Oop$_{e}xt$ Documentation

ESSS

Mar 19, 2021

CONTENTS:

CHAPTER

ONE

OOP EXTENSIONS

1.1 What is OOP Extensions ?

OOP Extensions is a set of utilities for object oriented programming which is missing on Python core libraries.

1.2 Usage

`oop_ext` brings a set of object oriented utilities, it supports the concept of interfaces, abstract/overridable methods and more. `oop_ext` carefully checks that implementations have the same method signatures as the interface it implements and raises exceptions otherwise.

Here's a simple example showing some nice features:

```
from oop_ext.interface import Interface, ImplementsInterface


class IDisposable(Interface):
    def dispose(self):
        """
        Clears this object
        """

    def is_disposed(self) -> bool:
        """
        Returns True if the object has been cleared
        """


@ImplementsInterface(IDisposable)
class MyObject(Disposable):
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        super().__init__()
        self._data = [0] * 100
        self._is_disposed = False

    def is_disposed(self) -> bool:
        return self._is_disposed

    def dispose(self):
        self._is_disposed = True
        self._data = []
```

If any of the two methods in `MyObject` are not implemented or have differ signatures than the ones declared in `IDisposable`, the `ImplementsInterface` decorator will raise an error during import.

Arbitrary objects can be verified if they implement a certain interface by using `IsImplementation`:

```
from oop_ext.interface import IsImplementation

my_object = MyObject()
if IsImplementation(my_object, IDisposable):
    # my_object is guaranteed to implement IDisposable completely
    my_object.dispose()
```

Alternatively you can assert that an object implements the desired interface with `AssertImplements`:

```
from oop_ext.interface import AssertImplements

my_object = MyObject()
AssertImplements(my_object, IDisposable)
my_object.dispose()
```

1.2.1 Type Checking

As of `1.1.0`, `oop-ext` includes inline type annotations and exposes them to user programs.

If you are running a type checker such as mypy on your tests, you may start noticing type errors indicating incorrect usage. If you run into an error that you believe to be incorrect, please let us know in an issue.

The types were developed against `mypy` version 0.800.

See the docs for more information.

1.2.2 Contributing

For guidance on setting up a development environment and how to make a contribution to oop_ext, see the contributing guidelines.

1.2.3 Release

A reminder for the maintainers on how to make a new release.

Note that the VERSION should follow the semantic versioning as X.Y.Z Ex.: v1.0.5

1. Create a `release-VERSION` branch from `upstream/master`.
2. Update `CHANGELOG.rst`.
3. Push a branch with the changes.
4. Once all builds pass, push a `VERSION` tag to `upstream`.
5. Merge the PR.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install Oop_ext, run this command in your terminal:

```
$ pip install oop_ext
```

This is the preferred method to install Oop_ext, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Oop_ext can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/ESSS/oop-ext
```

Or download the tarball:

```
$ curl  -OL https://github.com/ESSS/oop-ext/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

CALLBACKS

Callbacks provide an interface to register other callbacks, that will be *called back* when the `Callback` object is called.

A `Callback` is similar to holding a pointer to a function, except it supports multiple functions.

Example:

```
class Data:

    def __init__(self, x: int) -> None:
        self._x = x
        self.on_changed = Callback()

    @property
    def x(self) -> int:
        return self._x

    @x.setter
    def x(self, x: int) -> None:
        self._x = x
        self.on_changed(x)
```

In the code above, `Data` contains a `x` property, which triggers a `on_changed` callback whenever `x` changes.

We can be notified whenever `x` changes by registering a function in the callback:

```
def on_x(x: int) -> None:
    print(f"x changed to {x}")

data = Data(10)
data.on_changed.Register(on_x)
data.x = 20
```

The code above will print `x changed to 20`, because changing `data.x` triggers all functions registered in `data.on_changed`.

An important feature is that the functions connected to the callback are *weakly referenced*, so methods connected to a callback won't keep the method instance alive due to the connection.

We can unregister functions using `Unregister`, check if a function is registered with `Contains`, and unregister all connected functions with `UnregisterAll`.

3.1 Type Checking

New in version 1.1.0.

`oop-ext` also provides type-checked variants, `Callback0`, `Callback1`, `Callback2`, etc, which explicitly declare the number of arguments and types of the parameters supported by the callback.

