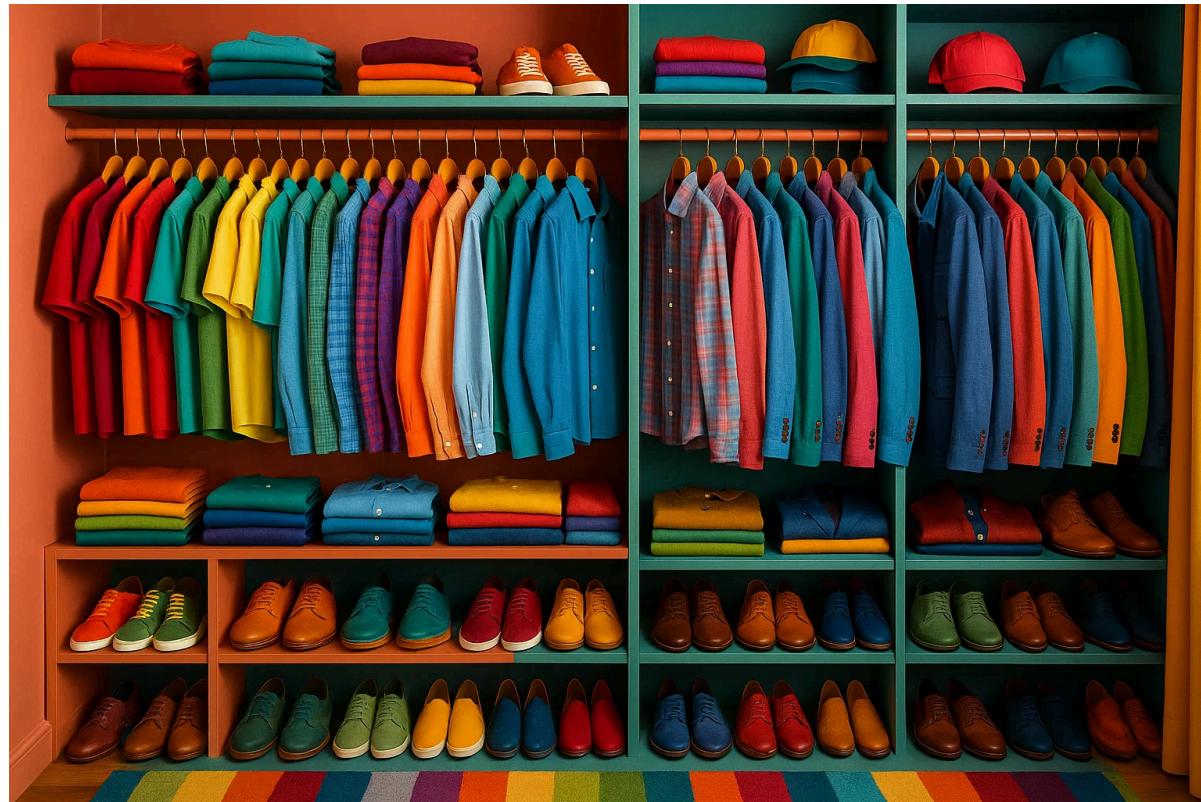
- Maintenance overhead
- IDE is confused
- Hard to automate consistency

## Consistency reduces decision fatigue



"... the more decisions you make throughout the day, the worse you are at making them."<sup>[2]</sup>

# Python importing without consistency



# How importing in Python works?



# Python interpreter

```
(base) jan@jan-Inspiron-3543:~/consistent_importing$ python
Python 3.12.8 [GCC 11.2.0] on linux
>>> def hello_world() -> None:
...     print("Hello world!")
...
>>> hello_world()
Hello world!
>>> exit()
(base) jan@jan-Inspiron-3543:~/consistent_importing$ python
Python 3.12.8 [GCC 11.2.0] on linux
>>> hello_world()
Traceback (most recent call last):
  File "", line 1, in 
NameError: name 'hello_world' is not defined
>>>
```

# Python module

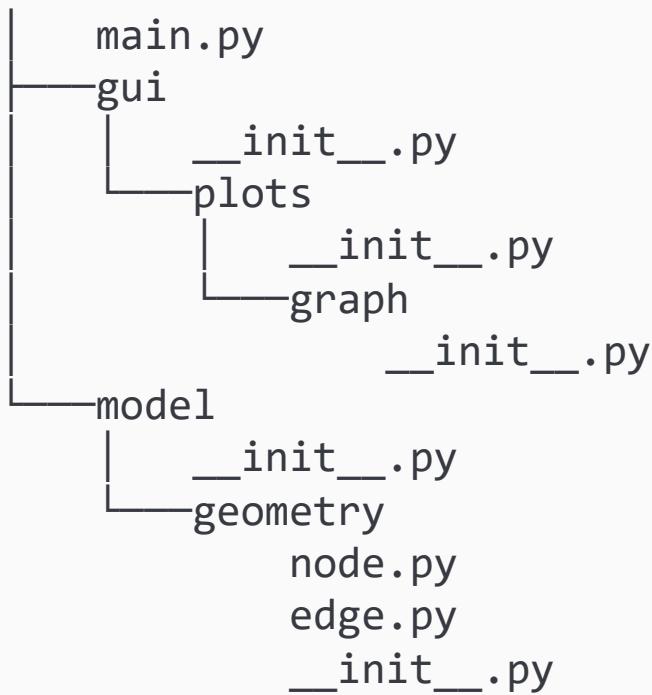
```
# my_module.py
def hello_world() -> None:
    print("Hello world!")

print("Hello from my_module!")
```

```
(base) jan@jan-Inspiron-3543:~/consistent_importing$ python
Python 3.12.8 [GCC 11.2.0] on linux
>>> import my_module
Hello from my_module!
>>> dir(my_module) # properties and methods of the object
[..., '__name__', '__package__', '__spec__', 'hello_world']
>>> my_module.hello_world()
Hello world!
>>>
```

# Python package

```
(base) :~/consistent_importing$ tree /f
```



# Why modular structure?

```
(base) jan@jan-Inspiron-3543:~/> python
Python 3.12.8 [GCC 11.2.0] on linux
>>> def hi() -> None:
...     print("Hello world!")
...
>>> hi()
Hello world!
>>> exit()
(base) jan@jan-Inspiron-3543:~/> python
Python 3.12.8 [GCC 11.2.0] on linux
>>> hi()
Traceback (most recent call last):
  File "", line 1, in 
NameError: name 'hi' is not defined
```

