List Comprehensions and Generator Expressions in Python - Answers

Release 1.0

Trey Hunner

May 23, 2018

CHAPTER

ONE

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: You should use at least one list comprehension in each of these exercises!

Starting with a vowel

Edit the `get_vowel_names` function 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:

```
>>> from lists import get_vowel_names
>>> names = ["Alice", "Bob", "Christy", "Jules"]
>>> get_vowel_names(names)
['Alice']
>>> names = ["scott", "arthur", "jan", "elizabeth"]
>>> get_vowel_names(names)
['arthur', 'elizabeth']
```

Answers

Without a list comprehension:

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

With a list comprehension:

```
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"
    ]
```

There are many variations of the vowel condition. Here's another:

```
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.startswith(tuple("aeiou"))
    ]
```

Power List By Index

Edit the `power_list` function 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

Without a list comprehension:

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

With a list comprehension:

```
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)]
```

Flatten a Matrix

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

```
>>> from lists 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

Without a list comprehension:

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

With a list comprehension (notice the order of the *for* clauses):

```
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
    ]
```

Reverse Difference

Edit the function `reverse_difference` so that it accepts a list of numbers and returns a new copy of the list with the reverse of the list subtracted.

Example usage:

```
>>> from lists import reverse_difference
>>> reverse_difference([9, 8, 7, 6])
[3, 1, -1, -3]
>>> reverse_difference([1, 2, 3, 4, 5])
[-4, -2, 0, 2, 4]
>>> reverse_difference([3, 2, 1, 0])
[3, 1, -1, -3]
>>> reverse_difference([0, 0])
[0, 0]
```

Answers

Without a list comprehension:

```
def reverse_difference(numbers):
    """Return list subtracted from the reverse of itself."""
    differences = []
    for n, m in zip(numbers, numbers[::-1]):
        differences.append(n - m)
    return differences
```

With a list comprehension:

```
def reverse_difference(numbers):
    """Return list subtracted from the reverse of itself."""
    return [
        (n - m)
```

```
        for n, m in zip(numbers, numbers[::-1])
    ]
```

Matrix Addition

Edit the function `matrix_add` so that it takes two matrices (lists of lists of numbers) and returns a new matrix (list of lists of numbers) with the corresponding numbers added together.

Example usage:

```
>>> from lists import matrix_add
>>> m1 = [[6, 6], [3, 1]]
>>> m2 = [[1, 2], [3, 4]]
>>> matrix_add(m1, m2)
[[7, 8], [6, 5]]
>>> m1
[[6, 6], [3, 1]]
>>> m2
[[1, 2], [3, 4]]
>>> matrix_add([[5]], [[-2]])
[[3]]
>>> m1 = [[1, 2, 3], [4, 5, 6]]
>>> m2 = [[-1, -2, -3], [-4, -5, -6]]
>>> matrix_add(m1, m2)
[[0, 0, 0], [0, 0, 0]]
```

Answers

Without list comprehensions:

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

With a list comprehension for creating rows:

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

With nested list comprehensions:

```
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)
    ]
```

Transpose

File: Edit the `transpose` function in the `lists.py` file.

Test: Run `python test.py transpose` in your `exercises` directory.

Exercise: Make a function `transpose` that accepts a list of lists and returns the transpose of the list of lists.

Example usage:

```
>>> from zip import transpose
>>> transpose([[1, 2], [3, 4]])
[[1, 3], [2, 4]]
>>> matrix = [['a','b','c'],['d','e','f'],['g','h','i']]
>>> transpose(matrix)
[['a', 'd', 'g'], ['b', 'e', 'h'], ['c', 'f', 'i']]
```

Answers

Without a list comprehension:

```
def transpose(matrix):
    """Return a transposed version of given list of lists."""
    transposed = []
    for row in zip(*matrix):
        transposed.append(list(row))
    return transposed
```

With a list comprehesion:

```
def transpose(matrix):
    """Return a transposed version of given list of lists."""
    return [
        list(row)
        for row in zip(*matrix)
    ]
```

Factors

File: Edit the `get_factors` function in the `lists.py` file.

Test: Run `python test.py get_factors` in your `exercises` directory.

Exercise: The function `get_factors` returns the factors of a given number.

Example:

```
>>> from lists import get_factors
>>> get_factors(2)
[1, 2]
>>> get_factors(6)
[1, 2, 3, 6]
```

```
>>> get_factors(100)
[1, 2, 4, 5, 10, 20, 25, 50, 100]
```

Answers

Without a list comprehension:

