



PyCon
Greece

29-30
August
2025

Technopolis City of Athens



Consistent importing

Jan Bielecki





Jan Bielecki

Senior Full Stack Developer



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]

Global leader in offshore wind



~8000

All employees in
19 countries



~1000

IT employees



~100

Pythonistas



Scan for full presentation
consistent_importing_all_slides.pdf

Agenda

1. Why am I talking about this?
2. How importing in Python works?
3. How to import in a consistent way?
4. How imports impact architecture?
5. How to configure related CI tools?

I. Why am I talking about this?



**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

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

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

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

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."^[2]

Python importing without consistency



II. 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 from hello_world function!")
...
>>> hello_world()
Hello from hello_world function!
>>> 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 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 from hello_world function!")
...
>>> hello_world()
Hello from hello_world function!
>>> 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 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 from hello_world function!")
...
>>> hello_world()
Hello from hello_world function!
>>> 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 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 from hello_world function!")
...
>>> hello_world()
Hello from hello_world function!
>>> 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 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 from hello_world function!")
...
>>> hello_world()
Hello from hello_world function!
>>> 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 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 from hello_world function!")
...
>>> hello_world()
Hello from hello_world function!
>>> 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 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 from hello_world function!")
...
>>> hello_world()
Hello from hello_world function!
>>> 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 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 from hello_world function!")
...
>>> hello_world()
Hello from hello_world function!
>>> 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 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 from hello_world function!")
...
>>> hello_world()
Hello from hello_world function!
>>> 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

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 print("Hello from my_module!")
```

Python module

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 print("Hello from my_module!")
```

Python module

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 print("Hello from my_module!")
```

Python module

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 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 from hello_world function!
```

Python module

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 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 from hello_world function!
```

Python module

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 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 from hello_world function!
```

Python module

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 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 from hello_world function!
```

Python module

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 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 from hello_world function!
```

Python module

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 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 from hello_world function!
```

Python module

```
1 # my_module.py
2 def hello_world() -> None:
3     print("Hello from hello_world function!")
4
5 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 from hello_world function!
```

Python package

```
main.py
gui
  __init__.py
  plots
    __init__.py
    graph
      __init__.py
model
  __init__.py
  geometry
    node.py
    edge.py
    __init__.py
```

sys.path

```
(.venv) ~\consistent_importing> python
>>> import sys
>>> print(sys.path)
[
    '',
    'C:\\\\Desktop\\\\projects\\\\consistent_importing',
    'C:\\\\Local\\\\Programs\\\\Python\\\\Python313\\\\DLLs',
    'C:\\\\Local\\\\Programs\\\\Python\\\\Python313\\\\Lib',
    'C:\\\\Desktop\\\\projects\\\\consistent_importing\\\\.venv\\\\Lib\\\\site-packages',
    ...
]
```

sys.path

```
(.venv) ~\consistent_importing> python
>>> import sys
>>> print(sys.path)
[
    '',
    'C:\\\\Desktop\\\\projects\\\\consistent_importing',
    'C:\\\\Local1\\\\Programs\\\\Python\\\\Python313\\\\DLLs',
    'C:\\\\Local1\\\\Programs\\\\Python\\\\Python313\\\\Lib',
    'C:\\\\Desktop\\\\projects\\\\consistent_importing\\\\.venv\\\\Lib\\\\site-packages',
    ...
]
```

sys.path

```
(.venv) ~\consistent_importing> python
>>> import sys
>>> print(sys.path)
[
    '',
    'C:\\\\Desktop\\\\projects\\\\consistent_importing',
    'C:\\\\Local\\\\Programs\\\\Python\\\\Python313\\\\DLLs',
    'C:\\\\Local\\\\Programs\\\\Python\\\\Python313\\\\Lib',
    'C:\\\\Desktop\\\\projects\\\\consistent_importing\\\\.venv\\\\Lib\\\\site-packages',
    ...
]
```

sys.path

```
(.venv) ~\consistent_importing> python
>>> import sys
>>> print(sys.path)
[
    '',
    'C:\\\\Desktop\\\\projects\\\\consistent_importing',
    'C:\\\\Local1\\\\Programs\\\\Python\\\\Python313\\\\DLLs',
    'C:\\\\Local1\\\\Programs\\\\Python\\\\Python313\\\\Lib',
    'C:\\\\Desktop\\\\projects\\\\consistent_importing\\\\.venv\\\\Lib\\\\site-packages',
    ...
]
```

sys.path manipulation (execute a module vs execute a script)

```
(.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!")
```

sys.path manipulation (execute a module vs execute a script)

```
(.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!")
```

sys.path manipulation (execute a module vs execute a script)

```
(.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!")
```

sys.path manipulation (execute a module vs execute a script)

```
(.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!")
```

sys.path manipulation (if the previous approach is not enough)

```
(.venv) ~\consistent_importing> PYTHONPATH="/example:anything-can-be-here" python
>>> import sys; print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 ...]
>>> sys.path.append(True); print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 True
 ...]
```

sys.path manipulation (if the previous approach is not enough)

```
(.venv) ~\consistent_importing> PYTHONPATH="/example:anything-can-be-here" python
>>> import sys; print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 ...]
>>> sys.path.append(True); print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 True
 ...]
```

sys.path manipulation (if the previous approach is not enough)

```
(.venv) ~\consistent_importing> PYTHONPATH="/example:anything-can-be-here" python
>>> import sys; print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 ...]
>>> sys.path.append(True); print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 True
 ...]
```

sys.path manipulation (if the previous approach is not enough)

```
(.venv) ~\consistent_importing> PYTHONPATH="/example:anything-can-be-here" python
>>> import sys; print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 ...]
>>> sys.path.append(True); print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 True
 ...]
```

sys.path manipulation (if the previous approach is not enough)

```
(.venv) ~\consistent_importing> PYTHONPATH="/example:anything-can-be-here" python
>>> import sys; print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 ...]
>>> sys.path.append(True); print(sys.path)
['',
 '/example',
 '/home/projects/consistent_importing/anything-can-be-here'
 '/home/projects/consistent_importing/.venv/lib/python3.12/site-packages',
 True
 ...]
```

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'
```

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'
```

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'
```

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'
```

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'
```

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'
```

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'
```

