Aspect-oriented Programming in Python

A Tutorial at Chemnitzer Linux-Tage 2025

March 23, 2025

Chemnitz

Author:	Dr.-Ing. Mike Müller
E-Mail:	mmueller@python-academy.de
Twitter:	pyacademy
Version:	1.0

Trainer:	Dr.-Ing. Mike Müller
E-Mail:	mmueller@python-academy.de

Python Academy – The Python training specialist

- Dedicated to Python training since 2006
- Wide variety of Python topics
- Experienced Python trainers
- Trainers are specialists and often core developers of taught software
- All levels of participants from novice to experienced software developers
- Courses for system administrators
- Courses for scientists and engineers
- Python for data analysis
- Open courses Europe-wide
- Customized in-house courses
- Python programming
- Development of technical and scientific software
- Code reviews
- Coaching of individuals and teams migrating to Python
- Workshops on developed software with detailed explanations

Current training modules – Python Academy

As of 2025

Module Topic	Length (days)	In-house	Remote	Open Training
Python for Programmers	4	yes	yes	yes
Python for Non-Programmers	5	yes	yes	yes
Advanced Python	3	yes	yes	yes
Python for Scientists and Engineers	3	yes	yes	yes
Python for Data Analysis	3	yes	yes	yes
Machine Learning with Python	3	yes	yes	yes
Professional Testing with Python	3	yes	yes	yes
High Performance Computing with Python	3	yes	yes	yes
Cython in Depth	2	yes	yes	yes
Introduction to Django	4	yes	yes	yes
Advanced Django	3	yes	yes	yes
Image Processing with Python	3	yes	yes	no
SQLAlchemy	3	yes	yes	no
High Performance XML with Python	1	yes	yes	no
Optimizing Python Programs	1	yes	yes	no
Python Extensions with Other Languages	1	yes	yes	no
Data Storage with Python	1	yes	yes	no
Introduction to Software Engineering with Python	1	yes	yes	no
Introduction to PySide/PyQt	1	yes	yes	no
Overview of the Python Standard Library	1	yes	yes	no
Threads and Processes in Python	1	yes	yes	no
Network Programming with Python	1	yes	yes	no

We offer on-site and open courses world-wide. We customize and extend training modules for your needs. We also provide consulting services such as code review, customized programming, and workshops. Open courses are offered as in-person training in Leipzig, Germany as well as remote trainer-led training. Individuals participants can enroll in open courses.

More information: www.python-academy.com

Contents

1 Basic Decorators

1.1 Wrapping Functions and Classes

Decorator provide a very useful method to add functionality to existing functions and classes. Decorators are functions that wrap other functions or classes. Decorators use the `@` syntax to "attach" a decorator to a function or class.

1.2 Examples from the Core Language

1.2.1 Static Methods

One example for the use of decorators are static methods. Static methods could be functions in the global scope but are defined inside a class. There is no `self` and no reference to the instance. Before Python 2.4 they had to defined like this:

```
class C:
    def func():
        """No self here."""
        print('Method used as function.')
    func = staticmethod(func)
```

```
c = C()
```

```
c.func()
```

```
Method used as function.
```

Because the `staticmethod` call is after the actual definition of the method, it can be difficult to read and easy to be overlooked. Therefore, the new `@` syntax is used before the method definition, but it does the same:

```
class C:
    @staticmethod
    def func():
        """No self here."""
        print('Method used as function.')
```

```
c = C()
```

```
c.func()
```

```
Method used as function.
```

The same works for class methods that take a class objects as argument instead of the instance (aka `self`).

1.2.2 Class Methods

Another example are class methods. While a "normal" method automatically receives `self` when called from an instance, a class method will be called on a class and receives the class as the first argument. Often the name `cls` is used for this first argument, i.e. the class. A typical example for a class method is an additional constructor, i.e. a method that creates a new instance with a different signature.