- ✓ **Reusability**
- ✓ Readability
- ✓ Concerns separation
- ✓ Encapsulation
- ✓ Collaboration
- ✓ Maintainability

# Why modular structure?

```
# >>> all in one file: messy
def calculate_distance(p1, p2): ...
def create_node(x, y): ...
def add_edge(graph, n1, n2): ...
def plot_graph(graph): ...
def validate_input(data): ...
# ... 500+ more lines
# >>> organized structure: clean
# model/geometry/node.py
class Node:
    def __init__(self, x, y): ...

# gui/plots/graph.py
class GraphPlotter:
    def plot(self, graph): ...
```

- ✓ Reusability
- ✓ **Readability**
- ✓ Concerns separation
- ✓ Encapsulation
- ✓ Collaboration
- ✓ Maintainability

# Why modular structure?

```
(base) jan@jan-Inspiron-3543:~/ $ tree /f
.
├── main.py
└── gui
    ├── __init__.py
    └── plots
        ├── __init__.py
        └── graph
            └── __init__.py
.
└── model
    ├── __init__.py
    ├── geometry
    │   ├── node.py
    │   ├── edge.py
    │   └── __init__.py
```

```
# running tests inside a specific package
python -m pytest model/geometry
# linting a specific package
ruff check gui/plots
```

- ✓ Reusability
- ✓ Readability
- ✓ **Concerns separation**
- ✓ Encapsulation
- ✓ Collaboration
- ✓ Maintainability

# Why modular structure?

```
(base) jan@jan-Inspiron-3543:~/ $ tree /f
.
├── main.py
└── os_ops
    ├── windows
    │   ├── __init__.py
    │   └── path.py
    └── linux
        ├── __init__.py
        └── path.py
    └── __init__.py
```

```
from os_ops import copy_file

copy_file(
    source="source.txt",
    destination="destination.txt"
)
```

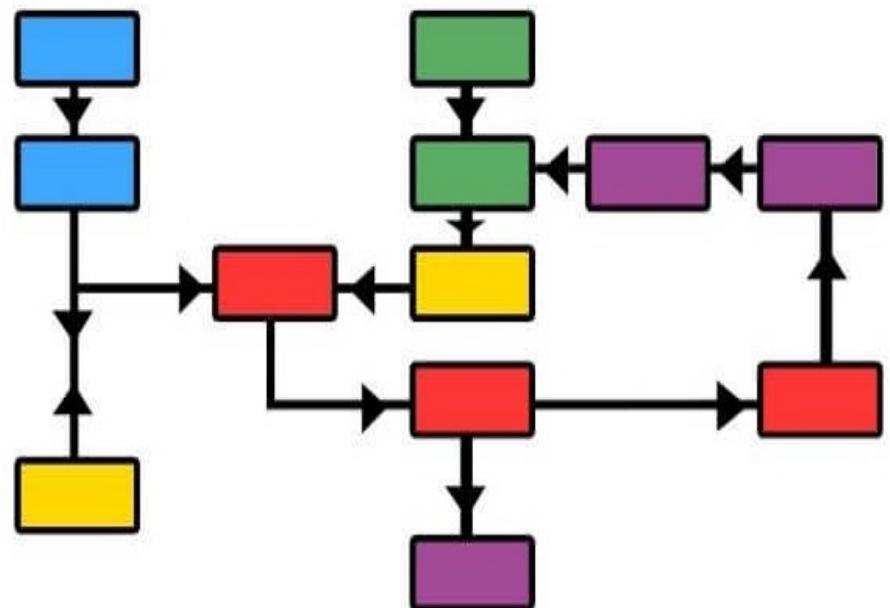
- ✓ Reusability
- ✓ Readability
- ✓ Concerns separation
- ✓ **Encapsulation**
- ✓ Collaboration
- ✓ Maintainability

# Why modular structure?



- ✓ Reusability
- ✓ Readability
- ✓ Concerns separation
- ✓ Encapsulation
- ✓ **Collaboration**
- ✓ Maintainability

# Why modular structure?



- ✓ Reusability
- ✓ Readability
- ✓ Concerns separation
- ✓ Encapsulation
- ✓ Collaboration
- ✓ **Maintainability**

# sys.path

```
(.venv) ~\consistent_importing>python
>>> import sys
>>> from pprint import pprint
>>> pprint(sys.path)
[
    '',
    'C:\\\\Users\\\\JBIEL\\\\Desktop\\\\projects\\\\report-filler',
    'C:\\\\Users\\\\JBIEL\\\\Desktop\\\\projects\\\\consistent_importing',
    'C:\\\\Users\\\\JBIEL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python313\\\\python313.zip',
    'C:\\\\Users\\\\JBIEL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python313\\\\DLLs',
    'C:\\\\Users\\\\JBIEL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python313\\\\Lib',
    'C:\\\\Users\\\\JBIEL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python313',
    'C:\\\\Users\\\\JBIEL\\\\Desktop\\\\projects\\\\consistent_importing\\\\.venv',
    'C:\\\\Users\\\\JBIEL\\\\Desktop\\\\projects\\\\consistent_importing\\\\.venv\\\\Lib\\\\site-packages'
]
```

# Package vs Namespace Package

```
—my_package (/some/path)
```

```
| __init__.py  
|—plot  
| __init__.py  
|—model  
| __init__.py
```

```
...
```

```
—my_package (/some/another_path)
```

```
| __init__.py  
|—plot  
| __init__.py  
|—computation  
| __init__.py
```

```
...
```

```
(.venv) ~\consistent_importing>python
```

```
>>> import sys  
>>> sys.path  
[  
    '/some/path',  
    '/some/another_path'  
]  
>>> import my_package.model  
hello from path.my_package.model  
>>> import my_package.computation  
Traceback (most recent call last):  
  File "", line 1, in  
    import my_package.computation  
ModuleNotFoundError: No module named 'my_package.computation'
```