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
```

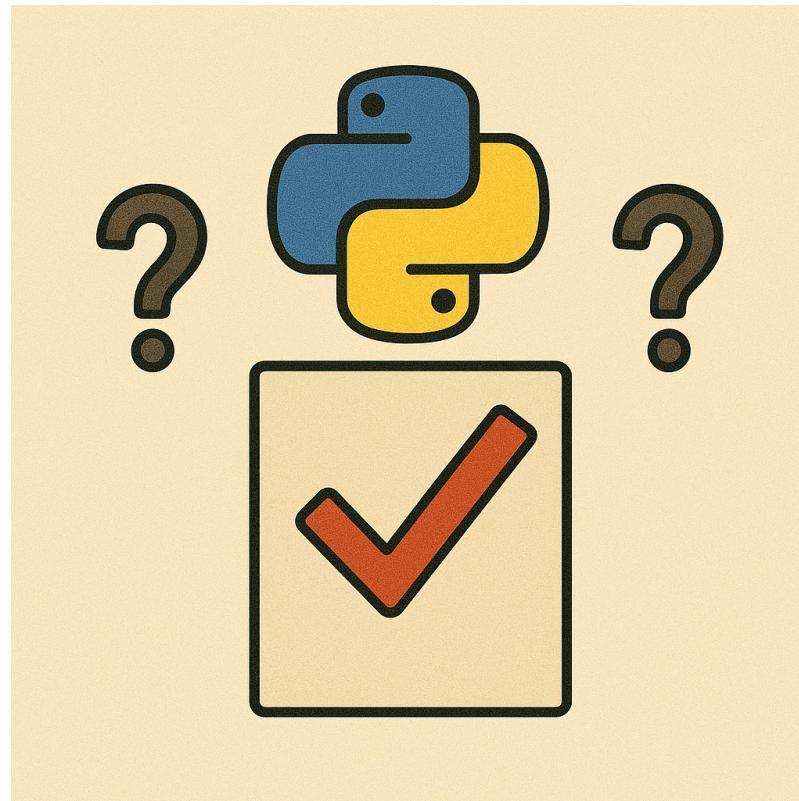
```
Traceback (most recent call last):
```

```
  File "", line 1, in
```

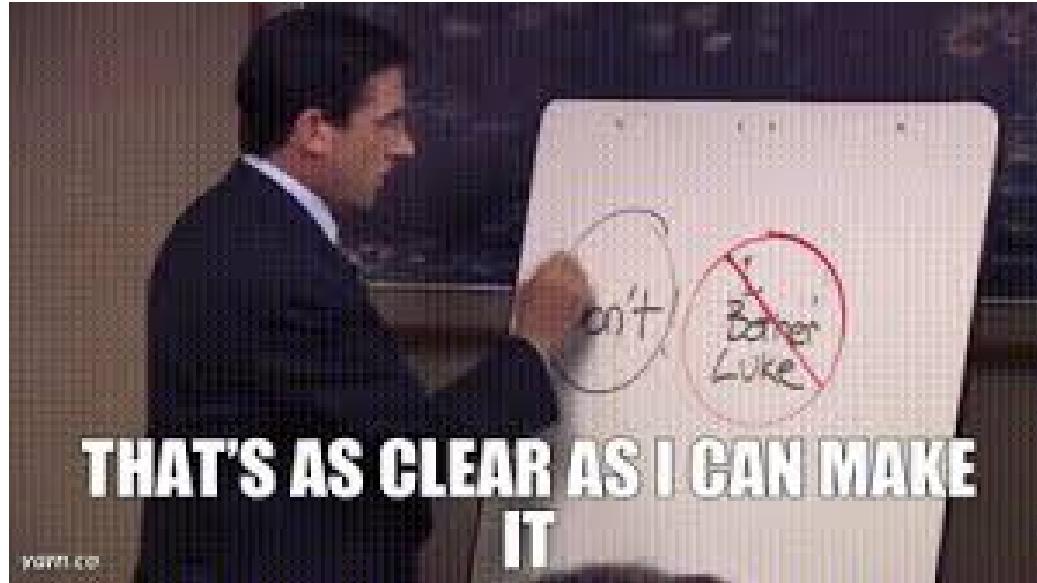
```
    import my_package.computation
```

```
ModuleNotFoundError: No module named 'my_package.computation'
```

III. 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
    ...

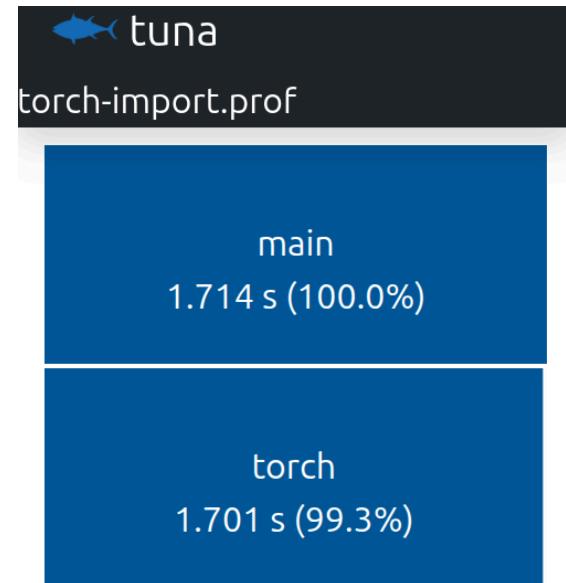
# src/_main_.py

from src.evaluation import evaluate_model

def main() -> None:

    if do_evaluate:
        evaluate_model(model)

    ...
```



Benefit: Reduction of startup time at some conditions

3. Do not use "if TYPE_CHECKING" to avoid ImportError

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

# Lazy import to reduce startup time
if TYPE_CHECKING:
    import torch

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

Benefit: Circular imports most of the time signal a design issue. Fix them to improve the design.

