Декораторы

Задача 1: Суммы прогрессий

Необходимо реализовать функционал для подсчета суммы первых n + 1 - членов арифметической и геометрической прогрессии с возможностями настройки шага и значения первого члена. n соответствует числу вызовов функции по подсчету суммы.

Предполагаемые сценарии использования:

```
sum_arithmetic = make_arithmetic_progression_sum(first_member=2,
step=0.5)
sum_geometric = make_geometric_progression_sum(first_member=1,
step=0.5)

print(sum_arithmetic())
print(sum_arithmetic())
print('')
print(sum_geometric())
print(sum_geometric())
```

Вывод:

```
4.5
7.5

1.5
1.75
```

```
def make_arithmetic_progression_sum(first_member: float, step: float):
    # ваш код
    pass


def make_geometric_progression_sum(first_member: float, step: float):
    # ваш код
    pass
```

Задача 2: Среднее

Предположим, что мы занимаемся инвистициями и у нас есть некоторый портфель акций. Каждый день наш портфель приносит нам некоторый доход или убыток. Мы задались целью: каждый день фиксированного периода определять средний доход (или убыток),

который мы получаем. С этой целью мы реализовали функцию get_avg(), принимающую на вход значение заработка на сегодняшний день. Наша функция вычисляет среднее в течении определнного фиксированного периода, скажем, 30 дней, после чего обнуляется и начинает вычислять среднее заново, но уже по новым значениям.

Также у нас есть друзья инвесторы, которые оценили разработанный нами функционал и хотели бы заполучить свой экземпляр функции get_avg, для подсчета своего дохода в течении интересующего их промежутка времени.

Ваша задача: реализовать функционал, для получения произвольного числа независимых функций get_avg(). В момент создания функции сообщается длительность периода расчета среднего, по достижении которого среднее начинает расчитываться заново, а также наш начальный доход. При каждом вызове функции передается число - заработок в текущий день.

Предполагаемые сценарии использования:

```
get_avg1 = make_averager(accumulation_period=2)
print(get_avg1(78))
print(get_avg1(-17))
print(get_avg1(52))
```

Вывод:

```
78.0
30.5
52.0

# ваш код
```

Задача 3: Сбор статистик

Предположим, что мы работаем в отделе аналитики некоторой компании. В компании также существуют другие отделы, которые разрабатывают некоторые функции для осуществления сложных вычислений. Также в нашей компании существует отдел планирования, который следит за исполнением сроков реализации той или иной функции, и в случае, если разработка затягивается, начинает торопить разработчиков. В таком случае разработчики пишут медленный код на скорую руку, что расстраивает заказчиков.

Наша задача, как аналитиков, собрать статистику по проблемным функциям. Нас интересует количество вызовов функции, а также среднее время выполнения функции. Все статистики собираются в отдельную базу данных - специальный единый словарь. Более того, статистика должна собираться не для всех функций, а только для функций, зарегестрированных в базе данных. Затем эта информация будет передана начальству, чтобы в скорейшее время заняться переписанием долгих и популярных функций.

Ваша задача реализовать функционал для регистрации функций в БД и сбора статистик.

```
functions_register = {}

# ваш код
```

Задача 4: наивный LRU-кэш

Представим, что вы являетесь сотрудником департамента оптимизаций некоторой компании, занимающейся разработкой ПО для научных вычислений. Раздел аналитиков предоставил вам исследование, согласно которому значительная часть функций в вашей компании работает очень медленно и должна быть оптимизирована. Более того, согласно исследованию, вызовы этих функций в основном осуществляются с ограниченным множеством аргументов. Для оптимизации этих функций вы решили использовать LRU-кэш:

- вы заранее фиксируете размер кэша - памяти, выделенной для хранения результатов вычислений функции;
- в кэше хранится следующая информация: аргументы вызова - результат;
- помимо этого для каждой пары хранится время их последнего вызова;
- в случае достижения объема кэша установленной границы удаляются значения, чьи времена последних вызовов являются самыми старыми;

Для применения данного подхода ко всем проблемным функциям, не переписывая сами функции, вы решили реализовать параметрический декоратор, т.к. для разных функций требуется разный размер кеша.

Ваша задача: реализовать описанный декоратор.

```
# ваш код
```