Targ

Daniel Townsend

Mar 21, 2024

CONTENTS:

Build a Python CLI for your app, just using type hints and docstrings.

Just register your type annotated functions, and that's it - there's no special syntax to learn, and it's super easy.

```python
# main.py
from targ import CLI


def add(a: int, b: int):
    """
    Add the two numbers.

    :param a:
        The first number.
    :param b:
        The second number.
    """
    print(a + b)


if __name__ == "__main__":
    cli = CLI()
    cli.register(add)
    cli.run()
```

And from the command line:

```
>>> python main.py add 1 1
2
```

To get documentation:

```
>>> python main.py add --help

add
===
Add the two numbers.

Usage
-----
add a b

Args
----
a
The first number.

b
The second number.
```

CHAPTER

ONE

INSTALLATION

Targ is supported on Python 3.7 and above.

```
pip install targ
```

CHAPTER

TWO

SUPPORTED TYPES

Targ currently supports basic Python types:

- str
- int
- bool
- float
- Decimal
- Optional

You should specify a type annotation for each function argument, so Targ can convert the input it receives from the command line into the correct type. Otherwise, the type is assumed to be a string.

2.1 str

```
def say_hello(name: str):
    print(f'hello {name}')
```

Example usage:

```
>>> python main.py say_hello bob
'bob'

>>> python main.py say_hello --name=bob
'bob'
```

When your string contains spaces, use quotation marks:

```
>>> python main.py say_hello --name="bob jones"
'bob jones'
```

2.2 int

```python
def add(a: int, b: int):
    print(a + b)
```

Example usage:

```
>>> python main.py add 1 2
3

>>> python main.py add --a=1 --b=2
3
```

2.3 bool

```python
def print_pi(precise: bool = False):
    if precise:
        print("3.14159265")
    else:
        print("3.14")
```

Example usage:

```
>>> python main.py print_pi
3.14

>>> python main.py print_pi true
3.14159265

>>> python main.py print_pi --precise
3.14159265

>>> python main.py print_pi --precise=true
3.14159265
```

You can use `t` as an alias for `true`, and likewise `f` as an alias for `false`.

```
>>> python main.py print_pi --precise=t
3.14159265
```

2.4 float

```
def compound_interest(interest_rate: float, years: int):
    print(((interest_rate + 1) ** years) - 1)
```

Example usage:

```
>>> python main.py compound_interest 0.05 5
0.27628156250000035
```

2.5 Decimal

```
from decimal import Decimal

def compound_interest(interest_rate: Decimal, years: int):
    print(((interest_rate + 1) ** years) - 1)
```

Example usage:

```
>>> python main.py compound_interest 0.05 5
0.2762815625
```

2.6 Optional

```
from typing import Optional

def print_address(
    number: int, street: str, postcode: str, city: Optional[str] = None
):
  address = f"{number} {street}"
  if city is not None:
      address += f", {city}"
  address += f", {postcode}"

  print(address)
```

Example usage:

```
>>> python print_address --number=1 --street="Royal Avenue" --postcode="XYZ 123" --
↪city=London
1 Royal Avenue, London, XYZ 123

>>> python print_address --number=1 --street="Royal Avenue" --postcode="XYZ 123"
1 Royal Avenue, XYZ 123
```

CHAPTER

THREE

DOCSTRINGS

Docstrings are used to document your CLI.

ReST-style and Google-style docstrings are supported - use whichever you prefer the look of, as functionally they are basically the same.

3.1 ReST

```
def say_hello(name: str):
    """
    Say hello to someone.

    :param name: The person to say hello to.

    """
    print(f'hello {name}')
```

3.2 Google

```
def say_hello(name: str):
    """
    Say hello to someone.

    Args:
        name:
            The person to say hello to.

    """
    print(f'hello {name}')
```

3.3 Output

Targ automatically documents your API using the docstring:

```
python main.py say_hello --help
```

```
say_hello
Say hello to someone.

Usage:
say_hello name

Args:
name       The person to say hello to.
```

CHAPTER

FOUR

CLI

4.1 Registering functions

Once you've defined your functions, you need to register them with a `CLI` instance.