4. (PEP) Imports should be grouped

```
1 # standard library imports
2 import filecmp
3 import shutil
4 from pathlib import Path
5
6 # related third party imports
7 import pandas as pd
8 import pytest
9 import numpy as np
10
11 # Local application/library specific imports
12 from common.constants import current_version
13 from common.constants import default_model_type
14 from dashboard.model.model import Model
15 from dashboard.model.model import ModelType
16 from source.db import DB
17 from source.connector.adapter import TimeSeriesAdapter
```

Benefit: Immediate visual separation of different types of dependencies

4. (PEP) Imports should be grouped

```
1 # standard library imports
2 import filecmp
3 import shutil
4 from pathlib import Path
5
6 # related third party imports
7 import pandas as pd
8 import pytest
9 import numpy as np
10
11 # Local application/library specific imports
12 from common.constants import current_version
13 from common.constants import default_model_type
14 from dashboard.model.model import Model
15 from dashboard.model.model import ModelType
16 from source.db import DB
17 from source.connector.adapter import TimeSeriesAdapter
```

Benefit: Immediate visual separation of different types of dependencies

4. (PEP) Imports should be grouped

```
1 # standard library imports
2 import filecmp
3 import shutil
4 from pathlib import Path
5
6 # related third party imports
7 import pandas as pd
8 import pytest
9 import numpy as np
10
11 # Local application/library specific imports
12 from common.constants import current_version
13 from common.constants import default_model_type
14 from dashboard.model.model import Model
15 from dashboard.model.model import ModelType
16 from source.db import DB
17 from source.connector.adapter import TimeSeriesAdapter
```

Benefit: Immediate visual separation of different types of dependencies

4. (PEP) Imports should be grouped

```
1 # standard library imports
2 import filecmp
3 import shutil
4 from pathlib import Path
5
6 # related third party imports
7 import pandas as pd
8 import pytest
9 import numpy as np
10
11 # Local application/library specific imports
12 from common.constants import current_version
13 from common.constants import default_model_type
14 from dashboard.model.model import Model
15 from dashboard.model.model import ModelType
16 from source.db import DB
17 from source.connector.adapter import TimeSeriesAdapter
```

Benefit: Immediate visual separation of different types of dependencies

5. Import one definition per line



```
1 import filecmp
2 import shutil
3 from pathlib import Path
4
5 import pandas as pd
6 import pytest
7 import numpy as np
8
9 from common.constants import current_version
10 from common.constants import default_model_type
11 from dashboard.model.model import Model
12 from dashboard.model.model import ModelType
13 from source.db import DB
14 from source.connector.adapter import TimeSeries
```

Benefit: Reduction of version control conflicts

5. Import one definition per line



```
1 import filecmp
2 import shutil
3 from pathlib import Path
4
5 import pandas as pd
6 import pytest
7 import numpy as np
8
9 from common.constants import current_version
10 from common.constants import default_model_type
11 from dashboard.model.model import Model
12 from dashboard.model.model import ModelType
13 from source.db import DB
14 from source.connector.adapter import TimeSeries
```

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. Avoid using Namespace Packages

```
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

```
1 # base_implementation.py
2 def fast_add(a: float, b: float) -> float:
3     return a + b
4
5 def fast_multiply(a: float, b: float) -> float:
6     return a * b
7
8 # cython_accelerated_implementation.py (optional module)
9 def fast_add(a: float, b: float) -> float: ...
10
11 # public_api.py
12 from base_implementation import * # noqa: F403
13
14 try: # (cython_accelerated_implementation.py could be missing)
15     from cython_accelerated_implementation import * # noqa: F403
16 except ImportError:
17     pass
18
19 # main.py
20 from public_api import fast_add
21 from public_api import fast_multiply
```

Benefit: Avoid ambiguity and namespace pollution

9. (PEP) Avoid using wildcard imports

```
1 # base_implementation.py
2 def fast_add(a: float, b: float) -> float:
3     return a + b
4
5 def fast_multiply(a: float, b: float) -> float:
6     return a * b
7
8 # cython_accelerated_implementation.py (optional module)
9 def fast_add(a: float, b: float) -> float: ...
10
11 # public_api.py
12 from base_implementation import * # noqa: F403
13
14 try: # (cython_accelerated_implementation.py could be missing)
15     from cython_accelerated_implementation import * # noqa: F403
16 except ImportError:
17     pass
18
19 # main.py
20 from public_api import fast_add
21 from public_api import fast_multiply
```

Benefit: Avoid ambiguity and namespace pollution

9. (PEP) Avoid using wildcard imports

```
1 # base_implementation.py
2 def fast_add(a: float, b: float) -> float:
3     return a + b
4
5 def fast_multiply(a: float, b: float) -> float:
6     return a * b
7
8 # cython_accelerated_implementation.py (optional module)
9 def fast_add(a: float, b: float) -> float: ...
10
11 # public_api.py
12 from base_implementation import * # noqa: F403
13
14 try: # (cython_accelerated_implementation.py could be missing)
15     from cython_accelerated_implementation import * # noqa: F403
16 except ImportError:
17     pass
18
19 # main.py
20 from public_api import fast_add
21 from public_api import fast_multiply
```

Benefit: Avoid ambiguity and namespace pollution

9. (PEP) Avoid using wildcard imports

```
1 # base_implementation.py
2 def fast_add(a: float, b: float) -> float:
3     return a + b
4
5 def fast_multiply(a: float, b: float) -> float:
6     return a * b
7
8 # cython_accelerated_implementation.py (optional module)
9 def fast_add(a: float, b: float) -> float: ...
10
11 # public_api.py
12 from base_implementation import * # noqa: F403
13
14 try: # (cython_accelerated_implementation.py could be missing)
15     from cython_accelerated_implementation import * # noqa: F403
16 except ImportError:
17     pass
18
19 # main.py
20 from public_api import fast_add
21 from public_api import fast_multiply
```

Benefit: Avoid ambiguity and namespace pollution

9. (PEP) Avoid using wildcard imports

```
1 # base_implementation.py
2 def fast_add(a: float, b: float) -> float:
3     return a + b
4
5 def fast_multiply(a: float, b: float) -> float:
6     return a * b
7
8 # cython_accelerated_implementation.py (optional module)
9 def fast_add(a: float, b: float) -> float: ...
10
11 # public_api.py
12 from base_implementation import * # noqa: F403
13
14 try: # (cython_accelerated_implementation.py could be missing)
15     from cython_accelerated_implementation import * # noqa: F403
16 except ImportError:
17     pass
18
19 # main.py
20 from public_api import fast_add
21 from public_api import fast_multiply
```

Benefit: Avoid ambiguity and namespace pollution

10. Import directly from module where a definition is defined

```
main.py
os_ops
  windows
    __init__.py
    path.py
    ...
  linux
    __init__.py
    path.py
    ...
__init__.py
```

```
# os_ops/__init__.py

import os

if os.name == 'nt':
    from ._windows.path import copy_file
elif os.name == 'posix':
    from ._linux.path import copy_file
else:
    raise OSError(f"Unsupported OS: {os.name}")
```