# Execute a script vs execute a module

```
(.venv) ~\consistent_importing>python -m src.gui_model.main
[
  '~\\consistent_importing',
  ...
  '~\\consistent_importing\\.venv'
]
Success!

(.venv) ~\consistent_importing>python src\gui_model\main.py
[
  '~\\consistent_importing\\src\\gui_model',
  ...
  '~\\consistent_importing\\.venv'
]
Success!
```

```
# src\gui_model\main.py

from pprint import pprint

pprint(sys.path)
print("Success!")
```

```
# src\on_import\module_b.py
print("greetings from module_b.py")
a = 123

# src\on_import\module_a.py
print("hello from module_a.py")

def function_a() -> None:
    import src.on_import.module_b
    print(dir(src.on_import.module_b))

print("goodbye from module_a.py")
# src\on_import\main.py
from src.on_import.module_a import function_a
import inspect

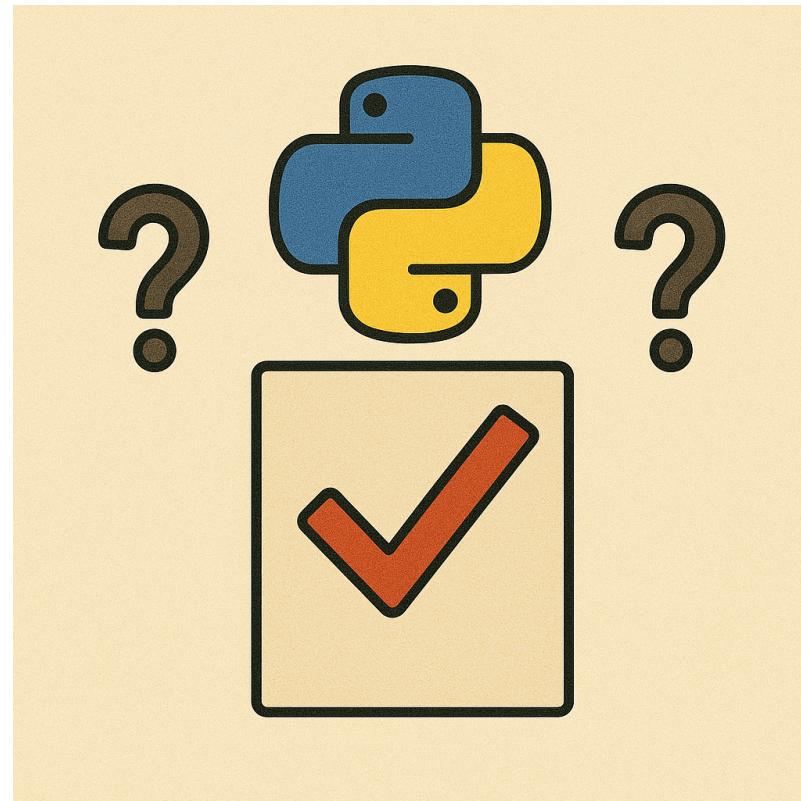
print("hello from main.py\n")
print(inspect.getsource(function_a))
function_a()
print("goodbye from main.py")
```

```
(.venv) >python -m src.on_import.main
hello from module_a.py
goodbye from module_a.py
hello from main.py

def module_function() -> None:
    import src.on_import.module_b
    print(dir(src.on_import.module_b))
```

```
greetings from module_b.py
[..., '__name__', '__package__', 'a']
goodbye from main.py
```

## II. What is the best importing strategy?



## 1. (PEP) Place imports at the top of a file[3]



**Benefit:** Transparency about dependencies

## 2. Lazy imports of heavy modules

```
python -X importtime -c 'import torch' 2> torch-import.prof  
uvx tuna torch-import.prof
```

```
# src/evaluation.py

def evaluate_model(model: Model) -> None:
    import torch # noqa: PLC0415
    ...

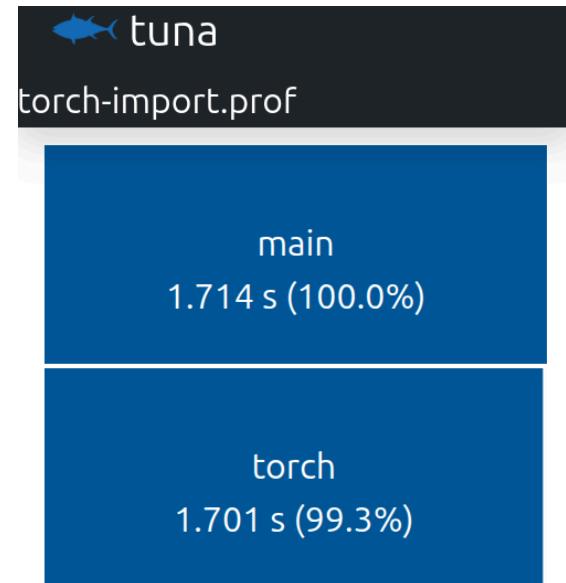
# src/_main_.py

from src.evaluation import evaluate_model

def main() -> None:

    if do_evaluate:
        evaluate_model(model)

    ...
```



**Benefit:** Reduction of startup time at some circumstances

### 3. Do not overuse "if TYPE\_CHECKING:"

```
# src/model.py
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    import torch

class Model:
    def add_module(self, module: "torch.Module") -> None:
        print(f"Adding module: {module}")
        ...
```

**Benefit:** Avoiding circular dependencies

## 4. (PEP) Imports should be grouped

```
# standard Library imports
import filecmp
import shutil
from pathlib import Path

# related third party imports
import pandas as pd
import pytest
import numpy as np

# Local application/library specific imports
from common.constants import current_version
from common.constants import default_model_type
from dashboard.model.model import Model
from dashboard.model.model import ModelType
from source.db import DB
from source.connector.adapter import TimeSeriesAdapter
```

**Benefit:** Immediate visual separation of different types of dependencies

## 5. Import one definition per line



```
import filecmp
import shutil
from pathlib import Path

import pandas as pd
import pytest
import numpy as np

from common.constants import current_version
from common.constants import default_model_type
from dashboard.model.model import Model
from dashboard.model.model import ModelType
from source.db import DB
from source.connector.adapter import TimeSeriesAdapter
```

**Benefit:** Reduction of version control conflicts

## 6. (PEP) Absolute imports

- Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on `sys.path`):

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

```
from . import sibling
from .sibling import example
```

Standard library code should avoid complex package layouts and always use absolute imports.

