**29.6 Практическая работа**

Цели практической работы

Отработать:

* создание и использование декораторов, принимающие аргументы, использование вложенных декораторов;
* декорирование классов, особенности декорирования классов и модификацию поведения внутри класса с помощью специальных функций.

Что входит в работу

* Права доступа.
* Функция обратного вызова.
* Логирование в формате.
* Синглтон.

Задача 1. Права доступа

Что нужно сделать

Перед вами стоит задача создать и поддерживать специализированный форум. Вы только приступили и сейчас работаете над действиями, которые могут совершать посетители форума. Для разных пользователей прописаны разные возможности.

Напишите декоратор check\_permission, который проверяет, есть ли у пользователя доступ к вызываемой функции, и если нет, то выдаёт исключение PermissionError.

Пример кода:

user\_permissions = ['admin']  
  
@check\_permission('admin')  
def delete\_site():  
    print('Удаляем сайт')  
  
@check\_permission('user\_1')  
def add\_comment():  
    print('Добавляем комментарий')  
  
delete\_site()  
add\_comment()

**Результат:**  
Удаляем сайт  
PermissionError: у пользователя недостаточно прав, чтобы выполнить функцию add\_comment

Что оценивается в задаче

* Результат вычислений корректен.
* Формат вывода соответствует примеру.
* Переменные, функции и собственные методы классов имеют значащие имена (не a, b, c, d).
* Классы и методы/функции имеют прописанную документацию.
* Есть аннотация типов для методов/функций и их аргументов, кроме args и kwargs. Если функция/метод ничего не возвращает, то используется None.

Задача 2. Функция обратного вызова

Что нужно сделать

При работе с сетью и веб-сервисами иногда используется функция callback, так называемая функция обратного вызова. Это функция, которая вызывается при срабатывании определённого события: переходе на страницу, получении сообщения или окончании обработки процессором. В неё можно передать функцию, чтобы она выполнилась после определённого события. Это используется, например, в HTTP-серверах в ответ на URL-запросы. Реализуйте такую функцию.

Пример функции:

@callback('//')  
def example():  
    print('Пример функции, которая возвращает ответ сервера')  
    return 'OK'  
  
Основной код:  
route = app.get('//')  
if route:  
    response = route()  
    print('Ответ:', response)  
else:  
    print('Такого пути нет')

Ожидаемый результат: пример функции, которая возвращает ответ сервера.  
Ответ: OK.

**Подсказка:** функция callback, в зависимости от условия, может быть вызвана следующим действием или просто так.

Что оценивается в задаче

* Результат вычислений корректен.
* Формат вывода соответствует примеру.
* Переменные, функции и собственные методы классов имеют значащие имена (не a, b, c, d).
* Классы и методы (функции) имеют прописанную документацию.
* Есть аннотация типов для методов (функций) и их аргументов, кроме args и kwargs. Если функция или метод ничего не возвращает, то используется None.

Задача 3. Логирование в формате

Что нужно сделать

Реализуйте декоратор, который будет логировать все методы декорируемого класса (кроме магических методов) и в который можно передавать формат вывода даты и времени логирования.

Пример кода, передаётся формат «месяц день год — часы:минуты:секунды»:

@log\_methods("b d Y — H:M:S")  
class A:  
    def test\_sum\_1(self) -> int:  
        print('test sum 1')  
        number = 100  
        result = 0  
        for \_ in range(number + 1):  
            result += sum([i\_num \*\* 2 for i\_num in range(10000)])  
  
        return result  
  
@log\_methods("b d Y - H:M:S")  
class B(A):  
    def test\_sum\_1(self):  
        super().test\_sum\_1()  
        print("Наследник test sum 1")  
  
  
    def test\_sum\_2(self):  
        print("test sum 2")  
        number = 200  
        result = 0  
        for \_ in range(number + 1):  
            result += sum([i\_num \*\* 2 for i\_num in range(10000)])  
  
        return result  
  
my\_obj = B()  
my\_obj.test\_sum\_1()  
my\_obj.test\_sum\_2()

**Результат:**  
Запускается 'B.test\_sum\_1'. Дата и время запуска: Apr 23 2021 — 21:50:37.   
Запускается 'A.test\_sum\_1'. Дата и время запуска: Apr 23 2021 — 21:50:37.   
Тут метод test\_sum\_1.  
Завершение 'A.test\_sum\_1', время работы = 0,187 s.   
Тут метод test\_sum\_1 у наследника.  
Завершение 'B.test\_sum\_1', время работы = 0,187 s.   
Запускается 'B.test\_sum\_2'. Дата и время запуска: Apr 23 2021 — 21:50:37.   
Тут метод test\_sum\_2 у наследника.  
Завершение 'B.test\_sum\_2', время работы = 0,370 s.

