Lazy Looping in Python - Answers

Trey Hunner

Apr 27, 2020

CONTENTS

CHAPTER

ONE

LIST COMPREHENSION EXERCISES

These exercises are all in the `lists.py` file in the `exercises` directory. Edit the file to add the functions or fix the error(s) in the existing function(s).

To run the test: from the `exercises` folder, type `python test.py <function_name>`, like this:

```
$ python test.py get_vowel_names
```

Tip: Start with a `for` loop and then copy-paste your way into a list comprehension.

1.1 Starting with a vowel

Edit the function `get_vowel_names` so that it accepts a list of names and returns a new list containing all names that start with a vowel. It should work like this:

```
>>> names = ["Alice", "Bob", "Christy", "Jules"]
>>> get_vowel_names(names)
['Alice']
>>> names = ["Scott", "Arthur", "Jan", "elizabeth"]
>>> get_vowel_names(names)
['Arthur', 'elizabeth']
```

Answers

```
def get_vowel_names(names):
    """Return a list containing all names given that start with a vowel."""
    return [
        name
        for name in names
        if name[0].lower() in "aeiou"
    ]
```

1.2 Flatten a Matrix

Edit the function `flatten`, that will take a matrix (a list of lists) and return a flattened version of the matrix.

```
>>> from loops import flatten
>>> matrix = [[row * 3 + incr for incr in range(1, 4)] for row in range(4)]
>>> matrix
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
>>> flatten(matrix)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Answers

```
def flatten(matrix):
    """Return a flattened version of the given 2-D matrix (list-of-lists)."""
    return [
        item
        for row in matrix
        for item in row
    ]
```

1.3 Matrix From String

Edit the function `matrix_from_string` so it accepts a string and returns a list of lists of integers (found in the string).

```
>>> matrix_from_string("1 2\n10 20")
[[1, 2], [10, 20]]
```

Answers

```
def matrix_from_string(string):
    """Convert rows of numbers to list of lists."""
    return [
        [int(x) for x in row.split()]
        for row in string.splitlines()
    ]
```

1.4 Power List By Index

Edit the function `power_list` so that it accepts a list of numbers and returns a new list that contains each number raised to the `i`-th power where `i` is the index of that number in the given list. For example:

```
>>> from lists import power_list
>>> power_list([3, 2, 5])
[1, 2, 25]
>>> numbers = [78, 700, 82, 16, 2, 3, 9.5]
>>> power_list(numbers)
[1, 700, 6724, 4096, 16, 243, 735091.890625]
```

Answers

```
def power_list(numbers):
    """Return a list that contains each number raised to the i-th power."""
    return [
        n ** i
        for i, n in enumerate(numbers)
    ]
```

1.5 Matrix Addition

Edit the function `matrix_add` so it accepts two matrices (lists of lists of numbers) and returns one matrix that includes each corresponding number in the two lists added together.

You should assume the lists of lists provided will always be the same size/shape.

```
>>> from ranges import matrix_add
>>> m1 = [[1, 2], [3, 4]]
>>> m2 = [[5, 6], [7, 8]]
>>> matrix_add(m1, m2)
[[6, 8], [10, 12]]
>>> m1 = [[1, 2, 3], [0, 4, 2]]
>>> m2 = [[4, 2, 1], [5, 7, 0]]
>>> matrix_add(m1, m2)
[[5, 4, 4], [5, 11, 2]]
```

Answers

```
def matrix_add(matrix1, matrix2):
    """Add corresponding numbers in given 2-D matrices."""
    return [
        [n + m for n, m in zip(row1, row2)]
        for row1, row2 in zip(matrix1, matrix2)
    ]
```

1.6 Identity Matrix

Edit the function `identity` so that it takes as input a number `size` for the size of the matrix and returns an identity matrix of `size` x `size` elements. It should work like this:

An identity matrix is a square matrix with ones on the main diagonal and zeros elsewhere. A 3 by 3 identity matrix looks like:

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

```
>>> from lists import identity
>>> identity(3)
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>> identity(4)
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
>>> identity(2)
[[1, 0], [0, 1]]
```

Answers

With a helper function for readability:

```
def same_value(x, y):
    """Returns 1 if x == y, else returns 0."""
    return 1 if x == y else 0

def identity(size):
    """Return an identity matrix of size x size."""
    return [
        [same_value(row, col) for row in range(size)]
        for col in range(size)
    ]
```

Or, we could take advantage of the internal values of `True` and `False`. This is a legal and reasonable thing to do in Python!