**Benefit:** More informative and less ambiguous

## 7. Use only well-known abbreviations

```
import os
import sys

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import sklearn as sk
import tensorflow as tf

import windows_temporary_file as wtf
```

**Benefit:** Avoid confusion on what the abbreviation stands for

## 8. Package vs Namespace Package

```
my_package (/some/path)
├── __init__.py
├── plot
│   └── __init__.py
└── model
    └── __init__.py
...
...my_package (/some/another_path)
├── __init__.py
├── plot
│   └── __init__.py
└── computation
    └── __init__.py
...
```

```
(.venv) ~\consistent_importing>python
>>> import sys
>>> sys.path
[
    '/some/path',
    '/some/another_path'
]
>>> import my_package.model
hello from path.my_package.model
>>> import my_package.computation
Traceback (most recent call last):
  File "", line 1, in 
        import my_package.computation
ModuleNotFoundError: No module named 'my_package.computation'
```

**Benefit:** Avoid uncertain behavior of namespace packages

## 9. (PEP) Avoid using wildcard imports

```
# base_implementation.py
def fast_add(a: float, b: float) -> float:
    return a + b

def fast_multiply(a: float, b: float) -> float:
    return a * b

# cython_accelerated_implementation.py (optional module)
def fast_add(a: float, b: float) -> float: ...

# public_api.py
from base_implementation import * # noqa: F403

try: # (cython_accelerated_implementation.py could be missing)
    from cython_accelerated_implementation import * # noqa: F403
except ImportError:
    pass

# main.py
from public_api import fast_add
from public_api import fast_multiply
```

**Benefit:** Avoid ambiguity and namespace pollution

## 10. Do not re-export definition through package (`__init__.py`)

```
(base):~/ $ tree /f
.
├── main.py
└── os_ops
    ├── windows
    │   ├── __init__.py
    │   ├── path.py
    │   ...
    └── linux
        ├── __init__.py
        ├── path.py
        ...
    └── __init__.py
└── __init__.py
```

```
# main.py
from os_ops import copy_file

copy_file(
    "source.txt",
    "destination.txt"
)
```

```
# os_ops/__init__.py

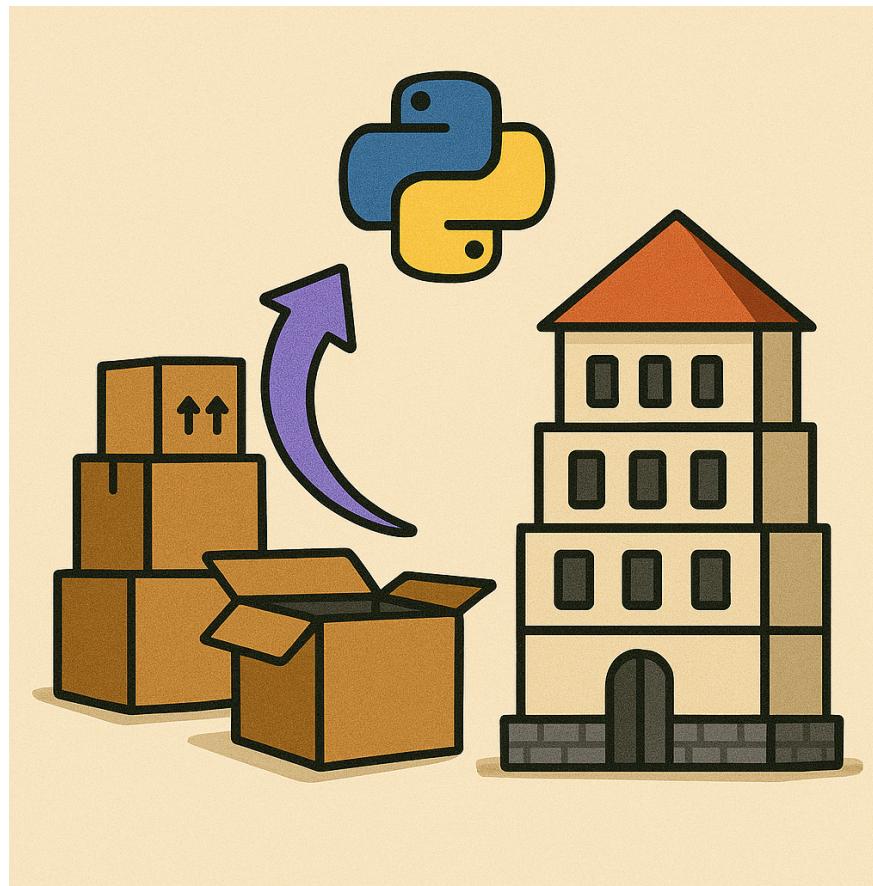
import os

from ._windows.path import copy_file as _windows_copy_file
from ._linux.path import copy_file as _linux_copy_file

def copy_file(source: str, destination: str) -> None:
    print(f"Copying file from {source} to {destination}")

    if os.name == 'nt':
        _windows_copy_file(source, destination)
    elif os.name == 'posix':
        _linux_copy_file(source, destination)
    else:
        raise OSError(f"Unsupported OS: {os.name}")
```