**Совет:** внимательно пересмотрите видео 29.4, если сталкиваетесь с трудностями в этой задаче.

Что оценивается в задаче

* Результат вычислений корректен.
* Модели реализованы в стиле ООП, основной функционал описан в методах классов и в отдельных функциях.
* При написании классов соблюдаются основные принципы ООП: инкапсуляция, наследование и полиморфизм.
  + Для получения и установки значений у приватных атрибутов используются сеттеры и геттеры с соответствующими декораторами.
  + Для создания нового класса на основе уже существующего используется наследование.
  + Для статических и классовых методов используется декоратор classmethod.
* Формат вывода соответствует примеру.
* Переменные, функции и собственные методы классов имеют значащие имена (не a, b, c, d).
* Классы и методы/функции имеют прописанную документацию.
* Есть аннотация типов для методов/функций и их аргументов, кроме args и kwargs. Если функция/метод ничего не возвращает, то используется None.

Задача 4. Синглтон

Что нужно сделать

Синглтон — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к этому экземпляру глобальную точку доступа. Синглтонами мы уже пользовались, к ним относятся, например, None, True и False. Благодаря тому, что None — синглтон, можно использовать оператор is: он возвращает True только для объектов, представляющих одну и ту же сущность.

Реализуйте декоратор singleton, который превращает класс в одноэлементный. При множественной инициализации объекта этого класса будет сохранён только первый инстанс, а все остальные попытки создания будут возвращать первый экземпляр.

**Пример кода:**

@singleton  
class Example:  
    pass  
  
my\_obj = Example()  
my\_another\_obj = Example()  
  
print(id(my\_obj))  
print(id(my\_another\_obj))  
  
print(my\_obj is my\_another\_obj)

**Результат:**  
1986890616688  
1986890616688  
True

Что оценивается в задаче

* Результат вычислений корректен.
* Формат вывода соответствует примеру.
* Переменные, функции и собственные методы классов имеют значащие имена (не a, b, c, d).
* Классы и методы/функции имеют прописанную документацию.
* Есть аннотация типов для методов/функций и их аргументов (кроме args и kwargs). Если функция или метод ничего не возвращает, то используется None.

Что оценивается в практической работе

* Практическая работа сдана через GitLab.
* Структура папок и файлов репозитория соответствует репозиторию python\_basic.
* Все задачи выполнены в соответствующих папках и файлах main.py.
* Описания коммитов осмысленны и понятны: 111, done, «я сделалъ» — неверно; added m15 homework, 14.3 fix: variables naming — верно.
* Использованы именованные индексы, не просто i (подробнее об этом — в видео 7.2).
* Использованы правильные числа, без дополнительных действий со стороны пользователя, без +1 (подробнее об этом в видео 7.4).
* Правильно оформлен input, без пустого приветствия для ввода (подробнее об этом в видео 2.3).
* Переменные и функции имеют значащие имена, не только a, b, c, d (подробнее об этом в видео 2.3).
* Есть пробелы после запятых и при бинарных операциях.
* Нет пробелов после имён функций и перед скобками: print (), input () — неверно; print() — верно.
* Правильно оформлены блоки if-elif-else, циклы и функции; отступы одинаковы во всех блоках одного уровня.
* Все входные и выходные файлы называются так, как указано в заданиях.
* Работа с файлами осуществляется с помощью контекстного менеджера with.
* Для обработки исключений используются блоки try-except.
* Модели реализованы в стиле ООП, основной функционал описан в методах классов и в отдельных функциях.
* При написании классов соблюдаются основные принципы ООП: инкапсуляция, наследование и полиморфизм.
  + Для получения и установки значений у приватных атрибутов используются сеттеры и геттеры с соответствующими декораторами.
  + Для создания нового класса на основе уже существующего используется наследование.
  + Для статических и классовых методов используется декоратор classmethod.
* Если классы вынесены в отдельный модуль, то импортируются определённые классы (запись вида from garden import \* считается плохим тоном).
* Классы и методы/функции имеют прописанную документацию, хотя бы минимальную).
* Есть аннотация типов для методов/функций и их аргументов, кроме args и kwargs. Если функция/метод ничего не возвращает, то используется None.
* Во всех декораторах используется functools.wraps.

Рекомендации

* Арифметические операции [PEP 8](https://docs.python.org/3.7/reference/expressions.html#operator-precedence) остаются в приоритете. Необходимо вводить and, or.
* [Руководство по стилю Python PEP 8](https://www.python.org/dev/peps/pep-0008/) (на английском)
* [Руководство по стилю Python PEP 8](https://pythonworld.ru/osnovy/pep-8-rukovodstvo-po-napisaniyu-koda-na-python.html) (на русском)
* [Список встроенных функций](https://docs.python.org/3.7/library/functions.html)

Как отправить работу на проверку

Чтобы выполнить практическую работу, обновите репозиторий python\_basic на своём компьютере при помощи IDE PyCharm. Задачи находятся в папке Module29.

Сдайте практические работы этого модуля через систему контроля версий Git сервиса Skillbox GitLab. В материалах с практической работой напишите «Сделано» и прикрепите ссылку на репозиторий. Ссылки на реплит оставлять не нужно.