Generalitat de Catalunya
Departament d'Educació
Institut Thos i Codina

El mòdul 'sys' de Python

Domini Públic (CC0)

Índex

1. Arguments de la línia d'ordres: `sys.argv` 1
Exemple amb validació . . . 1

2. Sortir del programa: `sys.exit()` 2
Què fa realment? . . . 2
Codis de sortida . . . 2

3. Entrada, sortida i error estàndard 3
Per què separar `stdout` i `stderr`? . . . 3
Llegir de `stdin` . . . 3
Buffering . . . 4
Codificació . . . 4
Redirigir temporalment la sortida . . . 4

4. Cerca de mòduls: `sys.path` i `sys.modules` 5
`sys.path` . . . 5
`sys.modules` . . . 5
Altres relacionats . . . 6

5. Informació de l'intèrpret 6
Comprovar la versió . . . 6
Detectar la plataforma . . . 6
Detectar si s'executa dins un entorn virtual . . . 7
Usar el mateix intèrpret per llançar subprocessos . . . 7

6. Memòria i límits 7
`sys.getsizeof()` . . . 7
`sys.getrefcount()` . . . 7
Límit de recursivitat . . . 8
Valors límit . . . 8

7. Excepcions i depuració 9
`sys.exc_info()` . . . 9
`sys.excepthook`: gestor global d'excepcions no capturades . . . 9
Traça d'execució . . . 10

8. Altres utilitats 10
Nom i arguments originals de l'intèrpret . . . 10
Optimitzacions i flags . . . 10
Enllaços als originals . . . 10
Sortida per consola d'objectes: `sys.displayhook` . . . 10

9. Exemples complets 11
9.1. Script amb comprovació de versió i ús correcte . . . 11
9.2. Filtre que processa `stdin` i escriu a `stdout` . . . 11
9.3. Informe de l'entorn Python . . . 12

10. Exercicis proposats 12

11. Resum 12

12. Referències

El mòdul `sys` dOna accés a paràmetres i funcions que depenen directament de **l'intèrpret de Python** i del seu entorn d'execució: arguments de la línia d'ordres, fluxos d'entrada i sortida estàndard, camins de cerca de mòduls, versió de l'intèrpret, límits interns, etc.

NOTA

Diferència amb os: `os` parla amb el **sistema operatiu** (fitxers, processos, variables d'entorn). `sys` parla amb **l'intèrpret de Python** (com s'ha llançat, què té carregat, com es comporta). Per a informació detallada de la màquina (CPU, nom del SO, arquitectura) existeix un tercer mòdul, `platform`.

```
import sys
```

1. Arguments de la línia d'ordres: `sys.argv`

`sys.argv` és una llista de cadenes amb els arguments passats al script.

```
# fitxer: saluda.py
import sys

print(sys.argv)
```

```
$ python saluda.py Ramon 42
['saluda.py', 'Ramon', '42']
```

- `sys.argv[0]` és el nom del script (tal com s'ha invocat). Si s'executa amb `python -c "..."`, val `'-c'`; amb `python -m modul`, és el camí complet del mòdul.
- Els elements següents (`argv[1:]`) són els arguments, **sempre com a cadenes**. Cal convertir-los: `int(sys.argv[2])`.

Exemple amb validació

```
import sys

if len(sys.argv) != 3:
    print(f"Ús: {sys.argv[0]} <nom> <edat>", file=sys.stderr)
    sys.exit(1)

nom = sys.argv[1]
try:
    edat = int(sys.argv[2])
except ValueError:
    print("L'edat ha de ser un enter", file=sys.stderr)
    sys.exit(2)

print(f"Hola {nom}, tens {edat} anys")
```

Per a programes amb opcions (`--verbose`, `-o fitxer`...) no cal parsejar `sys.argv` a mà: el mòdul `argparse` ho fa i genera l'ajuda automàticament. `sys.argv` és el que `argparse` fa servir per sota.

