swansonk14/typed-argument-parser

github.com/swansonk14/typed-argument-parser

swansonk14

Typed Argument Parser (Tap)

Tap is a typed modernization of Python's argparse library.

Tap provides the following benefits:

- Static type checking
- Code completion
- Source code navigation (e.g. go to definition and go to implementation)


```python
from tap import Tap


class MyArgumentParser(Tap):
    package: str
    is_cool: bool = False
    stars: int = 0


args = MyArgumentParser().parse_args()

print(f'{args.package} has {args.star} stars!')
```

variable "package"
Inferred type: str

stars MyArgumentParser

^↓ and ^↑ will move caret down and up in the editor Next Tip

See this poster, which we presented at PyCon 2020, for a presentation of some of the relevant concepts we used to guide the development of Tap.

As of version 1.8.0, Tap includes `tapify`, which runs functions or initializes classes with arguments parsed from the command line. We show an example below.

```
# square.py
from tap import tapify

def square(num: float) -> float:
    return num ** 2

if __name__ == '__main__':
    print(f'The square of your number is {tapify(square)}.')
```

Running `python square.py --num 2` will print `The square of your number is 4.0.`. Please see tapify for more details.

Installation

Tap requires Python 3.7+

To install Tap from PyPI run:

```
pip install typed-argument-parser
```

To install Tap from source, run the following commands:

```
git clone https://github.com/swansonk14/typed-argument-parser.git
cd typed-argument-parser
pip install -e .
```

Table of Contents

Tap is Python-native

To see this, let's look at an example:

```
"""main.py"""

from tap import Tap

class SimpleArgumentParser(Tap):
    name: str  # Your name
    language: str = 'Python'  # Programming language
    package: str = 'Tap'  # Package name
    stars: int  # Number of stars
    max_stars: int = 5  # Maximum stars

args = SimpleArgumentParser().parse_args()

print(f'My name is {args.name} and I give the {args.language} package
'
      f'{args.package} {args.stars}/{args.max_stars} stars!')
```

You use Tap the same way you use standard argparse.

```
>>> python main.py --name Jesse --stars 5
My name is Jesse and I give the Python package Tap 5/5 stars!
```

The equivalent argparse code is:

```
"""main.py"""

from argparse import ArgumentParser

parser = ArgumentParser()
parser.add_argument('--name', type=str, required=True,
                    help='Your name')
parser.add_argument('--language', type=str, default='Python',
                    help='Programming language')
parser.add_argument('--package', type=str, default='Tap',
                    help='Package name')
parser.add_argument('--stars', type=int, required=True,
                    help='Number of stars')
parser.add_argument('--max_stars', type=int, default=5,
                    help='Maximum stars')
args = parser.parse_args()

print(f'My name is {args.name} and I give the {args.language} package
'
      f'{args.package} {args.stars}/{args.max_stars} stars!')
```

The advantages of being Python-native include being able to:

- Overwrite convenient built-in methods (e.g. `process_args` ensures consistency among arguments)
- Add custom methods
- Inherit from your own template classes

Tap features

Now we are going to highlight some of our favorite features and give examples of how they work in practice.

Arguments

Arguments are specified as class variables defined in a subclass of `Tap`. Variables defined as `name: type` are required arguments while variables defined as `name: type = value` are not required and default to the provided value.

```
class MyTap(Tap):
    required_arg: str
    default_arg: str = 'default value'
```

Tap help

Single line and/or multiline comments which appear after the argument are automatically parsed into the help string provided when running `python main.py -h`. The type and default values of arguments are also provided in the help string.

```
"""main.py"""

from tap import Tap

class MyTap(Tap):
    x: float  # What am I?
    pi: float = 3.14  # I'm pi!
    """Pi is my favorite number!"""

args = MyTap().parse_args()
```

Running `python main.py -h` results in the following:

```
>>> python main.py -h
usage: demo.py --x X [--pi PI] [-h]

optional arguments:
  --x X       (float, required) What am I?
  --pi PI     (float, default=3.14) I'm pi! Pi is my favorite number.
  -h, --help  show this help message and exit
```

Configuring arguments

To specify behavior beyond what can be specified using arguments as class variables, override the `configure` method. `configure` provides access to advanced argument parsing features such as `add_argument` and `add_subparser`. Since Tap is a wrapper around argparse, Tap provides all of the same functionality. We detail these two functions below.

