Exploring Python's Powerful Iterable-Generating Functions

Functions that return iterable objects as output are incredibly useful in Python programming.

In this blog, we'll explore some of the lesser-known functions from various libraries that can generate iterable objects, and discuss how to use them effectively in our code.From itertools to more obscure libraries, you'll discover a wealth of tools that can help you streamline your code and make your programming more efficient.

**Iterable Objects**

Let’s get started by knowing about “**Iterable Objects**”

In Python, an iterable object is any object that can be looped over using a for loop. It is an object that implements the iterable protocol, which requires that the object has a method called **\_\_iter\_\_()** that returns an iterator object. The iterator object, in turn, must have a method called **\_\_next\_\_()** that returns the next item in the sequence, and raises a **StopIteration** exception when there are no more items.They make it possible to write code that can operate on any type of data, as long as it can be represented as a sequence of items.This includes simple data types like lists, tuples, and strings, and more complex objects like files, databases, and network connections.

Iterables are a fundamental part of many Python libraries and frameworks, and are used extensively in data processing, machine learning, web development, and other areas of programming

Now that we have understood what iterable objects are, let’s dive into some **Python functions** that **generate iterable objects** and how to use them effectively.In Python, many functions can generate iterable objects as output. Some of these functions are built-in functions in Python, while others are part of different libraries.

**Built-in functions in Python**

Let’s discuss various **Built-in functions in Python** that provide an efficient way to work with iterable objects.

* ‘bytearray()‘-This function returns a mutable sequence of integers in the

range 0 <= x < 256, representing a bytes object.

>>>my\_bytes = bytearray([97, 98, 99])

>>>for byte in my\_bytes:

print(byte)

#Output:

>>>97

>>>98

>>>99

* zip() - This function returns an iterator that aggregates elements from each of the iterables.

>>>numbers= [1, 2, 3]

>>>letters = ['a', 'b', 'c']

>>>zipped\_list = zip(numbers, letters)

>>>for pair in zipped\_list:

print(pair)

#Output:

>>> (1, 'a')

>>> (2, 'b')

>>> (3, 'c')

* enumerate() - The enumerate() function returns an iterable object that generates pairs of indices and corresponding values from an iterable. This is useful when you need to iterate over an iterable and keep track of the index of each item

>>> my\_list = ['a', 'b', 'c']

>>> for i, value in enumerate(my\_list):

... print(i, value)

#Output:

0 a

1 b

2 c

Let's now move on to explore functions that generate iterable objects from **other libraries** in Python.

**Itertools module**

The most commonly used library for iterable objects is the **Itertools module,** which contains several functions that return iterable objects such as permutations, combinations, and product.

* compress(iterable, selector): This function takes two iterables, iterable and selector, and returns an iterator consisting of only those elements from the iterable for which the corresponding element in selector is true.

from itertools import compress

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

selectors = [True, False, True, False, True]

result = compress(data, selectors)

print(list(result))

# Output:

[1, 3, 5]

* accumulate(iterable, func=None): This function returns an iterator that produces the accumulated results of applying the func function to the elements of the iterable. If func is not specified, the default function is the addition.

(Default Use):

from itertools import accumulate

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

result = accumulate(data)

print(list(result))

# Output:

[1, 3, 6, 10, 15]

(Can be used to calculate factorial):

from itertools import accumulate

numbers = [2, 3, 4, 5]

result = accumulate(numbers, func=lambda x, y: x\*y)

print(list(result))

# Output:

[2, 6, 24, 120]

* groupby(iterable, key=None): This function takes an iterable and returns an iterator that produces (key, group) pairs, where key is a value returned by the key function (or the element itself if key is not specified), and group is an iterator containing all the elements from the iterable that have the same key.

from itertools import groupby

data = [("A", 1), ("A", 2), ("B", 3), ("B", 4), ("B", 5)]

# grouping the data based on first element of each tuple

groups = groupby(data, lambda x: x[0])

# iterating through the groups and printing the keys and items

for key, group in groups:

print(key)

for item in group:

print(item)

# Output:

A

('A', 1)

('A', 2)

B

('B', 3)

('B', 4)

('B', 5)

* permutations(iterable, r=None) : This function returns an iterator that produces all possible permutations of the elements in the iterable. The optional parameter r specifies the length of the permutations (defaults to the length of the iterable).

(With ‘r’=2 )

from itertools import permutations

data = ['A', 'B', 'C']

result = permutations(data, 2)

print(list(result))

# Output:

[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]

(For all Permutations of ‘ABC”):

from itertools import permutations

def get\_permutations(s: str):

arr = []

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

for i in permutations(s, r):

arr.append(''.join(i))

return arr

print(get\_permutations('ABC'))

#Output: ['A', 'B', 'C', 'AB', 'AC', 'BA', 'BC', 'CA', 'CB', 'ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA']

**NumPy library**

**Numpy** offers several functions to generate iterable objects for different mathematical operations, making it easier to work with large datasets and perform computations efficiently.

* ndenumerate(): This function returns an iterator yielding pairs of array coordinates and values. It is useful for iterating over multi-dimensional arrays.

import numpy as np

arr = np.array([[1, 2], [3, 4]])

for index, value in np.ndenumerate(arr):

print(f"Index: {index}, Value: {value}")

#Output:

Index: (0, 0), Value: 1

Index: (0, 1), Value: 2

Index: (1, 0), Value: 3

Index: (1, 1), Value: 4

* ndindex(): This function returns an iterator yielding tuples of indices that can be used to index into an array. It is useful for creating and iterating over multi-dimensional index arrays.

import numpy as np

# create a 2D numpy array

arr = np.array([[1, 2], [3, 4]])

# iterate over all the indices of the array

for index in np.ndindex(arr.shape):

print(index, arr[index])

#Output:

(0, 0) 1

(0, 1) 2

(1, 0) 3

(1, 1) 4

* fromiter(): This function returns an iterator that iterates over elements from an iterable object and returns an array.

import numpy as np

it = iter(range(5))

arr = np.fromiter(it, dtype=int)

print(arr)

#Output:

[0 1 2 3 4]

**Pandas Library**

Besides the functions we already discussed, **Pandas** also has several other functions like iterrows(), itertuples(), groupby(), etc. that generate iterable objects as outputs.