2. Sortir del programa: `sys.exit()`

```
sys.exit()        # codi de sortida 0 (èxit)
sys.exit(0)       # idem
sys.exit(1)       # error genèric
sys.exit("Error: fitxer no trobat")   # imprimeix el missatge a stderr
 ↪  i surt amb codi 1
```

Què fa realment?

`sys.exit()` **no mata el procés directament**: llança l'excepció `SystemExit`. L'intèrpret la captura al nivell més alt i acaba amb el codi indicat. Conseqüències:

1. **S'executen els blocs `finally`** i els gestors de context (`with`) pel camí, de manera que els fitxers es tanquen correctament.
2. Es pot interceptar amb `try/except SystemExit`.
3. Un `except Exception:` **no** la captura, perquè `SystemExit` hereta de `BaseException`, no d'`Exception`. En canvi, un `except:` sense tipus (o `except BaseException:`) sí que la captura, i per això és una mala pràctica.

```
try:
    sys.exit(3)
except SystemExit as e:
    print("Capturat, codi:", e.code)   # 3
```

Si necessites sortir immediatament sense netejar res (per exemple, en un procés fill després d'un `fork`), existeix `os._exit(codi)`.

Codis de sortida

Per convenció, `0` vol dir èxit i qualsevol altre valor és un error. La shell el recull a `$?`:

```
$ python script.py; echo $?
```

Si es passa `None` es considera `0`; si es passa un objecte que no és enter (com una cadena), s'imprimeix a `stderr` i el codi és `1`.

3. Entrada, sortida i error estàndard

`sys.stdin`, `sys.stdout` i `sys.stderr` són els objectes fitxer (flux de text) que connecten el programa amb la consola o amb les redireccions de la shell.

```
sys.stdout.write("Hola\n")          # com print, però sense afegir
 ↪  salt de línia
print("Hola", file=sys.stdout)      # equivalent al print per defecte
print("Alerta!", file=sys.stderr)   # missatges d'error
```

Per què separar `stdout` i `stderr`?

Perquè la shell els pot redirigir de manera independent:

```
$ python programa.py > sortida.txt 2> errors.txt
$ python programa.py | grep resultat      # el pipe només rep stdout
```

Així, els missatges de diagnòstic no embruten les dades que un altre programa vol consumir.

Llegir de `stdin`

```
# Llegir línia a línia (funciona amb pipes: cat fitxer | python
 ↪  script.py)
for linia in sys.stdin:
    print(linia.rstrip().upper())

# Llegir-ho tot de cop
dades = sys.stdin.read()
```

Exemple de filtre estil Unix:

```
import sys

total = 0
for linia in sys.stdin:
    linia = linia.strip()
    if linia:
        total += int(linia)
print(total)
```

```
$ printf "10\n20\n30\n" | python suma.py
60
```

Buffering

Quan la sortida va a un pipe o a un fitxer, Python l'acumula en un **buffer** i no l'escriu fins que s'omple o s'acaba el programa. Això explica per què de vegades un `print` "no apareix" en temps real en un log.

```
print("progrés...", flush=True)   # força l'escriptura immediata
sys.stdout.flush()                # equivalent
```

També es pot desactivar el buffering amb `python -u script.py` o amb la variable d'entorn `PYTHONUNBUFFERED=1`.

Codificació

```
print(sys.stdout.encoding)   # normalment 'utf-8'

# Forçar UTF-8 (Python 3.7+), útil quan la redirecció dóna problemes
 ↪  d'accents
sys.stdout.reconfigure(encoding="utf-8")
```

Redirigir temporalment la sortida

```
import io

captura = io.StringIO()
original = sys.stdout
sys.stdout = captura
try:
    print("això no es veu per pantalla")
finally:
    sys.stdout = original     # restaurar sempre

print("Capturat:", captura.getvalue())
```

Per a aquest cas, la forma neta és `contextlib.redirect_stdout`.