### III. How internal dependencies impact a codebase architecture?

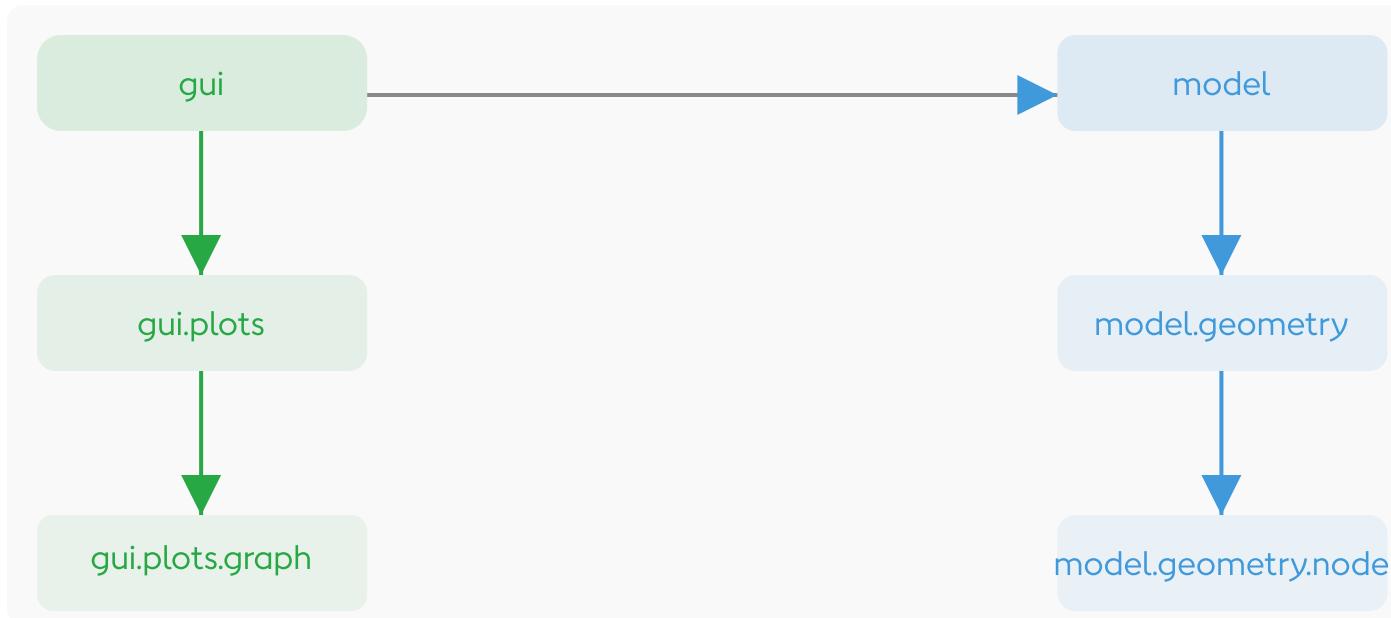


# Python circular imports. A bug or a feature?

```
File "gui/__init__.py", line 1, in
    from model import Node
File "model/__init__.py", line 1, in
    from .geometry import Node
File "model/geometry/__init__.py", line 3, in
    from gui import Graph
^^^^^^^^^^^^^^^^^^^^^^^^^
ImportError: cannot import name 'Graph' from partially initialized module 'gui'
(most likely due to a circular import) (/gui/__init__.py)
```

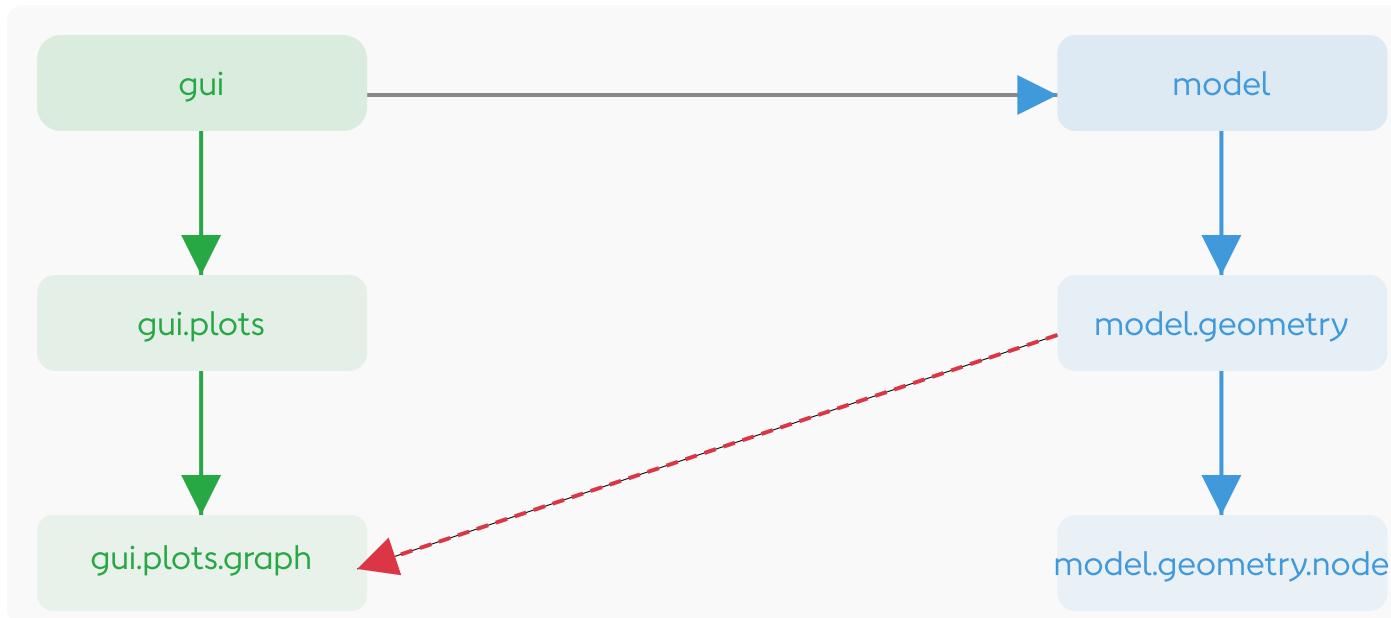
Circular imports often signal a design issue, but sometimes are used intentionally for plugin systems or late binding.

## Circular dependencies between packages



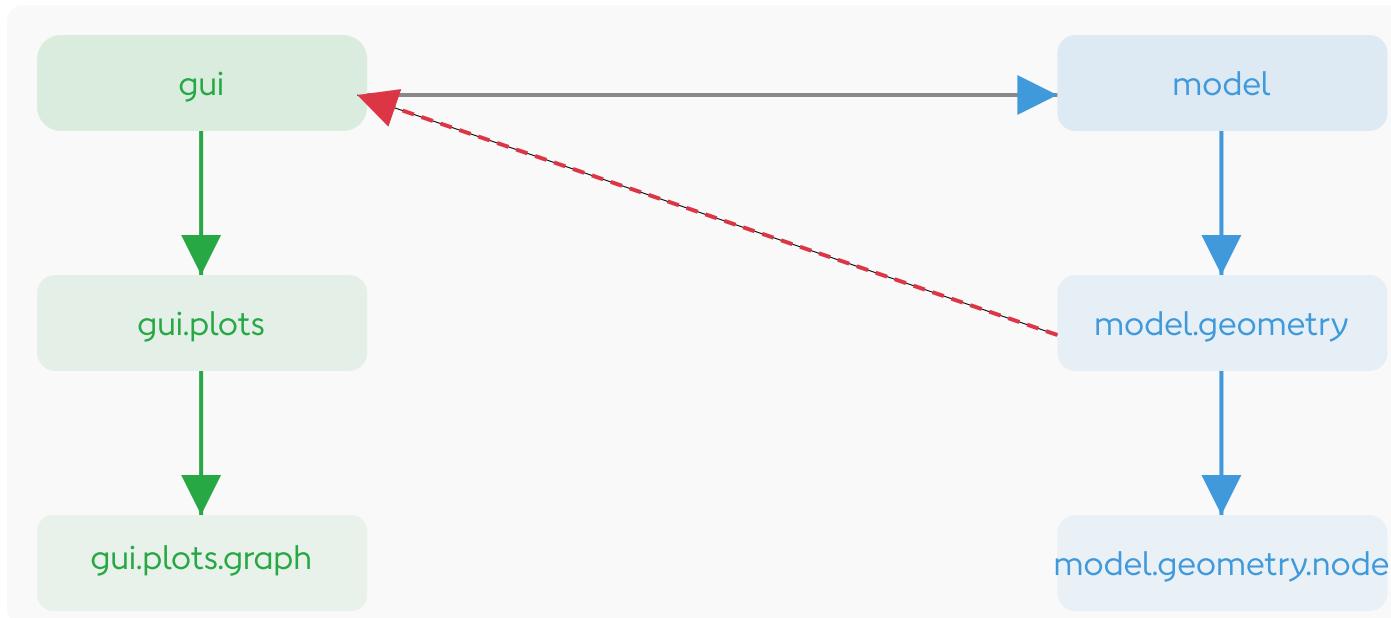
Packages "model" and "gui" re-exporting everything through `__init__.py`.  
Package "gui" imports from "model".

