Übersicht über CLI-Frameworks

Wie kann ich sys.argv komfortabel auswerten?

Detlef Lannert, PyDdf 17. 04. 2024

Das Problem

argv konventionsgerecht auswerten

- Es gibt mehrere Konventionen!
- Auf Unix-Systemen ist die Auswertung der Argumente Sache des Programms.
- Klassische Unix-Konventionen erschweren das: `ls -l -d`, `ls -ld`; `ls -w 50`, `ls -w50`; `ls -ldw50`
- GNU-Optionen kommen hinzu: `ls --all`, `ls --width=50`, `ls --width 50`
- Es kann ein oder mehrere unbenannte Argumente geben; `--` als optionaler Trenner
- Zusätzlich gibt es hierzu inkompatible „Standards“ (ip, MS-DOS, `+name`, …) → hier nicht weiter betrachtet

Die Lösungen

Verschiedene Wege zur Lösung

- Handarbeit: sys.argv
- Standardlösung: argparse
- Metadaten im Decorator: Click
- Metadaten als Annotations: cyclopts
- Metadaten als dataclass: cappa
- Metadaten wahlweise in Annotations oder dataclass: tyro, jsonargparse
- außer Konkurrenz (TUI statt CLI): textual

(ohne Anspruch auf Vollständigkeit!)

sys.argv

Beispiel

```python
#! /usr/bin/env python3

import sys

def main():
    if len(sys.argv) > 1 and sys.argv[1] == "-h":
        print(f"Usage:\n{sys.argv[0]} [-v] ...")
        sys.exit(0)

    if len(sys.argv) > 1 and sys.argv[1] == "-v":
        print("running")
        del sys.argv[1]  # there might be more args

if __name__ == "__main__":
    main()
```

Nachteile

- umständlich, selbst in ganz einfachen Fällen
- (De-facto-) Standards werden meist nicht eingehalten
- Hilfetext muss manuell angepasst werden

Fazit: Gar nicht erst damit anfangen.

argparse

Beispiel

```python
#! /usr/bin/env python3

import argparse

parser = argparse.ArgumentParser(prog="ex01", description="Beispiel für argparse")
parser.add_argument("-v", "--verbose", action="store_true", help="Geschwätzigkeit")
parser.add_argument("-c", "--count", type=int, default=1, help="Anzahl Wiederholungen")
parser.add_argument("names", nargs="+", metavar="NAME", help="Namen")

def main():
    args = parser.parse_args()
    if args.verbose:
        print("running")
    print(args.count * f"Hallo {', '.join(name for name in args.names)}! ")

if __name__ == "__main__":
    main()
```

Vorteile

- Bestandteil der Standardbibliothek von Python ⇒ immer da!
- gut dokumentiert

- sehr flexibel
- stellt `--help` bereit
- unterstützt die üblichen Konventionen, einschließlich Subkommandos
- verwendbar z. B. für Django-Management-Kommandos

Nachteile

- aufwendig zu schreiben (und sich zu merken)
- nur bei richtiger Anwendung konventionsgerecht
- keine Typechecks, keine gute IDE-Unterstützung
- imperativ, nicht deklarativ
- Definitionen haben keinen Bezug zu Funktionen und ihren Argumenten
- in bestimmten Fällen unpräzise (siehe Click-Doku)

Click

Beispiel

```python
#! /usr/bin/env python3

import click

@click.command()
@click.option("--verbose/--no-verbose", "-v", default=False, help="mehr ausgeben")
@click.option("--count", "-c", default=1, help="Anzahl der Grüße")
@click.argument("names", nargs=-1)
def greet(verbose, count, names):
    """Ein grußfreudiges Beispielprogramm."""

    if verbose:
        click.secho("running", fg="green")
    click.echo(count * f"Hallo {', '.join(name for name in names)}! ")

if __name__ == '__main__':
    greet(auto_envvar_prefix="PYDDF")
```

(siehe auch hhufw)

Vorteile

- Click arbeitet nicht nur als Parser, sondern auch als Dispatcher
- beliebige Verschachtelung von Unterkommandos …
- … mit der Möglichkeit, Routinen dynamisch nachzuladen
- mehrere eigene nützliche Datentypen
- großer Funktionsumfang
- einfache Shell-Autocompletion
- eigener REPL ist möglich
- weit verbreitet

