Functional Programming using itertools and functools modules in python

In this article we will take a look on two important modules of python’s standard library and various functions that helps dealing with iterables and higher-order function using functional programming method .

**Itertools**

Before talking about itertools module lets take quick look at what are iterators .

What are iterators ?

Iterator is an objects which contains countable number of objects . The most common iterator in python is List .

fruits = [‘apple’ , ‘banana’ , ‘orange’ , ‘strawberry’ ]

Coming back to itertools ,

Itertools is module in python’s standard library which is used for efficiently looping over iterators .

Types of iterator :

1. Infinite iterator : continue to run indefinitely if no stopping condition is placed
2. Finite iterator : Terminates on shortest input sequence
3. Combinatoric iterator : These iterators deals with counting that is enumeration , permutation and combination .

Infinite Iterators

There are multiple ways to iterate over a iterator . For simplicity purpose we will be working with list data structure .

Traditional way : Using a for loop

>> fruits = [‘apple’ , ‘banana’ , ‘orange’ , ‘strawberry’ ]

>> for fruit in fruits:

print(fruit)

>> apple

>> banana

>> orange

>> strawberry

1. count( ) : Returns evenly spaced values starting with number start .

syntax : itertools.count(start,step)

* start : determines the starting (initial) value
* step : determines the increment in each step

>> import itertools

# find all the odd numbers from 0 to 10

>> numbers = itertools.count(start=1 , step=2)

>> for number in numbers :

if number < 10 :

print(number , end=” “ )

else :

break

>> 1 3 5 7 9

Similar result can be achieved using a for loop :

>> for number in range(1 , 10 , 2):

print( number , end=” “ )

>> 1 3 5 7 9

1. cycle( ) : Iterates over the iterable’s element one at a time indefinitely until a termination condition is not met .

syntax : itertools.cycle(iterable)

>> import itertools

>> alphabets = [‘A’ ,’B’ ,’C’ ,’D’]

>> letters = itertools.cycle(alphabets)

>> count = 0

>> for letter in letters :

if count < 10 :

print(letter , end=” “ )

count += 1

else :

break

>> A B C D A B C D A B

1. repeat( ) : Returns object over and over again indefinitely unless the time keyword argument is specified .

syntax : itertools.repeat(object[,times])

* repeat takes an optional parameter times to specify number of times to repeat the object .

>> import itertools

# print 10 three times

>> numbers = itertools.repeat(object=10 , times=3)

>> for number in numbers :

print(number , end=” “ )

>> 10 10 10

>> import itertools

>> alphabets= itertools.repeat(“codespeedy ” , 3)

>> for alphabet in alphabets:

print(alphabet)

>> codespeedy

codespeedy

codespeedy

Finite Iterators

1. accumulate( ) : Returns the accumulated sum based on the function provided.

syntax : itertools.accumulate(iterable[,func,\*,initial=None])

* if the initial keyword argument is provided then the accumulation leads with the initial value resulting an extra element in output .

>> import itertools

>> import operator

>> foo = [ 1 , 2 , 3 , 4 , 5 ]

>> numbers = itertools.accumulate( foo , operator.add)

>> for number in numbers :

print(number , end=” “)

>> 1 3 6 10 15

How did it calculate ?

Initially it took the first value that is 1 and then ,

1 + 2 = 3

3 + 3 = 6

6 + 4 = 10

10 + 5 = 15

Roughly equivalent to (((((1+2)+3)+4)+5)+6)

With keyword argument initial :

>> import itertools

>> import operator

>> foo = [ 1 , 2 , 3 , 4 , 5 ]

>> numbers = itertools.accumulate( foo , operator.add , initial = 100)

>> for number in numbers :

print(number , end=” “)

>> 100 101 103 106 110 115

Note : 1 more element is added to the output that is the initial element 100 .

1. chain( ) : Returns element from the first iterable to the last iterable until all the iterables are exhausted .

loosely speaking ,

combines the elements of all iterable one by one into one iterable .

syntax : itertools.chain(\*iterables)

>> import itertools

>> foo =[‘Codespeedy’, ‘is’]

>> bar =[‘awesome’]

>> results = list(itertools.chain(foo, bar))

>> for result in results :

print(result , end =" ")

>> Codespeedy is awesome

1. pairwise( ) : Introduced in python 3.10 version , returns successive overlapping pairs taken from the input iterable.