```
# main.py
from os_ops import copy_file

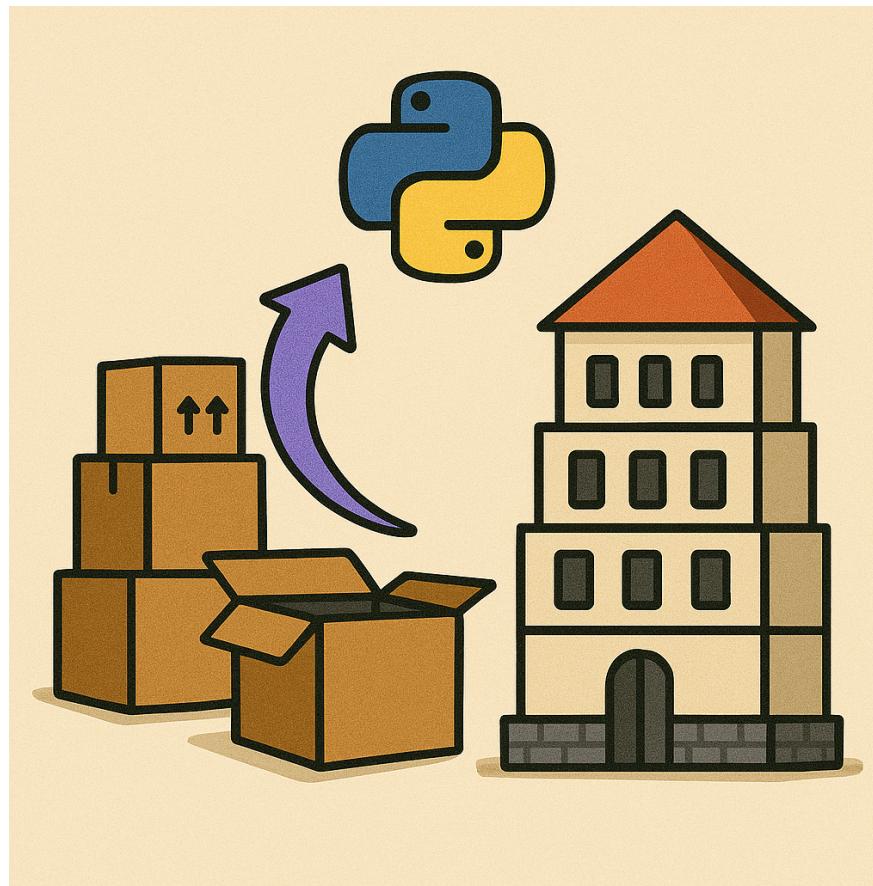
copy_file(
    "source.txt",
    "destination.txt"
)
```

Benefit: Avoid ambiguity

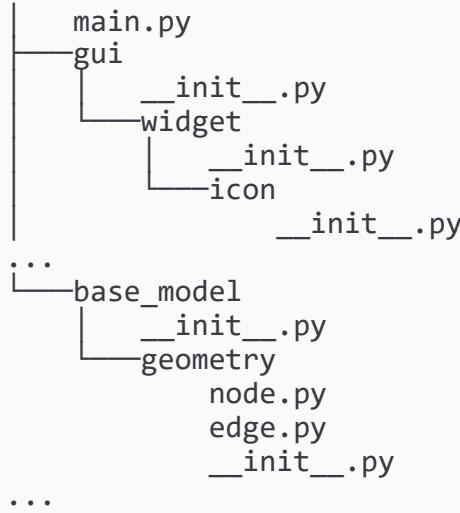
Re-exporting definitions through __init__.py

```
# 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
```

IV. How internal dependencies impact a package architecture?

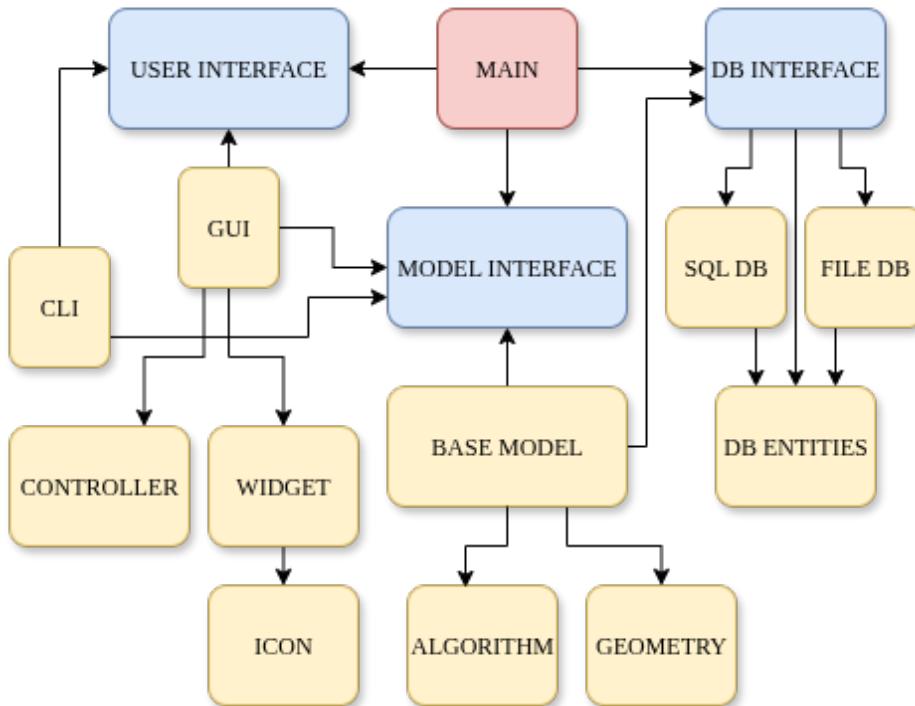


Why to care about clean modular structure?

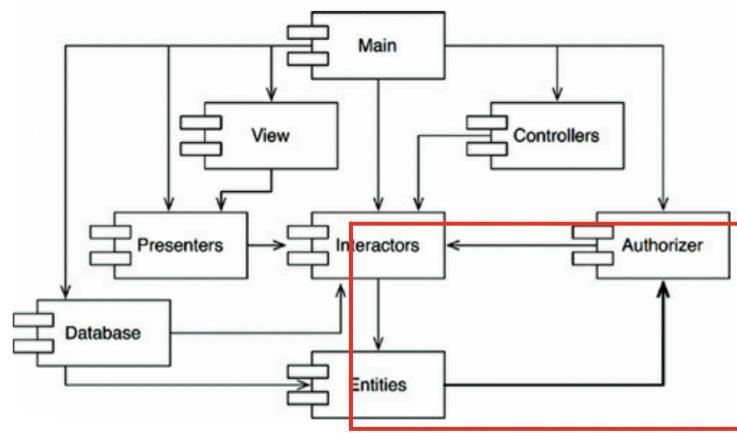