```
def get_factors(number):
    factors = []
    for n in range(1, number+1):
        if number % n == 0:
            factors.append(n)
    return factors
```

With a list comprehension:

```
def get_factors(number):
    return [
        n
        for n in range(1, number+1)
        if number % n == 0
    ]
```

Pythagorean Triples

Edit the `triples` function 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

Without a comprehension:

```
def triples(num):
    """Return list of Pythagorean triples less than input num"""
    triples_list = []
    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:
                    triples_list.append((a, b, c))
    return triples_list
```

With a comprehension:

```
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 EXPRESSION 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 is_prime
```

Primality

Edit the function `is_prime` so that it returns `True` if a number is prime and `False` otherwise.

Example:

```
>>> from generators import is_prime
>>> is_prime(21)
False
>>> is_prime(23)
True
```

Hint: You might want to use `any` or `all` for this.

Answers

With a *for* loop and *return* statements:

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

With a generator expression and *any*:

```
def is_prime(candidate):
    """Return True if candidate number is prime."""
    return candidate >= 2 and not all(
        candidate % n == 0
        for n in range(2, candidate)
    )
```

With a generator expression and *all*:

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

More performant:

```
from math import sqrt

def is_prime(candidate):
    """Return True if candidate number is prime."""
    return candidate >= 2 and all(
        candidate % n != 0
        for n in range(2, int(sqrt(candidate))+1)
    )
```

All Together

Edit the function `all_together` so that it takes any number of iterables and strings them together. Try using a generator expression to do it.

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
    )
```

Interleave

Edit the `interleave` function 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).

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

Without a generator expression (doesn't pass tests):

```
def interleave(iterable1, iterable2):
    """Return iterable of one item at a time from each list."""
    interleaved = []
    for item1, item2 in zip(iterable1, iterable2):
        interleaved.append(item1)
        interleaved.append(item2)
    return interleaved
```

With just a single *append* (so we can copy-paste it to a generator expression):

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

With a generator expression (passes the tests):

```
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
    )
```

Translate

Edit the function `translate` so that it takes a string in one language and transliterates each word into another language, returning the resulting string.

Here is an (over-simplified) example translation dictionary for translating from Spanish to English:

```
>>> words = {'esta': 'is', 'la': 'the', 'en': 'in', 'gato': 'cat', 'casa': 'house',
↪'el': 'the'}
```

Translate a sentence using your algorithm. An example of how this function should work:

```
>>> from generators import translate
>>> translate("el gato esta en la casa")
'the cat is in the house'
```

Answers

With string concatenation and no generator expression:

```
def translate(sentence):
    """Return a transliterated version of the given sentence."""
    translated = ""
    for w in sentence.split():
        translated += words[w] + " "
    return translated.rstrip()
```

With string *join* method:

```
def translate(sentence):
    """Return a transliterated version of the given sentence."""
    translated_words = []
    for w in sentence.split():
        translated_words.append(words[w])
    return " ".join(translated_words)
```

With string *join* method and a list comprehension:

```
def translate(sentence):
    """Return a transliterated version of the given sentence."""
    translated_words = [
        words[w]
        for w in sentence.split()
    ]
    return " ".join(translated_words)
```

With generator expression:

```
def translate(sentence):
    """Return a transliterated version of the given sentence."""
    return " ".join(
        words[w]
        for w in sentence.split()
    )
```

Parse Number Ranges

Edit the `parse_ranges` function so that it accepts a string containing ranges of numbers and returns a generator 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

With single *for* loop:

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

With pre-processing of "," and "-" splitting

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

With two generator expressions:

```
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)
    )
```

Primes Over

Edit the function `first_prime_over` so that it returns the first prime number over a given number.

Example:

```
>>> from generators import first_prime_over
>>> first_prime_over(1000000)
1000003
```

Answers

With early break:

```
def first_prime_over(n):
    """Return the first prime number over a given number."""
    for n in range(n+1, n**2):
        if is_prime(n):
            return n
```

With generator expression and *next* call:

```
def first_prime_over(n):
    """Return the first prime number over a given number."""
    primes = (
        n
        for n in range(n+1, n**2)
        if is_prime(n)
    )
    return next(primes)
```

With generator expression passed directly into *next*:

```
def first_prime_over(n):
    """Return the first prime number over a given number."""
    return next(
        n
        for n in range(n+1, n**2)
        if is_prime(n)
    )
```

Anagrams

Edit the function `is_anagram` so that it accepts two strings and returns True if the two strings are anagrams of each other. The function should use generator expressions. Make sure your function works with mixed case.