```
def identity(size):
    """Return an identity matrix of size x size."""
    return [
        [int(row == col) for row in range(size)]
        for col in range(size)
    ]
```

1.7 Pythagorean Triples

Edit the function `triples` so that it takes a number and returns a list of tuples of 3 integers where each tuple is a Pythagorean triple, and the integers are all less then the input number.

A Pythagorean triple is a group of 3 integers `a`, `b`, and `c`, such that they satisfy the formula `a**2 + b**2 = c**2`

```
>>> from lists import triples
>>> triples(15)
[(3, 4, 5), (5, 12, 13), (6, 8, 10)]
>>> triples(30)
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9, 12, 15), (10, 24,␣
↪26), (12, 16, 20), (15, 20, 25), (20, 21, 29)]
```

Answers

```
def triples(num):
    """Return list of Pythagorean triples less than input num"""
    return [
        (a, b, c)
        for a in range(1, num)
        for b in range(a+1, num)
        for c in range(b+1, num)
        if a**2 + b**2 == c**2
    ]
```

CHAPTER

TWO

GENERATOR EXERCISES

These exercises are all in the `generators.py` file in the `exercises` directory. Edit the file to add the functions or fix the error(s) in the existing function(s).

To run the test: from the `exercises` folder, type `python test.py <function_name>`, like this:

```
$ python test.py all_together
```

2.1 Sum All

Edit the function `sum_all` so that it accepts a list of lists of numbers and returns the sum of all of the numbers Use a generator expression.

```
>>> from loops import sum_all
>>> matrix = [[1, 2, 3], [4, 5, 6]]
>>> sum_all(matrix)
21
>>> sum_all([[0, 1], [4, 2], [3, 1]])
11
```

Answers

With two loops:

Using the `sum` function and a loop:

```
def sum_all(number_lists):
    """Return the sum of all numbers in the given list-of-lists."""
    total = 0
    for numbers in number_lists:
        total += sum(numbers)
    return total
```

Using the `sum` function and a generator expression:

```
def sum_all(number_lists):
    """Return the sum of all numbers in the given list-of-lists."""
    return sum(
        n
        for numbers in number_lists
        for n in numbers
    )
```

2.2 All Together

Edit the function `all_together` so that it takes any number of iterables and strings them together.

Make sure the return value of your function is a generator.

Example:

```
>>> from generators import all_together
>>> list(all_together([1, 2], (3, 4), "hello"))
[1, 2, 3, 4, 'h', 'e', 'l', 'l', 'o']
>>> nums = all_together([1, 2], (3, 4))
>>> list(all_together(nums, nums))
[1, 2, 3, 4]
```

Answers

```
def all_together(*iterables):
    """String together all items from the given iterables."""
    return (
        item
        for iterable in iterables
        for item in iterable
    )
```

2.3 Interleave

Edit the function `interleave` so that it accepts two iterables and returns a generator object with each of the given items “interleaved” (item 0 from iterable 1, then item 0 from iterable 2, then item 1 from iterable 1, and so on).

Hint: The built-in `zip` function will be useful for this.

Example:

```
>>> from generators import interleave
>>> list(interleave([1, 2, 3, 4], [5, 6, 7, 8]))
[1, 5, 2, 6, 3, 7, 4, 8]
>>> nums = [1, 2, 3, 4]
>>> list(interleave(nums, (n**2 for n in nums)))
[1, 1, 2, 4, 3, 9, 4, 16]
```

Answers

```
def interleave(iterable1, iterable2):
    """Return iterable of one item at a time from each list."""
    return (
        item
        for pair in zip(iterable1, iterable2)
        for item in pair
    )
```

2.4 Deep Add

Edit the `deep_add` function so that it accepts an iterable of iterables of numbers of unknown depth and returns the sums of all the numbers.

Example:

```
>>> from exception import deep_add
>>> deep_add([1, 2, 3, 4])
10
>>> deep_add([(1, 2), [3, {4, 5}]])
15
```

Answers

With a counter variable:

```
def deep_add(iterable_or_number):
    total = 0
    try:
        for x in iterable_or_number:
            total += deep_add(x)
    except TypeError:
        total += iterable_or_number
    return total
```

With the `sum` function and exception handling:

```
def deep_add(iterable_or_number):
    try:
        numbers = (deep_add(x) for x in iterable_or_number)
    except TypeError:
        return iterable_or_number
    else:
        return sum(numbers)
```

With the `sum` function and a check for a `__iter__` method:

```
def deep_add(iterable_or_number):
    """Return sum of values in given iterable, iterating deeply."""
    if hasattr(iterable_or_number, '__iter__'):
        return sum(deep_add(x) for x in iterable_or_number)
    else:
        return iterable_or_number
```

With a slightly more idiomatic `if` statement and `isinstance` check:

```
from collections.abc import Iterable

def deep_add(iterable_or_number):
    """Return sum of values in given iterable, iterating deeply."""
    if isinstance(iterable_or_number, Iterable):
        return sum(deep_add(x) for x in iterable_or_number)
    else:
        return iterable_or_number
```

2.5 Parse Number Ranges

Edit the `parse_ranges` function so that it accepts a string containing ranges of numbers and returns a list of the actual numbers contained in the ranges. The range numbers are inclusive.

It should work like this:

```
>>> from generators import parse_ranges
>>> parse_ranges('1-2,4-4,8-10')
[1, 2, 4, 8, 9, 10]
>>> parse_ranges('0-0, 4-8, 20-21, 43-45')
[0, 4, 5, 6, 7, 8, 20, 21, 43, 44, 45]
```

Answers

```
def parse_ranges(ranges_string):
    """Return a list of numbers corresponding to number ranges in a string"""
    pairs = (
        group.split('-')
        for group in ranges_string.split(',')
    )
    return [
        num
        for start, stop in pairs
        for num in range(int(start), int(stop)+1)
    ]
```

2.6 Is Prime

Rewrite the `is_prime` function in one expression.

```
def is_prime(candidate):
    for n in range(2, candidate // 2):
        if candidate % n == 0:
            return False
    return True
```

Hint: Use the `any` or `all` built-in functions and a generator expression.

It should work like this:

```
>>> from generators import is_prime
>>> is_prime(9)
False
>>> is_prime(11)
True
>>> is_prime(23)
True
```

Answers

```
def is_prime(candidate):
    """Return True if candidate number is prime."""
```

(continues on next page)

(continued from previous page)

```
    return not any(
        candidate % n == 0
        for n in range(2, candidate)
    )
```

```
def is_prime(candidate):
    """Return True if candidate number is prime."""
    return all(
        candidate % n != 0
        for n in range(2, candidate)
    )
```

CHAPTER

THREE

GENERATOR FUNCTION EXERCISES

These exercises are all in the `functions.py` file in the `exercises` directory. Edit the file to add the functions or fix the error(s) in the existing function(s).

To run the test: from the `exercises` folder, type `python test.py <function_name>`, like this:

```
$ python test.py unique
```

3.1 Unique

Edit the function `unique` to be a generator function that takes an iterable and yields the iterable elements in order, skipping duplicate values.

Example:

```
>>> from functions import unique
>>> list(unique([6, 7, 0, 9, 0, 1, 2, 7, 7, 9]))
[6, 7, 0, 9, 1, 2]
>>> list(unique([]))
[]
>>> ''.join(unique("hello there"))
'helo tr'
```

Answers

```
def unique(elements):
    """Yield iterable elements in order, skipping duplicate values."""
    seen_items = set()
    for item in elements:
        if item not in seen_items:
            yield item
            seen_items.add(item)
```

3.2 Float Range

Edit the `float_range` function so that it works like `range` except that the `start`, `stop`, and `step` can be fractional.

You can ignore negative `step` values (I'll only ever provide positive `step` values to you).

```
>>> list(float_range(2.5, 5))
[2.5, 3.5, 4.5]
>>> list(float_range(2.5, 5, 0.5))
[2.5, 3.0, 3.5, 4.0, 4.5]
>>> list(float_range(2.5, 5, step=0.5))
[2.5, 3.0, 3.5, 4.0, 4.5]
```

Answers

```
def float_range(start, stop, step=1):
    """Return iterable of numbers from start to stop by step."""
    i = start
    while i < stop:
        yield i
        i += step
```

3.3 Head

Edit the `head` function to lazily gives the first `n` items of a given iterable.

Try to use a generator function for this.

