Gerne! Hier kommt ein ausführlicher Lernzettel zu **`subprocess`** in Python — so detailliert und praxisnah wie der zu `concurrent.futures`.

Lernzettel: Python-Modul `subprocess`

---

1. Was ist `subprocess`?

- `subprocess` ist ein Standardmodul in Python, mit dem du externe Programme, Skripte oder Shell-Befehle ausführen kannst.
- Du kannst damit z.B. Bash-Skripte starten, Kommandozeilen-Tools aufrufen oder komplexe Shell-Kommandos ausführen.

---

2. Warum `subprocess` verwenden?

- Es ersetzt veraltete Funktionen wie `os.system()` und ist flexibler und sicherer.
- Du kannst Eingaben an das gestartete Programm geben, Ausgaben abfangen und Fehler kontrollieren.
- Du kannst sowohl synchron (Warten auf Fertigstellung) als auch asynchron arbeiten.

---

3. Grundlegende Funktionen

- `subprocess.run()` — führt ein Kommando aus und wartet, bis es fertig ist.
- `subprocess.Popen()` — startet ein Programm und erlaubt dir, später mit dem Prozess zu interagieren.
- `subprocess.call()` — wie `run()`, aber gibt nur den Rückgabecode zurück (ältere Methode).

---

4. `subprocess.run()` — Das Standardwerkzeug

```
import subprocess

# Einfaches Kommando ausführen und auf Ergebnis warten
result = subprocess.run(["ls", "-l", "/home/user"], capture_output=True,
text=True)

print("Rückgabecode:", result.returncode)
print("Standardausgabe:")
print(result.stdout)
print("Fehlerausgabe:")
print(result.stderr)
```

Erklärung:

- `["ls", "-l", "/home/user"]` ist eine Liste von Argumenten (Programm + Parameter).
- `capture_output=True` fängt stdout und stderr ab.
- `text=True` gibt die Ausgaben als Strings (statt Bytes) zurück.
- `result.returncode` ist 0 bei Erfolg, sonst Fehlercode.

---

5. Wichtige Parameter von `subprocess.run()`

Parameter	Funktion
`args`	Liste oder String mit dem Befehl und Argumenten
`capture_output`	stdout und stderr abfangen (True/False)
`stdout`	z.B. `subprocess.PIPE` um stdout zu capturen
`stderr`	wie stdout für Fehlerausgaben
`text` oder `universal_newlines`	Ausgaben als Text (statt Bytes)
`shell`	Kommando durch Shell ausführen (Achtung: Sicherheitsrisiko)
`timeout`	Max. Zeit in Sekunden, nach der abgebrochen wird
`check`	Bei Fehler (returncode != 0) eine Exception werfen
`input`	Daten an das Programm über stdin schicken

---

6. Beispiel: Kommando mit Timeout und Fehlerbehandlung

```
try:
    result = subprocess.run(
        ["sleep", "5"],  # Programm, das 5 Sekunden wartet
        timeout=3,       # Wir erlauben nur 3 Sekunden
        check=True       # Fehler werfen, wenn returncode != 0
    )
except subprocess.TimeoutExpired:
    print("Das Kommando hat zu lange gebraucht und wurde abgebrochen!")
except subprocess.CalledProcessError as e:
    print(f"Kommando schlug fehl mit Fehlercode {e.returncode}")
else:
    print("Kommando erfolgreich beendet")
```

---

7. `subprocess.Popen()` — Für mehr Kontrolle

- `Popen` startet den Prozess, ohne auf Fertigstellung zu warten.
- Du kannst Ein-/Ausgaben manuell lesen oder schreiben.
- Ideal für asynchrone oder interaktive Programme.

```
import subprocess
```

```
p = subprocess.Popen(["grep", "python"], stdin=subprocess.PIPE,
stdout=subprocess.PIPE, text=True)

# Schreibe Text in das Programm (stdin)
stdout, stderr = p.communicate("python ist toll\njava auch\npython rocks\n")

print("Ergebnis von grep:")
print(stdout)
```

Erklärung:

- `communicate()` sendet Daten an stdin und liest stdout/stderr, wartet bis Prozess fertig ist.
- Ohne `communicate()` kannst du mit `p.stdin.write()`, `p.stdout.read()` etc. arbeiten, musst dann aber selbst auf Prozessende warten.

---

8. `shell=True` — Wann und wie?

- Wenn du z.B. einen Shell-Befehl mit Pipes oder Variablen ausführst, brauchst du oft `shell=True`:

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

- **Warnung:** Benutze `shell=True` nur, wenn du die Eingaben kontrollierst — sonst kann es zu Sicherheitslücken (Injection) kommen!

---

9. Rückgabewerte verstehen

- `returncode` ist 0 → Erfolg
- `returncode` ungleich 0 → Fehler, oft systemabhängig
- Exceptions: `TimeoutExpired`, `CalledProcessError` (wenn `check=True` gesetzt ist)

---

10. Beispiel: Bash-Skript ausführen und Rückgabewert prüfen

```
result = subprocess.run(["/path/to/script.sh"], capture_output=True, text=True)

if result.returncode == 0:
    print("Script lief erfolgreich:")
    print(result.stdout)
else:
    print("Script schlug fehl:")
    print(result.stderr)
```

---

11. Nützliche Patterns für deine Bash-Skripte im Python-Programm

- **Input an Skript übergeben:**

  ```
  subprocess.run(["./myscript.sh", "argument1", "argument2"])
  ```

- **Output aus Skript lesen:**

```
result = subprocess.run(["./myscript.sh"], capture_output=True, text=True)
print(result.stdout)
```

- **Mit Fehler umgehen:**

```
try:
    subprocess.run(["./myscript.sh"], check=True)
except subprocess.CalledProcessError:
    print("Skript ist fehlgeschlagen!")
```

- **Timeout setzen:**

```
subprocess.run(["./myscript.sh"], timeout=10)
```

---

12. Häufig genutzte Module mit `subprocess`

- `os` für Pfad- und Umgebungsvariablen
- `shlex` um Strings sicher in Kommando-Argumentlisten zu zerlegen (wichtig bei `shell=False`)
- `logging` für Ausgabe- und Fehlerprotokollierung

---

13. Zusammenfassung

- `subprocess` ist dein Werkzeug, wenn du externe Prozesse, Bash-Skripte oder Shell-Befehle aus Python starten willst.
- Nutze `subprocess.run()` für einfache, synchrone Ausführung mit gutem Fehlerhandling.
- Nutze `subprocess.Popen()` für mehr Kontrolle (z.B. Kommunikation mit dem Prozess).
- Vermeide wenn möglich `shell=True` aus Sicherheitsgründen.
- Fang Fehler und Timeouts ab, um robuste Programme zu schreiben.
- Prüfe Rückgabewerte und Fehlerausgaben, besonders bei der Arbeit mit Bash-Skripten.

Falls du magst, kann ich dir noch Beispiele zeigen, wie du mit `subprocess` parallele Bash-Aufrufe mit `concurrent.futures` kombinierst — das ist super für dein Programm! Möchtest du das?