Adding special argument behavior

In the `configure` method, call `self.add_argument` just as you would use argparse's `add_argument`. For example,

```
from tap import Tap

class MyTap(Tap):
    positional_argument: str
    list_of_three_things: List[str]
    argument_with_really_long_name: int

    def configure(self):
        self.add_argument('positional_argument')
        self.add_argument('--list_of_three_things', nargs=3)
        self.add_argument('-arg', '--argument_with_really_long_name')
```

Adding subparsers

To add a subparser, override the `configure` method and call `self.add_subparser`. Optionally, to specify keyword arguments (e.g., `help`) to the subparser collection, call `self.add_subparsers`. For example,

```
class SubparserA(Tap):
    bar: int  # bar help

class SubparserB(Tap):
    baz: Literal['X', 'Y', 'Z']  # baz help

class Args(Tap):
    foo: bool = False  # foo help

    def configure(self):
        self.add_subparsers(help='sub-command help')
        self.add_subparser('a', SubparserA, help='a help')
        self.add_subparser('b', SubparserB, help='b help')
```

Types

Tap automatically handles all the following types:

```
str, int, float, bool
Optional, Optional[str], Optional[int], Optional[float],
Optional[bool]
List, List[str], List[int], List[float], List[bool]
Set, Set[str], Set[int], Set[float], Set[bool]
Tuple, Tuple[Type1, Type2, etc.], Tuple[Type, ...]
Literal
```

If you're using Python 3.9+, then you can replace `List` with `list`, `Set` with `set`, and `Tuple` with `tuple`.

Tap also supports `Union`, but this requires additional specification (see Union section below).

Additionally, any type that can be instantiated with a string argument can be used. For example, in

```
from pathlib import Path
from tap import Tap

class Args(Tap):
   path: Path

args = Args().parse_args()
```

`args.path` is a `Path` instance containing the string passed in through the command line.

`str`, `int`, and `float`

Each is automatically parsed to their respective types, just like argparse.

`bool`

If an argument `arg` is specified as `arg: bool` or `arg: bool = False`, then adding the `--arg` flag to the command line will set `arg` to `True`. If `arg` is specified as `arg: bool = True`, then adding `--arg` sets `arg` to `False`.

Note that if the `Tap` instance is created with `explicit_bool=True`, then booleans can be specified on the command line as `--arg True` or `--arg False` rather than `--arg`. Additionally, booleans can be specified by prefixes of `True` and `False` with any capitalization as well as `1` or `0` (e.g. for True, `--arg tRu`, `--arg T`, `--arg 1` all suffice).

`Optional`

These arguments are parsed in exactly the same way as `str`, `int`, `float`, and `bool`. Note bools can be specified using the same rules as above and that `Optional` is equivalent to `Optional[str]`.

`List`

If an argument `arg` is a `List`, simply specify the values separated by spaces just as you would with regular argparse. For example, `--arg 1 2 3` parses to `arg = [1, 2, 3]`.

`Set`

Identical to `List` but parsed into a set rather than a list.

`Tuple`

Tuples can be used to specify a fixed number of arguments with specified types using the syntax `Tuple[Type1, Type2, etc.]` (e.g. `Tuple[str, int, bool, str]`). Tuples with a variable number of arguments are specified by `Tuple[Type, ...]` (e.g. `Tuple[int, ...]`). Note `Tuple` defaults to `Tuple[str, ...]`.

`Literal`

Literal is analagous to argparse's choices, which specifies the values that an argument can take. For example, if arg can only be one of 'H', 1, False, or 1.0078 then you would specify that `arg: Literal['H', 1, False, 1.0078]`. For instance, `--arg False` assigns arg to False and `--arg True` throws error. The `Literal` type was introduced in Python 3.8 (PEP 586) and can be imported with `from typing_extensions import Literal`.

