Generalitat de Catalunya
Departament d'Educació
Institut Thos i Codina

Mòduls en Python

Domini Públic (CC0)

Índex

1. Què és un mòdul **1**
Primer mòdul 1

2. Com estructurar un projecte **2**

3. Formes d'importar mòduls **3**
3.1. `import mòdul` 3
3.2- `from mòdul import component` 3
3.3. `from mòdul import *` 3
3.4. Importar diversos mòduls 4
3.5. Col·lisió de noms 4

4. Àlies (as) **4**

5. Paquets i subcarpetes **5**
5.1. El paper de `__init__.py` 5
5.2. Importacions relatives 6
5.3. Importacions absolutes vs. relatives 6
5.4. Paquets d'espai de noms 6

6. `sys.path`: on cerca Python els mòduls **6**
6.1. Modificar `sys.path` en temps d'execució 7
6.2. Maneres permanents de fer-ho 7
6.3. El problema de l'ombra (*shadowing*) 7

7. La funció `dir()` **8**
7.1. Sense arguments: noms de l'espai de noms actual 8
7.2. Amb un mòdul: els seus atributs 8
7.3. Amb un objecte qualsevol 8
7.4. Per què és útil 9

8. Metadades d'un mòdul **9**
8.1. Exemple 9
8.2. Metadades personalitzades 10
8.3. Docstrings 10

9. `__main__`: mòdul o programa? **10**
9.1. Executar un mòdul amb `-m` 11
9.2. `__main__.py` en un paquet 11
9.3. Bones pràctiques 11

10. Gestió d'errors en mòduls no trobats **12**
10.1. `ModuleNotFoundError` i `ImportError` 12
10.2. Capturar l'error amb `try/except` 12
10.3. Alternatives (*fallback*) 12
10.4. Dependències opcionals 13
10.5. Comprovar l'existència **sense** importar 13
10.6. Capturar només el que toca 13
10.7. Importar per nom dinàmic 14

11. Recarregar mòduls: `reload` **14**
11.1. Exemple pas a pas 14
11.2. Limitacions importants 15
11.3. Quan s'utilitza 15
11.4. Eliminar un mòdul de la memòria cau 16

12. Exemple integrador **16**

13. Resum **17**

1. Què és un mòdul

Un **mòdul** és un fitxer `.py` que conté definicions (funcions, classes, variables) i, opcionalment, codi executable. El nom del mòdul és el nom del fitxer sense l'extensió.

Un **paquet** és un directori que agrupa mòduls (i, opcionalment, altres paquets). Un **espai de noms** (*namespace*) és el context en què viuen els noms: cada mòdul té el seu propi, i això evita col·lisions entre fitxers diferents.

Per què fer servir mòduls:

Avantatge	Descripció
Reutilització	Escrius la funció una vegada i la importes des de molts llocs
Organització	Un fitxer de 3000 línies es divideix en peces amb responsabilitat clara
Espai de noms propi	`a.saluda()` i `b.saluda()` poden coexistir
Manteniment	Un error es corregeix en un sol lloc
Proves	Cada mòdul es pot provar de manera aïllada

Primer mòdul

Fitxer `geometria.py`:

```
"""Funcions bàsiques de geometria plana."""

PI = 3.14159265

def area_cercle(radi):
    return PI * radi ** 2

def area_rectangle(base, altura):
    return base * altura
```

Fitxer `principal.py`, al mateix directori:

```
import geometria

print(geometria.area_cercle(2))      # 12.5663706
print(geometria.area_rectangle(3, 4))  # 12
```

Quan es fa `import geometria`, Python executa el fitxer **una sola vegada**, guarda el resultat a la memòria cau (`sys.modules`) i reutilitza aquest objecte en qualsevol importació posterior.

