**Itertools**

lunes, 20 de marzo de 2023

3:46 p. m.

Sources

Corey's tutorial - [Python Tutorial: Itertools Module - Iterator Functions for Efficient Looping](https://www.youtube.com/watch?v=Qu3dThVy6KQ)

Python Doc - <https://docs.python.org/3/library/itertools.html?highlight=itertools#module-itertools>

This module contains 3 infinite iterators, meaning that the could run for ever if next() is call upon them.

itertools.count*( start, step* ): creates an iterator that counts. - *This is an infinite iterator*

counter = itertools.count()

print(next(counter)) = 0

print(next(counter)) = 1

print(next(counter)) = 2

print(counter) = count(3)

print(type(counter)) = <class 'itertools.count'>

Just as an illustration of one use case of count iterator, if there is not way to know how many item are in a list or if they're going to be updated, a count could be useful.

data = [125, 478, 856, 936]

daily\_data = list(zip(itertools.count(), data))

print(daily\_data) = [(0, 125), (1, 478), (2, 856), (3, 936)]

Like the range function is possible to have arguments in the count function itertools.count*(star, step*)

daily\_data = list(zip(itertools.count(start=5, step=5), data))

print(daily\_data) = [(5, 125), (10, 478), (15, 856), (20, 936)]

The steps could be floating point numbers and could be negative as well.

itertools.cycle( i*terable* ): This function creates an iterable that creates a copy of the iterable and run through each element and when exhausted, starts over. - *This is an infinite iterator*

Note: *this member of the toolkit may require significant auxiliary storage (depending on the length of the iterable).*

data = ['a', 'b', 'c']

cycle\_data = itertools.cycle(data)

print(next(cycle\_data)) = a

print(next(cycle\_data)) = b

print(next(cycle\_data)) = c

print(next(cycle\_data)) = a

print(next(cycle\_data)) = b

It could be very useful is we were talking about (on/off) switches*.*

switch = itertools.cycle(['On', 'Off'])

print(next(switch)) = On

print(next(switch)) = Off

print(next(switch)) = On

itertools.repeat( *object, times = None* ): This function creates an iterable that repeats the same value indefinitely if no *times* parameter is specified. - *This is an infinite iterator*

*A common use for this function is to be meant as a support for map and zip functions, creating a stream of constants.*

repeated\_valued = itertools.repeat(2)

print(next(repeated\_valued)) = 2

print(next(repeated\_valued)) = 2

print(next(repeated\_valued)) = 2

repeated\_valued = itertools.repeat(2, times = 2)

print(next(repeated\_valued)) = 2

print(next(repeated\_valued)) = 2

print(next(repeated\_valued)) = StopIteration Exception!

This is an example from Python Doc. Regarding the use of repeat in a map function.

squares = list( map(pow, range(10), itertools.repeat(2)))

print(squares) = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

or

matches = list(range(5))

matched = list( zip(matches, itertools.repeat('x')) )

print(matched) = [(0, 'x'), (1, 'x'), (2, 'x'), (3, 'x'), (4, 'x')]

itertools.zip\_longest( *\*iterables, fillvalue= None*  ): works similar to the built-in zip with the difference that his functions matches up to the last value of the longest iterable, pairing the unpaired ones with a specified value, which by default will be None.

data = [125, 478, 856, 936]

daily\_data = list(itertools.zip\_longest(range(6),data))

print(daily\_data) = [(0, 125), (1, 478), (2, 856), (3, 936), (4, None), (5, None)]

itertools.starmap( *function, iterable* ): Works pretty similar to map() with the difference that receives on iterable that has to have paired values according to the function. In the case of pow(base, exp) it should receive tuples or list of pairs in the same manner [base, exp].

This is the same case of the squares:

squares = list( itertools.starmap( pow, [(x, 2) for x in range(5)] ) )

print(squares) = [0, 1, 4, 9, 16]

itertools.combinations( *iterable, r* ): Returns r length subsequences of elements from the *iterable* combined.

