Generalitat de Catalunya
Departament d'Educació
Institut Thos i Codina

El mòdul 'subprocess' de Python

Domini Públic (CC0)

Índex

1. `subprocess.run()`: el cas habitual **1**
Capturar la sortida . . . 1
L'objecte `CompletedProcess` . . . 2
Els codis de sortida . . . 2

2. Llista d'arguments o cadena? `shell=False` vs. `shell=True` **2**
Forma recomanada: llista, sense shell . . . 2
Amb shell: només quan realment cal . . . 2
Convertir una cadena en llista: `shlex.split` . . . 3
Substituir pipes i redireccions de la shell . . . 3

3. Gestionar errors **4**
Ordre que falla: `check=True` . . . 4
El programa no existeix: `FileNotFoundError` . . . 4
Temps màxim: `timeout` . . . 4
Jerarquia d'excepcions . . . 5

4. Entrada, directori de treball i entorn **5**
Enviar dades a `stdin`: `input` . . . 5
Directori de treball: `cwd` . . . 5
Variables d'entorn: `env` . . . 5
Redirigir fluxos . . . 6

5. Popen: control fi del procés **6**
Mètodes i atributs principals . . . 6
`communicate()` i el risc de bloqueig (*deadlock*) . . . 7
`communicate()` amb `timeout` . . . 7
Com a gestor de context . . . 7

6. Llegir la sortida en temps real **8**

7. Connectar processos amb un pipe (`ordre1 | ordre2`) **8**

8. Funcions antigues (només per reconèixer-les) **9**

9. Altres opcions útils **9**

10. Exemples complets **10**
10.1. Funció auxiliar robusta . . . 10
10.2. Estat d'un repositori Git . . . 10
10.3. Comprovar si una màquina respon al ping . . . 11
10.4. Executar diverses tasques en paral·lel . . . 11
10.5. Redirigir sortida i errors a fitxers separats . . . 11

11. Errors habituals **12**

12. Exercicis proposats **12**

13. Resum **13**

14. Referències

El mòdul `subprocess` permet **llançar programes externs** des de Python, connectar-ne les entrades i sortides, esperar que acabin i recollir-ne el codi de sortida. És l'eina estàndard per automatitzar tasques d'administració (cridar `git`, `ping`, `rsync`, `systemctl`, `ffmpeg`...) i substitueix mètodes antics i insegurs com `os.system()` o `os.popen()`.

```
import subprocess
```

NOTA

Un procés fill és un programa independent amb el seu propi espai de memòria, el seu propi entorn i els seus propis fluxos `stdin`, `stdout` i `stderr`. `subprocess` el crea (internament amb `fork+exec` a Unix o `CreateProcess` a Windows) i et dona eines per comunicar-t'hi.

1. `subprocess.run()`: el cas habitual

Des de Python 3.5, `run()` és la funció recomanada per al 90 % dels casos: llança l'ordre, **espera que acabi** i retorna un objecte `CompletedProcess`.

```
r = subprocess.run(["ls", "-l", "/tmp"])
print(r.returncode)   # 0 si ha anat bé
```

Per defecte, la sortida del fill va directament a la terminal del programa pare.

Capturar la sortida

```
r = subprocess.run(
    ["ls", "-l", "/tmp"],
    capture_output=True,   # Python 3.7+: equival a stdout=PIPE,
     ↪  stderr=PIPE
    text=True,             # decodifica bytes a str
)

print(r.stdout)       # sortida estàndard
print(r.stderr)       # sortida d'error
print(r.returncode)   # codi de sortida
```

- Sense `text=True`, `stdout` i `stderr` són **bytes** (`b"..."`).
- `text=True` usa la codificació preferida del sistema. Si el programa produeix UTF-8 i no és la codificació del sistema, indica-ho: `encoding="utf-8"`, i, si cal, `errors="replace"`.