## Circular dependencies between packages



Attempting to add a dependency from "model" to "gui" directly from the module "gui.plots.graph".

## Circular dependencies between packages



Attempting to add a dependency from "model" to "gui", using re-exported entity, creates a circular dependency.

# Circular dependencies between packages

```
# model/geometry/node.py
class Node: ...

# model/geometry/__init__.py
from .node import Node

# model/__init__.py
from .geometry import Node

# gui/__init__.py
from model import Node

# model/geometry/__init__.py
from gui import Graph
```

```
File "gui/__init__.py", line 1, in 
  from model import Node
File "model/__init__.py", line 1, in 
  from .geometry import Node
File "model/geometry/__init__.py", line 3, in 
  from gui import Graph
^^^^^^^^^^^^^^^^^^^^^^^^^^^

ImportError: cannot import name 'Graph' from partially initialized module 'gui'
(most likely due to a circular import) (/gui/__init__.py)
```

## (EXTRA) Python code that can adapt to changing requirements





## Spaghetti Code

Code without structure



## Lasagna Code

Code with layers



## Ravioli Code

Code with many modules



## Pizza Code

Code in a single layer

Using Python, it is easy to cook some italian pasta code [4]

# Clean code principles

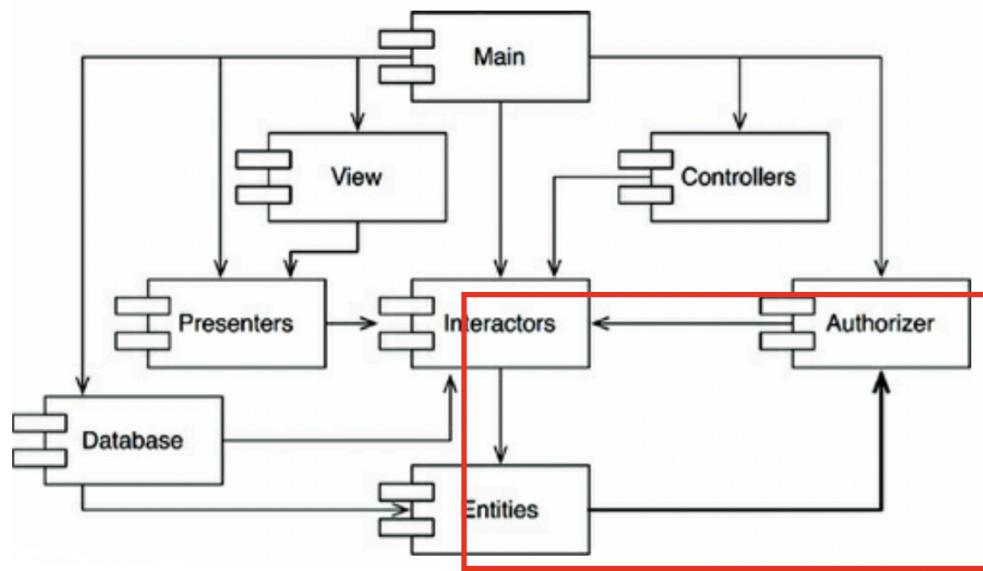


**Uncle Bob**

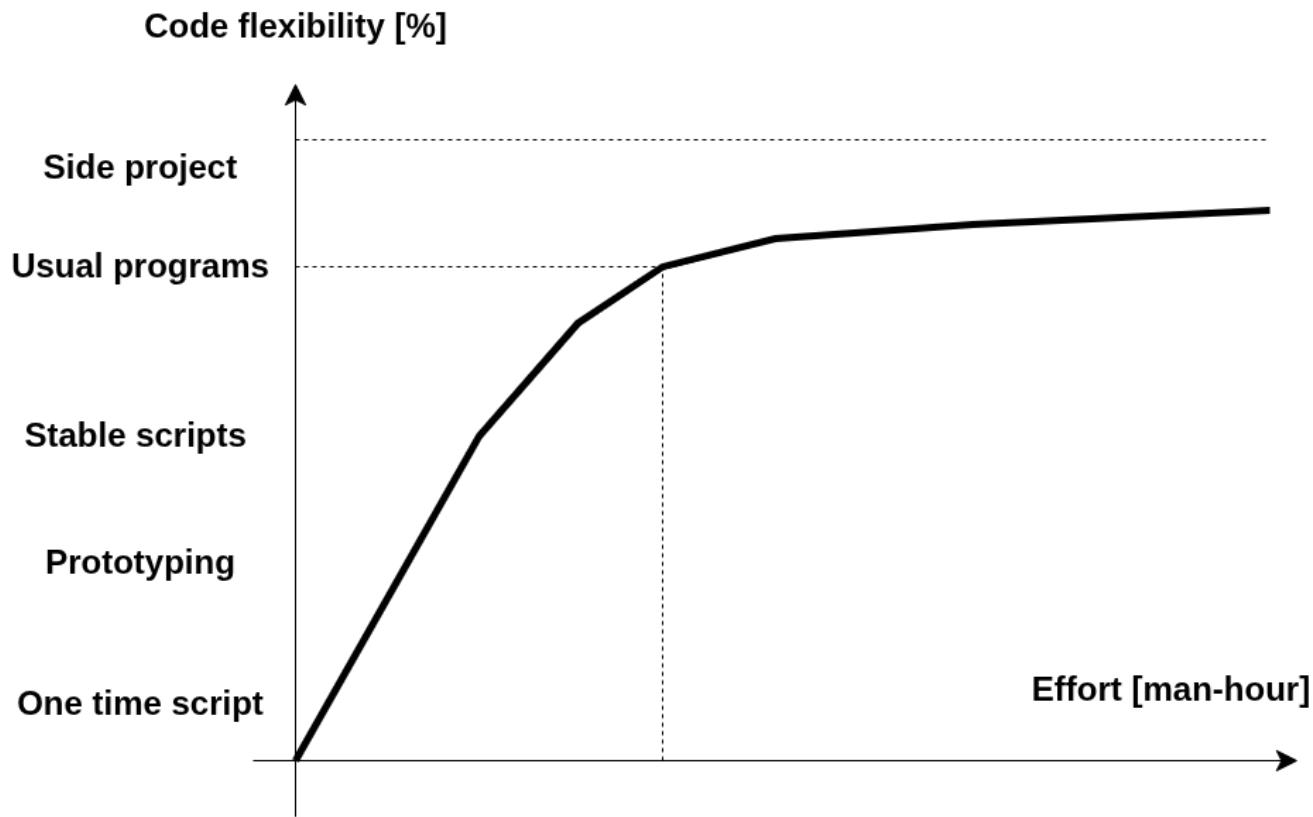
*Clean Code: A Handbook of Agile Software Craftsmanship*  
2008

