Useful Code Snippets

martes, 18 de octubre de 2022

11:04 a. m.

Prime numbers' Sieve:

From the Erastothenes Sieve, the one who first thought of the Idea to reduce the effort to find prime numbers in a range by just adding the first prime numbers and them multiples and only pick what's left as the result, this code encompass that:

def primeNumbers (cap):

    primes, non\_primes  = [1], list()

    for i in range(2, cap + 1):

        if i not in non\_primes:

            primes.append(i)

            for j in range( i\*i , cap + 1, i):

non\_primes.append(j)

return primes

prime\_test = primeNumbers(30)

print(prime\_test) = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

Filter non-unique-values from a list - [Link](http://Link):

Using Counter from the collections built-in module in python. Counter creates a dictionary that counts how many time a item appears on an iterable and adds it to a dictionary with the item as a key.

It's equivalent to write this function:

def my\_counter(list):

    my\_dict = {}

    for i in list:

        my\_dict[f'{i}'] = list.count(i)

    return my\_dict

The tip here is that if we call the Counter, in a one liner we can filter out repeated or non-repeated values, as follows:

Non-repeated:

from collections import \*

l = [1,1,2,3,0,1,4,5,8,9,7]

output = [ x for x in Counter(l) if Counter(l)[x] == 1 ]

print(output) = [2, 3, 0, 4, 5, 8, 9, 7]

Repeated:

from collections import \*

l = [1,1,2,3,0,1,4,5,8,9,7]

output = [ x for x in Counter(l) if Counter(l)[x] != 1 ]

print(output) = [1]

*Note from 27 Feb '23: Now that I know little more about comprehensions, this code imports without necessity, listcomps already filter this kind of things.*

Digitalizing a number - [Link](https://www.30secondsofcode.org/python/s/digitize):

If one would like to slice a full number and make it an iterable like is possible with strings, a quick function definition may come handy with a combination of map and int functions casting the number into a string:

def digitize(n):

  return list(map(int, str(n)))

x = 124895762

l1 = digitize(x)

print(l1) = [1,2,4,8,9,5,7,6,2]

Newbie mistake: Checking types with 'equality':

Sometimes is require to check if something is a certain type, and someone non-experienced could try to check with the following syntax:

Point = namedtuple('Point', ['x', 'y'])

p = Point(1,2)

if type(p) == tuple: print('its a tuple')

else: print('Not a tuple') = Not a tuple

The problem with this is that the syntax written would compare if 'p' is the built-in tuple type, and it isn't technically speaking. By heritance, p is a subclass of a P which is a tuple, so, to correct this, is better to use the isinstance() function.

if isinstance(p, tuple): print('its a tuple')

else: print('Not a tuple') = its a tuple

If \_\_name\_\_ == '\_\_main\_\_' idiom - [Link](https://www.youtube.com/watch?v=g_wlZ9IhbTs):

This little function at the end of the code where the body of the script is called is used to differentiate when a file is intended to be executed or not.

I am not quite familiar yet with all the implications, but what I got from the video is that a file without that clause at the end is a sign to let the reader of the code that the file is intended to be used as a library but not do be executed.

From the exercise # 254 from lists exercises:

The problem required to multiply the respective elements from two lists, and while there is a way defining an intermediate function, with two one-liners combining zip, map and lambda function will do the trick as well:

li = [10, 50, 40]

w = [2, 5, 3]

res = list(zip(li, w))

avg = list(map(lambda x: x[0]\*x[1], res))

print(sum(avg)/sum(w)) = 39.0

Using eval( ) to evaluate something for each element of a list

The combinations of all or any functions and the listcomp alongside with eval function is quite powerful.