```
# running tests inside a specific package
python -m pytest base_model/geometry
# linting a specific package
ruff check gui/widget/icon
```

Why to care about clean modular structure?



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

Python circular imports. A bug or a feature?

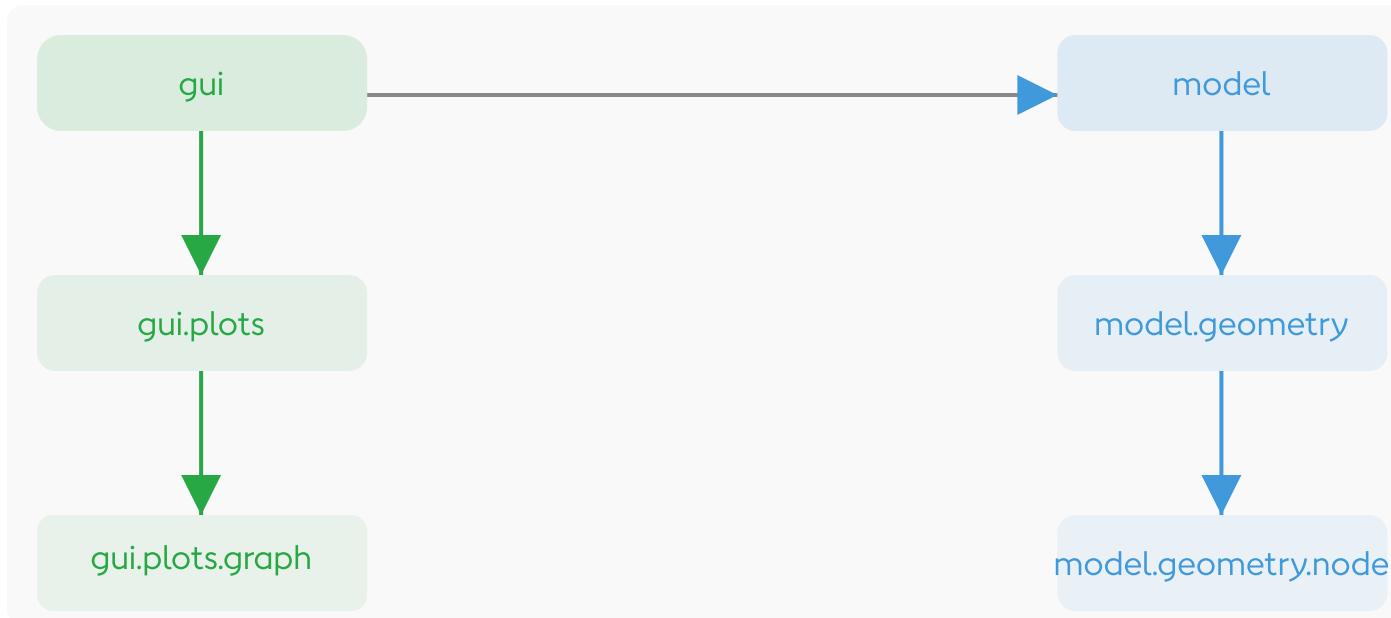
```
File "gui/__init__.py", line 1, in
    from model import Node
File "model/__init__.py", line 1, in
    from .geometry import Node
File "model/geometry/__init__.py", line 3, in
    from gui import Graph
^^^^^^^^^^^^^^^^^^^^^^^^^
ImportError: cannot import name 'Graph' from partially initialized module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

Circular imports most of the time signal a design issue, but sometimes are used intentionally for plugin systems or late binding.

Re-exporting definitions through __init__.py

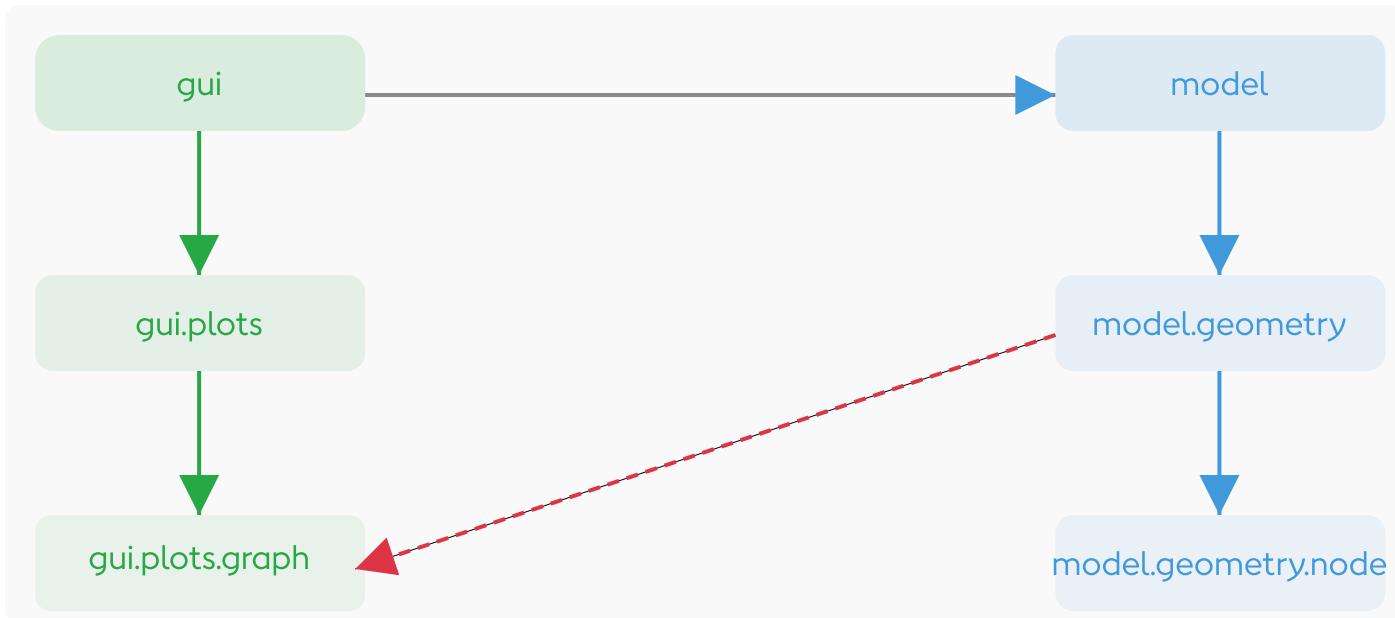
```
# model/geometry/node.py
class Node: ...
# model/geometry/__init__.py
from .node import Node, function_a
from .edge import Edge, function_b
# model/__init__.py
from .geometry import Node, Edge, function_a, function_b
from .other_module import SomeOtherClass
# gui/__init__.py
from model import Node, Edge, function_a, function_b, SomeOtherClass
```