Let's look at an example. We create a class that represents a point in the plain with x and y coordinates. This point has a `__repr__` method that shows the same text that was used to create an instance:

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'{self.__class__.__name__}({self.x!r}, {self.y!r})'
```

After creating an instance:

```
point1 = Point(10, 20)
```

The point represents itself just like the entered source code:

```
point1
```

```
Point(10, 20)
```

Now, we add a classmethod `from_point` that takes the class as first and a point as second argument. It returns a new instance of the current class, which is created based on the values of `x` and `y` of the given class:

```
class FlexiblePoint:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    def from_point(cls, point):
        return cls(point.x, point.y)

    def __repr__(self):
        return f'{self.__class__.__name__}({self.x!r}, {self.y!r})'
```

Now, we can make an instance from existing instance of point without supplying `x` and `y` as single values:

```
point2 = FlexiblePoint.from_point(point1)
```

```
point2
```

```
FlexiblePoint(10, 20)
```

1.3 Examples from Other Libraries

There are many Python libraries that use decorators as a central element.

These include:

- Click - Add command line arguments to a function
- Django - Regulate user permissions
- Flask - Do routing
- Cython - Add types to a function
- Numba - Add types to a function

1.4 Closures

We can use the concept of a closure for writing a function decorator. Using this technique, we can create a function that takes a function as argument and returns a new function. The closure allows to access arguments of the outer function from within the inner function:

```
def outer(outer_arg):
    def inner(inner_arg):
        return inner_arg + outer_arg
    return inner
```

Now, we can create a new function by calling `outer`:

```
new_func = outer(10)
```

This functions looks strange:

```
new_func
```

```
<function __main__.outer.<locals>.inner(inner_arg)>
```

Calling this function:

```
new_func(7)
```

```
17
```

calculates the sum of the given `7` and the `10` provided when creating this function. How does the new function have access to this `10`? It is stored in the attribute `__closure__`:

```
new_func.__closure__[0].cell_contents
```

```
10
```

Wikipedia[1] defines a closure as:

> In programming languages, a closure, also lexical closure or function closure, is a technique for implementing lexically scoped name binding in a language with first-class functions.

Since "everything is an object", functions and classes are Python objects too. Furthermore, in Python all objects are first-class.

[1] https://en.wikipedia.org/wiki/Closure_(computer_programming)

1.5 Writing a Simple Decorator

Writing you own decorator is simple:

```python
def hello(func):
    print('Hello')
```

Now apply it to a function:

```python
@hello
def add(a, b):
    return a + b
```

```
Hello
```

The `Hello` got printed. But calling our `add` doesn't work:

```python
add(10, 20)
```

```
TypeError: 'NoneType' object is not callable
```

This might become clearer if we look at it the old way:

```python
def add(a, b):
    return a + b

add = hello(add)
```

```
Hello
```

```python
add(20, 30)
```

```
TypeError: 'NoneType' object is not callable
```

So, even it is not enforced by the interpreter, decorators usually make sense (at least the way they are intended to be used) if they behave in a certain way. It is strongly recommended that a function decorator always returns a function object and a class decorator always returns a class object. A function decorator should typically either return a function that returns the result of the call to the original function and do something in addition, or return the original function itself.

This is a more useful example:

```python
def hello(func):
    """Decorator function."""
    def call_func(*args, **kwargs):
        """Takes a arbitrary number of positional and keyword arguments."""
        print('Hello')
        # Call original function and return its result.
        return func(*args, **kwargs)
    # Return function defined in this scope.
    return call_func
```

Now we can create our decorated function and call it:

```
@hello
def add(a, b):
    return a + b
```

```
add(20, 30)
```

```
Hello
```

```
50
```

and again:

```
add(20, 300)
```

```
Hello
```

```
320
```

1.6 Best Practice

When we use docstrings, as we always do:

```
def add(a, b):
    """Add two objects."""
    return a + b
```

we can access them later:

```
add.__doc__
```

```
'Add two objects.'
```

When we now wrap our function:

```
def hello(func):
    def call_func(*args, **kwargs):
        """Wrapper."""
        print('Hello')
        return func(*args, **kwargs)
    return call_func
```

and decorate it:

```
@hello
def add(a, b):
    """Add two objects."""
    return a, b
```

we loose our docstring:

```
add.__doc__
```

```
'Wrapper.'
```

We could manually set the docstring of our wrapped function to remain the same. But the module `functools` in the standard library helps us here:

```python
import functools

def hello(func):
    @functools.wraps(func)
    def call_func(*args, **kwargs):
        """Wrapper."""
        print('Hello')
        return func(*args, **kwargs)
    return call_func
```

Now we have a nice docstring after decorating our function:

```python
@hello
def add(a, b):
    """Add two objects."""
    return a + b
```

```python
add.__doc__
```

```
'Add two objects.'
```

Python allows to call function recursively:

```python
def recurse(x):
    if x:
        x -= 1
        print(x)
        recurse(x)
```

```python
recurse(5)
```

```
4
3
2
1
0
```

When we decorate a recursive function the wrapper will also be called recursively:

```python
@hello
def recurse(x):
    if x:
        x -= 1
        print(x)
        recurse(x)
```

```python
recurse(5)
```

```
Hello
4
```

(continues on next page)

(continued from previous page)

```
Hello
3
Hello
2
Hello
1
Hello
0
Hello
```

In most cases this is not desirable. Therefore, recursive function should not be decorated. Don't assume you have only one decorator.

1.7 Use cases

Decorators can be used for different purposes. Here are some common use cases.

1.7.1 Caching

Expensive but often repeated calculations can be cached. A simple function cache that never expires and grows without limit could look like this:

```
"""Caching results with a decorator."""

import functools
import pickle


def cached(func):
    """Cache with a decorator."""
    cache = {}

    @functools.wraps(func)
    def _cached(*args, **kwargs):
        """Take the function arguments."""
        # dicts cannot be use as dict keys
        # dumps are strings and can be used
        key = pickle.dumps((args, kwargs))
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]

    return _cached
```

Now we can decorate our expensive function:

```
@cached
def add(a, b):
    print('calc')
    return a + b
```

Only the first call will print `calc`. All subsequent calls get the value from cache without newly calculating it:

```
add(10, 10)
```

```
calc
```

```
20
```

```
add(10, 10)
```

```
20
```

```
add(10, 10)
```

```
20
```

1.7.2 Logging

Another use case is logging. We log things if the global variable `LOGGING` is true:

```
"""Helper to switch on and off logging of decorated functions."""

import functools

LOGGING = False


def logged(func):
    """Log with a decorator."""

    @functools.wraps(func)
    def _logged(*args, **kwargs):
        """Take the function arguments."""
        if LOGGING:
            print('logged')  # do proper logging here
        return func(*args, **kwargs)

    return _logged
```

After decorating our function:

```
@logged
def add(a, b):
    return a + b
```

and setting `LOGGING` to true:

```
LOGGING = True
```

we log:

```
add(10, 10)
```

```
logged
```

```
20
```

or not:

```
LOGGING = False
```

```
add(10, 10)
```

```
20
```

1.8 Exercises

1.8.1 Exercise 1

Write a function decorator that can be used to measure the run time of a function. Use `timeit.default_timer()` to get time stamps.

1.8.2 Exercise 2

Use `functools.wraps()` to preserve the function attributes including the docstring that you wrote.

2 Advanced Decorators

2.1 Parameterized Decorators

Decorators can take arguments. We redefine our decorator. The outermost function takes the arguments, the next more inner function takes the function and the innermost function will be returned and will replace the original function:

```
def say(text):
    def _say(func):
        def call_func(*args, **kwargs):
            print(text)
            return func(*args, **kwargs)
        return call_func
    return _say
```

Now we decorate with `Hello` to get the same effect as before:

```
@say('Hello')
def add(a, b):
    return a, b
```

```
add(10, 20)
```

```
Hello
```

```
(10, 20)
```

or with `Goodbye`:

```
@say('Goodbye')
def add(a, b):
    return a, b
```

```
add(10, 20)
```

```
Goodbye
```

```
(10, 20)
```

2.2 Chaining Decorators

We can use more than one decorator for one function:

```
@say('A')
@say('B')
@hello
def add(a, b):
    return a, b
```

```
add(10, 20)
```

```
A
B
Hello
```

```
(10, 20)
```

2.3 Callable Instances

So far we used functions to write decorators. Actually, Python uses a callable. Often the terms function and callable are used interchangeable. Any object that reacts to an added pair of parenthesis (`callable_name()`) is callable. We can check if an object is callable:

```
callable(sum)
```

```
True
```

```
callable(int)
```

```
True
```

Both are calable even though they are of diffremt types:

```
type(sum)
```

```
builtin_function_or_method
```

```
type(int)
```

```
type
```

So, even classes are callable. The syntax of calling a function and creating an instance is the same. In the background Python calls the special method `__call__` when the `()` is added after an object. Therefore, we can implement `__call__` on our own class:

```
class CallCounter:

    def __init__(self):
        self.count = 0
```

(continues on next page)

(continued from previous page)

```python
    def __call__(self):
        self.count +=1
```

Creating an instance (we name it like a function ;)):

```python
my_func = CallCounter()
```

We can now look at the attribute `count``:

```python
my_func.count
```

```
0
```

Calling our instance:

```python
my_func()
```

increments the count:

```python
my_func.count
```

```
1
```

We can repeat this:

```python
my_func()
```

```python
my_func.count
```

```
2
```

We can use this to write a decorator:

```python
from functools import wraps

class Say:

    def __init__(self, text):
        self.text = text

    def __call__(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(self.text)
            return func(*args, **kwargs)
        return wrapper
```

Apply just like our function-based decorator:

```python
@Say('Hello')
def add(a, b):
    return a + b
```

```
add(10, 20)
```

```
Hello
```

```
30
```

2.4 Use Cases

2.4.1 Argument Checking

We check if the positional arguments to a function call are of a certain type. First we define our decorator:

```
"""Check function arguments for given type."""

import functools


def check(*argtypes):
    """Check function argument types."""

    def _check(func):
        """Take function as argument."""

        @functools.wraps(func)
        def __check(*args):
            """Take function arguments."""
            if len(args) != len(argtypes):
                msg = f'Expected {len(argtypes)} but got {len(args)} arguments'
                raise TypeError(msg)
            for arg, argtype in zip(args, argtypes):
                if not isinstance(arg, argtype):
                    msg = f'Expected {argtypes} but got '
                    msg += f'{tuple(type(arg) for arg in args)}'
                    raise TypeError(msg)
            return func(*args)

        return __check

    return _check
```

Then we decorate our function:

```
@check(int, int)
def add(x, y):
    """Add two integers."""
    return x + y
```

We have our docstring:

```
add.__doc__
```

```
'Add two integers.'
```

and can call it with two integers:

```
add(1, 2)
```

```
3
```

But calling with an integer and a float doesn't work:

```
add(1, 2.0)
```

```
TypeError: Expected (<class 'int'>, <class 'int'>) but got (<class 'int'>,
↪<class 'float'>)
```

Also the wrong number of parameters won't work:

```
add(1)
```

```
TypeError: Expected 2 but got 1 arguments
```

```
add(1, 1, 1)
```

```
TypeError: Expected 2 but got 3 arguments
```

We can't use our function if we have a different number of parameters in the decorator than in the function definition:

```
@check(int, int, int)
def add(x, y):
    """Add two integers."""
    return x + y
```

```
add(1, 2)
```

```
TypeError: Expected 3 but got 2 arguments
```

2.4.2 Registration

Another useful application is registration. We would like to register functions. The first way is to make them append themselves to a list when they are called. We use a dictionary `registry` to store these lists. This is our decorator:

```
"""A function registry."""

import functools

registry = {}


def register_at_call(name):
    """Register the decorated function at call time."""

    def _register(func):
        """Take the function."""

