PyCon Greece
29-30 August 2025
Technopolis City of Athens

Consistent importing

Jan Bielecki

Intro

Jan Bielecki

Senior Full Stack Developer at Ørsted with 6 years of professional experience in software development. Interested in science, digitalization, electrification, sports, music and comedy.

Scan for full presentation

consistent_importing_all_slides.pdf

Ørsted

Ørsted is a Danish multinational power company and a global leader in renewable energy. The company is committed to sustainability and aims to create a world that runs entirely on green energy.[1]

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

Re-exporting definitions through __init__.py

```python
# 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 ambiguous suggestions
- Hard to automate consistency

Consistency reduces decision fatigue

"the more decisions you make throughout the day, the worse you are at making them"[2]

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

```python
# 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)  # all properties and methods of the object
[..., '__name__', '__package__', '__spec__', 'hello_world']
>>> my_module.hello_world()
Hello world!
>>>
```

Python package

```
(base) jan@jan-Inspiron-3543:~/consistent_importing$ tree /f

│   main.py
├───gui
│   │   __init__.py
│   └───plots
│       │   __init__.py
│       └───graph
│               __init__.py
└───model
    │   __init__.py
    └───geometry
            node.py
            edge.py
            __init__.py
```

Why modular structure?

```
(base) jan@jan-Inspiron-3543:~/$ 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:~/$ 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
>>>
```

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

Why modular structure?

```python
# >>>>> Without modularity - all in one file
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
# >>>>> With modularity - organized structure
# 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
- ✓ 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
- ✓ Maintainability

