Chapter 05 - The Decorator Pattern

September 18, 2021

0.1 Overview

- also a `structural pattern`
- structural patterns encourage composition over inheritance
- the `decorator` pattern allows a programmer to add responsibilities to an object dynamically, and in a transparent manner

- in python, we can write decorators in a pythonic way using the build in decorator feature
- a python decorator is a `callable` (function, method, or class) that gets a function object `func_in` as input and returns another function object `func_out`
- there is a one-to-one relationship between a decorator pattern and python's decorator feature
- python's decorators can actually do much more than the decorator pattern

0.2 Use Cases

- the decorator pattern shines when used for implementing cross-cutting concerns
- example of cross-cutting concerns are:
  - data validation
  - caching
  - logging
  - monitoring
  - debugging
  - business rules
  - encryption

- in general, all parts of an application that are generic and can be applied to many other parts of it are considered to be corss-cutting concerns
- another popular example of using the decorator pattern is `GUI` toolkits
- in a GUI toolkit, we want to be able to add features such as borders, shadows, colors and scrolling to individual components/widgets

0.3 Implementation

- as an example we will implment a memoization decorator

[2]:
```
def number_sum(n):
    '''Returns the sum of the first n numbers'''
    assert(n >= 0), 'n must be >= 0'
    if n == 0:
        return 0
```

```
    else:
        return n + number_sum(n-1)

if __name__ == '__main__':
    from timeit import Timer
    t = Timer('number_sum(30)', 'from __main__ import number_sum')
    print('Time: ', t.timeit())
```

```
Time:  6.310352899999998
```

- it takes 6 seconds to do this calculation
- we can memoize to improve the performance number
- in the code, we use a `dict` for caching the already computed sums
- we will also change the parameter passed to the `number_sum()` function
- we want to calculate the sum of the first 300 numbers instead of only the first 30

[3]:
```
sum_cache = {0:0}

def number_sum(n):
    assert (n >= 0), 'n numst be >= 0'
    if n in sum_cache:
        return sum_cache[n]
    res = n + number_sum(n-1)
    # add the value to the cache
    sum_cache[n] = res
    return res

if __name__ == '__main__':
    from timeit import Timer
    t = Timer('number_sum(30)', 'from __main__ import number_sum')
    print('Time: ', t.timeit())
```

```
Time:  0.25533679999989545
```

- there are already a few problems with this approach
- the code is not as clean as it was when not using memoization
- the code is also not reusable, if for instance, we wanted to get the fibonacci number

- to solve the problem, we first create a `memoize()` decorator
- our decorator accepts the function `fn` that needs to be memoized as an imput
- it uses a `dict` names `cache` as the cached data container
- the `functools.wraps()` function is used for convenience when creating decorators
- it is not mandatory but a good pratice to use it, since it makes sure that the documentation, and the signature of the function that is decorated, are preserved
- the argument list `*args` is required in this case because the functions that we want to decorate accepts input arguments

[4]:
```
import functools

def memoize(fn):
```

```
    cache = dict()

    @functools.wraps(fn)
    def memoizer(*args):
        if args not in cache:
            cache[args] = fn(*args)
        return cache[args]
    return memoizer
```

- now we can use our `memoize()` decorator with the naive version of our functions
- we apply the decorator using the `@name` syntax

[6]:
```
@memoize

def number_sum(n):
    '''Returns the sum of the first n numbers'''
    assert(n >= 0), 'n must be >= 0'
    if n == 0:
        return 0
    else:
        return n + number_sum(n-1)

@memoize
def fibonacci(n):
    '''Returns the suite of Fibonacci numbers'''
    assert(n >= 0), 'n must be >= 0'
    if n in (0, 1):
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

- in the last part oif the code, via `main()` we can see how to use the decorated functions and measure their performance
- the `to_execute` variable is used to hold a list of tuples containing the refrence to each function and the corresponding `timeit.Timer()` call

[7]:
```
def  main():
    from timeit import Timer

    to_execute = [
        (number_sum,
        Timer('number_sum(300)', 'from __main__ import number_sum')),
        (fibonacci,
        Timer('fibonacci(100)', 'from __main__ import fibonacci'))
    ]

    for item in to_execute:
        fn = item[0]
```

```
        print(f'Function "{fn.__name__}": {fn.__doc__}')
        t = item[1]
        print(f'Time: {t.timeit()}')
        print()

if __name__ == '__main__':
    main()
```

```
Function "number_sum": Returns the sum of the first n numbers
Time: 0.2522417000000132

Function "fibonacci": Returns the suite of Fibonacci numbers
Time: 0.25787690000015573
```