I've defined an intermediate function that generates prime numbers up some cap value.

li = [0, 3, 4, 7, 9]

cap = max(li)

def isprime (n, cap): "/Function defined"

print( all( [ eval( f'isprime({i},{cap})') for i in li ] ) ) = True

print( ( [ eval( f'isprime({i},{cap})') for i in li ] ) ) = [True, True, True, True]

Ordering a dict by values

I really liked this syntax to have a dictionary ordered from highest to lowest key values.

dict1 = { k: v for k, v in sorted( dict1.items(), key= lambda x: x[1], reverse=True ) }

Flattening an Array with ListComp

vec = [[1,2,3], [4,5,6], [7,8,9]]

new\_list = [num for elem in vec for num in elem]

print(new\_list) = [1, 2, 3, 4, 5, 6, 7, 8, 9]

Transposing a matrix with ListComp & zip( ) functions

Listcomp:

matrix = [

[1,2,3,4],

[5,6,7,8],

[9,10,11,12]

]

transposed\_matrix = [ [row[i] for row in matrix ] for i in range(len(matrix[0])) ]

print(transposed\_matrix) = [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

Zip( ):

transposed\_matrix = list(zip(\*matrix))

print(transposed\_matrix) = [(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

The result is a list of tuples, so for have it exactly as the listcomp way

transp\_matrix = list( map( list, zip(\*matrix) ) )

print(transp\_matrix) = [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

Creating multiple lists quickly

This snippet is a really quick way to create and name up to some number of lists.

with DictComp

lists = { f'list {x}': list() for x in range(1, n+1) }

print(lists) = {'list 1': [], 'list 2': [], 'list 3': []}

Sorting Algorithm

This is the base of a sorting algorithm in its most basic form:

arr = [9, 1, 10, 8, 9, 9, 8, 7, 1, 9]

for i in range( len(arr) ):

    for j in range( i + 1, len(arr) ):

        if(arr[i] > arr[j]):

            arr[i], arr[j] = arr[j], arr[i]

print(arr) = [1, 1, 7, 8, 8, 9, 9, 9, 9, 10]

Now, it double iterates and that could consume resources, so to avoid the double iteration, this is my proposal based on slicing and python built-in functions min( ) and list method *List.*index( ):

for i in range( len(arr) - 1 ):

    li = arr[ i - len(arr) : ]

    index = li.index( min( li )) + i

    if arr[i] > arr[ index ]:

        arr[i], arr[ index ] = arr[ index ], arr[i]

Taking unique values from a list of dicts

This is a setcomp approach to pull up unique values from a list of dicts.

base = [{"V":"S001"}, {"V": "S002"}, {"VI": "S001"}, {"VI": "S005"}, {"VII":"S005"}, {"V":"S009"}, {"VIII":"S007"}]

base = { v for item in base for v in item.values() }

print(base)

Combining sub items with other item's subitems in a iterable thru looping

By 4 nested loop, here is it a way to make all the possible pairings within item's subitems within an iterable.

base = [['a', 'b'], ['g'], ['i', 'j']]

# The expected result would be:

# ag, ai, aj, bg, bi, bj, gi, gj

# This will be the combination container

comb = list()

# The looping solution would be:

for i in range( len(base) -1 ):

    for j in range( len(base[i]) ):

        for k in range( i+1, len(base) ):

            for l in range( len(base[k]) ):

                comb.append(f'{base[i][j]}{base[k][l]}')

print(comb) = ['ag', 'ai', 'aj', 'bg', 'bi', 'bj', 'gi', 'gj']

This algorithm is basically the same of using itertools.combiantions()

First Recursion Problem

This is the first problem I review with recursion in it.

'''

54. Write a Python program to get the depth of a dictionary.

    Expected Output: 4

'''

d = {'a':1, 'b': {'c': {'d': {}}}}

def dict\_depth(d):

    if isinstance(d, dict):

        return 1 + ( max( map ( dict\_depth, d.values() ) ) if d else 0 )

    return 0

print(dict\_depth(d)) = 4

A Fibonacci Generator

def fibonacci\_nums(x=0, y=1):

    yield x

    while True:

        yield y

        x, y = y, x + y

Merging dictionaries

The bitwise operator '|=' when used on dictionaries does the same as the *dict*.update*(dict)* function.