- mittlerweile gibt es viele Erweiterungen

Nachteile

- die dekorierten Funktionen sind nicht mehr „normal“ aufrufbar
- Decorators müssen nicht mit den Argumenten der Funktion übereinstimmen → dann gibt’s Probleme
- Defaults gehen nicht automatisch in die Hilfetexte ein
- Subkommando und seine Gruppe müssen über den etwas unhandlichen Kontext kommunizieren
- keine Typenkontrolle
- alle Click-Typen benutzen dasselbe unübersichtliche Interface

cyclopts

Beispiel

```python
#! /usr/bin/env python3
import cyclopts

app = cyclopts.App()

@app.default
def greet(names: list[str], *, count: int = 1, verbose: bool = False) -> None:
    """Begrüße verschiedene Personen, auch mehrfach.

    Args:
        names: Liste der Namen
        count: Anzahl der Grüße
        verbose: Umfang der Ausgabe
    """
    if verbose:
        print("running")
    print(count * f"Hallo {', '.join(name for name in names)}! ")

if __name__ == "__main__":
    app()
```

Vorteile

- Type annotations können zugleich fürs CLI genutzt werden – bei Adhoc-Skripts (mit Typen) ist praktisch kein Mehraufwand erforderlich
- Das Paket Typer ist für diese Vorgehensweise das bekannteste; die cyclopts-Eigenwerbung sagt aber: „Cyclopts is what you thought Typer was.“
- Hilfetexte können ohne `Annotated` per Docstring angegeben werden
- gute Doku
- setzt (im Unterschied zu Typer) nicht auf Click auf

Nachteile

- Die Annotations werden etwas aufwendiger, wenn man Sonderwünsche hat:

```
count: Annotated[int, cyclopts.Parameter(
     name=("--count", "-c"),
     help="Anzahl der Grüße")] = 1,
```

Dafür können allerdings die entsprechenden Docstrings entfallen, und zusätzlich sind bspw. Konverter und Validierer möglich.

tyro & Co.

Abbildung von CLI-Argumenten auf Datenstrukturen

- Typisierte Datenstrukturen (dataclass, attrs, pydantic) sind äquivalent zu Funktionssignaturen (für das `__init__()`)
- sie können also ebenso auf ein CLI abgebildet werden
- Beispiele: *cappa*, *tyro*, *jsonargparse*
- *Tyro* und *jsonargparse* können das CLI sowohl aus Signaturen als auch aus Datenstrukturen generieren
- Integration mit Konfig-Dateien ist teilweise möglich
- Welches dieser Tools ist das beste? Es kommt drauf an ;-)

Beispiel mit Funktionsargumenten

```python
#! /usr/bin/env python3

import tyro

def greet(names: list[str], /, verbose: bool = False, count: int = 1) -> None:
    """Begrüßt die genannten Leute, auch mehrfach.

    Args:
        names: Liste der Namen
        verbose: Umfang der Ausgabe
        count: Anzahl der Grüße
    """
    if verbose:
        print("running")
    print(count * f"Hallo {', '.join(name for name in names)}! ")

tyro.cli(greet)
```

Beispiel mit Dataclass

```python
from attrs import define
import tyro

@define
class Args:
    "Begrüßung mehrerer Personen"

    # Liste der Namen
    names: tyro.conf.Positional[list[str]]

    # Umfang der Ausgabe
    verbose: bool = False

    # Anzahl der Grüße
    count: int = 1

def greet(args: Args) -> None:
    """Begrüßt die genannten Leute, auch mehrfach."""
    if args.verbose:
        print("running")
    print(args.count * f"Hallo {', '.join(name for name in args.names)}! ")

if __name__ == "__main__":
    args = tyro.cli(Args)
    greet(args)
```

Zusammenfassung

Meine (ganz subjektiven) Favoriten

- Click: vielseitig, robust, allerdings etwas umständlich
- cyclopts: wenn man mit Type annotations arbeiten mag
- tyro, cappa oder jsonargparse wären für komplexe Datenstrukturen nützlich (Vorzüge / Schwächen sind je nach Einsatzfall zu bewerten)

Anhang

Links

argparse: https://docs.python.org/3/library/argparse.html

Click: https://click.palletsprojects.com/

Typer: https://typer.tiangolo.com/

cyclopts: https://cyclopts.readthedocs.io/

tyro: https://brentyi.github.io/tyro/

cappa: https://cappa.readthedocs.io/

jsonargparse: https://jsonargparse.readthedocs.io/