2. Com estructurar un projecte

Una estructura simple i escalable:

```
projecte/
├── main.py
├── config.py
├── utils/
│   ├── __init__.py
│   ├── fitxers.py
│   └── text.py
└── xarxa/
    ├── __init__.py
    ├── ping.py
    └── ports.py
```

Regles pràctiques:

- **Un mòdul, una responsabilitat.** `fitxers.py` llegeix i escriu fitxers; `text.py` manipula cadenes.
- **Noms en minúscules i curts**, amb guions baixos si cal (`gestio_usuaris.py`). Cap guió (`-`) ni espai: no es poden importar.
- **No posar el nom d'un mòdul de la biblioteca estàndard** (`random.py`, `json.py`, `email.py`…): taparia el mòdul real (vegeu la secció de `sys.path`).
- **Evitar codi "solt"** en l'àmbit de mòdul que faci coses en importar (connectar a una base de dades, imprimir…). Les definicions sí; les accions, dins de funcions o sota `if __name__ == "__main__":`.
- **Evitar importacions circulars** (a importa b i b importa a). Si apareixen, normalment indiquen que cal extreure el codi comú a un tercer mòdul.

3. Formes d'importar mòduls

3.1. `import mòdul`

```
import math

print(math.sqrt(25))
print(math.pi)
```

Cal el prefix. El codi deixa clar d'on ve cada cosa.

3.2- `from mòdul import component`

```
from math import sqrt, pi

print(sqrt(25), pi)
```

Sense prefix. Només es creen al teu espai de noms els noms que demanes.

Per a llistes llargues, parèntesis:

```
from os.path import (
    join,
    exists,
    basename,
)
```

3.3. `from mòdul import *`

```
from math import *
```

Importa tots els noms públics: els que figuren a `__all__`, o, si no existeix, tots els que no comencen per _.

AVÍS

Es desaconsella en codi real: no es veu d'on ve cada nom i es poden sobreescriure noms existents sense avís. Només té sentit en sessions interactives.

Un mòdul pot limitar què exporta amb `import *`:

```
# geometria.py
__all__ = ["area_cercle", "area_rectangle"]   # PI queda fora de
 ↪  l'import *
```

`__all__` només afecta `import *`; `from geometria import PI` continua funcionant.

3.4. Importar diversos mòduls

```
import os, sys, json        # permès, però PEP 8 recomana una línia
 ↪  per mòdul
```

Forma recomanada:

```
import json
import os
import sys
```

Ordre PEP 8: primer biblioteca estàndard, després paquets de tercers, i finalment mòduls propis, amb una línia en blanc entre grups.

3.5. Col·lisió de noms

Si dues importacions donen el mateix nom, **la darrera guanya**, sense error:

```
from a import saluda
from b import saluda     # ara 'saluda' és la de b
```

Solució: importar el mòdul sencer o fer servir àlies.

4. Àlies (as)

L'àlies assigna un nom alternatiu en importar.

```
import numpy as np                  # conveni molt estès
import datetime as dt
from collections import defaultdict as dd
from a import saluda as saluda_a
from b import saluda as saluda_b
```

Usos habituals:

1. **Abreujar** noms llargs (`import matplotlib.pyplot as plt`).
2. **Evitar col·lisions** entre noms iguals de mòduls diferents.
3. **Triar una implementació** en funció de les circumstàncies:

```
try:
    import ujson as json      # més ràpida, si està instal·lada
except ImportError:
    import json               # alternativa estàndard

json.dumps({"a": 1})
```

L'àlies només existeix al teu espai de noms: el mòdul original no canvia ni es duplica.

5. Paquets i subcarpetes

Un directori amb mòduls es converteix en **paquet**. Tradicionalment, conté un fitxer `__init__.py` (pot estar buit).

```
projecte/
├── main.py
└── xarxa/
    ├── __init__.py
    ├── ping.py
    └── ports.py
```

`xarxa/ports.py`:

```
def port_obert(host, port):
    return f"Comprovant {host}:{port}"
```

`main.py`:

```
import xarxa.ports
print(xarxa.ports.port_obert("10.0.0.1", 22))

from xarxa import ports
print(ports.port_obert("10.0.0.1", 22))

from xarxa.ports import port_obert
print(port_obert("10.0.0.1", 22))

import xarxa.ports as p
print(p.port_obert("10.0.0.1", 22))
```

5.1. El paper de `__init__.py`

S'executa quan s'importa el paquet. Serveix per a:

- Marcar el directori com a paquet.
- **Exposar una API neta**, reexportant el que interessa:

```
# xarxa/__init__.py
from .ports import port_obert
from .ping import fer_ping

__all__ = ["port_obert", "fer_ping"]
```

Ara l'usuari pot escriure `from xarxa import port_obert` sense conèixer l'estructura interna.