Circular dependencies between packages



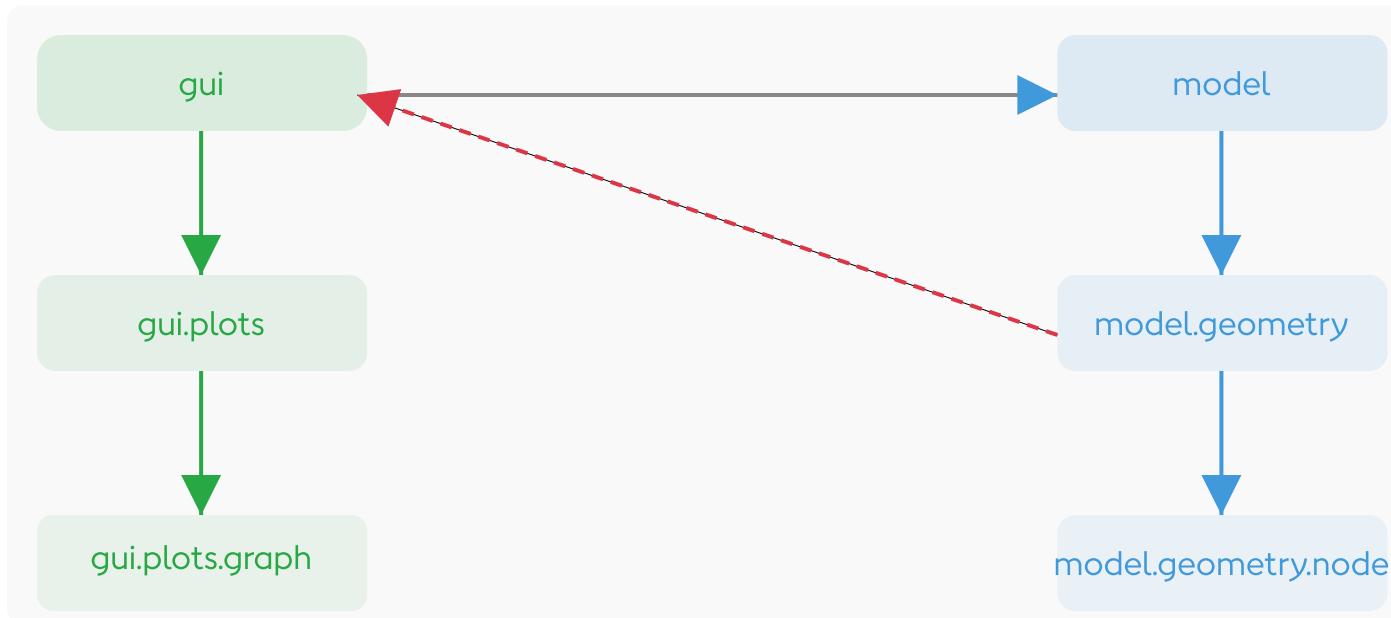
Packages "model" and "gui" re-exporting everything through `__init__.py`.
Package "gui" imports from "model".

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

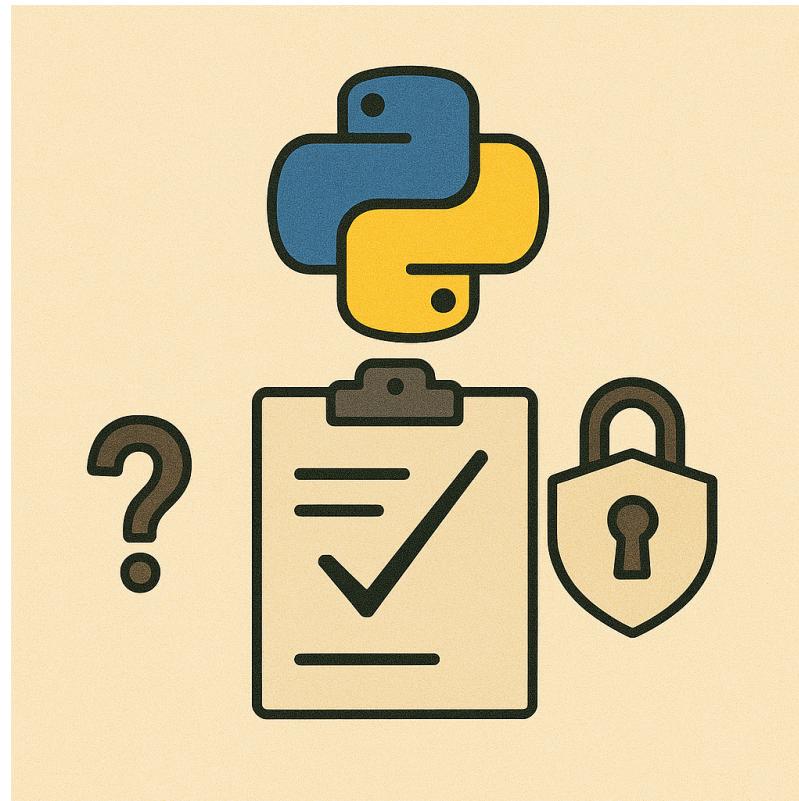
# gui/__init__.py
from model import Node

# 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)
```

V. 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
```

```
1 # ruff.toml
2
3 [lint]
4 select = [
5     "I",    # isort
6     "PLC0415",  # import-outside-top-level
7     "TID252",  # relative-imports
8     "F403",   # undefined-Local-with-import-star-usage
9 ]
10
11 [lint.isort]
12 force-single-line = true
13 known-first-party = ["my_another_package"]
```