Example:

```
class Point:
    def __init__(self, x: float, y: float) -> None:
        self._x = x
        self._y = y
        self.on_changed = Callback2[float, float]()

    def update(self, x: float, y: float) -> None:
        self._x = x
        self._y = y
        self.on_changed(x, y)


def on_point_changed(x: float, y: float) -> None:
    print(f"point changed: ({x}, {y})")


p = Point(0.0, 0.0)
p.on_changed.Register(on_point_changed)
p.update(100.0, 2.5)
```

In the example above, both the calls `self.on_changed` and `on_changed.Register` are properly type checked for number of arguments and types.

The method specialized signatures are only seen by the type checker, so using one of the specialized variants should have nearly zero runtime cost (only the cost of an empty subclass).

Note: The separate callback classes are needed for now, but if/when pep-0646 lands, we should be able to implement the generic variants into `Callback` itself.

CHAPTER

FOUR

INTERFACES

`oop-ext` introduces the concept of interfaces, common in other languages.

An interface is a class which defines methods and attributes, defining a specific behavior, so implementations can declare that they work with an specific interface without worrying about implementations details.

Interfaces are declared by subclassing *`oop_ext.interface.Interface`*:

```
from oop_ext.interface import Interface


class IDataSaver(Interface):
    """
    Interface for classes capable of saving a dict containing
    builtin types into persistent storage.
    """

    def save(self, data: dict[Any, Any]) -> None:
        """Saves the given list of strings in persistent storage."""
```

(By convention, interfaces start with the letter `I`).

We can write a function which gets some data and saves it to persistent storage, without hard coding it to any specific implementation:

```
def run_simulation(params: SimulationParameters, saver: IDataSaver) -> None:
    data = calculate(params)
    saver.save(data)
```

`run_simulation` computes some simulation data, and uses a generic `saver` to persist it somewhere.

We can now have multiple implementations of `IDataSaver`, for example:

```
from oop_ext.interface import ImplementsInterface


@ImplementsInterface(IDataSaver)
class JSONSaver:
    def __init__(self, path: Path) -> None:
        self.path = path

    def save(self, data: dict[Any, Any]) -> None:
        with self.path.open("w", encoding="UTF-8") as f:
            json.dump(f, data)
```

And use it like this:

```
run_simulation(params, JSONSaver(Path("out.json")))
```

4.1 What about duck typing?

In Python declaring interfaces is not really necessary due to *duck typing*, however interfaces bring to the table **runtime validation**.

If later on we add a new method to our `IDataSaver` interface, we will get errors at during *import time* about implementations which don't implement the new method, making it easy to spot the problems early. Interfaces also verify parameters names and default values, making it easy to keep implementations and interfaces in sync.

Note: Changed in version 2.0.0.

Interfaces do not check type annotations at all.

It was supported initially, but in practice this feature has shown up to be an impediment to adopting type annotations incrementally, as it discourages adding type annotations to improve existing interfaces, or annotating existing implementations without having to update the interface (and all other implementations by consequence).

It was decided to let the static type checker correctly deal with matching type annotations, as it can do so more accurately than `oop-ext` did before.

4.2 Type Checking

New in version 1.1.0.

The interfaces implementation has been implemented many years ago, before type checking in Python became a thing.

The static type checking approach is to use Protocols, which has the same benefits and flexibility of interfaces, but without the runtime cost. At ESSS however migrating the entire code base, which makes extensive use of interfaces, is a lengthy process so we need an intermediate solution to fill the gaps.

To bridge the gap between the runtime-based approach of interfaces, and the static type checking provided by static type checkers, one just needs to subclass from both *Interface* and `TypeCheckingSupport`:

```
from oop_ext.interface import Interface, TypeCheckingSupport


class IDataSaver(Interface, TypeCheckingSupport):
    """
    Interface for classes capable of saving a dict containing
    builtin types into persistent storage.
    """

    def save(self, data: dict[Any, Any]) -> None:
        """Saves the given list of strings in persistent storage."""
```

The `TypeCheckingSupport` class hides from the user the details necessary to make type checkers understand `Interface` subclasses.

Note that subclassing from `TypeCheckingSupport` has zero runtime cost, existing only for the benefits of the type checkers.

Note: Due to how `Protocol` works in Python, every `Interface` subclass **also** needs to subclass `TypeCheckingSupport`.

4.3 Proxies

Given an interface and an object that implements an interface, you can call *`GetProxy`* to obtain a *proxy object* which only contains methods and attributes defined in the interface.

For example, using the `JSONSaver` from the previous example:

```
def run_simulation(params, saver):
    data = calculate(params)
    proxy = GetProxy(IDataSaver, saver)
    proxy.save(data)
```

The `proxy` object contains a stub implementation which contains only methods and attributes in `IDataSaver`. This prevents mistakes like accessing a method that is defined in `JSONSaver`, but is not part of `IDataSaver`.