5.2. Importacions relatives

Dins d'un paquet es pot referir als mòduls germans amb punts:

```
# xarxa/ping.py
from . import ports              # mateix paquet
from .ports import port_obert    # component d'un mòdul germà
from ..utils import text         # un nivell amunt (paquet pare)
```

Sintaxi	Significat
`.`	Paquet actual
`..`	Paquet pare
`...`	Dos nivells amunt

NOTA

Les importacions relatives només funcionen quan el fitxer s'executa **com a part d'un paquet**. Si executes `python xarxa/ping.py` directament, obtindràs `ImportError: attempted relative import with no known parent package`. Cal fer-ho amb `python -m xarxa.ping` des del directori arrel del projecte.

5.3. Importacions absolutes vs. relatives

- **Absolutes** (`from xarxa.ports import port_obert`): més clares i robustes. PEP 8 les recomana per defecte.
- **Relatives** (`from .ports import port_obert`): útils quan el paquet es pot reanomenar o moure.

5.4. Paquets d'espai de noms

Des de Python 3.3, un directori **sense** `__init__.py` també es pot importar (paquet d'espai de noms implícit). Funciona, però es recomana mantenir `__init__.py` als paquets propis: és més explícit i evita sorpreses amb eines i empaquetatge.

6. `sys.path`: on cerca Python els mòduls

Quan escrius `import fitxer`, Python busca primer a `sys.modules` (ja importats), i si no hi és, recorre les rutes de la llista `sys.path` per ordre fins que el troba.

```
import sys
for ruta in sys.path:
    print(ruta)
```

Òbviament, la sortida varia per sistema. Ordre de contingut típic:

1. El directori de l'script que s'executa (o `''` en mode interactiu: el directori actual).

2. Les rutes de la variable d'entorn `PYTHONPATH`.
3. La biblioteca estàndard (p. ex. `/usr/lib/python3.12`).
4. `site-packages` (paquets instal·lats amb `pip`) i el *user site*.

Una ullada ràpida des de la terminal:

```
python3 -c "import sys; print('\n'.join(sys.path))"
python3 -m site
```

6.1. Modificar `sys.path` en temps d'execució

És una llista normal:

```
import sys
sys.path.append("/home/ramon/lib")        # al final: prioritat baixa
sys.path.insert(0, "/home/ramon/lib")     # al principi: prioritat
 ↪  màxima

import elmeumodul
```

6.2. Maneres permanents de fer-ho

Mètode	Exemple
Variable d'entorn	`export PYTHONPATH=/home/ramon/lib:$PYTHONPATH`
Fitxer `.pth`	Un fitxer `meu.pth` a `site-packages` amb una ruta per línia
Instal·lar el projecte	`pip install -e .` (mode editable, amb `pyproject.toml`)

La darrera és la forma professional: el teu codi queda importable des de qualsevol lloc sense tocar rutes.

6.3. El problema de l'ombra (*shadowing*)

Com que el directori de l'script va **primer**, un fitxer teu amb el mateix nom que un mòdul estàndard tapa el real:

```
projecte/
├── random.py        ← ombra el mòdul estàndard 'random'
└── main.py
```

```
# main.py
import random
print(random.randint(1, 6))    # AttributeError: module 'random' has
 ↪  no attribute 'randint'
```

Diagnòstic: mira on s'ha carregat realment el mòdul.

```
import random
print(random.__file__)     # ruta del fitxer carregat
```

Solució: reanomena el teu fitxer i elimina el directori `__pycache__` associat.

7. La funció `dir()`

`dir()` retorna la llista (ordenada alfabèticament) de noms disponibles.