You can register as many functions as you like with the `CLI` instance.

```
from targ import CLI


def add(a: int, b: int):
    print(a + b)


def subtract(a: int, b: int):
    print(a - b)


if __name__ == "__main__":
    cli = CLI()
    cli.register(add)
    cli.register(subtract)
    cli.run()
```

4.2 Coroutines

You can also register coroutines, as well as normal functions:

```
import asyncio

from targ import CLI


async def timer(seconds: int):
    print(f"Sleeping for {seconds}")
    await asyncio.sleep(seconds)
    print("Finished")
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    cli = CLI()
    cli.register(timer)
    cli.run()
```

4.3 Aliases

You can specify aliases for each command, which can make your CLI more convenient to use. For example, an alias could be an abbreviation, or a common mispelling.

```
from targ import CLI


def add(a: int, b: int):
    print(a + b)


if __name__ == "__main__":
    cli = CLI()
    cli.register(add, aliases=['+', 'sum'])
    cli.run()
```

All of the following will now work:

```
python main.py add 1 1
python main.py + 1 1
python main.py sum 1 1
```

4.4 Groups

You can add your functions / coroutines to a group:

```
cli.register(say_hello, 'greetings')
cli.register(add, 'maths')
```

And then call them as follows:

```
python main.py greetings say_hello 'bob'
python main.py maths add 1 2
```

4.5 Overriding the command name

By default the command name is the name of the function being registered. However, you can choose to override it:

```
cli.register(add, command_name='sum')
```

4.6 Traceback

By default, `targ` will print out an abbreviated error message if it encounters a problem. To see the full Python traceback, pass in the `--trace` argument.

```
python main.py maths add 1 'abc' --trace
```

4.7 Solo mode

Sometimes you'll just want to register a single command with your CLI, in which case, specifying the command name is redundant.

```
from targ import CLI


def add(a: int, b: int):
    print(a + b)


if __name__ == "__main__":
    cli = CLI()
    cli.register(add)
    cli.run(solo=True)
```

You can then omit the command name:

```
python main.py 1 1
```

4.8 Source

class targ.**CLI**(*description: str = 'Targ CLI'*)

The root class for building the CLI.

Example usage:

```
cli = CLI()
cli.register(some_function)
```

Parameters
description – Customise the title of your CLI tool.

command_exists(*group_name: str*, *command_name: str*) → bool

This isn't used by Targ itself, but is useful for third party code which wants to inspect the CLI, to find if a command with the given name exists.

register(*command: Callable*, *group_name: str | None = None*, *command_name: str | None = None*, *aliases: List[str] = []*)

Register a function or coroutine as a CLI command.

Parameters

- **command** – The function or coroutine to register as a CLI command.
- **group_name** – If specified, the CLI command will belong to a group. When calling a command which belongs to a group, it must be prefixed with the `group_name`. For example `python my_file.py group_name command_name`.
- **command_name** – By default, the name of the CLI command will be the same as the function or coroutine which is being called. You can override this here.
- **aliases** – The command can also be accessed using these aliases.

run(*solo: bool = False*)

Run the CLI.

Parameters
solo – By default, a command name must be given when running the CLI, for example `python my_file.py command_name`. In some situations, you may only have a single command registered with the CLI, so passing in the command name is redundant. If `solo=True`, you can omit the command name i.e. `python my_file.py`, and Targ will automatically call the single registered command.

CHAPTER

FIVE

RELATED PROJECTS

Targ is a general purpose library, but was created for the Piccolo ORM as an easy way of registering command line tools within an app.

CHAPTER

SIX

ALTERNATIVES

6.1 argparse

This is a stdlib module for building a CLI. It's powerful, but quite verbose. It's a good option if you don't want any external dependencies in your project.

6.2 click

Click is a popular third party package for building a CLI. It makes extensive use of decorators.

CHAPTER

SEVEN

CHANGES

7.1 0.4.0

General maintenance - dropping Python 3.7 support, adding Python 3.12, updating dependencies, and fixing linter errors.

7.2 0.3.8

Slackened dependencies to avoid clashes with other libraries, like `fastkafka`.

7.3 0.3.7

If an exception is raised when running a command, mention the `--trace` option, which will show a full stack trace.

Added docstring to `Command`.

7.4 0.3.6

Added Python 3.10 support.

7.5 0.3.5

Fixing a bug with the `--trace` option, which outputs a traceback if an exception occurs.

7.6 0.3.4

Commands will now work if the type annotation of an argument is missing - in this case the type of the argument is assumed to be a string.

7.7 0.3.3

Small help formatting change when a command has no args.

7.8 0.3.2

Add back *CLI.command_exists* - required by Piccolo.

7.9 0.3.1

Show aliases in command help text.

7.10 0.3.0

Added aliases for commands.

7.11 0.2.0

Added support for `Optional` and `Decimal`.

7.12 0.1.9

Added solo mode.

7.13 0.1.8

Fixing py.typed.

7.14 0.1.7

Loosening colorama dependency version.

7.15 0.1.6

Improving appearance when a command has no args.

7.16 0.1.5

Added –trace argument for getting Python traceback on error.

7.17 0.1.4

Can override the command name.

7.18 0.1.3

Removed cached_property to support Python 3.7.

7.19 0.1.2

Added support for groups and coroutines.

7.20 0.1.1

Add support for flag arguments, and improved help output.

7.21 0.1.0

Initial release.

INDEX

C

R