4.3.1 Legacy Proxies

With type annotations however, this is redundant: the type checker will prevent access to any method not declared in `IDataSaver`:

```
def run_simulation(params: SimulationParameters, saver: IDataSaver) -> None:
    data = calculate(params)
    saver.save(data)
```

However when adding type annotations to legacy code, one will encounter this construct:

```
def run_simulation(params, saver):
    data = calculate(params)
    proxy = IDataSaver(saver)
    proxy.save(data)
```

Here "creating an instance" of the interface, passing an implementation of that interface, returns the stub implementation. This API was implemented like this for historic reasons, mainly because it would trick IDEs into providing code completion for `proxy` as if a `IDataSaver` instance.

When adding type annotations, prefer to convert that to *`GetProxy`*, which is friendlier to type checkers:

```
def run_simulation(params: SimulationParameters, saver: IDataSaver) -> None:
    data = calculate(params)
    proxy = GetProxy(IDataSaver, saver)
    proxy.save(data)
```

Or even better, if you don't require runtime checking, let the type checker do its job:

```
def run_simulation(params: SimulationParameters, saver: IDataSaver) -> None:
    data = calculate(params)
    saver.save(data)
```

Note: As of `mypy 0.812`, there's a bug that prevents *`GetProxy`* from being properly type annotated. Hopefully this will be improved in the future.

CHAPTER

FIVE

API REFERENCE

Note: This page is WIP, PRs are welcome!

class oop_ext.foundation.callback.**Callback**

Object that provides a way for others to connect in it and later call it to call those connected.

Callbacks are stored as weakrefs to objects connected.

Determining kind of callable (Python 3)

Many parts of callback implementation rely on identifying the kind of callable: is it a free function? is it a function bound to an object?

Below there is a table to help understand how different objects are classified:

```
                    |has__self__|has__call__|has__call__self__
↪|isbuiltin|isfunction|ismethod
--------------------|-----------|-----------|-----------------|---------|---------
↪-|--------
free function       |False      |True       |True             |False    |True
↪ |False
bound method        |True       |True       |True             |False    |False
↪ |True
class method        |True       |True       |True             |False    |False
↪ |True
bound class method  |True       |True       |True             |False    |False
↪ |True
function object     |False      |True       |True             |False    |False
↪ |False
builtin function    |True       |True       |True             |True     |False
↪ |False
object              |True       |True       |True             |True     |False
↪ |False
custom object       |False      |False      |False            |False    |False
↪ |False
string              |False      |False      |False            |False    |False
↪ |False
```

where rows are:

```
def free_fn(foo):
    # `free function`
    pass
```

(continues on next page)

(continued from previous page)

```
class Foo:
    def bound_fn(self, foo):
        pass


class Bar:
    @classmethod
    def class_fn(cls, foo):
        pass


class ObjectFn:
    def __call__(self, foo):
        pass


foo = Foo()  # foo is `custom object`, foo.bound_fn is `bound method`
bar = Bar()  # Bar.class_fn is `class method`, bar.class_fn is `bound class␣
↪method`

object_fn = ObjectFn()  # `function object`

obj = object()  # `object`
string = "foo"  # `string`
builtin_fn = string.split  # `builtin function`
```

And where columns are:

- isbuiltin: inspect.isbuiltin
- isfunction: inspect.isfunction
- ismethod: inspect.ismethod
- has__self__: hasattr(obj, '__self__')
- has__call__: hasattr(obj, '__call__')
- has__call__self__: hasattr(obj.__call__, '__self__') if hasattr(obj, '__call__') else False

---

Note: After an internal refactoring, `__slots__` has been added, so, it cannot have weakrefs to it (but as it stores weakrefs internally, that shouldn't be a problem). If weakrefs are really needed, `__weakref__` should be added to the slots.

Contains (*func: Callable[..., Any]*, *extra_args: Sequence[object] = ()*) → bool

Parameters **func** (*object*) – The function that may be contained in this callback.

Return type bool

Returns True if the function is already registered within the callbacks and False otherwise.

Register (*func: Callable[..., Any]*, *extra_args: Sequence[object] = ()*) → oop_ext.foundation.callback._callback._UnregisterContext

Registers a function in the callback.

Parameters

- **func** – Method or function that will be called later.

- **extra_args** – Arguments that will be passed automatically to the passed function when the callback is called.