`Union`

Union types must include the `type` keyword argument in `add_argument` in order to specify which type to use, as in the example below.

```
def to_number(string: str) -> Union[float, int]:
    return float(string) if '.' in string else int(string)

class MyTap(Tap):
    number: Union[float, int]

    def configure(self):
        self.add_argument('--number', type=to_number)
```

In Python 3.10+, `Union[Type1, Type2, etc.]` can be replaced with `Type1 | Type2 | etc.`, but the `type` keyword argument must still be provided in `add_argument`.

Complex Types

Tap can also support more complex types than the ones specified above. If the desired type is constructed with a single string as input, then the type can be specified directly without additional modifications. For example,

```
class Person:
    def __init__(self, name: str) -> None:
        self.name = name

class Args(Tap):
    person: Person

args = Args().parse_args('--person Tapper'.split())
print(args.person.name)  # Tapper
```

If the desired type has a more complex constructor, then the `type` keyword argument must be provided in `add_argument`. For example,

```
class AgedPerson:
    def __init__(self, name: str, age: int) -> None:
        self.name = name
        self.age = age

def to_aged_person(string: str) -> AgedPerson:
    name, age = string.split(',')
    return AgedPerson(name=name, age=int(age))

class Args(Tap):
    aged_person: AgedPerson

    def configure(self) -> None:
        self.add_argument('--aged_person', type=to_aged_person)

args = Args().parse_args('--aged_person Tapper,27'.split())
print(f'{args.aged_person.name} is {args.aged_person.age}')  # Tapper
is 27
```

Argument processing

With complex argument parsing, arguments often end up having interdependencies. This means that it may be necessary to disallow certain combinations of arguments or to modify some arguments based on other arguments.

To handle such cases, simply override `process_args` and add the required logic. `process_args` is automatically called when `parse_args` is called.

```
class MyTap(Tap):
    package: str
    is_cool: bool
    stars: int

    def process_args(self):
        # Validate arguments
        if self.is_cool and self.stars < 4:
            raise ValueError('Cool packages cannot have fewer than 4
stars')

        # Modify arguments
        if self.package == 'Tap':
            self.is_cool = True
            self.stars = 5
```

Processing known args

Similar to argparse's `parse_known_args`, Tap is capable of parsing only arguments that it is aware of without raising an error due to additional arguments. This can be done by calling `parse_args` with `known_only=True`. The remaining un-parsed arguments are then available by accessing the `extra_args` field of the Tap object.

```
class MyTap(Tap):
    package: str

args = MyTap().parse_args(['--package', 'Tap', '--other_arg',
'value'], known_only=True)
print(args.extra_args)  # ['--other_arg', 'value']
```

Subclassing

It is sometimes useful to define a template Tap and then subclass it for different use cases. Since Tap is a native Python class, inheritance is built-in, making it easy to customize from a template Tap.

In the example below, `StarsTap` and `AwardsTap` inherit the arguments (`package` and `is_cool`) and the methods (`process_args`) from `BaseTap`.

```
class BaseTap(Tap):
    package: str
    is_cool: bool

    def process_args(self):
        if self.package == 'Tap':
            self.is_cool = True


class StarsTap(BaseTap):
    stars: int


class AwardsTap(BaseTap):
    awards: List[str]
```

Printing

Tap uses Python's pretty printer to print out arguments in an easy-to-read format.

```
"""main.py"""

from tap import Tap
from typing import List

class MyTap(Tap):
    package: str
    is_cool: bool = True
    awards: List[str] = ['amazing', 'wow', 'incredible', 'awesome']

args = MyTap().parse_args()
print(args)
```

Running `python main.py --package Tap` results in:

```
>>> python main.py
{'awards': ['amazing', 'wow', 'incredible', 'awesome'],
 'is_cool': True,
 'package': 'Tap'}
```

Reproducibility

Tap makes reproducibility easy, especially when running code in a git repo.