L'objecte `CompletedProcess`

Atribut	Significat
`args`	Els arguments amb què s'ha llançat
`returncode`	Codi de sortida (0 = èxit)
`stdout`	Sortida estàndard capturada (o `None`)
`stderr`	Sortida d'error capturada (o `None`)

Els codis de sortida

Per convenció, **0 és èxit** i qualsevol altre valor és un error. A POSIX, si el procés ha mort per un senyal, `returncode` és **negatiu**: `-9` vol dir que l'ha matat `SIGKILL`, `-15` `SIGTERM`, `-11` un *segmentation fault*.

2. Llista d'arguments o cadena? `shell=False` vs. `shell=True`

Aquesta és la decisió de seguretat més important del mòdul.

Forma recomanada: llista, sense shell

```
subprocess.run(["grep", "-r", "error", "/var/log"])
```

Cada element de la llista és **un argument**, exactament com arribarà al programa. No intervé cap shell: no s'interpreten espais, cometes, `*`, `|`, `;` ni $. Això evita el problema de l'*escapament* i, sobretot, la **injecció d'ordres**.

Amb shell: només quan realment cal

```
subprocess.run("ls -l | grep .py", shell=True)
```

Amb `shell=True`, la cadena es passa a `/bin/sh -c` (a Windows, `cmd.exe`), de manera que funcionen `|`, `>`, `*`, variables, etc. Però:

> **Perill: injecció d'ordres.** Si hi barreges dades de l'usuari, qualsevol pot executar el que vulgui.

```
nom = input("Fitxer: ")            # l'usuari escriu: a.txt;
 ↪  rm -rf ~
subprocess.run(f"cat {nom}", shell=True)   # ¡executa també
 ↪  rm -rf ~!
```

> Amb llista no hi ha problema: `subprocess.run(["cat", nom])` tractaria tot el text com **un únic nom de fitxer**.

Regla pràctica: **fes servir sempre la llista**. Si has d'usar `shell=True`, que la cadena sigui una constant del programa i no contingui res que vingui de fora. Si necessites escapar un valor, `shlex.quote()` ho fa:

```
import shlex
subprocess.run(f"cat {shlex.quote(nom)}", shell=True)
```

Convertir una cadena en llista: `shlex.split`

```
import shlex

ordre = 'grep -r "hola món" /home/ramon/docs'
args = shlex.split(ordre)
print(args)   # ['grep', '-r', 'hola món', '/home/ramon/docs']

subprocess.run(args)
```

L'invers és `shlex.join(args)` (Python 3.8+), útil per mostrar l'ordre en un log.

Substituir pipes i redireccions de la shell

Fer en Python allò que la shell faria amb `|` o `>` sol ser més segur i més portable:

```
# En lloc de: "ls -l | grep .py"
r = subprocess.run(["ls", "-l"], capture_output=True, text=True)
linies = [l for l in r.stdout.splitlines() if ".py" in l]

# En lloc de: "ordre > sortida.txt"
with open("sortida.txt", "w") as f:
    subprocess.run(["ordre"], stdout=f)
```

3. Gestionar errors

Ordre que falla: `check=True`

Per defecte, `run()` **no llança cap excepció** si el programa acaba amb error: només guarda el codi a `returncode`. És fàcil oblidar-se de mirar-lo. Amb `check=True`, si el codi no és 0 es llança `CalledProcessError`:

```
try:
    subprocess.run(["ls", "/no/existeix"], check=True,
     ↪  capture_output=True, text=True)
except subprocess.CalledProcessError as e:
    print("Codi:", e.returncode)
    print("Ordre:", e.cmd)
    print("Error:", e.stderr)
```

El programa no existeix: `FileNotFoundError`

```
try:
    subprocess.run(["programa_inexistent"])
except FileNotFoundError:
    print("L'ordre no existeix o no és al PATH")
```

Amb `shell=False`, si l'executable no es troba, es llança `FileNotFoundError`. Amb `shell=True`, en canvi, la shell s'executa correctament i retorna el codi **127** ("command not found"): per això és més fàcil no adonar-se de l'error.