Why modular structure?

```
(base) jan@jan-Inspiron-3543:~/$ tree /f
|   main.py
├── os
|   ├── _windows
|   |   ┬  __init__.py
|   |   |  path.py
|   |   |  ...
|   |
|   ├── _linux
|   |   ┬  __init__.py
|   |   |  path.py
|   |   |  ...
|   ├── __init__.py
|   ├── path.py
|   └──   ...
└── __init__.py
```

```python
from os.path import copy_file


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

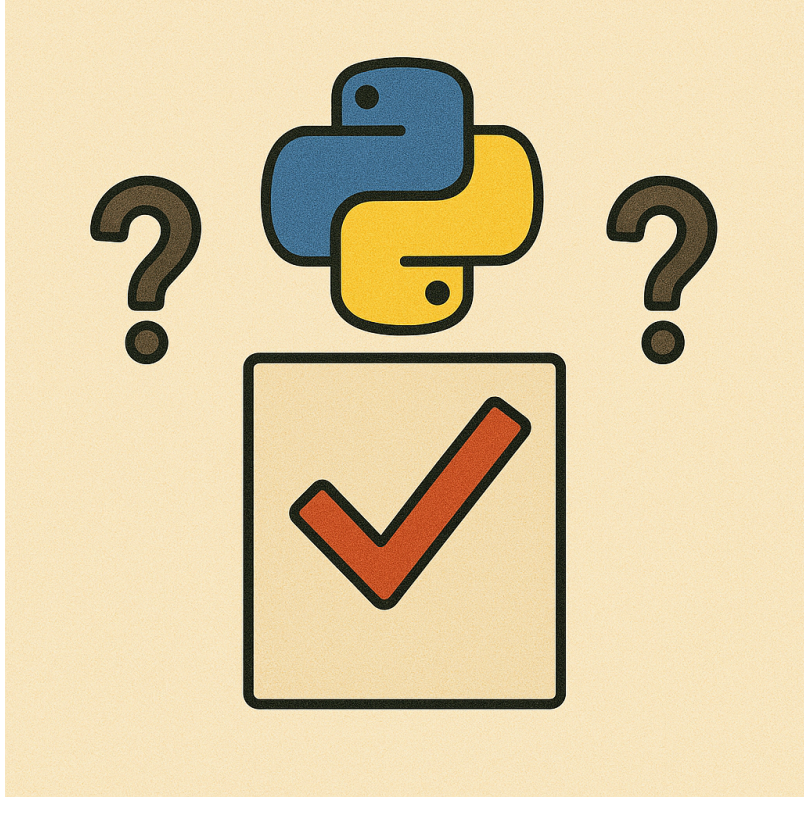
- ✓ Reusability
- ✓ Readability
- ✓ Concerns separation
- ✓ **Encapsulation**
- ✓ Maintainability

Why modular structure?

TODO sth here about dependencies management?

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

II. What is the best importing strategy?



1. Import one definition per line

TODO meme here ... - reduction of version control conflicts

2. Imports at the top of the file

TODO ... - clear and evident set of dependencies

3. Lazy imports of heavy modules

TODO something about "performance optimization" and exception of "imports at the top of the file" - improve startup time

4. Absolute imports

TODO ... - avoiding ambiguity (we can be sure what we import) - IDEs are handling absolute imports better - (exception) relative imports in an encapsulated package - recommendation: use absolute imports by default

5. from module import definition

TODO text here ... - the most clear and evident way to import specific definition - recommendation: use from module import definition by default

