23 – Packages

Bálint Aradi

Scientific Programming in Python (2025)

https://atticlectures.net/scipro/python-2025/

Module name collisions

- Module names in different projects can easily collide (e.g. solver, io)

```
import io                                    linsolver.py
```

Which io-module should be imported?

- the projects io-module
- Pythons io module
- some other projects io-module
- …

- Especially critical, if many Python projects are installed on the system (e.g. Conda)

Packages

- Each project should create a “**package**” (a namespace) with a unique name
- Each module of the project should be placed in this package
- Modules within a package can be accessed with the dot-notation:

```
import PACKAGENAME.MODULENAME
```

```
import linsolver.io
```

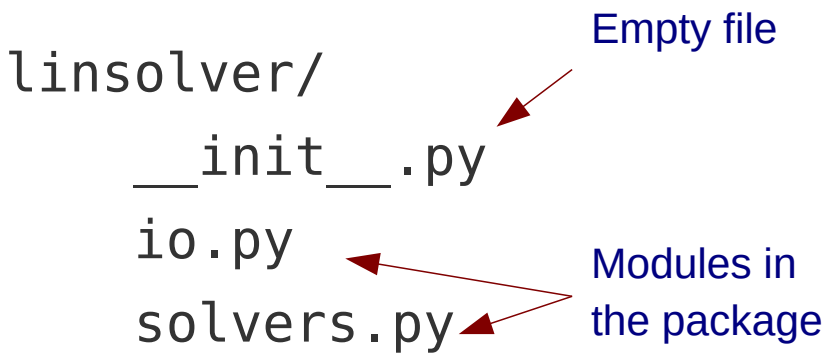
io module in the linsolver package

```
import numpy.linalg as linalg
```

linalg module in the numpy package

Defining packages

- Packages are simple directories in the file system
- In order to indicate that a directory is a package, it must contain an (evtl. empty) file **__init__.py**




```
import linsolver.solvers
import linsolver.io
…
aa, bb = linsolver.io.read_input()
xx = linsolver.solvers.solve(aa, bb))
```

Using the modules of the linsolver package

Package initialization

- The **__init__.py** file may also contain Python code (package initialization)
- The content of **__init__.py** is executed when the package is imported
- It is often used provide direct access to the quantities / modules which should be accessible from outside

```
linsolver/
    __init__.py
    io.py
    solver.py
```

__init__.py

```
import linsolver.solvers as solvers
import linsolver.io as io
```

equivalent

__init__.py

```
from . import solvers
from . import io
```

current directory

```
import linsolver
…
aa, bb = linsolver.io.read_input()
xx = linsolver.solvers.solve(aa, bb))
```

Using the modules of the linsolver package

Hiding internal structure of packages

- It is a good practice to hide the internal structure of a package
- **__init__.py** can be used to export all functionality without exposing internal structure

```
linsolver/
    __init__.py
    io.py
    solvers.py
```

```
from .solvers import solve
from .io import read_input
```

__init__.py

Importing selected entities from various modules into the packages name space

```
import linsolver
…
aa, bb = linsolver.read_input()
xx = linsolver.solve(aa, bb))
```

Using the (public) functionality of the linsolver package without knowing how it is split up into various modules

Hiding module internals

- A module should only export functions / objects which should be invoked from outside
- Implementational details (e.g. helper routines, internal constants, etc.) should be hidden / private.

```
_TOLERANCE = 1e-12                    Internally used constant (private)

def solve(aa, bb):                    Public routine (public)
    ...
    _eliminate(aa, pp)
    _make_back_subsitute(pp, bb)

def _eliminate(aa, pp):               Internal helper routine (private)
    ...

def _subsitute_back(pp, bb):          Internal helper routine (private)
    ...
```

Public / private entities

- Whether an entity should be accessed from outside is **indicated** by its **name**.
- If its name **starts with** a letter **[a-zA-Z]**, the entity is considered to be a **public** entity
- If its name **starts with** a **single underscore** (_), the entity is considered to be a **private** entity.
- Private entities should only be accessed from within the scope (module, object) where they are defined.

Note

- The public / private naming rule above is **only** a **convention** and is not enforced by the language.
- Theoretically, one can also access “private” entities from outside.

Stick to the public / private naming convention!

Make sure, you mark only the necessary entities in your module public and make everything else private (hiding implementation details)

Public entities should be always documented via their doc-strings