Homework 4: Iterators and Generators

hw04.zip (hw04.zip)

Due by 11:59pm on Wednesday, July 21

Instructions

Download hw04.zip (hw04.zip). Inside the archive, you will find a file called hw04.py (hw04.py), along with a copy of the `ok` autograder.

Submission: When you are done, submit with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be scored. Check that you have successfully submitted your code on okpy.org (https://okpy.org/). See Lab 0 (/~cs61a/su21/lab/lab00#submitting-the-assignment) for more instructions on submitting assignments.

Using Ok: If you have any questions about using Ok, please refer to this guide. (/~cs61a/su21/articles/using-ok)

Readings: You might find the following references useful:

- Section 4.2 (http://composingprograms.com/pages/42-implicit-sequences.html)

Grading: Homework is graded based on correctness. Each incorrect problem will decrease the total score by one point. There is a homework recovery policy as stated in the syllabus. **This homework is out of 3 points.**

Hint Video

Hint Video

Required questions

Q1: Repeated

Implement a function (not a generator function) that returns the first value in the iterator `t` that appears `k` times in a row.

As described in lecture, iterators can provide values using either the `next(t)` function or with a for-loop. Do not worry about cases where the function reaches the end of the iterator without finding a suitable value, all lists passed in for the tests will have a value that should be returned. If you are receiving an error where the iterator has completed then the program is not identifying the correct value.

Iterate through the items such that if the same iterator is passed into `repeated` twice, it continues in the second call at the point it left off in the first. An example of this behavior is shown in the doctests.

```
def repeated(t, k):
    """Return the first value in iterator T that appears K times in a row. Iterate t
    if the same iterator is passed into repeated twice, it continues in the second c
    in the first.

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s, 2)
    9
    >>> s2 = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s2, 3)
    8
    >>> s = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> repeated(s, 3)
    2
    >>> repeated(s, 3)
    5
    >>> s2 = iter([4, 1, 6, 6, 7, 7, 8, 8, 2, 2, 2, 5])
    >>> repeated(s2, 3)
    2
    """
    assert k > 1
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q repeated
```

Q2: Generate Permutations

Given a sequence of unique elements, a *permutation* of the sequence is a list containing the elements of the sequence in some arbitrary order. For example, `[2, 1, 3]`, `[1, 3, 2]`, and `[3, 2, 1]` are some of the permutations of the sequence `[1, 2, 3]`.

Implement `permutations`, a generator function that takes in a sequence `seq` and returns a generator that yields all permutations of `seq`.

Permutations may be yielded in any order. Note that the doctests test whether you are yielding all possible permutations, but not in any particular order. The built-in `sorted` function takes in an iterable object and returns a list containing the elements of the iterable in non-decreasing order.

> *Hint:* If you had the permutations of all the elements in `seq` not including the first element, how could you use that to generate the permutations of the full `seq`?
>
> *Hint:* If you're having trouble getting started, see the hints video for this question for tips on how to approach this question.

```
def permutations(seq):
    """Generates all permutations of the given sequence. Each permutation is a
    list of the elements in SEQ in a different order. The permutations may be
    yielded in any order.

    >>> perms = permutations([100])
    >>> type(perms)
    <class 'generator'>
    >>> next(perms)
    [100]
    >>> try: #this piece of code prints "No more permutations!" if calling next woul
    ...     next(perms)
    ... except StopIteration:
    ...     print('No more permutations!')
    No more permutations!
    >>> sorted(permutations([1, 2, 3])) # Returns a sorted list containing elements
    [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
    >>> sorted(permutations((10, 20, 30)))
    [[10, 20, 30], [10, 30, 20], [20, 10, 30], [20, 30, 10], [30, 10, 20], [30, 20,
    >>> sorted(permutations("ab"))
    [['a', 'b'], ['b', 'a']]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q permutations
```

Q3: Generators generator

Write the generator function `make_generators_generator`, which takes a zero-argument generator function `g` and returns a generator that yields generators. The behavior of each of these generators is as follows:

Since `g` is a generator function, calling `g()` will return a generator object that yields elements when `next` is called on it. For each element `e` that is yielded when `next` is called on the generator object returned by `g()`, your generator should yield a corresponding generator function that generates the first values, up until `e`, that are yielded by the generator returned by calling `g()`. See the doctests for examples.

```
def make_generators_generator(g):
    """Generates all the "sub"-generators of the generator returned by
    the generator function g.

    >>> def every_m_ints_to(n, m):
    ...     i = 0
    ...     while (i <= n):
    ...         yield i
    ...         i += m
    ...
    >>> def every_3_ints_to_10():
    ...     for item in every_m_ints_to(10, 3):
    ...         yield item
    ...
    >>> for gen in make_generators_generator(every_3_ints_to_10):
    ...     print("Next Generator:")
    ...     for item in gen:
    ...         print(item)
    ...
    Next Generator:
    0
    Next Generator:
    0
    3
    Next Generator:
    0
    3
    6
    Next Generator:
    0
    3
    6
    9
    """
    def gener(x):
        for e in ___________:
            ______________________________
            if ___________________________:
                ______________________________
    for e in ___________:
        ______________________________
```

Use Ok to test your code:

```
python3 ok -q make_generators_generator
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Just for Fun Question

Q4: Remainder Generator

Like functions, generators can also be *higher-order*. For this problem, we will be writing `remainders_generator`, which yields a series of generator objects. `remainders_generator` takes in an integer `m`, and yields `m` different generators. The first generator is a generator of multiples of `m`, i.e. numbers where the remainder is 0. The second is a generator of natural numbers with remainder 1 when divided by `m`. The last generator yields natural numbers with remainder `m - 1` when divided by `m`.

> *Hint*: To create a generator of infinite natural numbers, you can call the `naturals` function that's provided in the starter code.

> *Hint*: Consider defining an inner generator function. Each yielded generator varies only in that the elements of each generator have a particular remainder when divided by `m`. What does that tell you about the argument(s) that the inner function should take in?

```
def remainders_generator(m):
    """
    Yields m generators. The ith yielded generator yields natural numbers whose
    remainder is i when divided by m.

    >>> import types
    >>> [isinstance(gen, types.GeneratorType) for gen in remainders_generator(5)]
    [True, True, True, True, True]
    >>> remainders_four = remainders_generator(4)
    >>> for i in range(4):
    ...     print("First 3 natural numbers with remainder {0} when divided by 4:".fo
    ...     gen = next(remainders_four)
    ...     for _ in range(3):
    ...         print(next(gen))
    First 3 natural numbers with remainder 0 when divided by 4:
    4
    8
    12
    First 3 natural numbers with remainder 1 when divided by 4:
    1
    5
    9
    First 3 natural numbers with remainder 2 when divided by 4:
    2
    6
    10
    First 3 natural numbers with remainder 3 when divided by 4:
    3
    7
    11
    """
    "*** YOUR CODE HERE ***"
```

Note that if you have implemented this correctly, each of the generators yielded by `remainder_generator` will be *infinite* - you can keep calling `next` on them forever without running into a `StopIteration` exception.

Use Ok to test your code:

```
python3 ok -q remainders_generator
```