Reproducibility info

Specifically, Tap has a method called `get_reproducibility_info` that returns a dictionary containing all the information necessary to replicate the settings under which the code was run. This dictionary includes:

- Python command
  - The Python command that was used to run the program
  - Ex. `python main.py --package Tap`

- Time
  - The time when the command was run
  - Ex. `Thu Aug 15 00:09:13 2019`
- Git root
  - The root of the git repo containing the code that was run
  - Ex. `/Users/swansonk14/typed-argument-parser`
- Git url
  - The url to the git repo, specifically pointing to the current git hash (i.e. the hash of HEAD in the local repo)
  - Ex. https://github.com/swansonk14/typed-argument-parser/tree/446cf046631d6bdf7cab6daec93bf7a02ac00998
- Uncommitted changes
  - Whether there are any uncommitted changes in the git repo (i.e. whether the code is different from the code at the above git hash)
  - Ex. `True` or `False`

Saving and loading arguments

Save

Tap has a method called `save` which saves all arguments, along with the reproducibility info, to a JSON file.

```
"""main.py"""

from tap import Tap

class MyTap(Tap):
    package: str
    is_cool: bool = True
    stars: int = 5

args = MyTap().parse_args()
args.save('args.json')
```

After running `python main.py --package Tap`, the file `args.json` will contain:

```
{
    "is_cool": true,
    "package": "Tap",
    "reproducibility": {
        "command_line": "python main.py --package Tap",
        "git_has_uncommitted_changes": false,
        "git_root": "/Users/swansonk14/typed-argument-parser",
        "git_url": "https://github.com/swansonk14/typed-argument-
parser/tree/446cf046631d6bdf7cab6daec93bf7a02ac00998",
        "time": "Thu Aug 15 00:18:31 2019"
    },
    "stars": 5
}
```

Note: More complex types will be encoded in JSON as a pickle string.

Load

> ❗ ⚠️
> Never call `args.load('args.json')` on untrusted files. Argument loading uses the `pickle` module to decode complex types automatically. Unpickling of untrusted data is a security risk and can lead to arbitrary code execution. See the warning in the pickle docs.
> ❗ ⚠️

Arguments can be loaded from a JSON file rather than parsed from the command line.

```
"""main.py"""

from tap import Tap

class MyTap(Tap):
    package: str
    is_cool: bool = True
    stars: int = 5

args = MyTap()
args.load('args.json')
```

Note: All required arguments (in this case `package`) must be present in the JSON file if not already set in the Tap object.

Load from dict

Arguments can be loaded from a Python dictionary rather than parsed from the command line.

```
"""main.py"""

from tap import Tap

class MyTap(Tap):
    package: str
    is_cool: bool = True
    stars: int = 5

args = MyTap()
args.from_dict({
    'package': 'Tap',
    'stars': 20
})
```

Note: As with `load`, all required arguments must be present in the dictionary if not already set in the Tap object. All values in the provided dictionary will overwrite values currently in the Tap object.

Loading from configuration files

Configuration files can be loaded along with arguments with the optional flag `config_files: List[str]`. Arguments passed in from the command line overwrite arguments from the configuration files. Arguments in configuration files that appear later in the list overwrite the arguments in previous configuration files.

For example, if you have the config file `my_config.txt`

```
--arg1 1
--arg2 two
```

then you can write

```
from tap import Tap

class Args(Tap):
    arg1: int
    arg2: str

args = Args(config_files=['my_config.txt']).parse_args()
```

Config files are parsed using `shlex.split` from the python standard library, which supports shell-style string quoting, as well as line-end comments starting with `#`.

For example, if you have the config file `my_config_shlex.txt`

```
--arg1 21 # Important arg value

# Multi-word quoted string
--arg2 "two three four"
```

then you can write

```
from tap import Tap

class Args(Tap):
    arg1: int
    arg2: str

args = Args(config_files=['my_config_shlex.txt']).parse_args()
```

to get the resulting `args = {'arg1': 21, 'arg2': 'two three four'}`

The legacy parsing behavior of using standard string split can be re-enabled by passing `legacy_config_parsing=True` to `parse_args`.

tapify

`tapify` makes it possible to run functions or initialize objects via command line arguments. This is inspired by Google's Python Fire, but `tapify` also automatically casts command line arguments to the appropriate types based on the type hints. Under the hood, `tapify` implicitly creates a Tap object and uses it to parse the command line arguments, which it then uses to run the function or initialize the class. We show a few examples below.