- **SRP** - Single Responsibility Principle
- **OCP** - Open/Closed Principle
- **LSP** - Liskov Substitution Principle
- **ISP** - Interface Segregation Principle
- **DIP** - Dependency Inversion Principle
- **DRY** - Don't Repeat Yourself
- **KISS** - Keep It Simple, Stupid
- **YAGNI** - You Aren't Gonna Need It
- **POLA** - Principle of Least Astonishment
- **ADP** - Acyclic Dependencies Principle

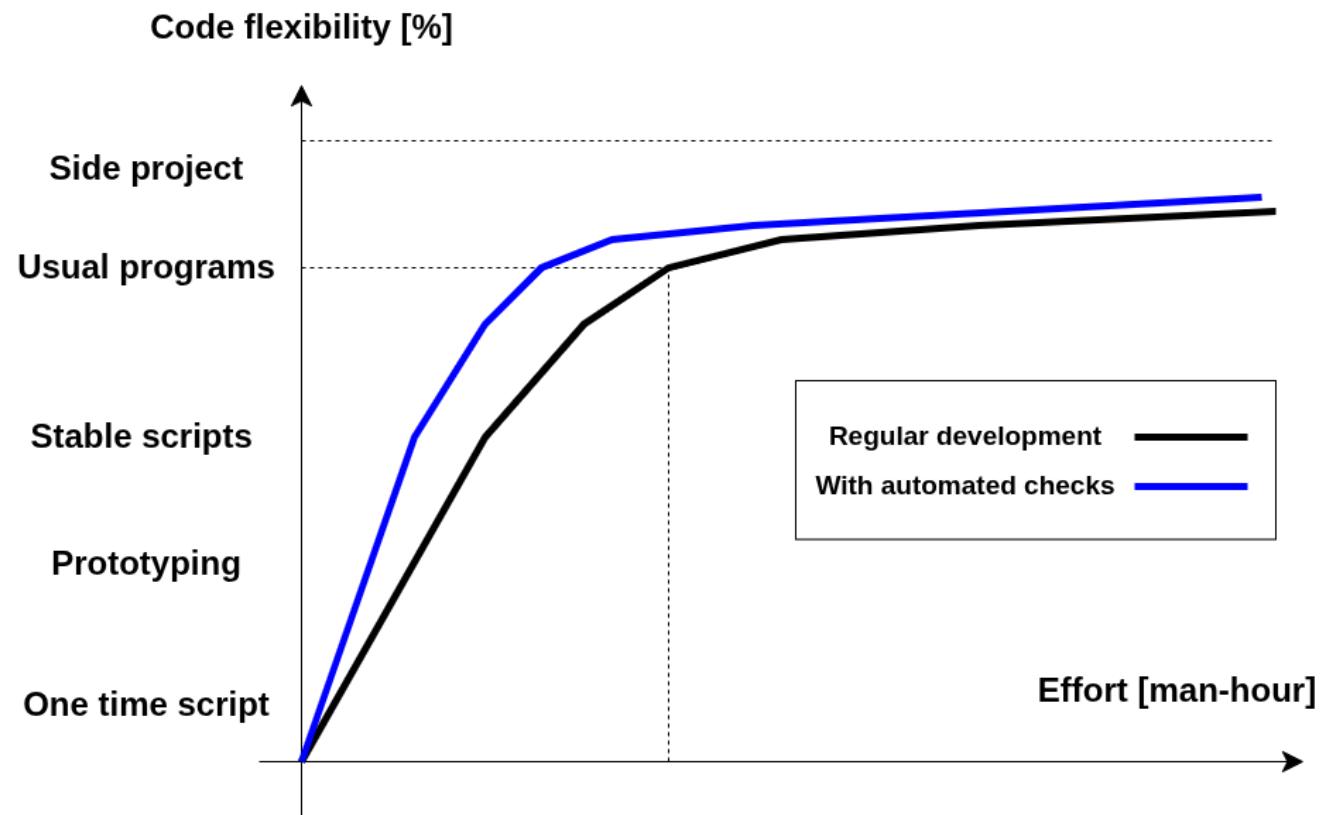
# Acyclic dependencies principle (ADP)



- make the code hard to understand, thus reducing maintainability
- harder to change one component without affecting another because of strong coupling
- impossible to test each individual component because they can't be separated since they act as a single component

