Python. Семинар 4

Преподаватели: Дмитрий Косицин, Светлана Боярович и Анастасия Мицкевич

Задание 1. (1 балл). Реализуйте класс **MidSkipQueue**, представляющий собой очередь, в которой хранятся только первые k и последние k добавленных элементов. В очереди должно быть реализовано следующее:

- конструктор, принимающий первым аргументом параметр k (проверьте, что он имеет допустимое значение), а вторым *опциональным* аргументом – iterable элементов, на основе которых нужно построить очередь.
- метод, преобразующий очередь в строку (переопределите "магический"метод *__str__* и используйте модуль *pprint*)
- оператор, позволяющий сравнивать на равенство 2 очереди
- "магический"метод, возвращающий длину очереди
- "магический"метод, позволяющий обратиться к элементу по индексу (используйте *assert* для проверки корректности индекса), либо взять слайс элементов
- метод *index*, возвращающий индекс элемента в очереди или -1, если такого элемента нет
- "магический"метод, позволяющий проверить, содержится ли некоторый элемент в очереди
- метод *append*, который в качестве агрумента принимает один и более (переменное число) объектов и добавляет все элементы в очередь.
- оператор сложения с iterable элементов

Обратите внимание, что удаление из очереди реализовывать не требуется.

От этого класса унаследуйте класс **MidSkipPriorityQueue**, в котором при добавлении элементов в очередь будет учитываться их значение так, что будут храниться не более k наименьших и k наибольших добавленных элементов. В начале очереди храните наименьший элемент. Предполагайте, что добавляемые элементы реализуют все необходимые операторы сравнения.

Классы сохраните в файле *mid_skip_queue.py*.

Примеры

```
q = MidSkipQueue(1)
q.append(-1)  # q: [-1]
q += (-2, -3)  # q: [-1, -3] - the first and the last remain
q.append(4)  # q: [-1, 4] - the last item has been replaced

q = MidSkipPriorityQueue(1)
q.append(-1)  # q: [-1]
q += (-2, -3)  # q: [-3, -1] - the smallest and the largest items
q.append(4)  # q: [-3, 4] - the largest item is replaced
q.append(-5)  # q: [-5, 4] - the smallest item is replaced
```

Задание 2. (0.3 балла). Реализуйте декоратор **memorize**, который будет сохранять результат выполнения некоторой функции и возвращать его, если функция была опять вызвана с этими же параметрами. Предполагайте, что значения всех параметров хешируемы. Предполагайте также, что количество комбинаций различных значений агрументов разумно и не приведет к ошибкам выдения памяти.

Декоратор сохраните в файле *decorators.py*.

Замечание. В Python 3 есть декоратор *functools.lru_cache*. Использовать его для решения задания не разрешается.

Задание 3. (0.3 балла). Реализуйте декоратор **profile**, который при вызове функции подсчитывает время выполния этой функции и выводит его на экран. Рассмотрите стандартный модуль *timeit* для измерения времени выполнения.

Декоратор сохраните в файле *decorators.py*.

Задание 4. (0.3 балла). Реализуйте декоратор **convolve**, принимающий своим аргументом некоторое натуральное число k и выполняющий свертку функции f соответствующее число раз. Проверьте коррекность параметра k.

Декоратор сохраните в файле *decorators.py*.

Пример

```
@convolve(3)
def f(some_argument):
    return 2 * some_argument

x = 1
assert f(x) == 2 * (2 * (2 * x))  # f(f(f(x)))
```

Задание 5. (1.1 балла). Реализуйте класс **DependencyHelper**, позволяющий эффективно проверить наличие циклических зависимостей. Реализуйте следующие методы:

- метод *add*, принимающий на вход 2 параметра – пару зависимых обьектов, где второй зависит от первого
- оператор сложения с кортежем (парой) зависимых элементов
- метод *remove* и соответствующий ему оператор вычитания
- метод копирования *copy*, возвращающий точную независимую копию данного объекта
- метод *get_dependent*, принимающий в качестве аргумента некоторый элемент и возвращающий последовательность непосредственно зависимых от него элементов
- метод *has_dependencies*, проверяющий наличие циклической зависимости между обьектами
- оператор преобразования к *bool*, возвращающий *True*, если зависимостей нет

Все добавляемые обьекты предполагаются хешируемыми. Использовать готовые алгоритмы не допускается.

Классы сохраните в файле *dependency_helper.py*.

Пример

```
dependency_helper = DependencyHelper()
dependency_helper.add(1, 2)
dependency_helper += (2, 1)
assert not dependency_helper  # helper must find out dependend items
```

Бонусное задание (0.5 балла).

От класса **DependencyHelper** унаследуйте класс **PriorityHelper**. В нем реализуйте метод *enumerate_priorities*, возвращающий словарь, содержащий пары обьектов и их приоритетов.

Приоритеты должны быть расставлены так, что объекты с меньшим приоритетом не зависят от объектов с большим приоритетом. Значения приоритетов могут совпадать. Количество различных приоритетов постарайтесь сделать минимальным.

Использовать готовые алгоритмы не допускается.

Общие замечания ко всем заданиям

За тесты к классам и декораторам (а также как и всегда за хороший стиль кода) будут начисляться бонусные баллы.