6. import module as abbreviation

TODO ... - numpy as np, pandas as pd, matplotlib.pyplot as plt - recommendation: use import module as abbreviation for well-known abbreviations of libraries

7. Package vs namespace

TODO text here ... - what is the difference between package and namespace? - naming matters -> maybe example with "utils" package in ori? - recommendation: use packages as default, namespaces only when needed

8. Wildcard imports

TODO text here ... - avoiding namespace pollution - avoiding ambiguity - recommendation: do not use wildcard imports at all

9. from package import definition (from __init__.py)

TODO ... add a code snippet with re-exporting definitions through __init__.py and renaming from private to public (`from ._windows import _copy_file as copy_file`) - recommendation: use for an encapsulated logic

10. Typing, if TYPE_CHECKING

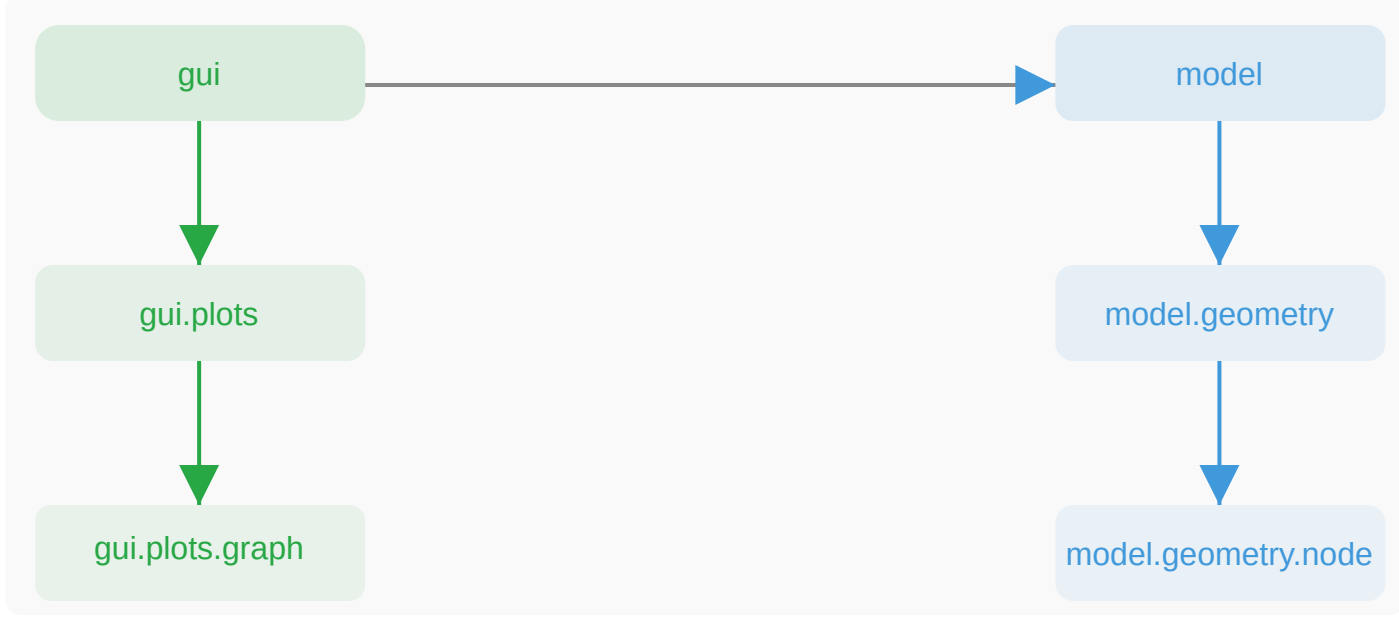
TODO ... Code snippet with typing and if TYPE_CHECKING - recommendation: solve your circular dependencies and do not use

III. How internal dependencies impact a codebase architecture?

Python circular imports -> bug or a feature?

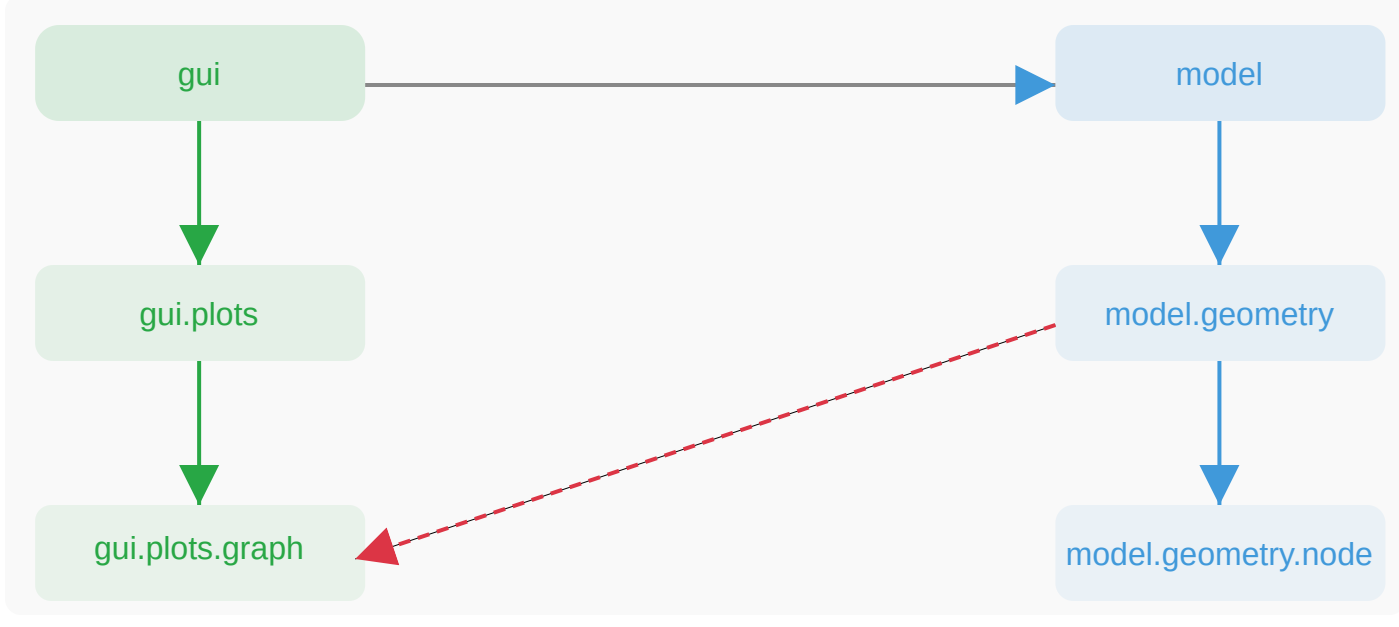
Circular imports can be both a bug and a feature: they often signal a design issue, but sometimes are used intentionally for plugin systems or late binding. In most cases, refactoring to avoid them leads to clearer architecture.

