Super! Hier kommt dein ausführlicher Lernzettel zu **`subprocess`** in Python – verständlich, praxisnah, mit Codebeispielen und Erklärungen:

Lernzettel: Python-Modul `subprocess` (für Bash-Skripte und Shell-Kommandos)

---

1. Was ist `subprocess`?

- Ein Standardmodul, mit dem du externe Programme oder Bash-Skripte aus Python heraus starten kannst.
- Ersetzt veraltete Funktionen wie `os.system()`.
- Ermöglicht das Ausführen von Kommandos, Abfangen von Ausgaben und Fehlern, sowie Steuerung über Ein- und Ausgabe.

---

2. Warum `subprocess` nutzen?

- Du kannst damit Bash-Skripte bequem und sicher ausführen.
- Erlaubt dir, Programm-Ausgaben und Fehler abzufangen und auszuwerten.
- Kontrolliertes Warten auf Fertigstellung oder asynchrones Arbeiten ist möglich.
- Sicherheit: Du kannst Befehle ohne Shell ausführen (besser vor Injection geschützt).

---

3. Wichtige Funktionen

Funktion	Zweck
`subprocess.run()`	Führt einen Befehl aus und wartet auf Ende
`subprocess.Popen()`	Startet Prozess und gibt mehr Kontrolle über Kommunikation
`subprocess.call()`	Ähnlich zu run, aber gibt nur Rückgabecode zurück

---

4. `subprocess.run()` – das Arbeitstier

```
import subprocess

# Beispiel: ls -l im Verzeichnis /home/user
result = subprocess.run(
    ["ls", "-l", "/home/user"],
    capture_output=True,  # stdout und stderr werden abgefangen
    text=True             # Ausgabe als String statt Bytes
)

print("Rückgabecode:", result.returncode)
print("Standardausgabe:\n", result.stdout)
```

```
print("Fehlerausgabe:\n", result.stderr)
```

- `result.returncode` ist 0 bei Erfolg, sonst Fehlercode.
- `capture_output=True` erlaubt dir, die Ausgabe in Python zu nutzen.
- `text=True` konvertiert Byte-Strings zu normalen Strings.

---

5. Sicherheit und `shell=True`

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

- `shell=True` führt das Kommando über die Shell aus, damit Pipes oder komplexe Shell-Syntax funktionieren.
- **Aber Vorsicht:** Wenn Benutzer-Eingaben in den Befehl einfließen, kann das ein Sicherheitsrisiko sein (Shell Injection).
- Wenn möglich, immer als Liste ausführen (`shell=False`, Standard).

---

6. Timeout und Fehlerbehandlung

```
try:
    subprocess.run(["sleep", "5"], timeout=3, check=True)
except subprocess.TimeoutExpired:
    print("Der Prozess hat zu lange gebraucht und wurde abgebrochen!")
except subprocess.CalledProcessError as e:
    print(f"Der Prozess ist mit Fehlercode {e.returncode} gescheitert.")
else:
    print("Prozess erfolgreich abgeschlossen.")
```

- `timeout` bricht den Prozess ab, wenn er zu lange läuft.
- `check=True` lässt Python eine Exception werfen, wenn der Rückgabewert ungleich 0 ist.

---

7. `subprocess.Popen()` – wenn du mehr Kontrolle brauchst

```
import subprocess

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

stdout, stderr = p.communicate("python ist toll\njava auch\npython rocks\n")

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

- `Popen` startet Prozess, erlaubt Lesen/Schreiben von Ein-/Ausgabe.

- `communicate()` schreibt Daten auf stdin und liest stdout/stderr gleichzeitig und wartet, bis Prozess fertig ist.
- Gut, wenn du mit Programmen interagieren möchtest.

---

8. Beispiel: Bash-Skript ausführen und Ergebnis prüfen

```
result = subprocess.run(["/pfad/zu/deinem_script.sh"], capture_output=True, text=True)

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

- Prüfe immer `returncode`, um sicher zu sein, dass das Skript erfolgreich war.
- `stdout` und `stderr` kannst du z.B. für Logging oder Fehlerdiagnose nutzen.

---

9. Übergabe von Argumenten an Bash-Skripte

```
subprocess.run(["/pfad/script.sh", "arg1", "arg2"])
```

- Argumente werden in der Liste einzeln übergeben (nicht alles als String in einer Shell).
- Das schützt vor vielen Fehlern und Injection.

---

10. Praktische Tipps

- Nutze `capture_output=True` für Ausgabe, wenn du die Ausgabe im Python-Programm brauchst.
- Sonst kannst du `stdout=None` lassen, um Ausgabe direkt im Terminal anzuzeigen.
- Bei Bedarf kannst du mit `env` eine eigene Umgebung übergeben (z.B. Variablen setzen).
- Verwende `shlex.split()` wenn du einen String in eine Argumentliste zerlegen willst (z.B. aus Benutzereingabe).

---

11. Typische Probleme und Lösungen

Problem	Lösung
Skript wird nicht gefunden	Pfad überprüfen, `./script.sh` oder absoluter Pfad
Kommando läuft ewig	Timeout nutzen (`timeout=...`)
Fehler werden nicht angezeigt	`capture_output=True` und `result.stderr` prüfen
Unsichere Eingaben führen zu	Niemals `shell=True` mit ungeprüften Strings

Problem	Lösung
Problemen	
Prozesse blockieren wegen voller Pipes	`communicate()` statt manuelles Lesen/Schreiben

---

12. Kombi mit `concurrent.futures` (parallel Bash-Skripte starten)

```
import concurrent.futures
import subprocess

def run_script(script_path):
    result = subprocess.run([script_path], capture_output=True, text=True)
    return (script_path, result.returncode, result.stdout, result.stderr)

scripts = ["/pfad/script1.sh", "/pfad/script2.sh", "/pfad/script3.sh"]

with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    futures = [executor.submit(run_script, s) for s in scripts]
    for future in concurrent.futures.as_completed(futures):
        script, code, out, err = future.result()
        print(f"Script: {script} | Code: {code}")
        if code == 0:
            print("Output:", out)
        else:
            print("Error:", err)
```

- So kannst du mehrere Bash-Skripte parallel starten, ohne den Mainthread zu blockieren.

---

13. Zusammenfassung

- `subprocess` ist das Python-Modul für alle Shell- und Skript-Aufrufe.
- `run()` ist einfach und sicher für synchronen Aufruf.
- `Popen()` für mehr Kontrolle bei Kommunikation mit Prozessen.
- Vermeide `shell=True` wenn möglich.
- Fang Fehler und Timeouts ab, checke Rückgabewerte.
- Du kannst `subprocess` prima mit `concurrent.futures` kombinieren, um mehrere Tasks parallel laufen zu lassen.

Wenn du möchtest, kann ich dir auch Beispiele mit Fehlerbehandlung, Loggings und komplexeren Kommunikationsmustern zeigen!
Oder ein kleines Tutorial, wie man eigene Bash-Skripte robust aus Python startet?