Walking through PEP 810: Lazy Imports

PYCON(IT) PyCon Italia 2026
2026/05/29

Antonio Spadaro

par-tec

Who I am

- I'm Antonio Spadaro,
  also known as: **@ilovelinux**
- Working as DevOps Engineer for **Par-Tec S.p.A.**
- Python enthusiastic user
- OSS Maintainer of **`datamodel-code-generator`**!
- Not a Python Core Developer

Who is par-tec?

Longtime friends of:





- Software & Infrastructure system integrator
- Active in the open source scene for 20+ years
- A team of 200+ employees with 200+ professional certifications
- Ridiculously good solutions in:
  AI & Automation; Cloud & Infrastructure;
  Cybersecurity; Financial Services

Very, very skilled in:

BROADCOM
REGISTERED
PARTNER

CONFLUENT
SYSTEMS
INTEGRATOR
REGISTERED

dynatrace
Authorized Solution Partner

GitLab
Select
Partner

Google Cloud
Partner

ONE IDENTITY
Silver+ Partner

ORACLE | Partner

opentext™ | Partner

Red Hat
Premier
Business Partner

Sophos
Silver
Partner

Also, tech journalists!

yum@par-tec.it

agenda

- Introduce Lazy Import, according to PEP 810
- Syntaxes not allowed
- Behavioral shifts
- A use case

PEP 810

Explicit lazy imports

Pablo Galindo Salgado
Python Core Developer
& Core Developer @ **HRT**

Germán Méndez Bravo
Software Engineer
@ **Meta**

Thomas Wouters
Python Core Developer
& Staff Software Engineer
@ **Meta**

Tim Stumbaugh
Core Developer
@ **HRT**

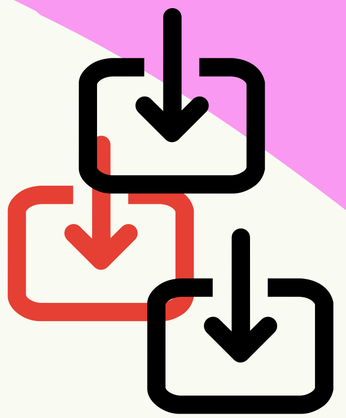
Dino Viehland
Python Core Developer
& Software Developer
@ **Microsoft**

Brittany Reynoso
Software Engineer
@ **Meta**

Noah Kim
Software Developer
@ **HRT**

Status: Accepted
Python Version: 3.15

Motivation



Importing the first module triggers an immediate cascade of imports.

Importing modules is computationally expensive.

The effect is especially costly for command-line tools with multiple subcommands, **where even running the command with** `--help` **can load dozens of unnecessary modules and take several seconds.**

Workarounds (until now)

Move the import into functions (inline imports), reducing readability

```python
def load_json(data:str) -> dict:
    from json import loads

    return loads(data)

def load_yaml(data:str) -> Any:
    from yaml import safe_load

    return safe_load(data)
```

Analysis of the Python standard library shows that **approximately 17% of all imports outside tests** (**nearly 3500 total imports across 730 files**) **are already placed inside functions or methods specifically to defer their execution.**

So, what's new?

PEP 810 introduces a new `lazy` keyword to import modules lazily.

```
import sys

lazy from json import dumps, loads

print('json'in sys.modules)     # False - module not loaded yet

# First use of 'dumps' triggers loading json and reifies ONLY 'dumps'
result = dumps({"hello":"world"})

print('json'in sys.modules)     # True - module now loaded

# Accessing 'loads' now reifies it (json already loaded, no re-import)
data = loads(result)
```

Reification

When a lazy object is used, it needs to be reified. This means resolving the important that point in the program and replacing the lazy object with the concrete one.
Reification imports the module at that point in the program.

That's so cool!

Which are the limitations?

Syntaxes not allowed


```
# SyntaxError: lazy import not allowed inside functions
def foo ():
      lazy import json
```

```
# SyntaxError: lazy import not allowed inside classes
class bar :
      lazy import json
```

Syntaxes not allowed


```
# SyntaxError: lazy import not allowed inside try/except blocks
# ... because it's not expected to raise.
try:
      lazy import json
except Import Error:
      pass
```

```
# SyntaxError: lazy from ... import * is not allowed
# ... because finding all the components inside a module
# ... would be almost as expensive as importing the module.
lazy from json import *
```

```
# SyntaxError: lazy from __future__ import is not allowed
# ... since future statements are special statements.
lazy from __future__ import annotations
```

Behavioral shifts

- **Error timing.** Import exceptions now occur at the use of the lazy name.

```
# With eager import - error at import statement
import broken_module # ImportError raised here
```

```
# With lazy import - error deferred
lazy import broken_module
print ("Import succeeded")
broken_module.foo() # ImportError raised here on use
```

Traceback

- **Error timing.** Import exceptions now occur at the use of the lazy name.

```
Import succeeded
Traceback (most recent call last):
  File "app.py", line 2, in <module>
ImportError: deferred import of 'broken_module' raised an exception du-
ring resolution

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "app. py", line 4, in <module>
    broken_module.foo() # ImportError raised here on use
    ^^^^^^^^^^

ModuleNotFoundError: No module named 'broken_module'
```

Behavioral shifts

- **Side-effect timing.** Import-time side effects in lazily imported modules occur at first use of the binding,not at module import time.

```
>>> import this
The Zen of Python, by Tim Peters
[...]
```

```
>>> lazy import this
>>> print ("Hello, PyCon Italy 2026!" )
Hello, PyCon Italy 2026 !

>>> this
The Zen of Python, by Tim Peters
[...]
```

Behavioral shifts

- **Import order.** Because modules are imported on first use, the order in whichmodules are imported may differ from how they appear in code.

- **Presence in** `sys.modules` **.** A lazily imported module does not appear insys.modules until first use. After reification, it must appear in sys.modules.

Behavioral shifts

Proxy visibility. Before first use, the bound name refers to a lazy proxy.

```
>>> lazy import json

>>> type(globals()['json'])
<class 'lazy_import'>

>>> json #First use loads the module

<module 'json' from 'lib/python3.15/json/__init__.py'>

>>> type(globals()['json'])
<class 'module'>
```

We've got retrocompatibility!

```
lazy_modules__ = ["json"] # No effect on Python 3.14 and below
import json
print('json' in sys.modules) # False
result = json.dumps({"PyCon Italia": "2026"})
print('json' in sys.modules) # True
```

We've got flags!

The global lazy imports flag can be controlled through:

- The `-X lazy_imports=<mode>` command-line option
- The `PYTHON_LAZY_IMPORTS=<mode>` environment variable
- The `sys.set_lazy_imports(mode)` function (primarily for testing)

Where `<mode>` can be:

- `"normal"` (or unset): Only explicitly marked lazy imports are lazy
- `"all"`: All module-level imports (except in try blocks and import *) become potentially lazy
- `"none"`: No imports are lazy, even those explicitly marked with lazy keyword

Use case: a back-end server

	Standard import	Lazy import
Fast start-up	✗	✓
Import error only at start-up	✓	✗
No latency spikes caused by lazy import reification	✓	✗
Hypotetical environment	Persistent Server	Serverless Function

Thanks to all the authors of the PEP 810!

PEP 810: LAZY IMPORTS

Thank You!

antonio.spadaro@par-tec.it
antonio-spadaro

Design • Elisabetta Rapini

GET THE SLIDES!

https://ilovelinux.dev/pyit26