Es pot comprovar abans amb `shutil.which`:

```
import shutil
if shutil.which("git") is None:
    raise SystemExit("Cal instal·lar git")
```

Temps màxim: `timeout`

```
try:
    subprocess.run(["sleep", "10"], timeout=3)
except subprocess.TimeoutExpired:
    print("L'ordre ha trigat massa")
```

Quan expira el temps, `run()` **mata el fill** i després llança `TimeoutExpired`. Compte: si el fill ha engegat al seu torn altres processos (com amb `shell=True`), aquests nets poden quedar vius.

Jerarquia d'excepcions

Excepció	Quan
`FileNotFoundError`	L'executable no existeix (sense shell)
`PermissionError`	L'executable no té permís d'execució
`CalledProcessError`	Codi de sortida ≠ 0 amb `check=True`
`TimeoutExpired`	S'ha superat el `timeout`
`SubprocessError`	Classe base de les dues excepcions de `subprocess`

4. Entrada, directori de treball i entorn

Enviar dades a `stdin`: `input`

```
r = subprocess.run(
    ["sort"],
    input="pera\nmanzana\nbanana\n",
    capture_output=True,
    text=True,
)
print(r.stdout)
```

Amb `text=True`, `input` és una cadena; sense, ha de ser `bytes`.

Directori de treball: cwd

```
subprocess.run(["git", "status"], cwd="/home/ramon/projecte")
```

El directori només s'aplica al fill. El del programa pare no canvia.

Variables d'entorn: env

`env` **substitueix tot l'entorn** del fill. Si només vols afegir o canviar una variable, parteix de l'entorn actual:

```
import os

entorn = {**os.environ, "LANG": "C", "MEVA_VAR": "valor"}
subprocess.run(["programa"], env=entorn)
```

Si passes `env={"MEVA_VAR": "valor"}` sol, el fill es quedarà sense `PATH`, `HOME`, etc., i molts programes deixaran de funcionar.

Redirigir fluxos

Valor	Efecte
`subprocess.PIPE`	Captura el flux per llegir-lo des de Python
`subprocess.DEVNULL`	Descarta el flux
`subprocess.STDOUT`	(només per a `stderr`) envia l'error al mateix lloc que `stdout`
un objecte fitxer o descriptor	Redirigeix al fitxer

```
# Descartar tota la sortida
subprocess.run(["programa"], stdout=subprocess.DEVNULL,
 ↪  stderr=subprocess.DEVNULL)

# Barrejar stderr dins stdout (com 2>&1)
r = subprocess.run(["programa"], stdout=subprocess.PIPE,
 ↪  stderr=subprocess.STDOUT, text=True)
print(r.stdout)    # inclou missatges d'error

# Escriure sortida a un fitxer de log
with open("execucio.log", "w") as log:
    subprocess.run(["programa"], stdout=log, stderr=subprocess.STDOUT)
```

5. Popen: control fi del procés

`run()` espera que el procés acabi. Si necessites **llegir la sortida mentre s'executa**, llançar diversos processos alhora, o interactuar amb el fill, cal `subprocess.Popen`, que és la classe sobre la qual `run()` està construïda.

```
p = subprocess.Popen(["ping", "-c", "3", "8.8.8.8"],
 ↪  stdout=subprocess.PIPE, text=True)
```

`Popen` **no espera**: retorna immediatament amb l'objecte procés.

Mètodes i atributs principals

Element	Funció
`p.pid`	Identificador del procés
`p.poll()`	Retorna el codi de sortida si ha acabat, o `None` si encara corre
`p.wait(timeout=None)`	Espera que acabi i retorna el codi
`p.communicate(input=None, timeout=None)`	Envia dades, llegeix tota la sortida i espera que acabi
`p.terminate()`	Envia `SIGTERM` (petició educada d'aturada)
`p.kill()`	Envia `SIGKILL` (mort immediata, no es pot ignorar)
`p.send_signal(sig)`	Envia el senyal que vulguis
`p.returncode`	Codi de sortida (`None` mentre corre)
`p.stdin`, `p.stdout`, `p.stderr`	Fluxos (si s'ha demanat `PIPE`)

