# Subprozesse in Python

Dominik Pataky

4. Dezember 2017

Python-Kurs 2017

# Gliederung

# Grundlagen

# Das Modul subprocess

Das Modul subprocess erlaubt die Ausführung externer Befehle und Skripte von einem Python Skript aus. Man kann sich die Funktionsweise ähnlich der eines Terminals vorstellen.

# Eigenschaften

- Subprozesse laufen asynchron
- Sie laufen direkt auf dem System, nicht in einer Shell (wenn nicht anders festgelegt)
- Verfügbare Programme hängen vom System ab, auf dem sie ausgeführt werden
- Der Aufruf ist allgemeingültig, die Bibliothek verwandelt den Aufruf unter Windows in einen kompatiblen `CreateProcess()` String

# Konstanten

# File Descriptoren

| | |
|---|---|
| DEVNULL | Der systeminterne ’Mülleimer’ |
| PIPE | Die Verbindung zwischen zwei Prozessen |
| STDOUT | Die Standardausgabe oder der laufende Prozess |

# Exceptions

**`SubprocessError`** Der Standardfehler dieses Moduls

**`TimeoutError`** Ein Timeout ist aufgetreten

**`CalledProcessError`** Der Subprozess endete auf eine unerwartete Art

# Popen Klasse

# Die Popen Klasse

Die **Popen** Klasse bildet die Basis des Moduls **subprocess**.
Die Funktionssignatur sieht wie folgt aus:

```
1 import subprocess
2
3 Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None
      ,
4       stderr=None, preexec_fn=None, close_fds=True, shell=False,
5       cwd=None, env=None, universal_newlines=False, startupinfo=
      None,
6       creationflags=0, restore_signals=True, start_new_session=
      False,
7       pass_fds=())
```

# Einige wichtige Argumente

**`args`** Die aufzurufenden Argumente. Sollten vom Typ tuple oder list sein. Im Prinzip splittet man den Konsolenbefehl an den Leerzeichen:
`ls -A *.md` entspricht `['ls', '-A', '*.md']`

**`shell`** Führt den Befehl in einer Shell aus. Sollte `False` sein (Standard), sonst ist der Aufruf unsicher.

**`stdout`** zusammen mit `stdin` und `stderr` die Input- und Output-Verbindungen des Subprozesses
(hier sind `DEVNULL`, `PIPE` und `STDOUT` nützlich)

**`env`** Umgebungsvariablen des Kindprozesses. Standard ist ein Subset von `os.environ` (dem Python Prozess Environment)

**`cwd`** Das Arbeitsverzeichnis des Subprozesses

# Popen Objekte

# Popen Objekte

Wenn Popen instanziiert wird, wird der darin enthaltene Prozess gestartet und das zurückgegebene **Popen** Objekt enthält Informationen über den laufenden Prozess.

# Informationen sammeln

- `process.args`
  Gibt die Argumente zurück, mit denen der Prozess aufgerufen wurde.
- `obj.stdout`, `obj.stdin`, `obj.stderr`
  Input- und Output-Verbindungen, die beim Start gesetzt wurden
- `process.pid`
  Vom System zugewiesene *Prozess ID*.
- `process.poll()`
  Prüft, ob der Prozess beendet wurde. Gibt den *Rückgabewert* des Prozesses zurück oder `None`, wenn der Prozess noch läuft.
- `process.returncode`
  Der Rückgabewert von `process.poll()` (der Rückgabewert des Prozesses)

# Interaktion mit dem Prozess

- `process.wait(timeout=None)`
  Wartet *timeout* Sekunden auf die Terminierung des Prozesses (wartet unendlich lang, wenn timeout *None* ist).
  Wirft nach Ablauf von *timeout* eine `TimeoutExpired` Exception.
- `process.send_signal(signal)`
  Sendet das Signal *signal* an den Prozess (z.B. `SIGTERM`).
- `process.communicate(input=None, timeout=None)`
  Schreibt die Daten aus *input* in den Standardinput (`stdin`) des Prozesses (wenn stdin PIPE ist), wartet auf die Terminierung des Prozesses und liest die Daten, die der Prozess in `stdout` geschrieben hat (wenn stdout PIPE ist). *timeout* funktioniert wie oben.

# Interaktion mit dem Prozess

- `process.terminate()`
  Sendet ein Terminationssignal an den Prozess (`SIGTERM`).
- `process.kill()`
  Erzwingt die Beendigung des Prozesses (`SIGKILL`).

# Popen im Kontextmanager

Popen kann mit dem Kontextmanager verwendet werden (siehe *File Handling*).
Der Code dafür würde wie folgt aussehen:

```
1 with subprocess.Popen(['ls']) as process:
2     pass
```

# Nützliche Funktionen

# Nützliche Funktionen

`subprocess` enthält einige Kurzfassungen für häufig genutzte Arbeitsabläufe. Intern rufen diese allerdings auch nur `Popen` auf.

# run

Die wohl nützlichste Funktion ist `subprocess.run()`:

```
run(args, *, stdin=None, input=None,
    stdout=None, stderr=None, shell=False, timeout=None, check=
    False)
```

- eingeführt in Python 3.5
- ruft durch `args` definierten Prozess auf
- wartet auf Beendigung des Prozesses (wenn *timeout* `None` ist)
- gibt ein `CompletedProcess` Objekt zurück
- **Beachte:** Einsatz eines unbenannten Aggregators

# CompletedProcess Objekt

Ist der Rückgabewert von `run()`, der zurückgegeben wird, wenn der Prozess beendet wurde. Enthält folgende Eigenschaften:

| | |
|---|---|
| **`args`** | Argumente, mit denen der Prozess aufgerufen wurde. |
| **`returncode`** | Rückgabewert des Prozesses |
| **`stdout`** | Ausgabe des Prozesses |
| **`stderr`** | stderr Output des Prozesses |

Außerdem gibt es die Funktion `check_returncode()`, die einen `CalledProcessError` beim Aufruf wirft, wenn der Rückgabewert nicht 0 ist.

# call

```
call(args, *, stdin=None, stdout=None,
     stderr=None, shell=False, timeout=None)
```

- Ruft einen Prozess auf, wartet auf Terminierung
- gibt Returncode des Prozesses zurück

Diese Funktion macht das selbe, wie

`run(...).returncode`,

nur dass *check* und *input* nicht unterstützt werden.

# check_call

Argumente entsprechen den Argumenten von `call()`.

- Ruft Prozess auf, wartet auf Terminierung
- gibt nichts zurück, wenn Ausführung erfolgreich (Returncode == 0)
- wenn Aufruf nicht erfolgreich, wird `CalledProcessError` geworfen

Diese Funktion macht das selbe, wie

```
run(..., check=True)
```

nur dass *input* nicht unterstützt wird.

# check_output

```
check_output(args, *, stdin=None, stdout=None,
             stderr=None, shell=False, timeout=None)
```

- führt Kommando aus und gibt den Prozessoutput zurück
- wirft ebenfalls `CalledProcessError`, wenn der Returncode nicht 0 ist

Diese Funktion macht das selbe, wie

```
run(..., check=True, stdout=PIPE).stdout
```