*Here the order does not count, meaning that AB is the same as BA.*

combinations = list( itertools.combinations('abcd', 2) )

print(combinations) = [('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd'), ('c', 'd')]

Or with groups of 3 elements

combinations = list( itertools.combinations('abcd', 3) )

print(combinations) = [('a', 'b', 'c'), ('a', 'b', 'd'), ('a', 'c', 'd'), ('b', 'c', 'd')]

itertools.combinations\_with\_replacement ( *iterable, r* ): Returns r length subsequences of elements from the *iterable* combined allowing individual to repeat themselves r times.

combinations = list( itertools.combinations\_with\_replacement('abcd', 2) )

print(combinations) = [('a', 'a'), ('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'b'), ('b', 'c'), ('b', 'd'), ('c', 'c'), ('c', 'd'), ('d', 'd')]

itertools.permutations( *iterable, r* ): Returns r length subsequences of elements from the *iterable* permuted.

*Here the order does matter*

permutations = list( itertools.permutations('abcd', 2) )

print(permutations) = [('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'a'), ('b', 'c'), ('b', 'd'), ('c', 'a'), ('c', 'b'), ('c', 'd'), ('d', 'a'), ('d', 'b'), ('d', 'c')]

Or with groups of 3 elements

permutations = list( itertools.permutations('abcd', 3) )

print(permutations) =

[('a', 'b', 'c'), ('a', 'b', 'd'), ('a', 'c', 'b'), ('a', 'c', 'd'), ('a', 'd', 'b'), ('a', 'd', 'c'), ('b', 'a', 'c'), ('b', 'a', 'd'), ('b', 'c', 'a'), ('b', 'c', 'd'), ('b', 'd', 'a'), ('b', 'd', 'c'), ('c', 'a', 'b'), ('c', 'a', 'd'), ('c', 'b', 'a'), ('c', 'b', 'd'), ('c', 'd', 'a'), ('c', 'd', 'b'), ('d', 'a', 'b'), ('d', 'a', 'c'), ('d', 'b', 'a'), ('d', 'b', 'c'), ('d', 'c', 'a'), ('d', 'c', 'b')]

itertools.product( *iterable, r* ): This returns the cartesian product of the iterable, or roughly means that a combiantion of up to r-lenght of elements, where each element will appear r times in one of the combinations.

numbers = [1,2,3,4]

product = itertools.product(numbers, repeat = 4 )

for i in product:

    print(i)

Console:

(1, 1, 1, 1)

(1, 1, 1, 2)

(1, 1, 1, 3)

(1, 1, 1, 4)

...

(4, 4, 3, 3)

(4, 4, 3, 4)

(4, 4, 4, 1)

(4, 4, 4, 2)

(4, 4, 4, 3)

(4, 4, 4, 4)

itertools.chain( \**iterables* ): This returns a generator containing all the elements in the order added to one single iterable and yield one by one the items.

The use case for this is, for example, if there is two or more huge iterables and to creates copies within a new iterable that concatenates the elements of all the arguments would be space inefficient. With the generator that problem won't be no more.

li1 = ['a','b','c']

li2 = (1,2,3,4)

li3 = {None, True, False}

comb = itertools.chain(li1, li2, li3)

print( [x for x in comb] ) = ['a', 'b', 'c', 1, 2, 3, 4, False, True, None]

itertools.islice( *iterable, stop | iterable, start, stop, step* ): Thisone works pretty much like the list slicing, passing firt an iterable and if there is only one additional argument, it will interpret it as the stop, but if more arguments are passed, it will change to the usual *start, stop, step*.

This is useful for the same reason of the chain() function. Say we are going through huge files with a lot of stuff, making a list to later slice it is a huge consumption of space, but with this iterator, problem solved.

itslice = itertools.islice(range(10), 2, 8, 2)

print( [x for x in itslice] ) = [2, 4, 6]

itertools.compress( *data, selectors* ): This function works as a filter, and it will return an iterator containing the data to which their respective selector are True

data = ['a','b','c','e']

selectors = [True, False, True, True]

compress = itertools.compress(data, selectors)

print([x for x in compress]) = ['a', 'c', 'e']

This also works with 1s and 0s as they correspond to True and False respectively .

data = ['a','b','c','e']

selectors = [1,0,1,1]

compress = itertools.compress(data, selectors)

print([x for x in compress]) = ['a', 'c', 'e']

itertools.filterfalse( *predicate, iterable* ): This function works also as a filter, and it will return an iterator containing the data to which their respective selector are False. But different to compress, the first positional argument is a predefined function, meaning it has to be defined and will be applied to each element of the iterable to check its truth value, if false then returned.