4. Cerca de mòduls: `sys.path` i `sys.modules`

`sys.path`

Llista de directoris on Python busca els mòduls quan fas `import`. S'inicialitza amb:

1. El directori del script que s'executa (a `sys.path[0]`; en mode interactiu o amb `-c` és la cadena buida, que vol dir el directori actual).
2. Els directoris de la variable d'entorn `PYTHONPATH`.
3. Els directoris per defecte de la instal·lació (biblioteca estàndard i `site-packages`).

```
for p in sys.path:
    print(p)
```

Afegir un directori en temps d'execució:

```
sys.path.append("/opt/meus_moduls")
import eina_meva
```

Això només afecta el procés actual. Si t'ho trobes sovint, és millor estructurar el projecte com a paquet instal·lable (`pip install -e .`) o usar `PYTHONPATH`.

IMPORTANT

Error típic: crear un fitxer propi anomenat `random.py`, `json.py` o `os.py` al costat del script. Com que el directori del script va primer a `sys.path`, el teu fitxer **ombreja** el mòdul de la biblioteca estàndard i apareixen errors incomprensibles.

`sys.modules`

Diccionari amb tots els mòduls ja carregats. És la memòria cau d'`import`: la segona vegada que importes un mòdul, Python el treu d'aquí en comptes de tornar a executar-lo.

```
print("json" in sys.modules)    # False si encara no s'ha importat
import json
print("json" in sys.modules)    # True

print(sys.modules["json"] is json)   # True
```

Per forçar la recàrrega d'un mòdul modificat (en depuració): `importlib.reload(modul)`.

Altres relacionats

```
print(sys.builtin_module_names)   # mòduls compilats dins l'intèrpret
print(sys.stdlib_module_names)    # noms de la biblioteca estàndard
 ↪  (Python 3.10+)
```

5. Informació de l'intèrpret

```
print(sys.version)         # cadena completa, amb compilador i data
print(sys.version_info)    # sys.version_info(major=3, minor=12,
 ↪  micro=3, ...)
print(sys.executable)      # camí de l'intèrpret en ús
print(sys.platform)        # 'linux', 'win32', 'darwin'...
print(sys.implementation.name)   # 'cpython', 'pypy'...
print(sys.byteorder)       # 'little' o 'big'
print(sys.getdefaultencoding())  # 'utf-8'
print(sys.prefix)          # directori d'instal·lació (o de l'entorn
 ↪  virtual)
```

Comprovar la versió

Compara **tuples**, no cadenes. Amb cadenes, `"3.10" < "3.9"` dona `True` perquè es compara caràcter a caràcter.

```
if sys.version_info < (3, 9):
    sys.exit("Cal Python 3.9 o superior")

if sys.version_info >= (3, 11):
    import tomllib
```

Detectar la plataforma

```
if sys.platform.startswith("linux"):
    print("Linux")
elif sys.platform == "win32":
    print("Windows (també en 64 bits)")
elif sys.platform == "darwin":
    print("macOS")
```

Detectar si s'executa dins un entorn virtual

Dins un `venv`, `sys.prefix` apunta a l'entorn, mentre que `sys.base_prefix` continua apuntant a la instal·lació base:

```
en_venv = sys.prefix != sys.base_prefix
print("Entorn virtual:", en_venv)
```

Usar el mateix intèrpret per llançar subprocessos

```
import subprocess
subprocess.run([sys.executable, "-m", "pip", "list"])
```

Això garanteix que s'usa el mateix Python (i el mateix entorn virtual) que el script en curs, en lloc de dependre del que trobi al `PATH`.

6. Memòria i límits

`sys.getsizeof()`

Retorna la mida en bytes d'un objecte, **sense incloure els objectes als quals fa referència** (és una mida superficial):

```
print(sys.getsizeof(0))          # ~28
print(sys.getsizeof("hola"))     # ~53
print(sys.getsizeof([]))         # ~56
print(sys.getsizeof([1, 2, 3]))  # mida de la llista, no dels enters
 ↪  que conté
```

Les mides exactes varien segons la versió i l'arquitectura. Per mesurar l'ús real d'una estructura complexa cal recórrer-la recursivament o usar eines com `tracemalloc` o `pympler`.

