Overloadlib

Niclas D. Gesing

Mar 18, 2022

CONTENTS

CHAPTER

ONE

FEATURES

- Introduces `@overload`, `@override` and `@<Function>.add` decorators, allowing one to overload and override functions. Functions are then called according to their argument types:

```
@overload
def func(var: str):
   return var

# via @<Function>.add
@func.add
def _(var: int) -> str:
   return str(var * 5)

# via @overload
@overload
def func() -> str:
   return "Functions don't need to have arguments."

# via @override
@override(funcs=[func])
def new(str_1: str, int_1: int):
   return str_1 * int_1

assert func("a") == "a" == new("a")
assert func(1) == "5" == new(1)
assert func() == "Functions don't need to have arguments." == new()
assert new("house", 2) == "househouse"
```

- Raises human readable errors, if no callable was determined with the given arguments. For example the following given:

```
@overload
def some_func(str_1: str, int_1: int):
   return str_1 + str(int_1)

@overload
def some_func(str_1: str):
   return str_1
```

Calling:

```
>>> some_func(str_1=2)
PyOverloadError:
Error when calling:
(__main__.some_func):
        def some_func(str_1: int):
                ...:
        'str_1' needs to be of type (<class 'str'>,) (is type <class 'int'>)
```

or

```
>>> some_func(10)
__main__.NoFunctionFoundError: No matching function found.
Following definitions of 'some_func' were found:
(__main__.some_func):
        def some_func(str_1: str, int_1: int):
                ...
(__main__.some_func):
        def some_func(str_1: str):
                ...
The following call was made:
(__main__.some_func):
        def some_func(int_1: int):
                ...
```

- Any type of variables is allowed: Build-in ones like `str, int, List` but also own ones, like classes etc.
- `@overload` uses `get_type_hints` to identify the right function call via type-checking. Hence, it may also be used as a type-checker for functions.
- Forgot, which overloads of a specific function have been implemented? No worries, you can print them with their typing information using *print(func_versions_info(<my_func>))*, e.g.

```
>>> print(func_versions_info(some_func))

Following overloads of 'some_func' exist:
(__main__.some_func):
         def some_func(str_1: str, int_1: int):
                 ...
(__main__.some_func):
         def some_func(str_1: str):
                 ...
```

CHAPTER

TWO

REQUIREMENTS

Requires Python 3.7+.

CHAPTER

THREE

INSTALLATION

You can install *Overloadlib* via pip from PyPI:

```
$ pip install overloadlib
```

or install with Poetry

```
$ poetry add overloadlib
```

Then you can run

```
$ overloadlib --help
```

or with Poetry:

```
$ poetry run overloadlib --help
```

<details> <summary>Installing Poetry</summary>

To download and install Poetry run (with curl):

```
$ curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/install-poetry.
↪py | python -
```

or on windows (without curl):

```
$ (Invoke-WebRequest -Uri https://raw.githubusercontent.com/python-poetry/poetry/master/
↪install-poetry.py -UseBasicParsing).Content | python -
```

 </details>

3.1 Uninstall

If you wan to uninstall the package, simply run

```
$ pip uninstall overloadlib
```

or with Poetry:

```
$ poetry remove overloadlib
```

CHAPTER

FOUR

USAGE

Please see the Command-line Reference for details.

CHAPTER

FIVE

CONTRIBUTING

Contributions are very welcome. To learn more, see the Contributor Guide.

CHAPTER

SIX

LICENSE

Distributed under the terms of the MIT license, *Overloadlib* is free and open source software.

CHAPTER

SEVEN

ISSUES

If you encounter any problems, please file an issue along with a detailed description.

CHAPTER

EIGHT

CREDITS

This project was generated by a template inspired by @cjolowicz's Hypermodern Python Cookiecutter template and @TezRomacH's python-package-template

8.1 Usage

Imports need to come form `overloadlib.overloadlib`. Importing via

```
from overloadlib.overloadlib import *
```

imports `overload` and `override` decorators, as well as `func_versions_info`.

Overloading of functions can be done via the `@overload` or `@override` decorator.