def fn(num):

    if num%2 == 0:

        return True

    else:

        return False

data = [x for x in range(21)]

filterfalse = itertools.filterfalse(fn, data)

print([x for x in filterfalse]) = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

data2 = [x for x in 'Ax.1e3"$sRT?']

filterfalse = itertools.filterfalse( str.isnumeric, data2 )

print([x for x in filterfalse]) = ['A', 'x', '.', 'e', '"', '$', 's', 'R', 'T', '?']

itertools.dropwhile( *predicate, iterable* ): This function works similar to filter but it will stop filtering when find a False value, from that point on, it will return the remaining values of the iterable.

Meaning that it will drop (take out) the values that returns True only until it finds one that is False.

data = [-4, -3, 17, 9, 14, 12, -7, -4, 16, 0, 19]

iterator = itertools.dropwhile(lambda x: x < 0, data)

print(list(iterator)) = [17, 9, 14, 12, -7, -4, 16, 0, 19]

Explanation: the first two elements (-4 and -3) were dropped, since are less than 0, so according to the lambda defined, they met the criteria that's why there were left out, but when reached the third element (17) which breaks the criteria, the iterator returns everything else without applying the rule.

itertools.takewhile( *predicate, iterable* ): This function works similar to filter but it will stop filtering when find a False value, from that point on, it WON'T return the remaining values of the iterable.

data = [-4, -3, 17, 9, 14, 12, -7, -4, 16, 0, 19]

iterator = itertools.takewhile(lambda x: x < 0, data)

print(list(iterator)) = [-4, -3]

itertools.accumulate( *iterable, function, initial = None* ): By default this takes the iterable elements and add one to another and returns the result, but if a function is defined is required that the function takes two arguments to be computed by the iterator.

data = [1, 2, 3, 4, 5, 6]

result = itertools.accumulate(data)

print([x for x in result]) = [1, 3, 6, 10, 15, 21]

The Python doc offers a practical example of an Amortization:

cashflows = [1000, -90, -90, -90, -90]

amortization = itertools.accumulate(cashflows, lambda bal, pmt: bal\*1.05 + pmt)

print([f'{x:.2f}' for x in amortization]) = ['1000.00', '960.00', '918.00', '873.90', '827.60']

itertools.groupby( *iterable, key = None* ): This function takes a iterable of groups or iterables and group them by a certain element called key and return a iterator of two elements tuples, where the first one will be the key and the second one an iterator containing the elements grouped.

**Note**: It's important that the base iterable is already sorted, otherwise the function would not recognize existing key to group, instead it would create a new key with the same name of the existing one.

people = [

    {

        'name': 'John Doe',

        'city': 'Gotham',

        'state': 'NY'

    },

    {

        'name': 'Jane Doe',

        'city': 'Kings Landing',

        'state': 'NY'

    },

    …

{

        'name': 'Jane Taylor',

        'city': 'Faketown',

        'state': 'NC'

    }

]

sorted\_people = [x for x in sorted(people, key = lambda x: x['state'], reverse = False)]

grouped\_people = itertools.groupby(sorted\_people, key = lambda x: x['state'] )

for k, g in grouped\_people:

    print(k, list(g))

Terminal:

CO [{'name': 'Corey Schafer', 'city': 'Boulder', 'state': 'CO'}, {'name': 'Al Einstein', 'city': 'Denver', 'state': 'CO'}]

NC [{'name': 'Nicole K', 'city': 'Asheville', 'state': 'NC'}, {'name': 'Jim Doe', 'city': 'Charlotte', 'state': 'NC'}, {'name': 'Jane Taylor', 'city': 'Faketown', 'state': 'NC'}]

NY [{'name': 'John Doe', 'city': 'Gotham', 'state': 'NY'}, {'name': 'Jane Doe', 'city': 'Kings Landing', 'state': 'NY'}]

WV [{'name': 'John Henry', 'city': 'Hinton', 'state': 'WV'}, {'name': 'Randy Moss', 'city': 'Rand', 'state': 'WV'}]

itertools.tee( *iterable, n = 2* ): This function creates n copies of the iterator avoiding side effects.