__wrapped__ in Python decorators

https://rmariano.eu/posts/wrapped-in-python-decorators/

Mariano Anaya Fri Jul, 30 01:26

This is another of the new interesting things that Python 3 has. Time ago, it was somehow tricky to work with decorated functions, because the decorator replaced the original object, and its behaviour became hard to reach.

So, for example, we all know that the following sentence:

```
@decorator
def function():
    pass
```

It's actually syntax sugar for:

```
def function():
    pass

function = decorator(function)
```

So, if for some reason, we need to work both with the original and decorated functions, we required tricks such as different names, something like:

```
def _function():
    pass

function = decorator(_function)
```

But in current Python, this should be no longer the case. As we can see from the code of the `functools` module 1, when decorating an object, there is an attribute named `__wrapped__` that holds the reference to the original one.

So now if we use this, we can access it directly without having to resort to the old quirks.

Example

Let's consider this example. Validating parameter types like this is far from a real implementation, but rather something for the sake of illustration.

```
from functools import wraps


def validate_parameters(kwargs, annotations):
    '''For a dictionary of kwargs, and another dictionary of annotations,
    mapping each argument name with its type, validate if all types on the
    keyword arguments match those described by the annotations.
    '''
    for arg, value in kwargs.items():
        expected_type = annotations[arg]
        if not isinstance(value, expected_type):
            raise TypeError(
                '{0}={1!r} is not of type {2!r}'.format(
                    arg, value, expected_type)
                )


def validate_function_parameters(function):
    @wraps(function)
    def inner(**kwargs):
        annotations = function.__annotations__
        validate_parameters(kwargs, annotations)
        return function(**kwargs)
    return inner




@validate_function_parameters
def test_function(x: int, y:float):
    return x * y


test_function.__wrapped__(x=1, y=3)  # does not validate
test_function(x=1, y=3)  # validates and raises
```

The statement in the penultimate line works because it's actually invoking the original function, without the decorator applied, whereas the last one fails because it's calling the function already decorated.

Potential use cases

In general this a nice feature, for the reason that in a way enables accessing both objects, (it could be argued that decorating a function causes some sort of temporal coupling).

Most importantly, it can be used in *unit tests*, whether is to test the original function, or that the decorator itself is acting as it is supposed to do.

Moreover, we could tests our own code, before being decorated by other libraries being used in the project (for example the function of a tasks without the `@celery.task` decorator applied, or an event of the database of `SQLAlchemy` before it was changed by the listener event decorator, etc.).

Just another trick to keep in the toolbox.

[1]
This attribute is documented, but I first found about it while reading at the code of CPython at https://github.com/python/cpython/blob/3405792b024e9c6b70c0d2355c55a23ac84e1e67/Lib/functools.py#L70