7.1. Sense arguments: noms de l'espai de noms actual

```
import math
x = 10
print(dir())
# ['__annotations__', '__builtins__', '__doc__', '__loader__',
 ↪  '__name__',
#  '__package__', '__spec__', 'math', 'x']
```

7.2. Amb un mòdul: els seus atributs

```
import math
print(dir(math))
# ['__doc__', '__loader__', '__name__', '__package__', '__spec__',
#  'acos', 'acosh', ..., 'sqrt', 'tan', 'tau', 'trunc']
```

Per filtrar només els noms públics:

```
print([n for n in dir(math) if not n.startswith("_")])
```

7.3. Amb un objecte qualsevol

```
print(dir("hola"))     # mètodes de str
print(dir([]))         # mètodes de list
```

7.4. Per què és útil

- Explorar un mòdul desconegut.
- Veure què ha exportat un `import *`.
- Comprovar si un nom existeix (`"sqrt" in dir(math)`), tot i que per atributs és millor `hasattr(math, "sqrt")`.

Combina-la amb `help()` per veure la documentació:

```
help(math.sqrt)
help(math)
```

NOTA

`dir()` és una eina pensada per a l'ús interactiu: mostra el que és "interessant", no garanteix una llista completa en tots els casos (un objecte pot personalitzar `__dir__`).

8. Metadades d'un mòdul

Tot mòdul té atributs especials, definits automàticament:

Atribut	Contingut
`__name__`	Nom del mòdul. Val `"__main__"` si s'executa com a programa principal
`__file__`	Ruta del fitxer `.py`
`__doc__`	*Docstring* del mòdul (la cadena a l'inici del fitxer)
`__package__`	Nom del paquet al qual pertany (`None` o `''` si no n'hi ha)
`__spec__`	Informació de com s'ha trobat i carregat (nom, ubicació, carregador)
`__path__`	Només als paquets: llista de directoris del paquet
`__cached__`	Ruta del bytecode compilat (`__pycache__/...pyc`)

8.1. Exemple

`eines.py`:

```
"""Eines de text per a les pràctiques d'SMX."""

__version__ = "1.2.0"
__author__ = "Ramon López"

def majuscules(text):
    return text.upper()
```

```
import eines

print(eines.__name__)      # eines
print(eines.__doc__)       # Eines de text per a les pràctiques d'SMX.
```

```
print(eines.__file__)      # /ruta/eines.py
print(eines.__version__)   # 1.2.0
print(eines.__spec__.name) # eines
```

8.2. Metadades personalitzades

`__version__`, `__author__`, `__license__`... no són obligatoris ni els interpreta Python: són una **convenció**. Si distribueixes el paquet, la informació oficial viu al `pyproject.toml` i es pot llegir amb:

```
from importlib.metadata import version
print(version("requests"))      # versió d'un paquet instal·lat
```

8.3. Docstrings

Una docstring al principi del mòdul n'és la documentació. `help(eines)` la mostra juntament amb les funcions i les seves docstrings.

9. `__main__`: mòdul o programa?

Un fitxer `.py` serveix de dues maneres:

1. Executat directament: `python eines.py`
2. Importat des d'un altre: `import eines`

Python posa `__name__ = "__main__"` només en el primer cas. Això permet escriure codi que s'executa **únicament** quan el fitxer és el programa principal:

```
# calcul.py
def suma(a, b):
    return a + b

def main():
    print("Prova:", suma(2, 3))

if __name__ == "__main__":
    main()
```

Situació	Valor de `__name__`	S'executa `main()`?
`python calcul.py`	`"__main__"`	Sí
`import calcul`	`"calcul"`	No

Sense aquesta protecció, importar el mòdul executaria el codi de prova cada vegada.

9.1. Executar un mòdul amb `-m`

```
python -m calcul            # busca 'calcul' a sys.path i l'executa
 ↪  com a __main__
python -m xarxa.ping        # mòdul dins d'un paquet (les
 ↪  importacions relatives funcionen)
python -m http.server 8000  # mòdul de la biblioteca estàndard
```

9.2. `__main__.py` en un paquet

Si un paquet conté `__main__.py`, es pot executar directament:

```
eina/
├── __init__.py
├── __main__.py
└── nucli.py
```

```
python -m eina
```

9.3. Bones pràctiques

- Posar la lògica en funcions i un `main()` que les crida.
- Un `if __name__ == "__main__":` curt: només crida `main()`.
- Facilita les proves: pots importar les funcions sense efectes secundaris.

