BIGDATA TEAM

#W7PA: Decorator Design Pattern

*задачи на Decorator составлены по мотивам занятий Николая Субоча, составлял задание Алексей Драль. Хотите стать автором задачи - пишите предложения на study@bigdatateam.org.

1. Описание задания

Шаблоны проектирования используются для создания дизайна (архитектуры) приложения. Для каждого приложения обычно можно выбрать несколько разных архитектур, которые могут быть “правильными”. Выбор наиболее “правильного” - это вопрос вкусовщины или внутренних правил команды или компании, в которой вы работаете. Проверять архитектуру приложения или валидность реализации шаблона проектирования в автоматическом режиме сложно, если вообще возможно (придумайте метрики “лаконичности” и “читабельности” кода). Поэтому, некоторые компании организуют архитектурные советы, куда включают опытных разработчиков, чтобы делать ревью предложенных архитектур и давать рекомендации по реализации / развитию проекта.

В рамках этого задания вам предлагается детально изучить что скрывается за символом @ языка программирования Python, каким идеологиям он соответствует и как удобно писать менеджеры контекстов с помощью декораторов. Интересные находки и вопросы выбора архитектуры предлагаем обсуждать в чате курса.

2. Decorator и @ в Python

Функция в Python – это тоже объект. Её можно передать как параметр в другую функцию:

```
def verbose_decorator(function):
    def wrapper(*args, **kwargs):
        print(f'wrap {function}')
        print('before')
        function(*args, **kwargs)
        print('after')

    return wrapper
```

Заворачиваем в декоратор:

```
@verbose_decorator
def hello(name, last_name, age):
    print(f'*** Hello {name} {last_name}! {age} ***')
```

При вызове задекорированной функции, получим:

```
>> hello('Nikolay', 'Suboch', age=18)
wrap <function hello at 0x7fbea7eebdd0[1]>
before
*** Hello Nikolay Suboch! 18 ***
after
```

Как уже было показано в предыдущих модуля, этот код с @ ровно тоже самое, что и:

```
def hello(name, last_name, age):
    print(f'*** Hello {name} {last_name}! {age} ***')

hello = verbose_decorator(hello)
```

Пользуясь этой идеологией создайте декоратор, который может принимать дополнительные аргументы. Например, декоратор repeater(n), который будет вызывать обернутую функцию ровно n раз. Подумайте, внимательно, что означает конструкция:

```
@repeater(n)
def my_fancy_function(arg_1, arg_2):
    # ...
```

подсказка: не бойтесь большой вложенности.

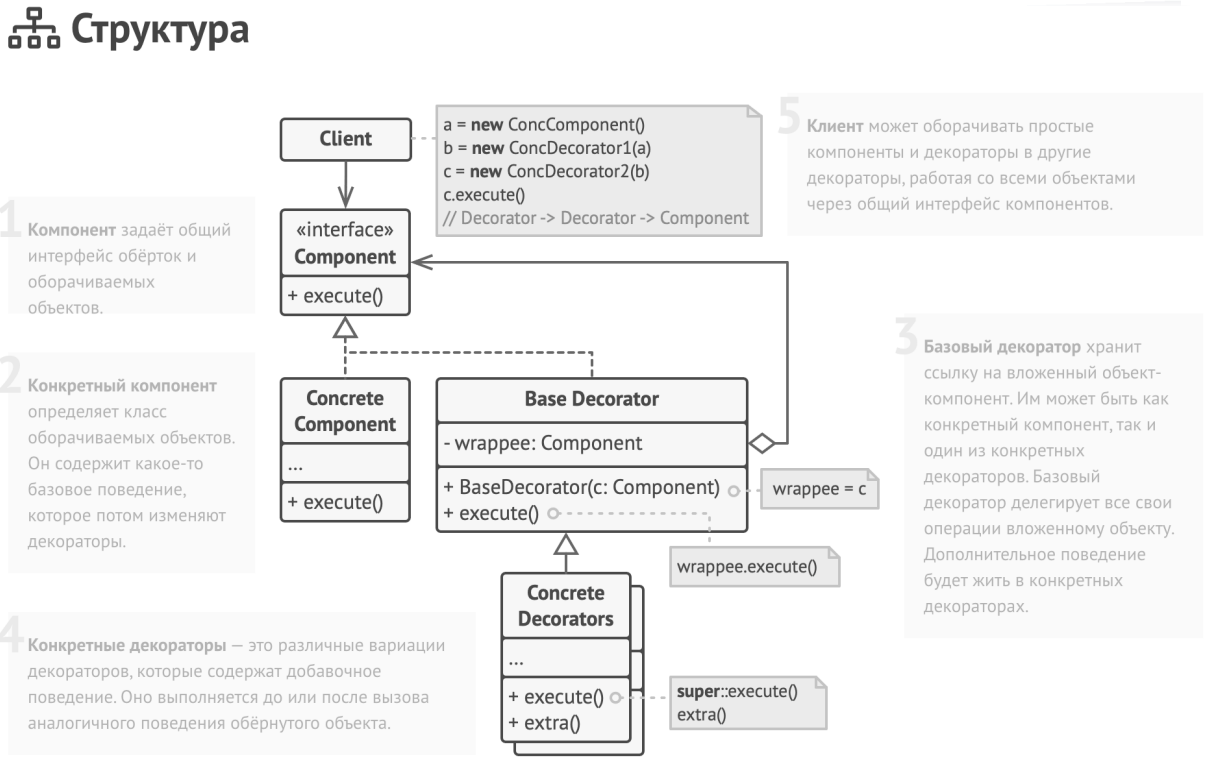
3. Decorator в классической литературе

Изучите шаблон проектирования декоратор:

- Для любителей англоязычной литературы, см.
  - https://sourcemaking.com/design_patterns/decorator
- Для любителей русской классики, от того же автора, см.
  - https://refactoring.guru/ru/design-patterns/decorator

Реализуйте шаблон проектирования декоратор самостоятельно и убедитесь, что понимаете смысл каждой связи на диаграмме классов:

[1] id объекта (функции) в runtime Python (по сути - адрес в памяти), при разных вызовах будет разный адрес



Сравните с выразительностью @ на языке Python.

4. Задания на Decorator и Context Manager

В рамках курса мы неоднократно видели конструкцию с “with” и в рамках одного из видео даже посмотрели на стандартную реализацию менеджера контекста в библиотеке mock. Вам предлагается освежить эти воспоминания и наработать практику в написании собственных менеджеров контекста. Благодаря ним, код становится более лаконичным и безопасным.

Изучите что такое contextmanager и ContextDecorator:

- https://docs.python.org/3/library/contextlib.html#contextlib.contextmanager
- https://docs.python.org/3/library/contextlib.html#contextlib.ContextDecorator

Реализуйте аналог менеджера контекста “open” для закрытия файлов после выхода из клаузы с with (или даже в случае получения exception):

1. Вариант с созданием класса, реализующим enter/exit;
2. Вариант с написанием функции, обернутой в @contextmanager.

Напишите тесты и убедитесь, что декоратор работает.

Бонусное задание. Реализуйте менеджер контекста, который будет реализовывать функциональность по хранению отступа. При входе в блок увеличиваем отступы всех следующих печатаемых текстов на 2 пробела, при выходе - возвращаем обратно. Пример:

```
with Indenter() as indent:
    indent.print("hi")
    with indent:
        indent.print("hello")
        with indent:
            indent.print("bonjour")
    indent.print("hey")
```

```
stdout:
  hi
    hello
      bonjour
  hey
```

Бонусное задание со звездочкой. Изучите, что такое шаблон проектирования RAII:

- https://en.cppreference.com/w/cpp/language/raii

Сравните его с contextmanager в Python. Чем они похожи, а чем отличаются?

5. Полезные материалы

Полезные материалы для расширения кругозора:

- Design Patterns: Elements of Reusable Object-Oriented Software by Gamma Erich, Helm Richard, Johnson Ralph , Vlissides John
- https://sourcemaking.com/ (En) + https://refactoring.guru/ (Ru)

Всем удачи!