```
>>> list(head([1, 2, 3, 4, 5], n=2))
[1, 2]
>>> first_4 = head([1, 2, 3, 4, 5], n=4)
>>> list(zip(first_4, first_4))
[(1, 2), (3, 4)]
```

Answers

```
def head(iterable, n):
    """Return the first n items of given iterable."""
    for item, _ in zip(iterable, range(n)):
        yield item
```

```
def head(iterable, n):
    """Return the first n items of given iterable."""
    for count, item in enumerate(iterable):
        if count >= n:
            break
        yield item
```

3.4 Interleave

Edit the `interleave` function in `generators.py` so that it two iterables and returns a generator object with each of the given items "interleaved" (e.g. first item from first iterable, first item from second, second item from first, second item from second, and so on). You may assume the input iterables have the same number of elements.

Try to use a generator function for this.

```
>>> list(interleave([1, 2, 3, 4], [5, 6, 7, 8]))
[1, 5, 2, 6, 3, 7, 4, 8]
>>> nums = [1, 2, 3, 4]
>>> list(interleave(nums, (n**2 for n in nums)))
[1, 1, 2, 4, 3, 9, 4, 16]
```

Answers

```
def interleave(iterable1, iterable2):
    for iterable in zip(iterable1, iterable2):
        for item in iterable:
            yield item
```

Or even better with `yield from` and an unlimited number of iterables:

```
def interleave(*iterables):
    for items in zip(*iterables):
        yield from items
```

3.5 Pairwise

Edit the function `pairwise` to be a generator function that accepts an iterable and yields a tuple containing each item and the item following it. The last item should treat the item after it as `None`.

Example:

```
>>> from functions import pairwise
>>> list(pairwise([1, 2, 3]))
[(1, 2), (2, 3), (3, None)]
>>> list(pairwise([]))
[]
>>> list(pairwise("hey"))
[('h', 'e'), ('e', 'y'), ('y', None)]
```

Answers

```
def pairwise(elements):
    """
    Yield a tuple containing each item and the item following it.

    The item after the last one is treated as ``None``.
    """
    previous, current = None, None
    for current in elements:
        if previous:
            yield previous, current
        previous = current
```

(continues on next page)

(continued from previous page)

```
    if current:
        yield current, None
```

This second answer is not ideal because it **only works on sequences** like lists. It won't work on sets, dictionaries, or any non-sequence iterable like generators.

```
def pairwise(elements):
    """
    Yield a tuple containing each item and the item following it.

    The item after the last one is treated as ``None``.
    """
    for i, item in enumerate(elements):
        if i < len(elements) - 1:
            next_item = elements[i + 1]
        else:
            next_item = None
        yield item, next_item
```

3.6 Stop On

Edit the function `stop_on` to be a generator function that accepts an iterable and a value and yields from the given iterable repeatedly until the given value is reached.

Example:

```
>>> from functions import stop_on
>>> list(stop_on([1, 2, 3], 3))
[1, 2]
```

```
>>> next(stop_on([1, 2, 3], 1), 0)
0
```

Answers

```
def stop_on(elements, stop_item):
    """Yield from the iterable until the given value is reached."""
    for item in elements:
        if item == stop_item:
            break
        yield item
```

3.7 Around

Edit the function `around` to be a generator function that accepts an iterable and yields a tuple containing the previous item, the current item, and the next item. The previous item should start at `None` and the next item should be `None` for the last item in the iterable.

Example:

```
>>> from functions import around
>>> list(around([1, 2, 3, 4]))
[(None, 1, 2), (1, 2, 3), (2, 3, 4), (3, 4, None)]
>>> list(around([]))
[]
>>> list(around("hey"))
[(None, 'h', 'e'), ('h', 'e', 'y'), ('e', 'y', None)]
```

Answers

```
def around(elements):
    """
    Yield a tuple of the previous, current, and next items.