Circular dependencies between packages



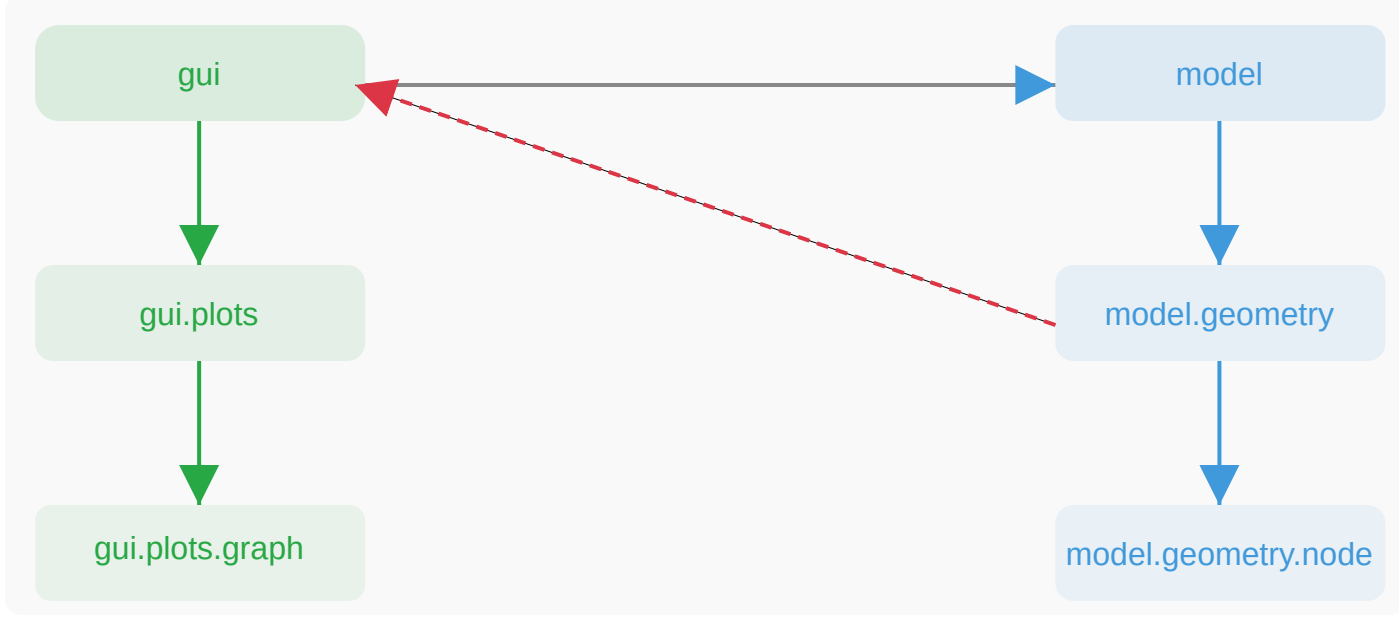
Packages "model" and "gui" re-exporting everything through __init__.py.
Package "gui" imports from "model".

Circular dependencies between packages



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

Circular dependencies between packages

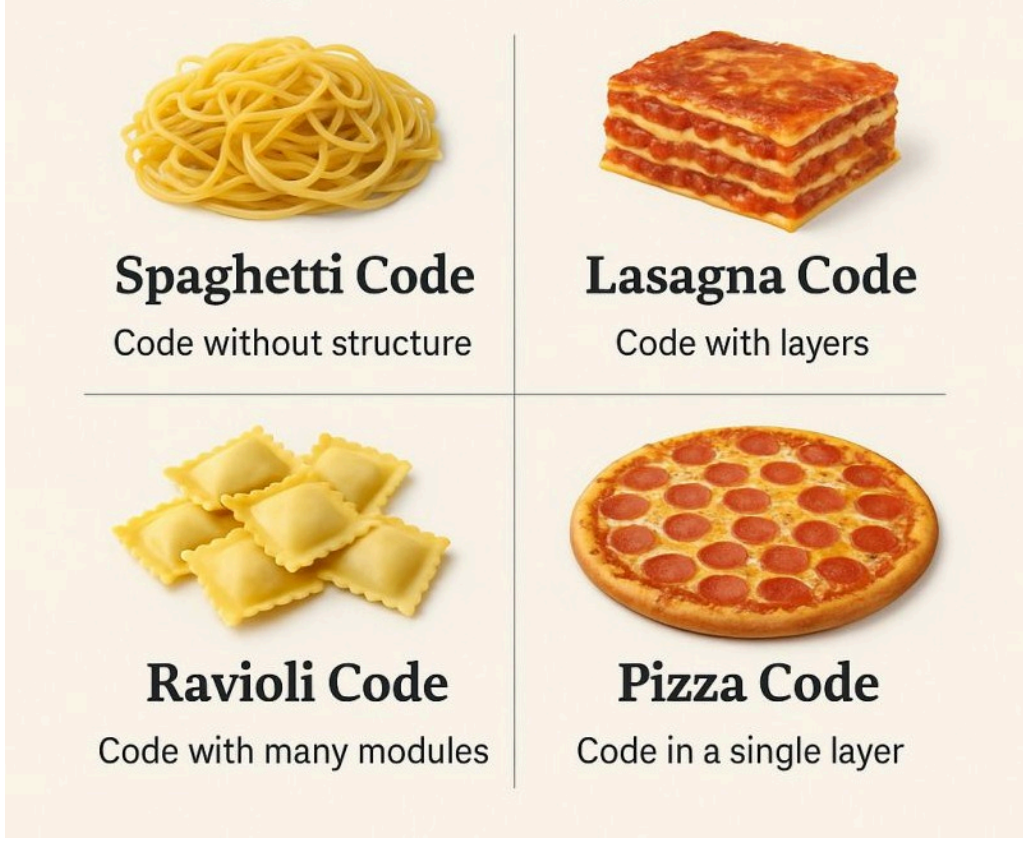


Attempting to add a dependency from "model" to "gui", using re-exported entity, creates a circular dependency.

Circular dependencies between packages

```python
# 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)
```



Using Python, it is easy to cook some italian pasta code [3]

Clean code principles

TODO ... content

Acyclic dependencies principle (ADP)

TODO content ...

Effort vs benefit of applying clean code principles

TODO ... content

Automated checks of clean code principles violations

TODO ... content

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

Ruff

TODO text here ...

Import Linter

TODO text here ...

AcyclicContract

TODO text here ...

Check upholding of AcyclicContract in django

TODO text here ...

Bibliography

1. Photo by Lester Hsu, https://orsted.com/en/media/news/2022/04/20220421515811
2. https://www.todaywellspent.com/blogs/articles/why-do-successful-people-wear-the-same-outfits-every-day
3. Italian pasta code, Lynn (Yu Ling) Wang, https://www.linkedin.com/in/yuling-wang-jobmenta/