Python (BSU FAMCS Fall'19)

Seminar 3

Advisor: Dzmitryi Kasitsyn

Task 1. (0.5 points). Implement a function `transpose` to transpose an *iterable* of nested *iterable*s. All nested *iterable*s are supposed to have the same length. In other words you're asked to transpose a two-dimensional rectangular array.

Consider to use *built-in* functions and functions from **itertools** module. Don't use any loops and do not explicitly convert the given *iterable* to a `list` or to a `tuple` implementing your solution.

Save your function into a *iter_helpers.py* file.

Example

```
expected = [[1, 2], [-1, 3]]
actual = transpose([[1, -1], [2, 3]])
assert expected == list(map(list, actual))
```

Task 2. (1 point). Implement a function `scalar_product` to calculate a scalar product of two *iterable*s that can contain items of type `int`, `float`, or `str`.

String items can contain:

- an integer representation (binary, octal, decimal, hexadecimal with corresponding prefixes) – use *built-in* `int` function to convert them to a number,
- some letters that cannot be interpreted as numbers – assume the result of the whole expression is `None` in that case.

Consider to use *built-in* functions and functions from **itertools** module. Don't use any loops implementing your solution.

Save your function into a *iter_helpers.py* file.

Example

```
expected = 1
actual = scalar_product([1, '2'], [-1, 1])
assert expected == actual

actual = scalar_product([1, 'xyz'], [-1, 1])
assert actual is None
```

Task 3. (0.5 points). Implement a decorator `profile` that calculates how long a given function is being executed, prints the execution time and returns the function result.

Consider to use `timeit` module and `default_timer` function to calculate function execution time.

Save your decorator into a *utils.py* file.

Example

```
@profile
def some_function():
    return sum(range(1000))

result = some_function()  # return a value and print execution time
```

Task 4. (0.5 points). Implement a context manager `timer` to calculate code block execution time and print it onto a screen.

Save the context manager into a *utils.py* file.

Example

```
with timer():
    print(sum(range(1000)))
    # print execution time when calculation is over
```

Task 5. (1.5 points). Suppose you are given a singly linked list node class `Node` (see below) so that a signly linked list is a sequence of `Node`s that have either other `Node` or `None` as node's `next_` value. The value `None` determines the end of a list.

Nested lists have other `Node`s stored in `value` field.

You are asked to do the next:

- Add check of argument types in the constructor (`__init__` method; use `assert`);
- Implement properties to get and set `_value` and `_next` values;
- Implement a «magic» (*dunder*) method `__iter__` to iterate over a list;
- Implement a `flatten_linked_list` function to flatten your list *inplace*. First, it has to modify a given list in a way that all `_values` don't have `Node` type however the sequence of values is the same as in the original list iterating over it. Second, your function have to return the object passed into it (refer to *r3* example below).

Save your solution in a *linked_list.py* file.

Example

```
class Node(object):
    def __init__(self, value, next_=None):
        self._value = value
        self._next = next_


r1 = Node(1)  # 1 -> None - just one node

r2 = Node(7, Node(2, Node(9)))  # 7 -> 2 -> 9 -> None

# 3 -> (19 -> 25 -> None) -> 12 -> None
r3 = Node(3, Node(Node(19, Node(25)), Node(12)))
r3_flattenned = flatten_linked_list(r3)  # 3 -> 19 -> 25 -> 12 -> None
r3_expected_flattenned_collection = [3, 19, 25, 12]
assert r3_expected_flattenned_collection == list(r3_flattenned)
```