10. Gestió d'errors en mòduls no trobats

10.1. `ModuleNotFoundError` i `ImportError`

```
import mod_inexistent
# ModuleNotFoundError: No module named 'mod_inexistent'
```

`ModuleNotFoundError` és una subclasse d'`ImportError`.

Excepció	Quan apareix
`ModuleNotFoundError`	Python no troba el mòdul a `sys.path`
`ImportError`	El mòdul existeix, però no es pot importar (o falta un nom: `cannot import name 'X'`)

Causes freqüents:

1. **Mòdul no instal·lat.** Solució: `pip install paquet` (o `python -m pip install paquet`, que garanteix que s'usa el mateix intèrpret).
2. **Entorn virtual incorrecte.** Instal·lat a un Python i executat amb un altre. Comprova-ho amb `which python` i `python -m pip list`.
3. **Nom mal escrit**, o diferència entre el nom del paquet a `pip` i el del mòdul (`pip install pillow` → `import PIL`).
4. **El mòdul no és a `sys.path`.** Comprova-ho amb `print(sys.path)` (secció 6).
5. **Falta `__init__.py`** o s'ha executat des d'un directori inadequat.
6. **Importació circular**, que provoca `ImportError: cannot import name ... (most likely due to a circular import)`.
7. **Shadowing** d'un mòdul estàndard (secció 6.3).

10.2. Capturar l'error amb `try/except`

```
try:
    import requests
except ModuleNotFoundError:
    print("Cal instal·lar 'requests': python -m pip install requests")
    raise SystemExit(1)
```

10.3. Alternatives (*fallback*)

```
try:
    import tomllib                  # Python 3.11+
except ModuleNotFoundError:
    import tomli as tomllib         # paquet extern per a versions
     ↪  anteriors
```

10.4. Dependències opcionals

```
try:
    import colorama
    HAS_COLOR = True
except ImportError:
    colorama = None
    HAS_COLOR = False

def avis(text):
    if HAS_COLOR:
        print(colorama.Fore.YELLOW + text)
    else:
        print(text)
```

10.5. Comprovar l'existència sense importar

```
import importlib.util

if importlib.util.find_spec("yaml") is None:
    print("PyYAML no està instal·lat")
else:
    print("PyYAML disponible")
```

10.6. Capturar només el que toca

Un `except ImportError` massa ampli pot amagar errors reals dins del mòdul importat. Si cal distingir quin mòdul falta, usa l'atribut `name` de l'excepció:

```
try:
    import elmeumodul
except ModuleNotFoundError as e:
    if e.name == "elmeumodul":
        print("No s'ha trobat el mòdul principal")
    else:
        print(f"El mòdul existeix, però li falta una dependència:
        ↪ {e.name}")
        raise
```

10.7. Importar per nom dinàmic

```
import importlib

nom = "json"
try:
    modul = importlib.import_module(nom)
except ModuleNotFoundError:
    print(f"No existeix el mòdul {nom}")
else:
    print(modul.dumps({"ok": True}))
```

Útil per a sistemes de *connectors*, on el nom del mòdul ve d'un fitxer de configuració.

11. Recarregar mòduls: `reload`

Com que un mòdul s'importa **una sola vegada**, si el modifiques mentre el programa (o la sessió interactiva) està en marxa, els canvis **no** es veuen:

```
import eines
# ... edites eines.py al disc ...
import eines          # no fa res: ja és a sys.modules
```

Per forçar la reexecució del codi cal usar `importlib.reload()`:

```
import importlib
import eines

# ... edites eines.py ...
importlib.reload(eines)
```

(`reload` era una funció integrada a Python 2; a Python 3 viu a `importlib`. El vell `imp.reload` ja no existeix a les versions modernes.)