    The previous item should start at ``None`` and the next item should
    be ``None`` for the last item in the iterable.
    """
    previous, current = None, None
    for next_item in elements:
        if current:
            yield previous, current, next_item
        previous, current = current, next_item
    if current:
        yield previous, current, None
```

3.8 Deep Flatten

Edit the function `deep_flatten` to be a generator function that “flattens” nested iterables. In other words the function should accept an iterable of iterables and yield non-iterable items in order.

Example:

```
>>> from functions import deep_flatten
>>> list(deep_flatten([0, [1, [2, 3]], [4]]))
[0, 1, 2, 3, 4]
>>> list(deep_flatten([[()]]))
[]
```

Answers

```
def deep_flatten(thing):
    """Flatten an iterable of iterables."""
    try:
        for item in thing:
            yield from deep_flatten(item)
    except TypeError:
        yield thing
```

If you’re worried about recursing forever on strings:

```
def deep_flatten(thing):
    """Flatten an iterable of iterables."""
    try:
        for item in thing:
            if isinstance(item, (str, bytes)):
```

(continues on next page)

(continued from previous page)

```
                yield item
            else:
                yield from deep_flatten(item)
    except TypeError:
        yield thing
```

3.9 Big Primes

Edit the `get_primes_over` function to return a given number of primes above 1,000,000. Make it a generator.

It should work like this:

```
>>> from generators import get_primes_over
>>> primes = get_primes_over(5)
>>> next(primes)
1000003
>>> next(primes)
1000033
>>> next(primes)
1000037
>>> next(primes)
1000039
>>> next(primes)
1000081
>>> next(primes)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> list(get_primes_over(3))
[1000003, 1000033, 1000037]
```

You can use this function to determine whether a number is prime:

```
def is_prime(candidate):
    """Return True if candidate number is prime."""
    for n in range(2, candidate):
        if candidate % n == 0:
            return False
    return True
```

Answers

```
def get_primes_over(limit):
    """Return given number of primes over 1,000,000."""
    candidate = 1000000
    count = 0
    while count < limit:
        if is_prime(candidate):
            yield candidate
            count += 1
            candidate += 1
        else:
            candidate += 1
```

CHAPTER

FOUR

ITERATOR EXERCISES

These exercises are all in the `iterators.py` file in the `exercises` directory. Edit the file to add the functions or fix the error(s) in the existing function(s). To run the test: from the `exercises` folder, type `python test.py <function_name>`, like this:

```
$ python test.py first
```

4.1 First

Edit the function `first` so that it returns the first item in any iterable:

```
>>> from iterators import first
>>> first(iter([1, 2]))
1
>>> first([1, 2])
1
```

Answers

```
def first(iterable):
    """Return the first item in given iterable."""
    return next(iter(iterable))
```

Note that this doesn't always work (try it on dictionaries or sets):

```
def first(iterable):
    """Return the first item in given iterable."""
    try:
        return next(iterable)
    except TypeError:
        return iterable[0]
```

4.2 Is Iterator

Edit the function `is_iterator` so that it accepts an iterable and returns `True` if the given iterable is an iterator.

Example:

```
>>> from iterators import is_iterator
>>> is_iterator(iter([]))
True
>>> is_iterator([1, 2])
False
>>> i = iter([1, 2])
>>> is_iterator(i)
True
>>> list(i)
[1, 2]
>>> def gen(): yield 4
...
>>> is_iterator(gen())
True
```

Answers

```
def is_iterator(iterable):
    """Return True if given iterable is an iterator."""
    return iter(iterable) is iterable
```

Note that this won't work because it will consume an item from the iterator:

```
def is_iterator(iterable):
    """Return True if given iterable is an iterator."""
    try:
        next(iterable)
    except TypeError:
        return False
    else:
        return True
```

4.3 Point

Make a `Point` class that stores 3-dimensional coordinates. Your `Point` class should work with multiple assignment, like this:

```
>>> p = Point(2, 3, 6)
>>> x, y, z = p
>>> x
2
>>> y
3
>>> z
6
```

Answers

```python
class Point:

    """Class representing a 3 dimensional point."""

    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __iter__(self):
        yield self.x
        yield self.y
        yield self.z
```

```python
class Point:

    """Class representing a 3 dimensional point."""

    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __iter__(self):
        yield from (self.x, self.y, self.z)
```

```python
class Point:

    """Class representing a 3 dimensional point."""