`communicate()` i el risc de bloqueig (*deadlock*)

```
p = subprocess.Popen(["ls", "-l"], stdout=subprocess.PIPE,
 ↪  stderr=subprocess.PIPE, text=True)
sortida, errors = p.communicate()
print(p.returncode)
```

NOTA

Per què no `p.wait()` amb `PIPE`? Els *pipes* tenen un buffer finit (típicament 64 KiB a Linux). Si el fill escriu més del que cap al buffer i el pare només fa `wait()` sense llegir, el fill es queda bloquejat esperant que algú llegeixi, i el pare esperant que el fill acabi: **bloqueig mutu**. `communicate()` llegeix `stdout` i `stderr` alhora i evita el problema. Si no necessites la sortida, no la demanis amb `PIPE`: envia-la a `DEVNULL` o a un fitxer.

`communicate()` amb `timeout`

A diferència de `run()`, aquí el fill **no es mata** quan expira el temps. Cal fer-ho tu:

```
p = subprocess.Popen(["sleep", "30"], stdout=subprocess.PIPE,
 ↪  text=True)
try:
    p.communicate(timeout=3)
except subprocess.TimeoutExpired:
    p.kill()
    p.communicate()    # recull el que quedi i evita processos zombi
```

Com a gestor de context

```
with subprocess.Popen(["ls", "-l"], stdout=subprocess.PIPE,
 ↪  text=True) as p:
    for linia in p.stdout:
        print(linia.rstrip())
# en sortir, es tanquen els pipes i s'espera el procés
```

6. Llegir la sortida en temps real

```
import subprocess

with subprocess.Popen(
    ["ping", "-c", "4", "8.8.8.8"],
    stdout=subprocess.PIPE,
    stderr=subprocess.STDOUT,
    text=True,
    bufsize=1,            # buffer per línies (només amb text=True)
) as p:
    for linia in p.stdout:
        print("[ping]", linia.rstrip())

print("Codi de sortida:", p.returncode)
```

NOTA

Per què de vegades no apareix res fins al final? Molts programes (inclosos els scripts de Python) usen **buffering complet** quan la seva sortida no és una terminal, sinó un pipe. Així, el fill acumula el text i no l'envia fins que s'omple el buffer o acaba. Ho has de resoldre en el programa fill: per a un script Python, `python -u script.py` o `PYTHONUNBUFFERED=1`; per a altres programes, mira si tenen una opció `--line-buffered`, o usa l'eina `stdbuf -oL`.

7. Connectar processos amb un pipe (`ordre1 | ordre2`)

```
p1 = subprocess.Popen(["ps", "aux"], stdout=subprocess.PIPE)
p2 = subprocess.Popen(["grep", "python"], stdin=p1.stdout,
 ↪  stdout=subprocess.PIPE, text=True)

p1.stdout.close()        # molt important (vegeu sota)
sortida, _ = p2.communicate()
print(sortida)
```

Es tanca `p1.stdout` al pare perquè, si `p2` acaba abans, `p1` rebi `SIGPIPE` i s'aturi, en lloc de quedar-se escrivint a un pipe sense lector. Sense aquesta línia, el pare manté una còpia oberta del pipe.

Tanmateix, si la canonada és fixa i de confiança, sovint és més senzill fer-ho en Python (com a l'apartat 2) o, de manera acceptable, amb `shell=True` i una cadena constant.

8. Funcions antigues (només per reconèixer-les)

Antiga	Equivalent actual
`subprocess.call(args)`	`subprocess.run(args).returncode`
`subprocess.check_call(args)`	`subprocess.run(args, check=True)`
`subprocess.check_output(args)`	`subprocess.run(args, check=True, capture_output=True).stdout`
`os.system("ordre")`	`subprocess.run(["ordre", ...])`
`os.popen("ordre")`	`subprocess.run(..., capture_output=True)` o `Popen`
`commands.getoutput` (Python 2)	`subprocess.run(..., capture_output=True, text=True)`

`check_output` encara és molt comú i és vàlid; retorna directament la sortida i llança `CalledProcessError` si falla.

```
versio = subprocess.check_output(["git", "--version"],
↪  text=True).strip()
```

9. Altres opcions útils

- **Llançar el mateix intèrpret de Python:**

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

  Usar `sys.executable` garanteix que s'executa el mateix Python (i el mateix entorn virtual) que el script actual.