It should work like this:

```
>>> from generators import is_anagram
>>> is_anagram("tea", "eat")
True
>>> is_anagram("tea", "treat")
False
>>> is_anagram("sinks", "skin")
False
>>> is_anagram("Listen", "silent")
True
```

The function should also ignore spaces and punctuation:

```
>>> is_anagram("coins kept", "in pockets")
True
>>> is_anagram("a diet", "I'd eat")
True
```

Answers

```
def is_anagram(word1, word2):
    """Return True if the given words are anagrams."""
    word1, word2 = word1.lower(), word2.lower()
    letters1 = sorted(c for c in word1 if c.isalpha())
    letters2 = sorted(c for c in word2 if c.isalpha())
    return letters1 == letters2
```

CHAPTER

THREE

MORE COMPREHENSION EXERCISES

These exercises are all in the `more.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 flip_dict
```

Flipped Dictionary

Edit the function `flip_dict`, that flips dictionary keys and values.

Example usage:

```
>>> from more import flip_dict
>>> flip_dict({'Python': "2015-09-15", 'Java': "2015-09-14", 'C': "2015-09-13"})
{'2015-09-13': 'C', '2015-09-15': 'Python', '2015-09-14': 'Java'}
```

Answers

Without dictionary comprehension:

```
def flip_dict(dictionary):
    """Return a new dictionary that maps the original values to the keys."""
    flipped = {}
    for old_key, old_value in dictionary.items():
        flipped[old_value] = old_key
    return flipped
```

With dictionary comprehension:

```
def flip_dict(dictionary):
    """Return a new dictionary that maps the original values to the keys."""
    return {
        old_value: old_key
        for old_key, old_value in dictionary.items()
    }
```

ASCII Strings

Edit the function `get_ascii_codes` so that it accepts a list of strings and returns a dictionary containing the strings as keys and a list of corresponding ASCII character codes as values.

```
>>> from more import get_ascii_codes
>>> words = ["hello", "bye", "yes", "no", "python"]
>>> get_ascii_codes(words)
{'yes': [121, 101, 115], 'hello': [104, 101, 108, 108, 111], 'python': [112, 121, 116,
↪ 104, 111, 110], 'no': [110, 111], 'bye': [98, 121, 101]}
```

Answers

Without any comprehensions:

```
def get_ascii_codes(words):
    """Return a dictionary mapping the strings to ASCII codes."""
    codes = {}
    for word in words:
        ascii_values = []
        for char in word:
            ascii_values[char] = ord(char)
        codes[word] = ascii_values
    return codes
```

Without a list comprehension:

```
def get_ascii_codes(words):
    """Return a dictionary mapping the strings to ASCII codes."""
    codes = {}
    for word in words:
        codes[word] = [ord(c) for c in word]
    return codes
```

Without a list comprehension inside a dictionary comprehension:

```
def get_ascii_codes(words):
    """Return a dictionary mapping the strings to ASCII codes."""
    return {
        word: [ord(c) for c in word]
        for word in words
    }
```

Double-valued Dictionary

Edit the function `dict_from_truple` so that it accepts a list of three-item tuples and returns a dictionary where the keys are the first item of each tuple and the values are a two-tuple of the remaining two items of each input tuple.

Example usage:

```
>>> from more import dict_from_truple
>>> dict_from_truple([(1, 2, 3), (4, 5, 6), (7, 8, 9)])
{1: (2, 3), 4: (5, 6), 7: (8, 9)}
```

Answers

Without unpacking:

```
def dict_from_truple(input_list):
    """Turn three-item tuples into a dictionary of two-valued tuples."""
    new_dict = {}
```

```
    for tup in input_list:
        new_dict[tup[0]] = (tup[1], tup[2])
    return new_dict
```

With unpacking (more idiomatic):

```
def dict_from_truple(input_list):
    """Turn three-item tuples into a dictionary of two-valued tuples."""
    truple_dict = {}
    for key, value1, value2 in input_list:
        truple_dict[key] = (value1, value2)
    return truple_dict
```

With unpacking (more idiomatic):