In other words , starts from the element which ended in the last iteration (successive overlapping) .

syntax : itertools.pairwise(*iterable*)

>> import itertools

>> list1=[‘apple’ ,’banana’, ‘orange’]

>> result = itertools.pairwise(list1)

>> for number in result:

print(number)

>>

('apple', 'banana')

('banana', 'orange')

Combinatoric iterators

1. permutations( ) : Returns successive r length permutations of elements in an iterable .

loosely speaking , it returns arrangement of values in definite order of length r .

syntax : itertools.permutations( iterable , r = None )

* If r is not specified or is None then r defaults automatically to the length of the iterables .

>> import itertools

>> fruits=['apple','banana','orange']

>> result= itertools.permutations(fruits , 2)

>> for fruit in result:

print(fruit)

>> ('apple', 'banana')

('apple', 'orange')

('banana', 'apple')

('banana', 'orange')

('orange', 'apple')

('orange', 'banana')

1. Combinations( ) : The function selects the elements from iterable and returns their combination with r members .

syntax : itertools.combinations( iterable , r )

>> import itertools

>> fruits=['apple','banana','orange']

>> result= itertools.combinations(fruits , 2)

>> for fruit in result:

print(fruit)

>> ('apple', 'banana')

('apple', 'orange')

('banana', 'orange')

Note : permutations refers to the arrangement of the object , whereas combinations refers to the way of picking up elements (selection) from a set of objects .

**functools**

(1)cache : helps reduce the complexity of a function using an optimization technique called memoization.

What is memoization ?

Memoization is done by storing the result in cache and using it again when needed instead of computing it again and again.

syntax : @functools.cache(user functions)

>> from functools import cache

>> @cache

>> def factorial(n):

return n \* factorial(n-1) if n else 1

>> start = time.time()

>> print(factorial(100))

>> end = time.time()

>> print("Time taken for function is : ",end-start)

>> 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

Time taken for function is : 0.0016701221466064453

Without @cache :

>> from functools import cache

>> def factorial(n):

return n \* factorial(n-1) if n else 1

>> start = time.time()

>> print(factorial(100))

>> end = time.time()

>> print("Time taken for function is : ",end-start)

>> 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

Time taken for function is : 0.0005016326904296875

You can see the difference in execution time of a function with and without @cache

Note : cache decorator is smaller and faster than the lru\_cache( ) which is with a size limit .

(2)reduce : Takes a function of two arguments and applies it cumulatively from left to right of the elements of the iterable .

syntax : functools.reduce(function ,iterable[,initializer])

>> from functools

>> def sum(a,b):

return a + b

>> result = functools.reduce(sum,[1,2,3,4,5])

>> print(result)

>> 15

Note : itertools.accumulate( ) Vs functools.reduce( )

* Accumulate( ) keeps the existing value whereas reduce( ) returns only single value .
* Simply speaking , reduce( ) returns the last element of the result of accumulate( ) function .

1. singledispatch : Suppose you want a single function to perform different operations based on the argument provided .

@singledispatch decorator modifies a function behavior based on the type of a single argument provided to it

syntax : @functools.singledispatch

>> from functools import singledispatch

>> @singledispatch

def fruits(arg:str):

print(f'Do you like {arg} ?')

>> @fruits.register

def \_(arg: int):

print(f'would you like to buy {arg} kg of fruits ? ')

>> @fruits.register

def \_(arg: list):

print("your fruit basket has : ")

for fruit in arg:

print(fruit)

>> fruits(‘apple’)

>> fruits(10)

>> fruits(['apple','banana','cherry'])

>> Do you like apple ?

>> would you like to buy 10 kg of fruits ?

>> your fruit basket has :

apple

banana

cherry

There are many more functions and decorators that will help you implement functional programming .

Keep experimenting with you code and don’t forget to check the <https://docs.python.org/3/> for more details .