`sys.getrefcount()`

Nombre de referències a un objecte. Retorna un valor una unitat superior a "l'esperat", perquè el paràmetre de la crida en compta una de temporal.

```
a = []
print(sys.getrefcount(a))   # 2 (la variable a + l'argument)
b = a
print(sys.getrefcount(a))   # 3
```

Límit de recursivitat

Python limita la profunditat de la pila de crides per evitar que una recursió infinita esgoti la memòria del procés:

```
print(sys.getrecursionlimit())   # 1000 per defecte

def infinita(n):
    return infinita(n + 1)

try:
    infinita(0)
except RecursionError as e:
    print("RecursionError:", e)
```

Es pot canviar amb `sys.setrecursionlimit(5000)`, però **amb precaució**: si el límit és massa alt, la pila nativa de C pot desbordar-se i l'intèrpret fallarà amb un *segmentation fault* en lloc de llançar una excepció. Si necessites molta profunditat, és preferible reescriure l'algorisme de manera iterativa.

Valors límit

```
print(sys.maxsize)       # 9223372036854775807 en sistemes de 64 bits
print(sys.float_info.max)      # el float més gran representable
print(sys.float_info.epsilon)  # diferència mínima entre floats
 ↪  propers a 1.0
```

`sys.maxsize` és el valor màxim de `Py_ssize_t` (mida màxima de contenidors). **No és el màxim d'un enter de Python**: els `int` de Python tenen precisió arbitrària.

NOTA

Des de Python 3.11 hi ha un límit per defecte de 4300 dígits en la conversió entre `int` i `str` (protecció contra atacs de denegació de servei). Es consulta amb `sys.get_int_max_str_digits()` i es canvia amb `sys.set_int_max_str_digits()`.

7. Excepcions i depuració

`sys.exc_info()`

Dins un bloc `except`, retorna una tupla `(tipus, valor, traceback)` de l'excepció que s'està gestionant:

```
try:
    1 / 0
except ZeroDivisionError:
    tipus, valor, tb = sys.exc_info()
    print(tipus.__name__, valor)   # ZeroDivisionError division by zero
    print(tb.tb_lineno)            # línia on s'ha produït
```

En codi nou sol bastar capturar l'excepció amb `except ... as e` i usar el mòdul `traceback`.

`sys.excepthook`: gestor global d'excepcions no capturades

Quan una excepció arriba al nivell més alt sense ser capturada, Python crida `sys.excepthook`. Es pot substituir per, per exemple, escriure-ho a un log:

```
import sys
import logging

logging.basicConfig(filename="errors.log", level=logging.ERROR)

def gestor(tipus, valor, tb):
    if issubclass(tipus, KeyboardInterrupt):
        sys.__excepthook__(tipus, valor, tb)   # mantenir el
        ↪  comportament normal de Ctrl+C
        return
    logging.error("Excepció no capturada", exc_info=(tipus, valor,
    ↪  tb))

sys.excepthook = gestor

raise RuntimeError("alguna cosa ha anat malament")
```

Traça d'execució

```
def traça(frame, event, arg):
    if event == "call":
        print("Crida a", frame.f_code.co_name)
    return traça

sys.settrace(traça)
```

`sys.settrace` és el mecanisme sobre el qual es construeixen els depuradors i les eines de cobertura. No és d'ús habitual en programes normals.

8. Altres utilitats

Nom i arguments originals de l'intèrpret

```
print(sys.orig_argv)   # Python 3.10+: inclou les opcions de
 ↪  l'intèrpret, p. ex. ['python', '-u', 'script.py']
```

Optimitzacions i flags

```
print(sys.flags)            # opcions amb què s'ha llançat l'intèrpret
print(sys.flags.optimize)     # 0, 1 o 2 segons -O / -OO
```

Enllaços als originals

`sys.__stdout__`, `sys.__stderr__`, `sys.__stdin__` i `sys.__excepthook__` conserven els valors originals, útils per restaurar després d'haver-los substituït.