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
```

```
1 # ruff.toml
2
3 [lint]
4 select = [
5     "I",    # isort
6     "PLC0415",  # import-outside-top-level
7     "TID252",  # relative-imports
8     "F403",   # undefined-Local-with-import-star-usage
9 ]
10
11 [lint.isort]
12 force-single-line = true
13 known-first-party = ["my_another_package"]
```

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
```

```
1 # ruff.toml
2
3 [lint]
4 select = [
5     "I", # isort
6     "PLC0415", # import-outside-top-level
7     "TID252", # relative-imports
8     "F403", # undefined-Local-with-import-star-usage
9 ]
10
11 [lint.isort]
12 force-single-line = true
13 known-first-party = ["my_another_package"]
```

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
```

```
1 # ruff.toml
2
3 [lint]
4 select = [
5     "I",    # isort
6     "PLC0415",  # import-outside-top-level
7     "TID252",  # relative-imports
8     "F403",   # undefined-Local-with-import-star-usage
9 ]
10
11 [lint.isort]
12 force-single-line = true
13 known-first-party = ["my_another_package"]
```

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
```

```
1 # ruff.toml
2
3 [lint]
4 select = [
5     "I",    # isort
6     "PLC0415",  # import-outside-top-level
7     "TID252",  # relative-imports
8     "F403",   # undefined-Local-with-import-star-usage
9 ]
10
11 [lint.isort]
12 force-single-line = true
13 known-first-party = ["my_another_package"]
```

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
```

```
1 # ruff.toml
2
3 [lint]
4 select = [
5     "I",    # isort
6     "PLC0415",  # import-outside-top-level
7     "TID252",  # relative-imports
8     "F403",   # undefined-Local-with-import-star-usage
9 ]
10
11 [lint.isort]
12 force-single-line = true
13 known-first-party = ["my_another_package"]
```

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
```

```
1 # ruff.toml
2
3 [lint]
4 select = [
5     "I",    # isort
6     "PLC0415",  # import-outside-top-level
7     "TID252",  # relative-imports
8     "F403",   # undefined-Local-with-import-star-usage
9 ]
10
11 [lint.isort]
12 force-single-line = true
13 known-first-party = ["my_another_package"]
```

Ruff

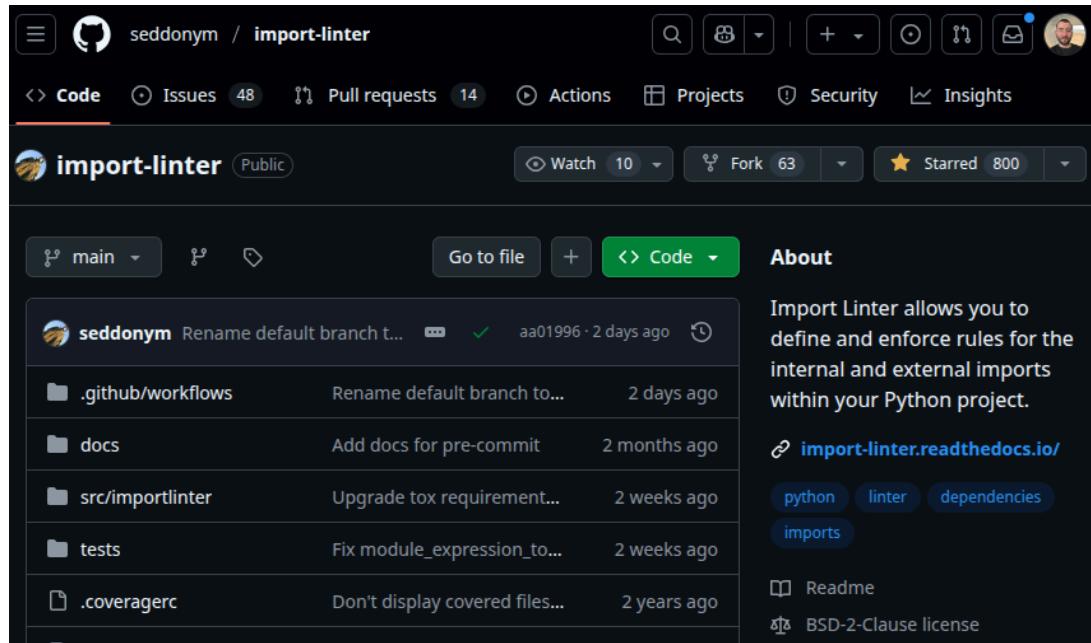
```
from functools import partial
import sys

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
```

```
1 # ruff.toml
2
3 [lint]
4 select = [
5     "I",    # isort
6     "PLC0415",  # import-outside-top-level
7     "TID252",  # relative-imports
8     "F403",   # undefined-Local-with-import-star-usage
9 ]
10
11 [lint.isort]
12 force-single-line = true
13 known-first-party = ["my_another_package"]
```

Import Linter[4]



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

```
[importlinter]
root_package = my_package

[importlinter:contract:1]
name=GUI cannot be imported from other packages
type=forbidden
source_modules=
    my_package.model
    my_package.db
forbidden_modules=
    my_package.gui
```