Returns

A context which can be used to unregister this call.

The context object provides this low level functionality, if you are registering many callbacks at once and plan to unregister them all at the same time, consider using *Callbacks* instead.

Unregister (*func: Callable[..., Any]*, *extra_args: Sequence[object] = ()*) → None
Unregister a function previously registered with Register.

Parameters **func** (*object*) – The function to be unregistered.

UnregisterAll () → None
Unregisters all functions

class oop_ext.foundation.callback.**Callbacks**
Holds created callbacks, making it easy to disconnect later.

This class provides two methods of operation:

- *Before()* and *After()*:

  This provides connection support for arbitrary functions and methods, similar to mocking them.

- *Register()*:

  Registers a function into a *Callback*, making the callback call the registered function when it gets itself called.

In both modes, *RemoveAll()* can be used to unregister all callbacks.

The class can also be used in context-manager form, in which case all callbacks are unregistered when the context-manager ends.

Note: This class keeps a strong reference to the callback and the sender, thus they won't be garbage-collected while still connected.

After (*sender: T*, *callback: Callable*, *, *sender_as_parameter: bool = False*) → T
Same as *Before()*, but will call the callback after the sender function has been called.

Before (*sender: T*, *callback: Callable*, *, *sender_as_parameter: bool = False*) → T
Registers a callback to be executed before an arbitrary function.

Example:

```
class C:
    def foo(self, x): ...

def callback(x): ...


Before(C.foo, callback)
```

The call above will result in callback to be called for *every instance* of C.

Register (*callback: oop_ext.foundation.callback._callback.Callback*, *func: Callable*) → None
Registers the given function into the given callback.

This will automatically unregister the function from the given callback when *Callbacks.RemoveAll()* is called or the context manager ends in the context manager form.

RemoveAll() → None
Remove all registered functions, either from *Before()*, *After()*, or *Register()*.

class oop_ext.interface.**Interface**(*class_: Any = <object object>*)
Base class for interfaces.

A interface describes a behavior that some objects must implement.

TypeCheckingSupport

New in version 1.1.0.

Interfaces that which to support static type checkers such as `mypy` also need to subclass from this class:

```
from oop_ext.interface import Interface, TypeCheckingSupport


class IDataSaver(Interface, TypeCheckingSupport):
    ...
```

The `TypeCheckingSupport` exists solely for the benefit of type checkers, and has zero runtime cost associated with it.

oop_ext.interface.**ImplementsInterface**(**interfaces: Any, no_check: bool = False*) → Callable[T, T]
Make sure a class implements the given interfaces. Must be used in as class decorator:

```
@ImplementsInterface(IFoo)
class Foo(object):
    ...
```

Parameters **no_check** – If `True`, does not check if the class implements the declared interfaces during import time.

oop_ext.interface.**GetProxy**(*interface: Type[Any], obj: T*) → T
Obtains a *proxy object* for `obj`, which contains only methods and attributes declared in `interface`.

Usage:

```
def run_simulation(params: SimulationParameters, saver: IDataSaver) -> None:
    data = calculate(params)
    proxy = GetProxy(IDataSaver, saver)
    proxy.save(data)
```

Note however this is redundant when used with type checkers: `saver: IDataSaver` already tells the type checker to only allow access to legal members. This is useful however when type annotating legacy code which doesn't have any type annotations and makes use of the runtime mechanism to ensure interface compliance.

Note: As of `mypy 0.812`, there's a bug that prevents *GetProxy* from being properly type annotated. Hopefully this will be improved in the future.

CHAPTER

SIX

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

6.1 Get Started!

Ready to contribute? Here's how to set up *oop_ext* for local development.

1. Fork the *oop_ext* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_github_username_here/oop-ext.git
```

3. Create a virtual environment and activate it:

```
$ python -m virtualenv .env

$ .env\Scripts\activate  # For Windows
$ source .env/bin/activate  # For Linux
```

4. Install the development dependencies for setting up your fork for local development:

```
$ cd oop_ext/
$ pip install -e .[testing,docs]
```

Note: If you use `conda`, you can install `virtualenv` in the root environment:

```
$ conda install -n root virtualenv
```

Don't worry as this is safe to do.

---

5. Install pre-commit:

```
$ pre-commit install
```

6. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

7. When you're done making changes, run the tests:

```
$ pytest
```

8. If you want to check the modification made on the documentation, you can generate the docs locally:

```
$ tox -e docs
```

   The documentation files will be generated in `docs/_build`.

9. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

10. Submit a pull request through the GitHub website.

6.2 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated.

CHAPTER

SEVEN

CHANGELOG

7.1 2.1.0 (2021-03-19)

- #48: New type-checker friendly `proxy = GetProxy(I, obj)` function as an alternative to `proxy = I(obj)`. The latter is not accepted by type checkers in general because interfaces are protocols, which can't be instantiated.

  Also fixed a type-checking error with `AsssertImplements`:

```
Only concrete class can be given where "Type[Interface]" is expected
```

  This happens due to python/mypy#5374.

7.2 2.0.0 (2021-03-10)

- #47: Interfaces no longer check type annotations at all.

  It was supported initially, but in practice this feature has shown up to be an impediment to adopting type annotations incrementally, as it discourages adding type annotations to improve existing interfaces, or annotating existing implementations without having to update the interface (and all other implementations by consequence).

  It was decided to let the static type checker correctly deal with matching type annotations, as it can do so more accurately than `oop-ext` did before.

7.3 1.2.0 (2021-03-09)

- #43: Fix support for type annotated `Attribute` and `ReadOnlyAttribute`:

```
class IFoo(Interface):
    value: int = Attribute(int)
```

7.4 1.1.2 (2021-02-23)

- #41: Fix regression introduced in `1.1.0` where installing a callback using `callback.After` or `callback.Before` would make a method no longer compliant with the signature required by its interface.

7.5 1.1.1 (2021-02-23)

- #38: Reintroduce `extra_args` argument to `Callback._GetKey`, so subclasses can make use of it.
- #36: Fix regression introduced in `1.1.0` where `Abstract` and `Implements` decorators could no longer be used in interfaces implementations.

7.6 1.1.0 (2021-02-19)

- #25: `oop-ext` now includes inline type annotations and exposes them to user programs.

  If you are running a type checker such as mypy on your tests, you may start noticing type errors indicating incorrect usage. If you run into an error that you believe to be incorrect, please let us know in an issue.

  The types were developed against `mypy` version 0.800.

- #26: New type-checked `Callback` variants, `Callback0`, `Callback1`, `Callback2`, etc, providing type checking for all operations(calling, `Register`, etc) at nearly zero runtime cost.

  Example:

```
from oop_ext.foundation.callback import Callback2


def changed(x: int, v: float) -> None:
    ...


on_changed = Callback2[int, float]()
on_changed(10, 5.25)
```

- Fixed `Callbacks.Before` and `Callbacks.After` signatures: previously their signature conveyed that they supported multiple callbacks, but it was a mistake which would break callers because every parameter after the 2nd would be considered the `sender_as_parameter` parameter, which was forwarded to `After` and `Before` functions of the `_shortcuts.py` module.

7.7 1.0.0 (2020-10-01)

- `Callbacks` can be used as context manager, which provides a `Register(callback, function)`, which automatically unregisters all functions when the context manager ends.
- `Callback.Register(function)` now returns an object with a `Unregister()` method, which can be used to undo the register call.

7.7.1 0.6.0 (2020-01-31)

- Change back the default value of `requires_declaration` to `True` and fix an error (#22) where the cache wasn't properly cleared.

7.8 0.5.1 (2019-12-20)

- Fixes an issue (#20) where mocked *classmethods* weren't considered a valid method during internal checks.

7.9 0.5.0 (2019-12-12)

- Add optional argument `requires_declaration` so users can decide whether or not `@ImplementsInterface` declarations are necessary.

7.10 0.4.0 (2019-12-03)

- Implementations no longer need to explicitly declare that they declare an interface with `@ImplementsInterface`: the check is done implicitly (and cached) by *AssertImplements* and equivalent functions.

7.11 0.3.2 (2019-08-22)

- Interface and implementation methods can no longer contain mutable defaults, as this is considered a bad practice in general.
- `Null` instances are now hashable.

7.12 0.3.1 (2019-08-16)

- Fix mismatching signatures when creating "interface stubs" for instances:

```
foo = IFoo(Foo())
```

7.13 0.3.0 (2019-08-08)

- Interfaces now support keyword-only arguments.

7.14 0.2.4 (2019-03-22)

- Remove `FunctionNotRegisteredError` exception, which has not been in use for a few years.

7.15 0.2.3 (2019-03-22)

- Fix issues of ignored exception on nested callback.

7.16 0.2.1 (2019-03-14)

- Fix issues and remove obsolete code.

7.17 0.1.8 (2019-03-12)

- First release on PyPI.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX