BIGDATA TEAM

HW #07: Decorator Design Pattern

---

*задачи на Decorator составлены по мотивам занятий Николая Субоча и Алексея Драля в рамках обучения на курсах промышленной разработки.

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. Цель задания: научить понимать и писать декораторы в Python.

1.1. Декоратор (Decorator) и @ в Python

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

```
def verbose(function):
    def wrapper(*args, **kwargs):
        print("before function call")
        outcome = function(*args, **kwargs)
        print("after function call")
        return outcome

    return wrapper
```

Заворачиваем в декоратор:

```
@verbose
def hello(name: str):
    print(f"*** Hello {name}! ***")
```

При вызове задекорированной функции, получим:

```
>> hello("Nikolay")
before function call
*** Hello Nikolay! ***
after function call
```

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

```
def hello(name):
    print(f"*** Hello {name}! ***")

hello = verbose(hello)
```

Пользуясь этой идеологией и дополнительными ссылками на документацию нужно будет создать несколько декораторов, которые могут принимать дополнительные аргументы и держать контекст использования в памяти.

1.2. Декоратор repeater

Создайте декоратор repeater(count: int), который будет вызывать обернутую функцию ровно count раз. Подумайте, внимательно, что означает конструкция:

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

подсказка: не бойтесь большой вложенности.

1.3. Декоратор и ContextDecorator

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 (см. пример @patch). Вам предлагается освежить эти воспоминания и наработать практику в написании собственных менеджеров контекста. Благодаря ним, код становится более лаконичным и безопасным.

Бонусом, оказывается, что менеджеры контекста можно удобно использовать и как декораторы для определения поведения до и после вызова функций. Такие

декораторы называются контекстные декораторы (ContextDecorator) и увидеть примеры использования можно по ссылке:

- https://docs.python.org/3/library/contextlib.html#contextlib.ContextDecorator (5 мин на чтение)

Что такое with и какие методы должны быть определены и с каким поведением можно узнать по ссылке:

- https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers (5 мин на чтение)

Ваша задача, написать класс ContextDecorator, который можно использовать по аналогии с декоратором @verbose:

- "class: before function call" - данное выражение должно выводиться до вызова функции
- "class: after function call" - данное выражение должно выводиться после вызова функции
- класс-декоратор должен называться verbose_context

Следующая конструкция

```
@verbose_context()
def hello(name: str):
    print(f"*** Hello {name}! ***")
```

Должна при вызове функции hello написать в терминал:

```
>> hello("Nikolay")
class: before function call
*** Hello Nikolay! ***
class: after function call
```

Если интересуют более глубокие детали использования with в Python читайте:

- https://docs.python.org/3/reference/compound_stmts.html#the-with-statement

1.4. Декоратор Indenter

Реализуйте менеджер контекста, который будет реализовывать следующую функциональность:

`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`

Indenter должен принимать на вход аргументы:

- `indent_str: str = " " * 4`
- `indent_level: int = 0`

Ниже представлено несколько test-кейсов, где передаются только указанные аргументы и какой ожидается stdout:

аргументы	stdout

`indent_str="--"`	`hi`
	`--hello`
	`----bonjour`
	`hey`

`indent_str="--", indent_level=1`	`--hi`
	`----hello`
	`------bonjour`
	`--hey`

Напишите сначала тесты, а затем реализуйте декоратор.

1.5. Требования к реализации

Реализуйте декораторы verbose, verbose_context и repeater в файле repeater.py, а тесты на них расположите в файле test_repeater.py. Обязательное условие - добейтесь того, чтобы результат задекорированной функции и ее docstring не изменялись после декорирования:

```
@verbose
@repeater(3)
@verbose_context()
def hello(name: str):
```

```
    """my hello docstring"""
    print(f"Hello {name}!")
```

Должен вернуть функцию, у которой можно получить .__name__ и .__doc__. Для простого решения этой задачи изучите возможности стандартного модуля functools:

- https://docs.python.org/3/library/functools.html

У вас также должны быть получены файлы indenter.py и test_indenter.py.

Тесты должны покрывать 100% функциональности в обоих случаях.

2. Рекомендации

Рекомендации по разработке:

- следите за качеством кода и проверяйте “глупые” ошибки с помощью pylint, следите за поддерживаемостью и читаемостью кода;
- держите уровень покрытия кода тестами на уровне 80+%, следуйте TDD (сначала тесты, потом реализация);
- отделяйте фазу рефакторинга от фазы добавления новой функциональности.
    - фиксируем функциональность, все тесты зеленые;
    - проводим рефакторинг;
    - по окончании фазы рефакторинга снова все тесты зеленые;
- следите за скоростью выполнения unit-test’ов, несколько секунд - это хорошо, в противном случае нужно уменьшать размер тестируемых датасетов или разделять тесты на фазы (см. обсуждение про mark.slow);

3. Критерии оценивания

Балл за задачу складывается из:

- **20%** - правильная реализация verbose, verbose_context и repeater
- **20%** - качество покрытия юнит-тестами test_*repeater.py
  - оценка качества проводится автоматически вызовом pytest:
    - `PYTHONPATH=. pytest -v --cov=task_*_repeater test_*_repeater.py`
    - уровень покрытия тестами должен **быть ровно 100%**
    - проверяем код Python версии 3.7 с помощью `pytest==6.0.1`
- **30%** - правильная реализация Indenter
- **20%** - качество покрытия юнит-тестами test_*indenter.py
  - оценка качества проводится автоматически вызовом pytest:
    - `PYTHONPATH=. pytest -v --cov=task_*_indenter test_*_indenter.py`
    - уровень покрытия тестами должен **быть ровно 100%**
    - проверяем код Python версии 3.7 с помощью `pytest==6.0.1`
- **10%** - поддерживаемость и читаемость кода
  - в общем случае см. Clean Code и Google Python Style Guide
  - оценка качества будет проводиться автоматическим вызовом pylint:
    - `pylint task_*.py`
    - качество кода должно оцениваться выше 8.0 / 10.0
    - проверяем код Python версии 3.7 с помощью `pylint==2.5.3`
    - точная формула: 10% x min([lint_quality / 8.0], 1.0)

Discounts (скидки и другие акции):

- **100%** за плагиат в решениях (всем участникам процесса)
- **100%** за посылку решения после hard deadline
- **30%** за посылку решения в после soft deadline и до hard deadline
- **5%** за каждую посылку после 2й посылки в день (каждый день можно делать до 2х посылок без штрафа)

лучший балл с 1-й попытки: 100%
лучший балл со 2-й попытки: 100%
лучший балл с 3-й попытки: 95%
лучший балл с 4-й попытки: 90%

4. Инструкция по отправке задания

Оформление задания:

- Код задания (Short name): **HW07:Decorator**
- Выполненное ДЗ запакуйте в архив **PY-MADE-2021-Q4**_<Surname>_<Name>_HW#**.zip**, пример -- **PY-MADE-2021-Q4**_Dral_Alexey_**HW07**.zip. (Проверяйте отсутствие пробелов и невидимых символов после копирования имени отсюда.[1]) Если ваше решение лежит в папке my_solution_folder, то для создания архива hw.zip на Linux и Mac OS выполните команду[2]:
  - `zip -r hw.zip my_solution_folder/*`
- На Windows 7/8/10: необходимо выделить все содержимое директории my_solution_folder/ нажать правую кнопку мыши на одном из выделенных объектов, выбрать в открывшемся меню "Отправить >", затем "Сжатая ZIP-папка". Теперь можно переименовать архив.
- Решение задания должно содержаться в одной папке.
- Перед проверкой убедитесь, что дерево вашего архива выглядит так:
  - **PY-MADE-2021-Q4_<Surname>_<Name>_HW07.zip**
  - **| ---- task_<Surname>_<Name>_indenter.py**
  - **| ---- test_<Surname>_<Name>_indenter.py**
  - **| ---- task_<Surname>_<Name>_repeater.py**
  - **| ---- test_<Surname>_<Name>_repeater.py**
  - ~~**| ---- *.txt**~~[3]
  - При несовпадении дерева вашего архива с представленным деревом, ваше решение не будет возможным автоматически проверить, а значит, и оценить его.
- Для того, чтобы сдать задание, необходимо:
  - Зарегистрироваться и залогиниться в сервисе Everest
  - Перейти на страницу приложения: MADE Python Grader
  - Выбрать вкладку Submit Job (если отображается иная).
  - Выбрать в качестве “Task” значение: **HW07:Decorator**[4]
  - Загрузить в качестве “Task solution” файл с решением
  - В качестве Access Token указать тот, который был выслан по почте
- **Перед отправкой задания**, оставьте, пожалуйста, отзыв о домашнем задании по ссылке: https://rebrand.ly/pymade2021q4_feedback_hw. Это позволит нам скорректировать учебную нагрузку по следующим заданиям (в зависимости от

[1] Онлайн инструмент для проверки: https://www.soscisurvey.de/tools/view-chars.php
[2] Флаг -r значит, что будет совершен рекурсивный обход по структуре директории
[3] Тестовые данные для тестирования данного решения не нужны
[4] Сервисный ID: python.decorator

того, сколько часов уходит на решение ДЗ), а также ответить на интересующие вопросы.

Внимание: если до дедлайна остается меньше суток, и вы знаете (сами проверили или коллеги сообщили), что сдача решений сломана, обязательно сдайте свое решение, прислав нам ссылку на выполненное задание (Job) на почту с темой письма “Short name. ФИО.”. Например: “**HW07:Decorator**. Иванов Иван Иванович.” Таким образом, мы сможем увидеть какое решение у вас было до дедлайна и сможем его оценить. Пример ссылки:

- https://everest.distcomp.org/jobs/67893456230000abc0123def

Любые вопросы / комментарии / предложения пишите согласно предложениям на портале.

Всем удачи!

5. FAQ (часто задаваемые вопросы)

"You are not allowed to run this application", что делать?

Если Вы видите надпись "You are not allowed to run this application" во вкладке Submit Job в Everest, то на данный момент сдача закрыта (нет доступных для сдачи домашних заданий, по техническим причинам или другое). Попробуйте, пожалуйста, еще раз через некоторое время. Если Вы еще ни разу не сдавали, у коллег сдача работает, но Вы видите такое сообщение, сообщите нам об этом.

Grader показывает 0 или < 0, а отчет (Grading report) не помогает решить проблему

Ситуации:

- система оценивания показывает оценку (Grade) < 0, а отчет (Grading report) не помогает решить проблему. Пример: в случае неправильно указанного access token система вернет -401 и информацию о том, что его нужно поправить;
- система показывает 0 и в отчете (Grading report) не указано, какие тесты не пройдены. Пример: вы отправили невалидный архив (rar вместо zip), не приложили нужные файлы (или наоборот приложили лишние - временные файлы от Mac OS и т.п.), рекомендуется проверить содержимое архива в консоли:

  ```
  unzip -l your_solution.zip
  ```

Если Вы столкнулись с какой-то из них присылайте ссылку на выполненное задание (Job) в чат курса. Пример ссылки:

https://everest.distcomp.org/jobs/67893456230000abc0123def

Как правильно настроить окружение, чтобы оно совпадало с тестовым окружением?

1. Если еще не установлено, то установите conda https://docs.conda.io/projects/conda/en/latest/user-guide/install/
2. Настройте окружение для разработки на основе README.md курса https://github.com/big-data-team/python-course
3. Скачайте необходимые датасеты для выполнения задания https://github.com/big-data-team/python-course#study-datasets

6. Дополнительные задания (не на оценку)

Задание #0. Декоратор и @contextmanager

Изучите что такое contextmanager:

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

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

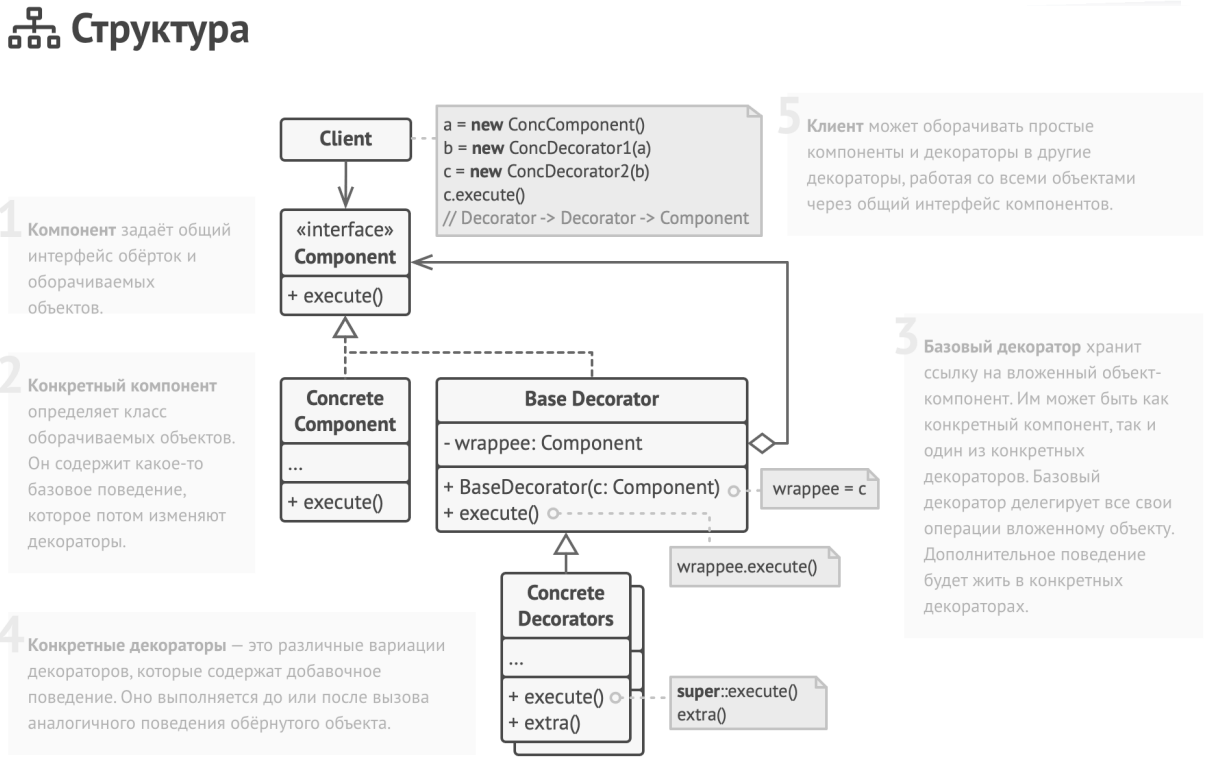
1. Вариант с созданием класса, реализующим enter/exit;
2. Вариант с написанием функции, обернутой в @contextmanager.

Задание #1 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

Убедитесь, что понимаете смысл каждой связи на диаграмме классов:



Реализуйте шаблон проектирования декоратор самостоятельно (с помощью создания классов и объектов классов), убедитесь, что функционал работает и сравните с выразительностью @ на языке Python.

Задание #2 RAII vs ContextManager

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?

Задание #3 pytest fixture взамен capsys

Для вдохновения обратите внимание на доступный ContextManager в стандартной библиотеке Python:

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

Реализуйте собственную pytest fixture взамен capsys с аналогичным интерфейсом и протестируйте реализацию.

Задание #4 ContextManager взамен @patch

Вспоминая логику работы модулей и подгрузки объектов из namespace по правилу LEGB реализуйте свой собственный ContextManager по аналогии с @patch, чтобы иметь возможность перегружать поведение произвольных функций.

Задание #5 memoization с immutable аргументами

В Python доступен инструмент для кеширования возвращаемого значения из функций (также изучите паттерн проектирования Memoization):

- https://docs.python.org/3/library/functools.html#functools.lru_cache[5]

Напишите @cache который сможет работать со словарями в качестве входных значений. Обсудите плюсы и минусы вашего подхода и реализации с коллегами в чате.

7. Полезные книги, ресурсы, дополнительные материалы

Полезные материалы для расширения кругозора:

- https://sourcemaking.com/ (En) + https://refactoring.guru/ (Ru)

Классика паттернов проектирования (GoF):

- Design Patterns: Elements of Reusable Object-Oriented Software by Gamma Erich, Helm Richard, Johnson Ralph , Vlissides John

[5] поскольку курс у нас проходит на версии Python 3.7, то функциональность языка версии 3.9 (@cache) мы здесь не рассматриваем.