user\_base\_data = {

    'name' : 'xxxxx',

    'last\_name' : 'xxxxx',

    'phone' : 'xxxxxx',

    'address' : 'xxxx'

}

george\_data = {

    'name' : 'George',

    'last\_name' : 'Smith',

    'phone' : '555-55-55',

}

user\_base\_data |= george\_data

print(user\_base\_data) =

{

    'name' : 'George',

    'last\_name' : 'Smith',

    'phone' : '555-55-55',

    'address' : 'xxxx'

}

user\_base\_data.update(george\_data)

print(user\_base\_data)

=

{

    'name' : 'George',

    'last\_name' : 'Smith',

    'phone' : '555-55-55',

    'address' : 'xxxx'

}

Sorting with two criteria

The goal is to pick the three most repeated letters and also in alphabetical order. First will be sorted by occurrence and the alphabetically

s = 'aaeebbbccd'

d = { c : s.count(c) for c in s }

print(d)    # {'a': 2, 'b': 3, 'c': 2, 'd': 1, 'e': 1}

d = sorted(d.items(), key = lambda x: x[1], reverse = True)

print(d)    # [('b', 3), ('a', 2), ('e', 2), ('c', 2), ('d', 1)]

print(d[:3])    # [('b', 3), ('a', 2), ('e', 2)]

Here there is the problem: 'a', 'e' and 'c' have the same occurrences and if we pick the top three just like that, 'c' would be cut out, giving that 'e' should not make the cut instead of 'c'.

Here is the solution: the key kwarg give the chance to pass a tuple to consider two criteria at the same time.

d = sorted(d.items(), key = lambda x: (x[1], x[0]) , reverse = True)

print(d)    # [('b', 3), ('e', 2), ('c', 2), ('a', 2), ('d', 1)]

But the thing is that as reverse kwarg is set in True, it also affects the second criteria x[0], so it returns alphabetically second to occurrence but in ascending order and we need it in descending order.

if we try just to set reverse to False, the problem is that also affects the criteria x[1]

d = sorted(d.items(), key = lambda x: (x[1], x[0]) , reverse = False)

print(d)    # [('d', 1), ('a', 2), ('c', 2), ('e', 2), ('b', 3)]

So to overcome this issue, the solution is to change the occurrence criteria from x[1] to -x[1], that will make it be descending without the reverse kwarg

d = sorted(d.items(), key = lambda x: (-x[1], x[0]))

print(d)    # [('b', 3), ('a', 2), ('c', 2), ('e', 2), ('d', 1)]

Solving definitively the problem in case

print(d[:3])    # [('b', 3), ('a', 2), ('c', 2)]

Factorial function with recursion

def factorial(n):

    if n == 1:

        return 1

    else:

        return n \* factorial(n-1)

Unknown number of inputs for a HackerRank problem

Handling an EOFerror exception apparently works just fine

'''

Task

Given n names and phone numbers, assemble a phone book that maps friends' names to their respective phone numbers.

You will then be given an unknown number of names to query your phone book for.

For each  queried, print the associated entry from your phone book on a new line in the form name=phoneNumber;

if an entry for  is not found, print Not found instead.

Input

... After the  lines of phone book entries, there are an unknown number of lines of queries.

Each line (query) contains a  to look up, and you must continue reading lines until there is no more input.

'''

Solution

N = int(input().strip())

d = dict()

for i in range(N):

    inp = input().strip().split()

    d[inp[0]] = inp[1]

while True:

    try:

        name = input().strip()

        inp = d.get(name,'Not found')

        if inp == 'Not found':

            print(inp)

        else:

            print(f'{name}={d[name]}')

    except EOFError:

        break

How to create all possible substrings from a string

s = 'abcd'

subs = list()

#[1,2,3,4]

for i in range(1, len(s)+1):

    # R/ 4:[0,1,2,3]

    for j in range( (len(s)+1) - i ):

        subs.append(s[j:j+i])

print(subs) # R/ ['a', 'b', 'c', 'd', 'ab', 'bc', 'cd', 'abc', 'bcd', 'abcd']