ВМК

Задание 4. Advanced Python. Итераторы и декораторы.

Практикум 317 группы 2017-2018, осенний семестр

Начало выполнения задания: 27 октября 2017 года.
Срок сдачи: **3 ноября 2017 года, 23:59.**

Решение каждой задачи должно быть описано в модуле `task_<номер задачи>.py`.

В систему anytask необходимо сдать `zip` архив, содержащий решения задач, называющийся `contest4_<фамилия студента>_<имя студента>.zip`.

Задачи основной части дополнительно сдаются в систему «Яндекс-контест», где проверяются с помощью автоматических тестов. Ссылку на страницу соревнования можно найти на странице курса. За каждую из задач основной части можно получить либо 5 баллов (вердикт 'OK' контеста), либо 0. Задание считается сданным, если хотя бы одна из задач получила вердикт 'OK'.

1. Написать собственный класс `RleSequence`, реализующий кодирование длин серий (Run-length encoding).

   Класс должен включать/перегружать следующие методы:

   - `__init__(self, input_sequence)` — конструктор класса. По входному вектору `input_sequence` строится два вектора одинаковой длины. Первый содержит числа, а второй — сколько раз их нужно повторить. Для реализации рекомендуется использовать функцию `encode_rle(x)` из второго домашнего задания.
     - `input_sequence` — одномерный `numpy.array`

   Экземпляры класса должны поддерживать протокол итераций, причём порядок элементов, выдаваемый в процессе итерирования по экземпляру `RleSequence`, должен совпадать с порядком элементов в исходном `input_sequence`, который подавался в конструктор класса. Экземпляры должны поддерживать простое индексирование (положительные и отрицательные целые индексы) и взятие срезов с положительным шагом (третий параметр). Экземпляры должны поддерживать проверку на вхождение (`in` и `not in`).

   Пример правильно работающего класса:

   ```
   >>> rle_seq = RleSequence(np.array([1, 1, 2, 2, 3, 4, 5]))
   >>> rle_seq[4]
   3
   >>> rle_seq[1:5:2]
   array([1, 2])
   >>> rle_seq[1:-1:3]
   array([1, 3])
   >>> 5 in rle_seq
   True
   >>> my_list = []
   >>> for elem in rle_seq:
   ...     my_list.append(elem)
   >>> my_list
   [1, 1, 2, 2, 3, 4, 5]
   ```

   Замечание. Некоторые операторы и функции перегружаются автоматически при определении других. Тем не менее, часто собственая реализация может работать эффективнее чем созданная по умолчанию.

2. Написать итератор `linearize`, принимающий на вход любой итерируемый объект и линеаризующий его, т.е. раскрывающий все вложенности.

   Пример правильно работающего кода:

   ```
   >>> my_list = list()
   >>> for elem in linearize([4, 'mmp', [8, [15, 1], [[6]], [2, [3]]], range(4, 2, -1)]):
   ...     my_list.append(elem)
   >>> my_list
   [4, 'm', 'm', 'p', 8, 15, 1, 6, 2, 3, 4, 3]
   ```

Замечание. Итератор можно реализовать как генератор. В этом случае будет полезно использовать конструкцию `yield from`, о которой можно прочитать, например, здесь: ссылка

3. Напишите декоратор `@check_arguments`, который будет проверять правильность типов входных позиционных аргументов функции.

   Декоратор принимает на вход типы аргументов и декорирует функцию таким образом, что она генерирует исключение `TypeError`, если хотя бы один из аргументов имеет неверный тип. Типов может быть меньше чем аргументов, в этом случае проверяются типы только первых аргументов, для которых типы прописаны. Типов может быть больше чем аргументов, в этом случае необходимо вывести ошибку. Декоратор должен корректно обрабатывать функции с переменным числом аргументов. Декоратор не обязан корректно работать с функциями с именованными аргументами.

   Примеры правильно работающего кода:

```
>>> @check_arguments(int, int)
... def f(x, y, *args):
...     return x + y + sum(args)
...
>>> f(1, 'a')
TypeError: wrong types
>>> f(1, 2, 3)
6
```

4. Напишите декоратор `substitutive`, который позволяет вызывать функцию от неполного множества аргументов. В этом случае аргументы, которые были переданы в функцию, фиксируются для использования в дальнейшем, а результатом функции будет функция от оставшихся аргументов. Декоратор должен работать корректно при любых вызовах функции с позиционными аргументами.

```
>>> @substitutive
... def f(x, y, z):
...     return x + y + z
...
>>> f(1, 2)
<function __main__.f>
>>> f(1, 2)(3)
6
```

   Замечание 1. У всех декорированных функций должны сохраняться атрибуты исходных функций.

   Замечание 2. Количество аргументов функции `f` можно узнать, вызвав `f.__code__.co_argcount`.

5. **Бонусное задание — 2 балла.** Добавьте в `RleSequence` поддержку срезов с отрицательным шагом.

   Пример правильно работающего кода:

```
>>> rle_seq = RleSequence(np.array([1, 1, 2, 2, 3, 4, 5]))
>>> rle_seq[-1:2:-1]
array([5, 4, 3, 2])
```

6. **Бонусное задание — 2 балла.** Напишите декоратор `@memoized`, кеширующий результаты последних `max_count` вычислений функции. При переполнении кеша из него выкидывается то, что использовалось давнее всего. Декоратор может не принимать аргументов, в этом случае размер кеша неограничен.

   Пример использования декоратора:

```
>>> @memoized
... def ackermann(m, n):
...     # код функции
>>> @memoized(max_count=1000)
... def ackermann(m, n):
...     # код функции
```