    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __iter__(self):
        return iter((self.x, self.y, self.z))
```

4.4 All Same

Edit the function `all_same` so that it accepts an iterable and `True` if all items in the iterable are equal to each other.

Example:

```python
>>> from iterators import all_same
>>> all_same(n % 2 for n in [3, 5, 7, 8])
False
>>> all_same(n % 2 for n in [3, 5, 7, 9])
True
```

Your function should work with any iterable and any items that can be compared (including unhashable ones). It should return as soon as an unequal value is found.

Answers

With a `for` loop to get the first item and a `for` loop to compare values:

```python
def all_same(iterable):
    """Return True if all items in the given iterable are the same."""
    for first_item in iterable:
        break
```

(continues on next page)

(continued from previous page)

```python
    for item in iterable:
        if item != first_item:
            return False
    return True
```

With `next` to get the first item and a `for` loop to compare values:

```python
def all_same(iterable):
    """Return True if all items in the given iterable are the same."""
    first_item = next(iter(iterable), None)
    for item in iterable:
        if item != first_item:
            return False
    return True
```

With a generator expression with `all` to compare values:

```python
def all_same(iterable):
    """Return True if all items in the given iterable are the same."""
    first_item = next(iter(iterable), None)
    return all(
        item == first_item
        for item in iterable
    )
```

4.5 minmax

Edit the function `minmax` to accept an iterable and return the minimum and maximum values of that iterable.

Example:

```python
>>> from iterators import minmax
>>> minmax(n**2 for n in [9, 5, 2, 8])
(4, 81)
```

Your `minmax` function should accept any iterable.

Note: This function **should not copy every item** in the supplied iterable into a new list. Process the items one by one so that your function won't have any memory concerns with extremely long/large iterables.

Answers

```python
def minmax(iterable):
    """Return minimum and maximum values from given iterable."""
    iterator = iter(iterable)
    try:
        minimum = maximum = next(iterator)
    except StopIteration as e:
        raise ValueError("Iterable empty") from e
    for item in iterator:
        if item < minimum:
            minimum = item
        if maximum < item:
```

(continues on next page)

(continued from previous page)

```
            maximum = item
    return (minimum, maximum)
```

4.6 Random Number

Make an inexhaustable iterator object `RandomNumberGenerator` that returns random integers between two numbers (inclusive).

Example:

```
>>> number_generator = RandomNumberGenerator(4, 8)
>>> next(number_generator)
4
>>> next(number_generator)
7
>>> next(number_generator)
8
>>> iter(number_generator) is number_generator
True
```

Answers

```
from random import randint


class RandomNumberGenerator:

    """Return infinite series of randomly generator numbers."""

    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __next__(self):
        return randint(self.start, self.end)

    def __iter__(self):
        return self
```

4.7 Dictionary Changes

1. Create an empty dictionary
2. Get an iterator for the dictionary
3. Add an item to the dictionary
4. Try to get the next item out of the iterator

What happened?

4.8 List Changes

1. Create a list with two items
2. Get an iterator from the list
3. Get the next item from the iterator
4. Insert an item at the **beginning** of the list
5. Get the next item from the iterator

What happened?

CHAPTER

FIVE

ITERTOOLS EXERCISES

Except as noted, these exercises are in the `iteration.py` file in the `exercises` directory. Edit the file to add the functions or fix the error(s) in the existing function(s).

To run the test: from the `exercises` folder, type `python test.py <function_name>`, like this:

```
$ python test.py get_primes_over
```

5.1 Head

Edit the `head` function we saw before in `functions.py` so that it returns an iterator which gives the first `n` items of a given iterable. Use something from `itertools` to solve this exercise.

Example:

```
>>> from generators import head
>>> list(head([1, 2, 3, 4, 5], n=2))
[1, 2]
>>> first_4 = head([1, 2, 3, 4, 5], n=4)
>>> list(zip(first_4, first_4))
[(1, 2), (3, 4)]
```

Answers

```
from itertools import islice


def head(iterable, n):
    """Return first n items of a given iterable."""
    return islice(iterable, n)
```

5.2 All Together

Edit the `all_together` function we saw before in `generators.py` so that it takes any number of iterables and strings them together. Use something from `itertools` to solve this exercise.

Example:

```
>>> from generators import all_together
>>> list(all_together([1, 2], (3, 4), "hello"))
```

(continues on next page)

(continued from previous page)