- **Processos fora de control del pare** (`start_new_session=True` a POSIX): el fill es desvincula de la terminal del pare i no rep els seus senyals (com `Ctrl+C`). A Windows s'usa `creationflags`.
- **Aplicacions asíncrones:** si el teu programa usa `asyncio`, existeix `asyncio.create_subprocess_exec()` i `asyncio.create_subprocess_shell()`, que no bloquegen el bucle d'esdeveniments.
- **`preexec_fn`**: permet executar codi al fill abans d'`exec`, però **no és segur amb fils** i s'ha desaconsellat. Per a la majoria dels usos (canviar d'usuari, límits…), existeixen paràmetres directes com `user=`, `group=` o `start_new_session=`.

10. Exemples complets

10.1. Funció auxiliar robusta

```
import subprocess
import sys

def executa(ordre, timeout=60):
    """Executa una ordre i retorna stdout. Surt amb un missatge clar
     ↪  si falla."""
    try:
        r = subprocess.run(
            ordre,
            capture_output=True,
            text=True,
            encoding="utf-8",
            errors="replace",
            timeout=timeout,
            check=True,
        )
    except FileNotFoundError:
        sys.exit(f"Ordre no trobada: {ordre[0]}")
    except subprocess.TimeoutExpired:
        sys.exit(f"Temps esgotat ({timeout} s): {' '.join(ordre)}")
    except subprocess.CalledProcessError as e:
        sys.exit(f"Error {e.returncode} a {'
         ↪  '.join(ordre)}:\n{e.stderr}")
    return r.stdout

print(executa(["uname", "-a"]))
```

10.2. Estat d'un repositori Git

```
import subprocess

def git(*args, cwd="."):
    return subprocess.run(
        ["git", *args], cwd=cwd, capture_output=True, text=True,
         ↪  check=True
    ).stdout.strip()

print("Branca:", git("rev-parse", "--abbrev-ref", "HEAD"))
print("Últim commit:", git("log", "-1", "--format=%h %s"))
canvis = git("status", "--porcelain")
print("Canvis pendents:" if canvis else "Repositori net")
print(canvis)
```

10.3. Comprovar si una màquina respon al ping

```
import subprocess

def respon(host):
    r = subprocess.run(
        ["ping", "-c", "1", "-W", "2", host],
        stdout=subprocess.DEVNULL,
        stderr=subprocess.DEVNULL,
    )
    return r.returncode == 0

for host in ["srv-dc01", "srv-fs01", "8.8.8.8"]:
    print(f"{host:12} {'OK' if respon(host) else 'sense resposta'}")
```

En Windows, la bandera del recompte és `-n` i el temps màxim `-w` (en mil·lisegons).

10.4. Executar diverses tasques en paral·lel

```
import subprocess

hosts = ["8.8.8.8", "1.1.1.1", "9.9.9.9"]

processos = {
    h: subprocess.Popen(
        ["ping", "-c", "3", h],
        stdout=subprocess.PIPE, stderr=subprocess.STDOUT, text=True,
    )
    for h in hosts
}

for host, p in processos.items():
    sortida, _ = p.communicate()
    print(f"== {host} (codi {p.returncode}) ==")
    print(sortida.splitlines()[-1])
```

Els tres `ping` s'executen alhora i només s'espera a recollir els resultats al final.