```
def dict_from_truple(input_list):
    """Turn three-item tuples into a dictionary of two-valued tuples."""
    return {
        key: (value1, value2)
        for key, value1, value2 in input_list
    }
```

Multi-valued Dictionary

Edit the function `dict_from_tuple` by starting with the code from your `dict_from_truple` function, above, and modify it to accept a list of tuples of any length and return a dictionary which uses the first item of each tuple as keys and all subsequent items as values.

Example usage:

```
>>> from more import dict_from_tuple
>>> dict_from_tuple([(1, 2, 3, 4), (5, 6, 7, 8)])
{1: (2, 3, 4), 5: (6, 7, 8)}
>>> dict_from_tuple([(1, 2, 3), (4, 5, 6), (7, 8, 9)])
{1: (2, 3), 4: (5, 6), 7: (8, 9)}
```

Answers

An answer with a loop:

```
def dict_from_tuple(tuple_list):
    """Turn multi-item tuples into a dictionary of two-valued tuples."""
    new_dict = {}
    for items in tuple_list:
        new_dict[items[0]] = items[1:]
    return new_dict
```

A better answer using multiple assignment:

```
def dict_from_tuple(tuple_list):
    """Turn multi-item tuples into a dictionary of two-valued tuples."""
    new_dict = {}
    for key, *values in tuple_list:
        new_dict[key] = values
    return new_dict
```

An answer using a dictionary comprehension:

```
def dict_from_tuple(tuple_list):
    """Turn multi-item tuples into a dictionary of two-valued tuples."""
    return {
        key: values
        for key, *values in tuple_list
    }
```

Factors

Edit the function `get_all_factors` so that it takes a set of numbers and makes a dictionary containing the numbers as keys and a list of factors as values.

```
>>> from more import get_all_factors
>>> get_all_factors({1, 2, 3, 4})
{1: [1], 2: [1, 2], 3: [1, 3], 4: [1, 2, 4]}
>>> get_all_factors({62, 293, 314})
{314: [1, 2, 157, 314], 293: [1, 293], 62: [1, 2, 31, 62]}
```

Hint: You can use this function to find the factors of any number:

```
def get_factors(number):
    """Get factors of the given number."""
    return [
        n
        for n in range(1, number + 1)
        if number % n == 0
    ]
```

Answers

```
def get_factors(number):
    """Get factors of the given number."""
    return [
        n
        for n in range(1, number + 1)
        if number % n == 0
    ]


def get_all_factors(numbers):
    """Return a dictionary mapping numbers to their factors."""
    return {
        n: get_factors(n)
        for n in numbers
    }
```

CHAPTER

FOUR

ADVANCED EXERCISES

These exercises are all in the `advanced.py` file (except as noted) 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 matrix_from_string
```

Matrix From String

Edit the `matrix_from_string` exercise in to accept a string and return a list of lists of integers (found in the string).

Example:

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

Answers

Without any comprehensions:

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

With comprehension for inner *for* loop:

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

With comprehension for outer *for* loop:

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

Atbash Cipher

Instructions for this one can be found here.

You can test this one by typing:

```
$ python test.py encode
```

And:

```
$ python test.py decode
```

Answers

```
from itertools import zip_longest
from string import ascii_lowercase as letters


atbash = dict(zip(letters, reversed(letters)))


def decode(string):
    """Return string of each character decoded."""
    return ''.join(
        atbash.get(c, c)
        for c in string.lower()
        if c.isalnum()
    )


def chunkify(string, n):
    """Return generator of n-letter word groups in string."""
    return (
        string[i:(i+n)]
        for i in range(0, len(string), n)
    )


def encode(string):
    """Decode each letter and group into 5-letter words."""
    return ' '.join(chunkify(decode(string), 5))
```

Memory-efficient CSV

Edit the function `parse_csv` so that it accepts a file object which contains a CSV file (including a header row) and returns a list of namedtuples representing each row. It contains a partially-implemented version that does not pass the

tests.

Note: Python's standard library has a csv module that makes reading and processing csv files easy. It has a `csv.reader` object for reading csv files that handles all the quoting and column separations for you. Each line in the file is read in as a list, with each element of the list being a column from the file. It also has a `csv.DictReader` object that will read the file into a list of dictionaries where the key is the column name and the value is the string from the corresponding column. Using `DictReader` to read CSV files is convenient because CSV columns can be referenced by name (instead of positional order). However there are some downsides to using `DictReader`. CSV column ordering is lost because dictionaries are unordered. The space required to store each row is also unnecessarily large because dictionaries are not a very space-efficient data structure.

There is discussion of adding a NamedTupleReader to the `csv` module, but this hasn't been implemented yet.

In the meantime, it's not too difficult to use a `csv.reader` object to open a CSV file and then use a `namedtuple` to represent each row.

Example with `us-state-capitals.csv`:

```
>>> with open('us-state-capitals.csv') as csv_file:
...     csv_rows = parse_csv(csv_file)
...
>>> csv_rows[:3]
[Row(state='Alabama', capital='Montgomery'), Row(state='Alaska', capital='Juneau'),␣
↪Row(state='Arizona', capital='Phoenix')]
```

Answers

```
from collections import namedtuple
import csv

def parse_csv(file):
    """Return namedtuple list representing data from given file object."""
    csv_reader = csv.reader(file)
    Row = namedtuple('Row', next(csv_reader))
    return [Row(*values) for values in csv_reader]
```

Deal Cards

Edit the `get_cards` and `deal_cards` functions. Some of them are partially implemented and may not pass the tests.

- `get_cards`: returns a list of namedtuples representing cards. Each card should have `suit` and `rank`.
- `shuffle_cards`: This function is provided for you.
- `deal_cards`: accepts a number as its argument, removes the given number of cards from the end of the list and returns them

Examples:

```
>>> from advanced import get_cards, shuffle_cards, deal_cards
>>> deck = get_cards()
>>> deck[:14]
[Card(rank='A', suit='spades'), Card(rank='2', suit='spades'), Card(rank='3', suit=
↪'spades'), Card(rank='4', suit='spades'), Card(rank='5', suit='spades'), Card(rank=
↪'6', suit='spades'), Card(rank='7', suit='spades'), Card(rank='8', suit='spades'),␣
↪Card(rank='9', suit='spades'), Card(rank='10', suit='spades'), Card(rank='J', suit=
↪'spades'), Card(rank='Q', suit='spades'), Card(rank='K', suit='spades'), Card(rank=
↪'A', suit='hearts')]
```

```
>>> len(deck)
52
>>> shuffle_cards(deck)
>>> deck[-5:]
[Card(rank='9', suit='diamonds'), Card(rank='6', suit='hearts'), Card(rank='7', suit=
↪'diamonds'), Card(rank='K', suit='spades'), Card(rank='7', suit='clubs')]
>>> hand = deal_cards(deck)
>>> hand
[Card(rank='9', suit='diamonds'), Card(rank='6', suit='hearts'), Card(rank='7', suit=
↪'diamonds'), Card(rank='K', suit='spades'), Card(rank='7', suit='clubs')]
>>> len(deck)
47
>>> deck[-5:]
[Card(rank='5', suit='spades'), Card(rank='Q', suit='clubs'), Card(rank='Q', suit=
↪'spades'), Card(rank='2', suit='diamonds'), Card(rank='6', suit='clubs')]
```

Answers

```
from collections import namedtuple
import random

def get_cards():
    """Create a list of namedtuples representing a deck of playing cards."""
    Card = namedtuple('Card', 'rank suit')
    ranks = ['A'] + [str(n) for n in range(2, 11)] + ['J', 'Q', 'K']
    suits = ['spades', 'hearts', 'diamonds', 'clubs']
    return [Card(rank, suit) for suit in suits for rank in ranks]

def shuffle_cards(deck):
    """Shuffles a deck in-place"""
    random.shuffle(deck)

def deal_cards(deck, count=5):
    """Remove the given number of cards from the deck and returns them"""
    return [deck.pop() for i in range(count)]
```

Meetup

Instructions for this one can be found here.

You can test this one by typing:

```
$ python test.py meetup_day
```

Answers

```
from calendar import day_name
import datetime


def weekdays_in_month(year, month, weekday):
    """Return list of all 4/5 dates with given weekday and year/month."""
    date = datetime.date(year, month, 1)
    date += datetime.timedelta(days=(7 + weekday - date.weekday()) % 7)
    first_to_fifth = (
        date + datetime.timedelta(days=7)*i
```

```
        for i in range(6)
    )
    return [
        date
        for date in first_to_fifth
        if date.month == month
    ]


def meetup_day(year, month, weekday, nth):
    weekday_names = list(day_name)
    shift_by = {'1st': 0, '2nd': 1, '3rd': 2, '4th': 3, '5th': 4, 'last': -1}
    dates = weekdays_in_month(year, month, weekday_names.index(weekday))
    if nth == 'teenth':
        return next(d for d in dates if d.day > 12)
    else:
        return dates[shift_by[nth]]
```