8.1.1 @overload

Overloading of functions:

```
@overload
def func(var: str):
    return var

@overload
def func(var: int):
    return str(var * 5)

func("a") == "a"  # True
"a: " + func(1)  # "a: 5"
```

Overloading of methods (or mixtures of both) are also possible using the same decorator:

```
@dataclass
class Hello:
    text: str = "Hello"
```

(continues on next page)

(continued from previous page)

```
class Some:
    def __init__(self) -> None:
        pass

    @overload
    def func(self, str_1: str, int_1: int) -> str:
        return str_1 + str(int_1)

    @overload
    def func(self, str_1: str) -> str:
        return str_1

    @overload
    def func(self, obj: Hello) -> str:
        return obj.text

@overload
def func(str_1: str) -> str:
    return "yummy " + str_1
```

Note that own classes, can be given as types to the function. Furthermore, methods and functions ma have the same name. Possible calls could now look like this:

```
# Giving only *args
some.func("Number: ", 1)  # "Number: 1"

# Giving **kwargs
some.func(str_1="Number: ", int_1=1)  # "Number: 1"

# An object as argument
some.func(Hello())  # "Hello"

# Calling the function not the method
func("yummy")  # "yummy cheese"
```

8.1.2 @override

You may also 'overload' functions using the `@override` decorator. This one overrides an list of callables or `Function` (function wrapper class of `overloadlib.py`.) via a given new 'parent' function.

```
def func_str(var: str) -> str:
    return "I am a string"

def func_int(var: int) -> str:
    return "I am an integer"

@overload
def func_both(var_1: int, var_2: str) -> str:
    return var_2 * var_1

@override(funcs=[func_str, func_int, func_both])  # callables and `Function` are given
```

(continues on next page)

(continued from previous page)

```
def new_func(fl: float) -> str:
    return "Float parameter"
```

Possible calls could now look like this:

```
new_func(1.0) == "Float parameter"  # True
new_func("a") == func_str("a") == "I am a string"  # True
new_func(1) == func_int(1) == "I am an integer"  # True
new_func(1, "a") == func_both(1, "a") == "a"  # True
```

Overriding `Function`'s (callables that are decorated with `@overload`) overrides every *version* of that `Function`:

```
@dataclass
class Some:
    text: str = "Hello"

@overload
def func(str_1: str) -> str:
    return str_1

@func.add
def _(obj: Some) -> str:
    return obj.text

@overload
def func() -> str:
    return "Functions don't need to have arguments."

# adds all previously defined overloads/'version' of `func` to `new`
@override(funcs=[func])
def new(str_1: str, int_1: int) -> str:
    return str_1 * int_1

assert new("a") == "a" == func("a")
assert new(Some()) == "Hello" == func(Some())
assert new() == "Functions don't need to have arguments." == func()
assert new("house", 2) == "househouse"
```

8.1.3 @<Function>.add

You can always add a new callable to an existing *overloaded* callable <func> using the `@<func>.add` decorator:

```
@overload
def some_func(str_1: str, int_1: int) -> str:
    return str_1 + str(int_1)

@some_func.add
def _(str_1: str) -> str:
    return str_1

@some_func.add
```

(continues on next page)

(continued from previous page)

```
def name_does_not_matter() -> str:
    return "I return some text."

@some_func.add
def _(str_1: str, str_2: str) -> str:
    return str_1 + str_2

assert some_func("This is a number: ", 10) == "This is a number: 10"
assert some_func("cheese") == "cheese"
assert some_func(Some()) == "Hello"
```

The name of the callable's you are adding don't matter and you can also always use the same name, when adding. However, as using the same name for added functions, clashes with `[no-redef]` error of mypy, it is recommended to use different ones (this also increases the readability of the code).[1]

Usage of `@override` and `@<Function>.add` is recommended over usage of `@overload` only, when working with static type checkers like mypy.

8.1.4 func_versions_info

If you want to get all versions of a certain function <myfunc>, use `func_versions_info(<myfunc>)`, e.g.

```
>>> print(func_versions_info(new_func))

Following overloads of 'new_func' exist:
(__main__.new_func):
        def new_func(var: str):
                ...
(__main__.new_func):
        def new_func(var: int):
                ...
(__main__.new_func):
        def new_func(var_1: int, var_2: str):
                ...
(__main__.new_func):
        def new_func(fl: float):
                ...
```

Common Mistakes and Limitations

- Overloading using overload raises problems with mypy. This can be circumvented using `@override` (or `@<func>.add`) instead of `@overload`.

[1] It should also be stressed, that `@<func>.add` only works with a previously with `@overload` decorated function `.

8.2 Reference

8.2.1 overloadlib.__main__

__main__ of overloadlib.

`overloadlib.__main__.version_callback()`
Print the version of the package.

Return type Any

8.3 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- Code of Conduct

8.3.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

8.3.2 How to request a feature

Request features on the Issue Tracker.

8.3.3 How to set up your development environment

You need Python 3.6+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
$ poetry run overloadlib
```

8.3.4 How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the `tests` directory, and are written using the pytest testing framework.

8.3.5 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before commiting your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

8.4 Contributor Covenant Code of Conduct

8.4.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

8.4.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

8.4.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

8.4.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

8.4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at nicdomgesing@gmail.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

8.4.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

8.4.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

8.5 MIT License

PYTHON MODULE INDEX

INDEX