10.5. Redirigir sortida i errors a fitxers separats

```
import subprocess

with open("sortida.log", "w") as out, open("errors.log", "w") as err:
    r = subprocess.run(["tar", "czf", "backup.tar.gz",
     ↪  "/home/ramon/docs"],
                       stdout=out, stderr=err)

print("Codi:", r.returncode)
```

11. Errors habituals

1. **Oblidar `check=True`** i continuar com si l'ordre hagués anat bé. Mira sempre `returncode` o activa `check`.
2. **`shell=True` amb dades de l'usuari** → injecció d'ordres. Usa llista.
3. **Passar una llista amb `shell=True` (POSIX):** només el primer element s'executa com a ordre, la resta es converteixen en arguments de la *shell*, no de l'ordre. És un error silenciós molt comú.
4. **Passar una cadena sense `shell=True`** (a POSIX): Python intenta executar un programa que es diu exactament com la cadena sencera (`"ls -l"`), i falla amb `FileNotFoundError`.
5. **`wait()` amb `PIPE` ple** → bloqueig. Usa `communicate()`.
6. **Treballar amb bytes sense adonar-te'n**: oblidar `text=True` i després comparar amb cadenes (`"error" in r.stdout` dona `TypeError`).
7. **Substituir tot l'entorn** amb `env={...}` i perdre `PATH`.
8. **Processos orfes/zombis:** no esperar un `Popen` (ni amb `wait()`/`communicate()` ni com a context manager) deixa processos zombi fins que el pare acaba.
9. **Esperar sortida en temps real** d'un programa que fa buffering quan va a un pipe.
10. **Problemes de codificació** (accents que es veuen malament): indica `encoding="utf-8"` o `errors="replace"`.

12. Exercicis proposats

1. Escriu un script que executi `df -h` i imprimeixi només les particions amb més del 80 % d'ocupació.
2. Fes una funció `existeix_servei(nom)` que usi `systemctl is-active` i retorni `True` o `False` segons el codi de sortida.
3. Llança `ping` a una llista d'adreces **en paral·lel** amb `Popen` i mostra un resum final.
4. Escriu un programa que executi una ordre amb un `timeout` de 5 segons i, si expira, mati el procés i registri l'incident en un fitxer de log.
5. Demostra la injecció d'ordres amb `shell=True` usant una ordre inofensiva (`echo`) i corregeix-ho amb una llista.
6. Llegeix en temps real la sortida d'un programa llarg (per exemple `ping -c 10`) i mostra-la amb una marca de temps davant de cada línia.
7. Implementa `ordre1 | ordre2` amb dos `Popen` connectats (p. ex. `cat /etc/passwd | cut -d: -f1`) i compara-ho amb una versió en Python pur.

13. Resum

Vull…	Eina
Executar una ordre i esperar	`subprocess.run([...])`
Capturar sortida com a text	`capture_output=True, text=True`
Fallar si l'ordre falla	`check=True`
Limitar el temps	`timeout=segons`
Enviar dades per `stdin`	`input="..."`
Canviar de directori / entorn	`cwd=...` / `env={**os.environ, ...}`
Descartar la sortida	`stdout=subprocess.DEVNULL`
Barrejar errors i sortida	`stderr=subprocess.STDOUT`
Llegir sortida en temps real	`Popen(..., stdout=PIPE)` i iterar `p.stdout`
Comunicar-me amb un procés sense bloquejar	`p.communicate()`
Aturar un procés	`p.terminate()` / `p.kill()`
Convertir una cadena en llista d'arguments	`shlex.split(cadena)`
Escapar un valor per a la shell	`shlex.quote(valor)`
Usar el mateix Python	`[sys.executable, ...]`

14. Referències

- Documentació oficial de `subprocess`: https://docs.python.org/es/3.15/library/subprocess.html
- `shlex`: https://docs.python.org/es/3.15/library/shlex.html
- `asyncio` i subprocessos: https://docs.python.org/es/3.15/library/asyncio-subprocess.html
- `shutil.which`: https://docs.python.org/es/3.15/library/shutil.html#shutil.which

Versions d'aquest document

- HTML
- PDF
- ODT
- MD

Domini Públic (CC0)