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

-----
Contracts
-----

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

GUI cannot be imported from other packages 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
```

Following ADP (Acyclic Dependencies Principle)



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

Summary

1. Importing in Python can be done in many different ways
2. Choosing one, consistent way, will reduce decision fatigue
3. Clean modular structure of your Python code matters
4. Following Acyclic Dependencies Principle (ADP) leads to better software design
5. Using tools like Ruff and Import Linter for automating the recommendations checks

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. Import Linter, <https://github.com/seddonym/import-linter>
5. Italian pasta code, Lynn (Yu Ling) Wang, <https://www.linkedin.com/in/yuling-wang-jobmenta/>

(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 [5]

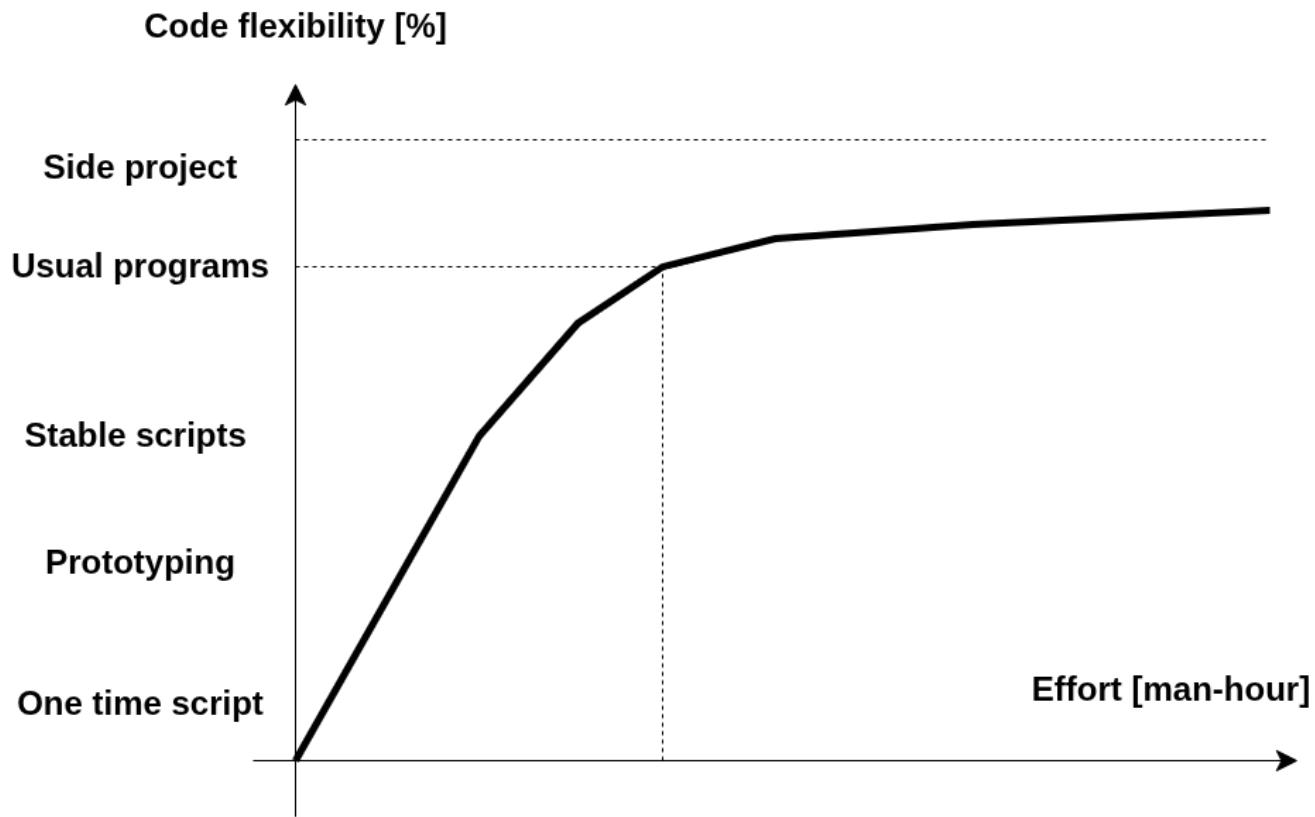
Clean code principles



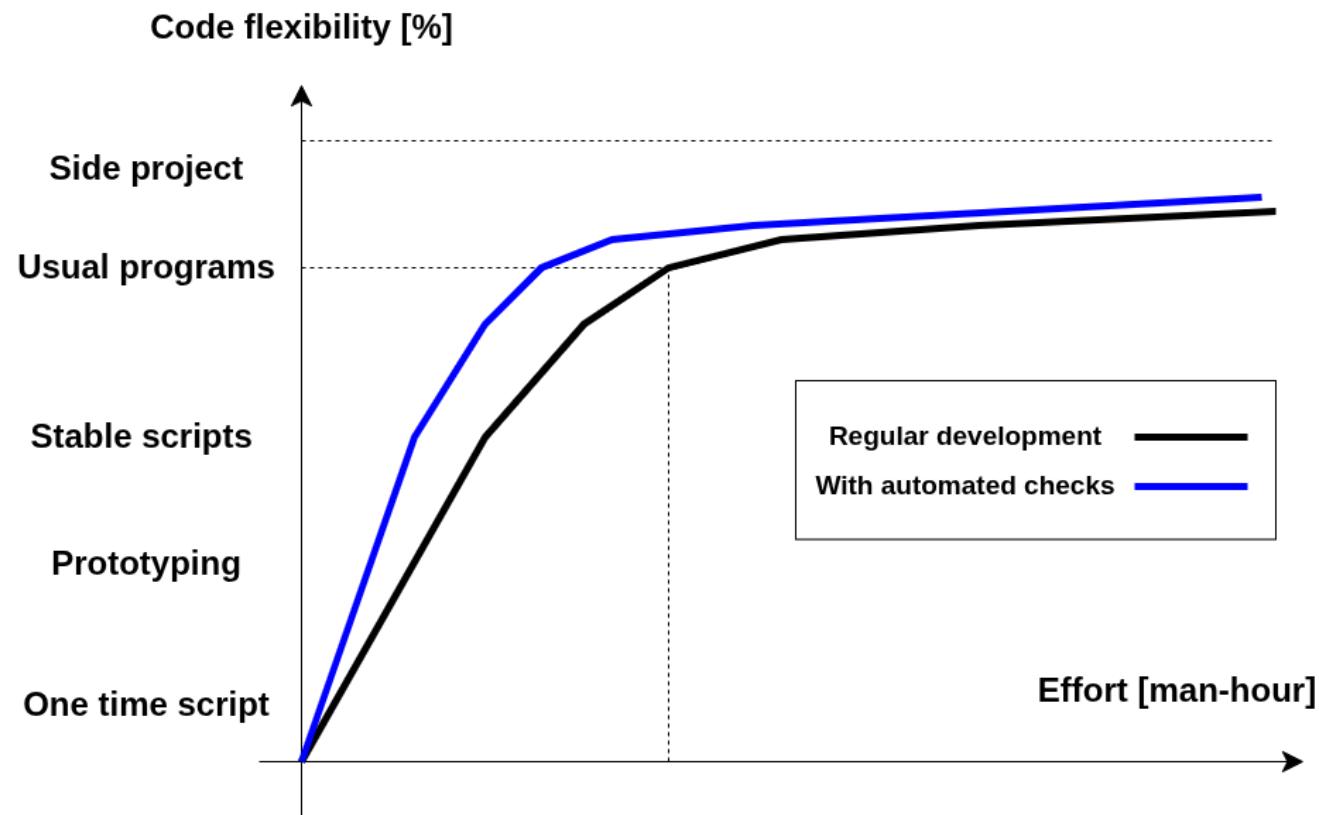
Uncle Bob

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



Effort vs benefit of applying clean code principles



Automated checks of clean code principles violations