Sortida per consola d'objectes: `sys.displayhook`

És la funció que s'invoca en mode interactiu per mostrar el resultat de cada expressió. Es pot personalitzar, però és un ús molt avançat.

9. Exemples complets

9.1. Script amb comprovació de versió i ús correcte

```
#!/usr/bin/env python3
import sys

if sys.version_info < (3, 8):
    sys.exit("Aquest programa requereix Python 3.8 o superior")

def main(args):
    if len(args) != 1:
        print(f"Ús: {sys.argv[0]} <fitxer>", file=sys.stderr)
        return 2
    try:
        with open(args[0], encoding="utf-8") as f:
            print(sum(1 for _ in f), "línies")
    except FileNotFoundError:
        print(f"No existeix: {args[0]}", file=sys.stderr)
        return 1
    return 0

if __name__ == "__main__":
    sys.exit(main(sys.argv[1:]))
```

El patró `sys.exit(main())` és molt habitual: `main` retorna el codi de sortida i queda net i testejable.

9.2. Filtre que processa `stdin` i escriu a `stdout`

```
import sys

for numero, linia in enumerate(sys.stdin, start=1):
    sys.stdout.write(f"{numero:4d}  {linia}")
```

```
$ cat /etc/hostname | python numera.py
   1  srv-dc01
```

9.3. Informe de l'entorn Python

```
import sys

print(f"Python:        {sys.version.split()[0]}")
print(f"Implementació: {sys.implementation.name}")
print(f"Executable:    {sys.executable}")
print(f"Plataforma:    {sys.platform}")
print(f"Entorn virtual: {'sí' if sys.prefix != sys.base_prefix else
 ↪  'no'}")
print(f"Codificació:   {sys.getdefaultencoding()}")
print(f"Límit recursió: {sys.getrecursionlimit()}")
print("Camins de cerca:")
for p in sys.path:
    print("  ", p or "(directori actual)")
```

10. Exercicis proposats

1. Escriu un script que rebi un nombre variable d'arguments numèrics per la línia d'ordres i en mostri la suma. Si algun no és vàlid, ha de sortir amb codi 1 i un missatge a `stderr`.
2. Fes un programa que llegeixi `stdin` i compti paraules, línies i caràcters (un `wc` senzill).
3. Comprova a mà que `sys.exit()` executa els blocs `finally`, i que un `except Exception` no atrapa `SystemExit`.
4. Crea un fitxer `random.py` a la carpeta d'un script que faci `import random` i explica per què falla. Mira `sys.path[0]` per confirmar-ho.
5. Escriu una funció recursiva que superi el límit de recursivitat; captura `RecursionError` i després reescriu-la de forma iterativa.
6. Instal·la un `sys.excepthook` que guardi les excepcions no capturades en un fitxer de log amb la data i l'hora.

11. Resum

Vull…	Eina
Llegir els arguments del programa	`sys.argv`
Acabar el programa amb un codi	`sys.exit(codi)`
Escriure errors separats de la sortida	`print(..., file=sys.stderr)`
Llegir dades d'un pipe	`sys.stdin`
Saber on busca mòduls Python	`sys.path`
Veure mòduls ja carregats	`sys.modules`
Comprovar la versió de Python	`sys.version_info >= (3, X)`
Saber la plataforma	`sys.platform`
Camí de l'intèrpret actual	`sys.executable`
Detectar un entorn virtual	`sys.prefix != sys.base_prefix`
Mida d'un objecte	`sys.getsizeof(obj)`
Consultar/canviar el límit de recursió	`sys.getrecursionlimit()` / `sys.setrecursionlimit(n)`
Gestionar excepcions no capturades	`sys.excepthook`

12. Referències

- Documentació oficial: https://docs.python.org/es/3.15/library/sys.html
- `argparse`: https://docs.python.org/es/3.15/library/argparse.html
- `platform`: https://docs.python.org/es/3.15/library/platform.html
- `traceback`: https://docs.python.org/es/3.15/library/traceback.html

Versions d'aquest document

- HTML
- PDF
- ODT
- MD

Domini Públic (CC0)