* iterrows(): This function returns an iterator that yields pairs of index and row data as pandas Series objects. Here's an example:

import pandas as pd

df = pd.DataFrame({'Name': ['John', 'Sara', 'David'],

'Age': [30, 25, 40],

'Country': ['USA', 'Canada', 'UK']})

for index, row in df.iterrows():

print(index, row['Name'], row['Age'], row['Country'])

#Output

0 John 30 USA

1 Sara 25 Canada

2 David 40 UK

* itertuples(): This function returns an iterator that yields namedtuples for each row in a DataFrame. Here's an example:

import pandas as pd

df = pd.DataFrame({'Name': ['John', 'Sara', 'David'],

'Age': [30, 25, 40],

'Country': ['USA', 'Canada', 'UK']})

for row in df.itertuples():

print(row.Name, row.Age, row.Country)

#Output:  
John 30 USA

Sara 25 Canada

David 40 UK

* groupby(): This function groups the DataFrame using one or more columns and returns a GroupBy object. The GroupBy object is an iterable that yields tuples containing the group name and the corresponding rows. Here's an example:

import pandas as pd

df = pd.DataFrame({'Name': ['John', 'Sara', 'David', 'Mary', 'Steve'],

'Age': [30, 25, 40, 28, 35],

'Country': ['USA', 'Canada', 'UK', 'USA', 'UK']})

groups = df.groupby('Country')

for name, group in groups:

print(name)

print(group)

#Output:

Canada

Name Age Country

1 Sara 25 Canada

UK

Name Age Country

2 David 40 UK

4 Steve 35 UK

USA

Name Age Country

0 John 30 USA

3 Mary 28 USA

**Scipy Library**

**Scipy** is a popular library used for scientific computing and technical computing in Python. It provides several functions that generate iterable objects as outputs, making it easier to perform complex calculations and analysis.

* The **combinations()** function of Scipy generates all possible combinations of a given sequence. For example:

from scipy.special import comb

arr = [1, 2, 3, 4]

combos = comb(arr, 2)

print(list(combos))

#Output:

[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]

* The **permutations()** function generates all possible permutations of a given sequence. For example:

from scipy.special import perm

arr = [1, 2, 3]

perms = perm(arr)

print(list(perms))

#Output:  
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]

* **distance\_matrix**: This function from the **scipy.spatial** library returns an iterable object that generates a two-dimensional array containing the pairwise distances between a set of points.

The **distance\_matrix** function takes two required arguments: **X** and **Y**, which are arrays of shape **(n\_samples\_X, n\_features)** and **(n\_samples\_Y, n\_features)**, respectively. It returns a two-dimensional array of shape **(n\_samples\_X, n\_samples\_Y)** where the element at index **(i, j)** is the distance between the i-th row of **X** and the jth row of **Y**.

Example:

from scipy.spatial import distance\_matrix

# Create an array of points in 2D space

points = [[0, 0], [1, 1], [2, 2]]

# Calculate the pairwise Euclidean distances between the points

distances = distance\_matrix(points, points)

# Print the resulting distances array

print(distances)

#Output:

[[0. 1.41421356 2.82842712]

[1.41421356 0. 1.41421356]

[2.82842712 1.41421356 0. ]]

**Collections**

The **collections** module in Python provides alternative implementations of several built-in types with additional features. It also provides some useful data structures such as deque, defaultdict, OrderedDict, and Counter.

Here are some examples of iterable generating functions from the **collections** module:

* deque(): The deque function creates a double-ended queue which supports adding and removing elements from both ends in O(1) time complexity. It returns an iterable object that allows us to iterate over the elements of the deque.

Example:

from collections import deque

my\_deque = deque([1, 2, 3, 4])

for item in my\_deque:

print(item)

#Output:  
1

2

3

4

* defaultdict(): The defaultdict function returns a dictionary-like object with a default value for missing keys. It takes a factory function as an argument which is used to provide a default value for any missing keys.

Example:

from collections import defaultdict

my\_dict = defaultdict(int)

my\_dict["a"] = 1

my\_dict["b"] = 2

for key, value in my\_dict.items():

print(key, value)

#Output:

a 1

b 2

* OrderedDict(): The OrderedDict function returns a dictionary-like object that maintains the order of keys as they are added. It returns an iterable object that allows us to iterate over the key-value pairs in the order in which they were added.

Example:  
from collections import OrderedDict

my\_dict = OrderedDict()

my\_dict["c"] = 3

my\_dict["b"] = 2

my\_dict["a"] = 1

for key, value in my\_dict.items():

print(key, value)

#Output:  
c 3

b 2

a 1

* Counter(): The Counter function returns a dictionary-like object that counts the number of occurrences of each element in a list. It returns an iterable object that allows us to iterate over the unique elements and their counts.

Example:

from collections import Counter

my\_list = [1, 2, 3, 2, 1, 2, 3, 1]

my\_counter = Counter(my\_list)

for key, value in my\_counter.items():

print(key, value)

#Output:

1 3

2 3

3 2