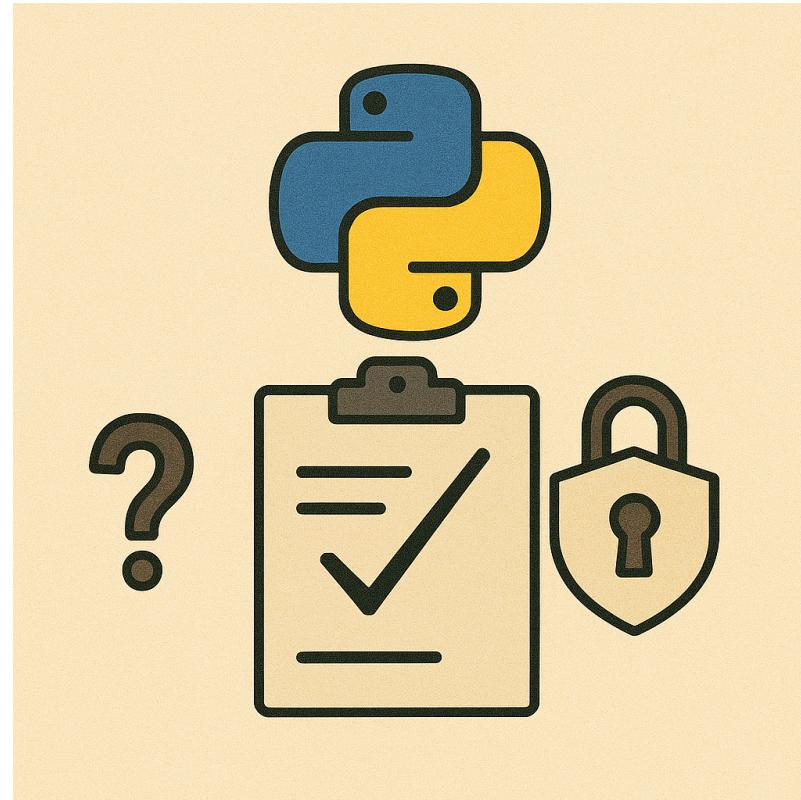


Effort vs benefit of applying clean code principles



Automated checks of clean code principles violations

## IV. How can we enforce the selected strategy in CI?



# Ruff

```
from functools import partial
import sys

import pandas as pd
from pydantic import BaseModel
from pydantic import Field

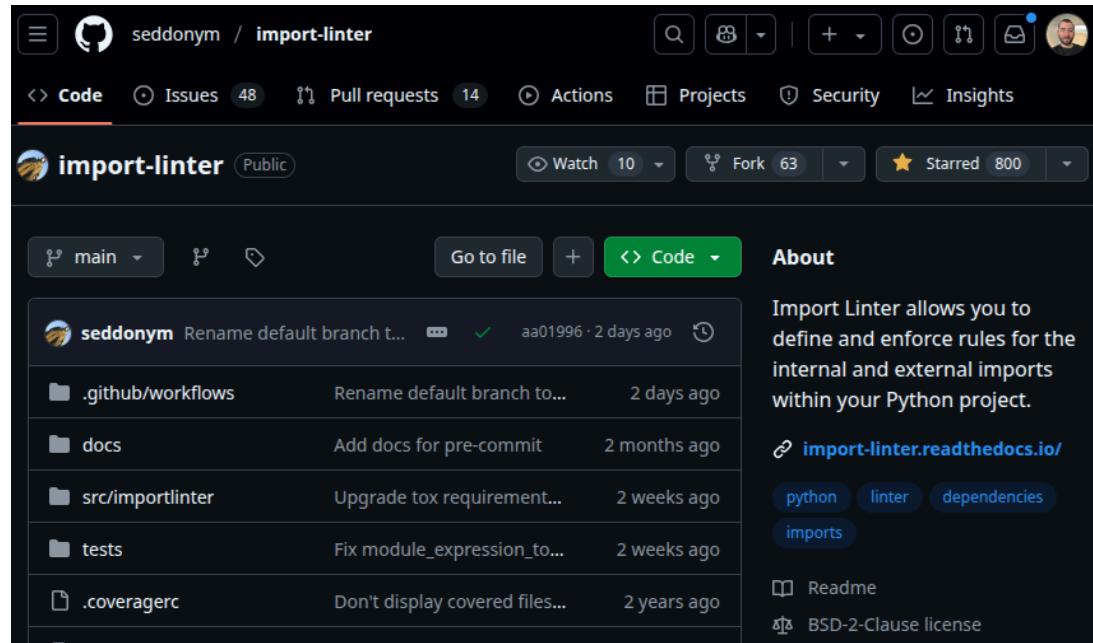
from my_another_package.graph import Graph
from my_another_package.graph import Node
from my_package.model import model

# ruff.toml

[lint]
select = [
    "I", # isort
    "PLC0415", # import-outside-top-level
    "TID252", # relative-imports
    "F403", # undefined-Local-with-import-star-usage
]

[lint.isort]
force-single-line = true
known-first-party = ["my_another_package"]
```

# Import Linter[5]



*"Import Linter allows you to define and enforce rules for the internal and external imports within your Python project."*

```
# .importlinter

[importlinter]
root_package = myproject

[importlinter:contract:1]
name=Foo cannot import bar or baz
type=forbidden
source_modules=
    myproject.foo
forbidden_modules=
    myproject.bar
    myproject.baz
```

```
(myproject) jan@jan-Inspiron-3543:~$ lint-imports
=====
Import Linter
=====

-----
Contracts
-----

Analyzed 134 files, 699 dependencies.
-----

Foo cannot import bar or baz KEPT

Contracts: 1 kept, 0 broken.
```

# AcyclicContract

```
pip install import-linter@git+https://github.com/K4liber/import-linter@issue/221_tree_contract
```

```
# .importlinter

[importlinter]
root_packages =
    django

[importlinter:contract:1]
name=Acyclic
type=acyclic
consider_package_dependencies=true
max_cycle_families=-1
exclude_parents=
    django.test
```

```
(django) jan@jan-Inspiron-3543:~$ lint-imports
```

# Check upholding of AcyclicContract in django

```
...
>>> Cycles family for parent module 'django'

Siblings:
(
    django.apps
    django.core
)
Number of cycles: 1

Cycle 1 (package dependency):

(
    -> django.apps (full path: 'django.apps.config')
    -> django.core (full path: 'django.core.exceptions')
    -> django.core (full path: 'django.core.checks.model_checks')
    -> django.apps
)
<<< Cycles family for parent module 'django'

Number of cycle families found for a contract 'AcyclicDependencies': 167
Number of cycle families with package dependencies: 88
```

## Acyclic dependencies



*Could a development have been more fun for django developers if they had used AcyclicContract from the start of the development?*

# Bibliography

1. Photo by Lester Hsu, <https://orsted.com/en/media/news/2022/04/20220421515811>
2. Why do successful people wear the same outfits every day?,  
<https://www.todaywellspent.com/blogs/articles/why-do-successful-people-wear-the-same-outfits-every-day>
3. PEP-0008, Imports, <https://peps.python.org/pep-0008/#imports>
4. Italian pasta code, Lynn (Yu Ling) Wang, <https://www.linkedin.com/in/yuling-wang-jobmenta/>
5. Import Linter, <https://github.com/seddonym/import-linter>