Decorators in Python

Ziming Xu
231220003@smail.nju.edu.cn

Abstract

This essay examines the versatile use of decorators in Python, focusing on their role in augmenting functions without altering their core behavior. It explores how decorators facilitate logging, timing, and authorization mechanisms. The essay also demonstrates decorators' integration into frameworks like Flask, showcasing their broad applicability in streamlining complex tasks within Python applications.

ACM Reference Format:
Ziming Xu. 2023. Decorators in Python. In *Proceedings of Hello! (PLP Workshop '23)*. ACM, New York, NY, USA, 2 pages. https://doi.org/404.404

1 Introduction

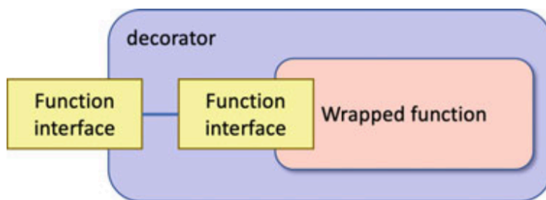

Figure 1. Decorator wrapping a callable objects [1]

During developing, it's necessary to add additional behaviors or functionalities to objects dynamically without altering their core structure or behavior. However, directly adding these codes inside the function changes the original behaivour of the function and is detrimental to further usage and adaption of the function. Moreover, it adds the code complexity and puts burden on developers. Thus, decorator pattern emerges. It creates a function which accepts another function as a parameter, execute the behaviours that decorate the original function and then execute the orginal function. It is adaptable and won't change original function's behavior. Thus it has wide application such as logging, timing, caching and integrating functions into existing behaviours. Moreover, Python supports decorator pattern grammatically. The contents below will talk about some of the applications of decorators in Python.

```python
def calc(x):
    print('Executing calc')
    print(x*2)
    print('Ending calc')
def calc2(x):
    print('Executing calc2')
    print(x*3)
    print('Ending calc2')
```

Listing 1. Traditionally adding behaviors are troublesome

2 Applications

2.1 Logging and timing

It is quite necessary to log which function is executing and when does it end. It provides history for debugging when an error occurs and reveal the sequence of events leading up to the problem. It can also provide real-time information about the program's status including the behaviours, performance and even unexpected events. Decorators can be applied to log and time any functions. Here is a simple example.

```python
import functools
def log(func):
    @functools.wraps(func)
    def wrapper(*args,**kwargs):
        print(f"Logging: Calling {func.__name__} with args={args} and kwargs={kwargs}")
        result = func(*args,**kwargs)
        if result: print(f"Logging: {func.__name__} returned {result}")
        return result
    return wrapper
@log
def calc(x):
    print(x*2)
@log
def calc2(x):
    print(x*3)
```

Listing 2. A decorator for logging

2.2 Authorization

In software developing, there exists a circumstance where multiple users are created and different users have different access to different objects or data. This type of security mechanism is called authorization. Typically, it's rather complex to define each user's permitted functions. However,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLP Workshop '23, December 23, 2023, Nanjing, Jiangsu
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/404.404

with decorators, the mechanism is easy-to-write and also well-structured.

```
import functools
user_permission = {
    'admin':['read','write'],
    'user1':['read'],
    'user2':['write']
}
def authorize(required_actions):
    def dec(func):
        @functools.wraps(func)
        def wrapper(user, *args,**kwargs):
            if all(action in user_permission.get(
                user) for action in
                required_actions):
                return func(user, *args, **kwargs)
            else:
                raise PermissionError('Permission
                    denied.')
        return wrapper
    return dec
@authorize(['read'])
def do_something(user):
    return 'something here'
```

It is easy and clear to write each function's behaviours and required actions before it and this kind of code style is also easy to debug. Moreover, add required actions after modifying each function won't take too much effort.

2.3 Integrating functions into existing framework

Flask is a lightweight Python network framework where decorators are also used. In *Flask*, the most common used decorator is @app.route that associates a URL and methods with a Python function. Here is an example.

```
from flask import *
app = Flask(__name__)
# Simple route returning a hello message
@app.route('/')
def index():
    return 'Welcome to my Flask application!'
# Route handling a form submission
@app.route('/submit', methods=['GET', 'POST'])
def submit():
    if request.method == 'POST':
        username = request.form['username']
        return f'Thank you, {username}!'
    return render_template('form.html')
if __name__ == '__main__':
    app.run(debug=True)
```

Listing 3. Applications of Python Decorators in Flask

@app.route('/') associates the / URL with the index() function. When a request is made to the root URL, Flask will execute the index() function and return the specified message.

3 Conclusion

The essay talks about decorators in Python, demonstrating its role in adding behaviours without changing the function. Moreover, the essay talks about three specific applications and their association with decorators.

References

[1] Hunt, J. *Decorators.* Springer International Publishing, Cham, 2023, pp. 339–351.