        @functools.wraps(func)
        def __register(*args, **kwargs):
            """Take the function arguments."""
```

(continues on next page)

(continued from previous page)

```
            registry.setdefault(name, []).append(func)
            return func(*args, **kwargs)

        return __register

    return _register
```

and this is our empty registry:

```
registry
```

```
{}
```

We define three decorated functions:

```
@register_at_call('simple')
def f1():
    pass
@register_at_call('simple')
def f2():
    pass
@register_at_call('complicated')
def f3():
    pass
```

The registry is still empty:

```
registry
```

```
{}
```

Now we call our functions and fill the registry:

```
f1()
registry
```

```
{'simple': [<function __main__.f1()>]}
```

```
f2()
registry
```

```
{'simple': [<function __main__.f1()>, <function __main__.f2()>]}
```

```
f3()
registry
```

```
{'simple': [<function __main__.f1()>, <function __main__.f2()>],
 'complicated': [<function __main__.f3()>]}
```

We can also look at the names of our functions:

```
f1.__name__
```

```
'f1'
```

```
[f.__name__ for f in registry['simple']]
```

```
['f1', 'f2']
```

```
[f.__name__ for f in registry['complicated']]
```

```
['f3']
```

Of course we will append a function every time we call it:

```
registry
```

```
{'simple': [<function __main__.f1()>, <function __main__.f2()>],
 'complicated': [<function __main__.f3()>]}
```

If we want to register our function at definition time, we have to change our decorator:

```
def register_at_def(name):
    """Register the decorated function at definition time."""

    def _register(func):
        """Take the function."""
        registry.setdefault(name, []).append(func)

        return func

    return _register
```

Now we add our function right when we define it:

```
registry = {}
@register_at_def('simple')
def f1():
    pass
```

```
registry
```

```
{'simple': [<function __main__.f1()>]}
```

Calling `f1()` doesn't change anything in the registry:

```
f1()
registry
```

```
{'simple': [<function __main__.f1()>]}
```

```
f1()
registry
```

```
{'simple': [<function __main__.f1()>]}
```

2.5 Class Decorators

We can use decorators for classes too:

```
def mark(cls):
    cls.added_attr = 'I am decorated.'
    return cls
```

```
@mark
class A:
    pass
```

```
A.added_attr
```

```
'I am decorated.'
```

It is important to always return a class object from the decorating function. Otherwise users cannot make instances from our class.

2.6 Use Cases

2.6.1 Verification

Verification is another useful way to use decorators. Let's make sure we have fluid water:

```
def assert_fluid(cls):
    assert 0 <= cls.temperature <= 100
    return cls
```

We decorate our class:

```
@assert_fluid
class Water:
    temperature = 20
```

```
@assert_fluid
class Steam:
    temperature = 120
```

```
AssertionError
```

It won't work if it is too hot or too cold:

2.6.2 Define-Time Checks of Naming Conventions

We can use class decorators to check if the names of methods of a class adhere to naming conventions. Let's say the name of a method should not be longer than a certain number of characters. This class decorator will raise a `NameError` if a name of a method is too long:

```python
"""Class decorator to check method name length."""


def check_name_length(max_len=30):
    """
    Check method name length.

    Raises a `NameError` if one method name of a decorated class is
    longer than `max_len`.
    """

    def _check_name_length(cls):
        for name, obj in cls.__dict__.items():
            if callable(obj) and len(name) > max_len:
                msg = (
                    f'name `{name}` too long\n'
                    f'found {len(name)} characters, '
                    f'only {max_len} are allowed'
                )
                raise NameError(msg)
        return cls

    return _check_name_length
```

Now, adding this decorator to a class will raise an exception, complaining about a method with too long name:

```python
@check_name_length(max_len=20)
class A:

    def good_meth(self):
        return 42

    def method_with_rather_long_name_that_is_difficult_to_use(self):
        return 43
```

```
NameError: name `method_with_rather_long_name_that_is_difficult_to_use` too long
found 53 characters, only 20 are allowed
```

2.7 Exercises

2.7.1 Exercise 3

Modify your solution from exercise 2. Measure the average run time for multiple runs of the function. To achieve this, make the decorator parameterized. It should take an integer that specifies how often the function has to be run. Make sure you divide the resulting run time by this number.

2.7.2 Exercise 4

Make the time measurement optional by using a global switch in the module that can be set to `True` or `False` to turn time measurement on or off.

2.7.3 Exercise 5

Write another decorator that can be used with a class and registers every class that it decorates in a dictionary. Use a string consisting of the module name (`cls.__module__`) and the class name (`cls.__name__`) as key for each class.