Examples

Function

```
# square_function.py
from tap import tapify

def square(num: float) -> float:
    """Square a number.

    :param num: The number to square.
    """
    return num ** 2

if __name__ == '__main__':
    squared = tapify(square)
    print(f'The square of your number is {squared}.')
```

Running `python square_function.py --num 5` prints `The square of your number is 25.0.`.

Class

```
# square_class.py
from tap import tapify

class Squarer:
    def __init__(self, num: float) -> None:
        """Initialize the Squarer with a number to square.

        :param  num: The number to square.
        """
        self.num = num

    def get_square(self) -> float:
        """Get the square of the number."""
        return self.num ** 2

if __name__ == '__main__':
    squarer = tapify(Squarer)
    print(f'The square of your number is {squarer.get_square()}.')
```

Running `python square_class.py --num 2` prints `The square of your number is 4.0.`.

Dataclass

```
# square_dataclass.py
from dataclasses import dataclass

from tap import tapify

@dataclass
class Squarer:
    """Squarer with a number to square.

     :param  num: The number to square.
    """
    num: float

    def get_square(self) -> float:
        """Get the square of the number."""
        return self.num ** 2

if __name__ == '__main__':
    squarer = tapify(Squarer)
    print(f'The square of your number is {squarer.get_square()}.')
```

Running `python square_dataclass.py --num -1` prints `The square of your number is 1.0.`.

tapify help

The help string on the command line is set based on the docstring for the function or class. For example, running `python square_function.py -h` will print:

```
usage: square_function.py [-h] --num NUM

Square a number.

options:
  -h, --help  show this help message and exit
  --num NUM   (float, required) The number to square.
```

Note that for classes, if there is a docstring in the `__init__` method, then `tapify` sets the help string description to that docstring. Otherwise, it uses the docstring from the top of the class.

Command line vs explicit arguments

`tapify` can simultaneously use both arguments passed from the command line and arguments passed in explicitly in the `tapify` call. Arguments provided in the `tapify` call override function defaults, and arguments provided via the command line override both arguments provided in the `tapify` call and function defaults. We show an example below.

```
# add.py
from tap import tapify

def add(num_1: float, num_2: float = 0.0, num_3: float = 0.0) ->
float:
    """Add numbers.

    :param num_1: The first number.
    :param num_2: The second number.
    :param num_3: The third number.
    """
    return num_1 + num_2 + num_3

if __name__ == '__main__':
    added = tapify(add, num_2=2.2, num_3=4.1)
    print(f'The sum of your numbers is {added}.')
```

Running `python add.py --num_1 1.0 --num_2 0.9` prints `The sum of your numbers is 6.0.`. (Note that `add` took `num_1 = 1.0` and `num_2 = 0.9` from the command line and `num_3=4.1` from the `tapify` call due to the order of precedence.)

Known args

Calling `tapify` with `known_only=True` allows `tapify` to ignore additional arguments from the command line that are not needed for the function or class. If `known_only=False` (the default), then `tapify` will raise an error when additional arguments are provided. We show an example below where `known_only=True` might be useful for running multiple `tapify` calls.

```
# person.py
from tap import tapify

def print_name(name: str) -> None:
    """Print a person's name.

    :param name: A person's name.
    """
    print(f'My name is {name}.')

def print_age(age: int) -> None:
    """Print a person's age.

    :param name: A person's age.
    """
    print(f'My age is {age}.')

if __name__ == '__main__':
    tapify(print_name, known_only=True)
    tapify(print_age, known_only=True)
```

Running `python person.py --name Jesse --age 1` prints `My name is Jesse.` followed by `My age is 1.`. Without `known_only=True`, the `tapify` calls would raise an error due to the extra argument.