```
[1, 2, 3, 4, 'h', 'e', 'l', 'l', 'o']
>>> nums = all_together([1, 2], (3, 4))
>>> list(all_together(nums, nums))
[1, 2, 3, 4]
```

Answers

```
from itertools import chain

def all_together(*iterables):
    """String together all items from the given iterables."""
    return chain.from_iterable(iterables)
```

5.3 Total Length

Edit the `total_length` function so that it calculates the total length of all given iterables.

Example:

```
>>> from iteration import total_length
>>> total_length([1, 2, 3])
3
>>> total_length()
0
>>> total_length([1, 2, 3], [4, 5], iter([6, 7]))
7
```

Answers

With a chained list:

```
from itertools import chain


def total_length(*iterables):
    """Return the total number of items in all given iterables."""
    return len(list(chain.from_iterable(iterables)))
```

With sum of a generator containing `1` for each item:

```
from itertools import chain


def total_length(*iterables):
    """Return the total number of items in all given iterables."""
    return sum(1 for _ in chain.from_iterable(iterables))
```

5.4 lstrip

Edit the `lstrip` function to accept an iterable and an item to strip from the beginning of the iterable. The `lstrip` function should return an iterator which, when looped over, will provide each of the items in the given iterable *after* all of the strip values have been removed from the beginnign.

Example:

```
>>> list(lstrip([0, 0, 1, 0, 2, 3, 0], 0))
[1, 0, 2, 3, 0]
>>> ''.join(lstrip('hhello there' 'h'))
'ello there'
```

Answers

```
def lstrip(iterable, strip_value):
    """Return iterable with strip_value items removed from beginning."""
    def is_strip_value(value): return value == strip_value
    return dropwhile(is_strip_value, iterable)
```

5.5 Stop On

Edit the `stop_on` function we saw before in `functions.py` so that it accepts an iterable and a value and yields from the given iterable repeatedly until the given value is reached. Use something from `itertools` to solve this exercise.

Example:

```
>>> from functions import stop_on
>>> list(stop_on([1, 2, 3], 3))
[1, 2]
>>> next(stop_on([1, 2, 3], 1), 0)
0
```

Answers

```
from itertools import takewhile

def stop_on(elements, stop_item):
    """Yield from the iterable until the given value is reached."""
    def not_equal(item): return item != stop_item
    return takewhile(not_equal, elements)
```

5.6 Interleave

Edit the `interleave` function we saw before in the `generators.py` file so that it works with iterables of different length. Any short iterables should be skipped over once exhausted.

For example:

```
>>> list(interleave([1, 2, 3], [6, 7, 8, 9]))
[1, 6, 2, 7, 3, 8, 9]
```

(continues on next page)

(continued from previous page)

```
>>> list(interleave([1, 2, 3], [4, 5, 6, 7, 8]))
[1, 4, 2, 5, 3, 6, 7, 8]
>>> list(interleave([1, 2, 3, 4], [5, 6]))
[1, 5, 2, 6, 3, 4]
```

Note: to test this exercise you'll need to comment out the appropriate `@unittest.skip` line in `generators_test.py`.

Answers

```
from itertools import zip_longest

def interleave(*iterables):
    """Return iterable of one item at a time from each list."""
    sentinel = object()
    return (
        item
        for items in zip_longest(*iterables, fillvalue=sentinel)
        for item in items
        if item is not sentinel
    )
```

5.7 Big Primes

Edit the `get_primes_over` function we saw before in the `functions.py` file so that it yields a specified number of prime numbers greater than 1,000,000.

Try doing this without using `while` loops or `for` loops, using `itertools`.

You can use this function to determine whether a number is prime:

```
def is_prime(candidate):
    """Return True if candidate is a prime number"""
    for n in range(2, candidate):
        if candidate % n == 0:
            return False
    return True
```

Answers

```
from itertools import count

def get_primes_over(limit):
    """Return given number of primes over 1,000,000."""
    primes = (n for n in count(1000000) if is_prime(n))
    return (next(primes) for _ in xrange(limit))
```

```
from itertools import count, islice

def get_primes_over(limit):
    """Return given number of primes over 1,000,000."""
    primes = (n for n in count(1000000) if is_prime(n))
    return islice(primes, limit)
```