11.1. Exemple pas a pas

`config.py`:

```
LIMIT = 10
```

Sessió interactiva:

```
>>> import config
>>> config.LIMIT
10
```

Edites `config.py` i poses `LIMIT = 99`:

```
>>> config.LIMIT
10                              # encara el valor antic
>>> import importlib
>>> importlib.reload(config)
<module 'config' from '/ruta/config.py'>
>>> config.LIMIT
99
```

11.2. Limitacions importants

1. **`from mòdul import nom` no s'actualitza.** Els noms ja copiats al teu espai de noms continuen apuntant als objectes antics:

   ```
   from config import LIMIT
   importlib.reload(config)    # (cal que 'config' també estigui
    ↪  importat)
   print(LIMIT)                # segueix sent el valor antic
   ```

   Solució: tornar a executar `from config import LIMIT` després del `reload`, o treballar amb `config.LIMIT`.

2. **Els objectes creats abans** (instàncies de classes) conserven la classe antiga.
3. **`reload` només actualitza el mòdul indicat**, no les seves dependències; cal recarregar-les explícitament, en l'ordre adequat.
4. **L'estat del mòdul es reinicia parcialment**: el diccionari del mòdul es manté i el codi es torna a executar sobre ell. Els noms eliminats del fitxer font no desapareixen.
5. **Només per a un mòdul ja importat.** Si dona error, comprova que s'hagi importat abans.

11.3. Quan s'utilitza

- Desenvolupament interactiu (REPL, Jupyter) per no haver de reiniciar la sessió.
- Configuracions que es poden canviar en calent.
- Sistemes de *connectors*.

NOTA

En entorns de producció, reiniciar el procés és més fiable. A Jupyter/IPython, la «màgia» `%autoreload 2` (després de `%load_ext autoreload`) recarrega els mòduls automàticament abans de cada execució.

11.4. Eliminar un mòdul de la memòria cau

Alternativa més dràstica: esborrar-lo de `sys.modules` perquè el pròxim `import` el torni a carregar des de zero:

```
import sys
sys.modules.pop("eines", None)
import eines        # es reexecuta des del disc
```

És un truc útil en proves, però deixa els objectes antics en circulació.

12. Exemple integrador

```
inventari/
├── main.py
└── gestio/
    ├── __init__.py
    ├── equips.py
    └── informes.py
```

`gestio/equips.py`:

```
"""Gestió bàsica d'equips informàtics."""

__version__ = "0.1"

_equips = []

def afegir(nom, ip):
    _equips.append({"nom": nom, "ip": ip})

def llistar():
    return list(_equips)
```

`gestio/informes.py`:

```
"""Generació d'informes."""

from . import equips

def resum():
    return "\n".join(f"{e['nom']:<10} {e['ip']}" for e in
     ↪  equips.llistar())
```

`gestio/__init__.py`:

```
"""Paquet de gestió d'inventari."""
```

```
from .equips import afegir, llistar
from .informes import resum

__all__ = ["afegir", "llistar", "resum"]
```

`main.py`:

```
import sys

try:
    from gestio import afegir, resum
except ModuleNotFoundError as e:
    print(f"Falta un mòdul: {e.name}")
    sys.exit(1)

def main():
    afegir("srv-dc01", "192.168.1.10")
    afegir("srv-fs01", "192.168.1.11")
    print(resum())

if __name__ == "__main__":
    main()
```

Execució:

```
cd inventari
python main.py
```

Sortida:

```
srv-dc01   192.168.1.10
srv-fs01   192.168.1.11
```

Aquí s'hi apliquen: separació en mòduls, paquet amb API neta a `__init__.py`, importacions relatives, docstrings i metadades, `__main__` i gestió d'errors d'importació.

13. Resum

Concepte	Clau
Mòdul	Fitxer `.py` reutilitzable
Paquet	Directori de mòduls amb `__init__.py`
`import m`	Accés amb prefix: `m.f()`
`from m import f`	Accés directe: `f()`
`import m as x`	Àlies per abreujar o evitar col·lisions
`sys.path`	Llista ordenada de rutes on se cerquen mòduls
`dir(obj)`	Llista els noms d'un mòdul o objecte
`__name__`, `__file__`, `__doc__`	Metadades automàtiques
`if __name__ == "__main__":`	Codi que només s'executa com a programa
`ModuleNotFoundError`	No es troba el mòdul a `sys.path`
`importlib.reload(m)`	Torna a executar un mòdul ja importat

Versions d'aquest document

- HTML
- PDF
- ODT
- MD

Domini Públic (CC0)