python_decorators_0324

March 24, 2023

1 Introduction to Decorators in Python:

1.1 Intro: Counting calls of function (manually implementing a decorator)

[ ]:
```
def fib_rec(n):
    """Naive recurse implementation to compute the Fibonacci numbers."""
    if n in [0,1] :
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)
```

[ ]:
```
[fib_rec(n) for n in range(10)]
```

[ ]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

[ ]:
```
# just to keep a reference to the original function (it'll get modified)
fib_rec_orig = fib_rec
```

When using this approach to compute fib_rec for not so small n's (say n>=40)
we notive that it is quite slow. We'd like keep track of the number of calls.

- Alternative1: Manually add an attribute n_calls to fib_rec.
- Alternative2: Try to come up with a more general solution that can be reused for other functions.

[ ]:
```
def call_counter(func):
        def helper(*args, **kwargs):
                helper.calls += 1
                return func(*args, **kwargs)
        helper.calls = 0
        helper.__name__ = func.__name__

        return helper
```

The classical way of using this would be:

[ ]:
```
fib_rec = call_counter(fib_rec)
```

[ ]:
```
fib_rec(20)
```

[ ]:
```
6765
```

[ ]:
```
print(fib_rec.calls)
```

```
21891
```

[ ]:
```
# just to illustrate that the function is really called quite often
print(fib_rec(30))
print(fib_rec.calls)
```

```
832040
2714428
```

2 Making a function "remember" values: Memoization

https://python-course.eu/advanced-python/memoization-decorators.php

2.1 Using decorators via the @ syntax

Suppose the call_counter function/decorator has been defined as above:

[ ]:
```
def call_counter(func):
        def helper(*args, **kwargs):
                helper.calls += 1
                return func(*args, **kwargs)
        helper.calls = 0
        helper.__name__ = func.__name__

        return helper
```

We can then wrap others functions using the synatax:

```
@call_counter
def function_to_be_wrapped(....) :
```

In general

```
@f
def g(...)
```

has the same effect as: g = f(g)

[ ]:
```
@call_counter
def fib_rec(n):
    """Naive recurse implementation to compute the Fibonacci numbers."""
    if n in [0,1] :
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)
```

[ ]:
```
print(fib_rec(20))
print(fib_rec.calls)
```

```
6765
21891
```

[ ]:
```
# remark: Here "calls" counts the total number of calls, not the calls used for
# one specific calculation.
print(fib_rec(20))
print(fib_rec.calls)
```

```
6765
43782
```

Decorators can be composed and the composition is done as we know it from mathematics:

```
@dec1
@dec2
def foo():
```

is the same as :

foo = dec1(dec2(foo)).

Thus the composition of decorators is just the same as the composition of functions.

2.2 Loss of docstrings and what to do about it

Remember we kept a 'backup' of our original fib_rec function before decorationg it: fib_rec_orig

[ ]:
```
print(fib_rec_orig.__doc__)
```

```
Naive recurse implementation to compute the Fibonacci numbers.
```

[ ]:
```
print(fib_rec.__doc__)
```

```
None
```

We see: The docstring disappeared. Of course it did, we just need to look at our defintion of call_counter.

To avoid this one usually use the functools.wraps which handles organizational stuff like docstrings for us:

[ ]:
```
import functools
```

[ ]:
```
def call_counter_via_wraps(func):
        @functools.wraps(func)
        def helper(*args, **kwargs):
                helper.calls += 1
                return func(*args, **kwargs)
        helper.calls = 0
        helper.__name__ = func.__name__
```

```
        return helper
```

[ ]:
```
@call_counter_via_wraps
def fib_rec(n):
    """Naive recurse implementation to compute the Fibonacci numbers."""
    if n in [0,1] :
        return n
    else:
        return fib_rec(n-1) + fib_rec(n-2)
```

[ ]:
```
print(fib_rec.__doc__)
```

```
Naive recurse implementation to compute the Fibonacci numbers.
```

3 A simpple logging Decorator

[ ]:
```
def logging_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Starting {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Finished {func.__name__}")
        return result
    return wrapper

@logging_decorator
def say_hello(name):
    print(f"Hello, {name}!")

say_hello("Alice")
```

```
Starting say_hello
Hello, Alice!
Finished say_hello
```

4 Detecting exections

[ ]:
```
def catch_exceptions_decorator(func):
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except Exception as e:
            print(f"An exception occurred in {func.__name__}: {e}")
            # You can handle the exception here or re-raise it if needed
    return wrapper

@catch_exceptions_decorator
```

```
def divide(a, b):
    return a / b

result = divide(4, 2)  # No exception, should return 2
print(f"Result: {result}")

result = divide(4, 0)  # ZeroDivisionError exception, should be caught by the␣
 ↪decorator
print(f"Result: {result}")
```

```
Result: 2.0
An exception occurred in divide: division by zero
Result: None
```

5 Decorators with Arguments

You can define a **decorator function that takes arguments** by defining a nested function that takes the arguments and returns the decorator function. For example:

[ ]:
```
def my_decorator_with_args(arg1, arg2):
    def decorator(func):
        def wrapper(*args, **kwargs):
            print(f"Decorator arguments: {arg1}, {arg2}")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@my_decorator_with_args("Hallo", "World")
def my_function():
    print("Hello, world!")

my_function()
```

```
Decorator arguments: Hallo, World
Hello, world!
```

6 Summary

[ ]:
```

```

Definition of Decorators:

Decorators are a feature of Python that allows you to modify the behavior of a function or class without changing the source code. They are more or less functions that take in a function as an argument and return a modified version of the same function.

Syntax of Decorators:

The syntax for a decorator is to place the decorator function above the function being modified, and to use the "@" symbol to indicate that the function is being decorated. For example:

Basic Use Cases

- add functionality to an existing functin, e.g. modify its input or output: Example: show the time it takes to run a function

Built-in Decorators:

- @staticmethod
- @classmethod
- @property

7 References

- https://python-course.eu/advanced-python/decorators-decoration.php
- https://www.python-kurs.eu/python3_dekorateure.php
- [Einführung in Python, Bernd Klein, 4e, Hanser Verlag], Kap.36 "Dokorateure"
- https://realpython.com/primer-on-python-decorators/
- This following link only gives a glimpse of decorators, however **Kristian Rothers** github repos are a great resource for learning python: https://github.com/krother/advanced_python/blob/master/functions/decorators.md
- https://towardsdatascience.com/5-real-handy-python-decorators-for-analyzing-debugging-your-code-c22067318d47

8