Практика 5. Изменяемость, nonlocal и итераторы

Изменяемость

Представь, что ты заказываешь пиццу в твоей любимой кафешке, тебе нужна пицца с грибами и сыром. Такой заказ можно представить так:

```
>>> pizza1 = ['сыр', 'грибы']
```

Через пять минут ты понимаешь, что нужно добавить в заказ и лучок. Стало быть нужно создать новый список.

```
>>> pizza2 = pizza1 + ['лучок']
>>> pizza2
['сыр', 'грибы', 'лучок']
>>> pizza1 # исходный список не изменился
['сыр', 'грибы']
```

Но ведь это глупо — делать новую пиццу `pizza2`, когда можно просто взять `pizza1` и добавить сверху лук.

Python позволяет *изменять* некоторые уже готовые объекты (например, списки и словари). Изменяемость означает, что содержимое объекта может измениться. То есть вместо создания нового объекта `pizza2`, достаточно использовать `pizza1.append('лучок')`. В этом случае:

```
>>> pizza1.append('лучок')
>>> pizza1
['сыр', 'грибы', 'лучок']
```

Хотя списки и словари могут *изменяться*, многие объекты (например, числовые типы, таплы, строки) являются *неизменяемыми*. Это значит, что после создания невозможно изменить их содержимое.

Методы списка — это функции, которые связаны с некоторым конкретным списком. Вызов методов предполагает использование *точечной нотации*, в форме `lst.method()`. Ниже представлены наиболее востребованные методы списков:

`lst.append(el)`

изменяет `lst`, добавляя `el` в конец.

`lst.insert(i, el)`

изменяет `lst`, добавляя `el` у индекса `i`.

`lst.sort(el)`

изменяет `lst`, сортируя элементы списка.

`lst.remove(el)`

изменяет `lst`, удаляя первый найденый элемент, равный `el`. Если указаный элемент отсутствует в списке, то возникает ошибка.

`lst.index(el)`

возвращает индекс первого найденого элемента, равного `el`. Если элемент `el` отсутствует в списке, то возникает ошибка. Этот метод не изменяет список `lst`.

Ни один изменяющий метод списка не возвращает новый список — вся работа ведётся непосредственно с указаным списком `lst`.

Вопрос 1

Что выведет Python? Нужно представить не только вывод, но и диаграмму Контеи□нер-и-Указатель.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = lst1
>>> lst1 is lst2
```

Ответ

```

```

```
>>> lst2.extend([5, 6])
>>> lst1[4]
```

Ответ

```

```

```
>>> lst1.append([-1, 0, 1])
>>> -1 in lst2
```

Ответ

```

```

```
>>> lst2[5]
```

Ответ

```

```

```
>>> lst3 = lst2[:]
>>> lst3.insert(3, lst2.pop(3))
>>> len(lst1)
```

Ответ

```

```

```
>>> lst1[4] is lst3[6]
```

Ответ

```

```

```
lst3[lst2[4][1]]
```

Ответ

```

```

```
lst1[:3] is lst2[:3]
```

Ответ

```

```

```
>>> lst1[:3] == lst3[:3]
```

Ответ

```

```

Вопрос 2

Напиши функцию, которая принимает значение `x`, значение `el` и список `lst`. Функция должна добавить в конец списка столько `el`, сколько `x` встречается в исходном списке. **Нужно модифицировать исходный список, а не создавать новый.**

```
def add_this_many(x, el, lst):
    """ Добавляет в конец списка столько el, сколько x встречается в исходном списке
lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
```

Ответ

```

```

Nonlocal

Ключевое слово `nonlocal` может использоваться для изменения переменной в родительском фрейме, за пределами текущего (конечно если родительский фрейм не является глобальным). Например, `make_step` использует `nonlocal` для изменения `num`:

```
def stepper(num):
    def step():
        nonlocal num        # указывает, что num нелокальное имя
        num = num + 1       # изменяет num в родительском фрейме
        return num
    return step

>>> step1 = stepper(10)
>>> step1()                 # изменяет и возвращает num
11
>>> step1()                 # num остается прежним в разных вызовах step1
12
>>> step2 = stepper(10)     # каждая созданная функция step хранит состояние
>>> step2()
11
```

Пример нам как бы говорит: «`nonlocal` полезен для хранения состояний внутри функции между вызовами».

Однако есть две важные особенности `nonlocal`:

- **Глобальные имена** не могут быть изменены таким образом.
- **Имена в текущем фрейме** не могут быть перекрыты через `nonlocal`. Это означает, что нельзя использовать и локальное, и нелокальное имя одновременно в одном фрейме.

Из-за того, что `nonlocal` позволяет изменять состояние родительского фрейма, функции с `nonlocal` называют *мутабельными*.

Вопрос 3

Нарисуй диаграмму окружения для следующего кода.

```
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step
s = stepper(3)
s()
s()
```

Ответ

```

```

Вопрос 4

Функция ниже имитирует эпическую битву между Рей и Кайло Реном в ванной, населённой резиновыми уточками. Заполни тело `ducky`, чтобы все доктесты прошли.

```
def bathtub(n):
    """
    >>> annihilator = bathtub(500) # Пробуждение силы...
    >>> kylo_ren = annihilator(10)
    >>> kylo_ren()
    490 резиновых уточек осталось
    >>> rey = annihilator(-20)
    >>> rey()
    510 резиновых уточек осталось
    >>> kylo_ren()
    500 резиновых уточек осталось
    """
    def ducky_annihilator(rate):
        def ducky():
```

Ответ

```
```

```
        return ducky
    return ducky_annihilator
```

Итераторы и генераторы

Вопрос 5

Что напечатает Python? Если произойдёт исключение `StopIteration` — так и надо написать. Если произойдёт другая ошибка, достаточно написать `Ошибка`.

```
>>> lst = [6, 1, "a"]
>>> next(lst)
```

Ответ

```

```

```
>>> lst_iter = iter(lst)
>>> next(lst_iter)
```

Ответ

```

```

```
>>> next(lst_iter)
```

Ответ

```

```

```
>>> next(iter(lst))
```

Ответ

```

```

```
>>> [x for x in lst_iter]
```

Ответ

Вопрос 6

Что напечатает Python? Если произойдёт исключение `StopIteration` — так и надо написать. Если произойдёт другая ошибка, достаточно написать `Error`.

```
>>> def weird_gen(x):
...     if x % 2 == 0:
...         yield x * 2
...     else:
...         yield x
...         yield from weird_gen(x - 1)
>>> next(weird_gen(2))
```

Ответ

```
>>> list(weird_gen(3))
```

Ответ

```
>>> def greeter(x):
...     while x % 2 != 0:
...         print('привет!')
...         yield x
...         print('пока!')
>>> greeter(5)
```

Ответ

```
>>> gen = greeter(5)
>>> next(gen)
```

Ответ

```

```

```
>>> next(gen)
```

Ответ

```

```

Вопрос 7

Напиши функцию-генератор `gen_all_items`, которая принимает список итераторов и выдаёт по очереди все их элементы.

```
def gen_all_items(lst):
    """
    >>> nums = [[1, 2], [3, 4], [[5, 6]]]
    >>> num_iters = [iter(l) for l in nums]
    >>> list(gen_all_items(num_iters))
    [1, 2, 3, 4, [5, 6]]
    """
```

Ответ

```

```