The zip function in Python

In Python and all programming languages, we can use `for` loops and `while` loops to iterate over arrays. Iterating over a single array is very simple. However, when it comes to iterating over multiple arrays together, things start becoming complex. If the size of all the arrays is the same, then it is a straightforward job. However, if array sizes are different, we have to make sure that we only consider the length of the smallest array to avoid errors and exceptions.

Python makes this task a lot easier. Instead of manually writing logic for iterating over arrays of different sizes, we can use an in-built utility, more precisely a method, provided by Python. This method is the `zip()` method.

In this article, we will learn how about the `zip()` method and how to use it.

The `zip()` method in Python

The `zip()` method accepts iterable objects such as lists, strings, and tuples as arguments and returns a single iterable; object that contains values from all the passed objects.

The returned iterable object has the length of the smallest iterables. For example, if two lists of size `5` and `10` are provided to the `zip()` function, the returned iterable object will have a length of `5`. This means that only the second list's first `5` elements will be a part of the iterable object. If an empty iterable or no iterable object is given to this function, it returns an empty iterable as well.

Now that we are done with the theory let us see how to use this method. Refer to the following examples to understand its usage.

No iterable object

In the following Python code, no iterable object will be passed to the `zip()` method.

```
result = zip()

for x in result:
    print(x)
```

Nothing will be printed on the console when we execute the above code. The reason behind the same is simple; if no iterable object were provided, we would have nothing to iterate over. Hence, an empty iterable object is returned.

Iterable objects of same lengths

In the following Python code, a tuple of integers, a list of floating values, a list of class objects, and a string of equal lengths will be passed to the `zip()` method.

```
```

```
class Number:
    def __init__(self, number):
        self.number = number

    def square(self):
        return number ** 2

    def __repr__(self):
        return f"Number({self.number})"

a = (11, 22, 33, 44, 55)
b = [1.1, 2.2, 3.3, 4.4, 5.5]
c = [Number(1), Number(23), Number(44.44), Number(0), Number(-9)]
d = "Hello"
result = zip(a, b, c, d)

for x in result:
    print(x)
```

Output:

```
(11, 1.1, Number(1), 'H')
(22, 2.2, Number(23), 'e')
(33, 3.3, Number(44.44), 'l')
(44, 4.4, Number(0), 'l')
(55, 5.5, Number(-9), 'o')
```

As we can see, the `zip()` method stores values across all the iterable objects together in tuples. The order of values inside the tuples is the same as the order in which their iterable objects were provided to the `zip()` function.

We can destructure or unpack these values using within the `for` loop for easy access. Refer to the following code for the same.

```
class Number:
    def __init__(self, number):
        self.number = number

    def square(self):
        return number ** 2

    def __repr__(self):
        return f"Number({self.number})"

a = (11, 22, 33, 44, 55)
b = [1.1, 2.2, 3.3, 4.4, 5.5]
c = [Number(1), Number(23), Number(44.44), Number(0), Number(-9)]
d = "Hello"
```

```
result = zip(a, b, c, d)

for p, q, r, s in result:
    print("A:", p)
    print("B:", q)
    print("C:", r)
    print("D:", s)
```

Output:

```
A: 11
B: 1.1
C: Number(1)
D: H
A: 22
B: 2.2
C: Number(23)
D: e
A: 33
B: 3.3
C: Number(44.44)
D: l
A: 44
B: 4.4
C: Number(0)
D: l
A: 55
B: 5.5
C: Number(-9)
D: o
```

Instead of using `for` loops, we can also iterable over an iterable object using a `while` loop. With `while` loops, we would require two additional things, the `next()` method and a `try-except` block. The `next()` method will be used to get values out of the iterable object returned by the `zip()` method, and the `try-except` block will be used to stop the iteration. Refer to the following Python code for the same.

```
class Number:
    def __init__(self, number):
        self.number = number

    def square(self):
        return number ** 2

    def __repr__(self):
        return f"Number({self.number})"

a = (11, 22, 33, 44, 55)
b = [1.1, 2.2, 3.3, 4.4, 5.5]
c = [Number(1), Number(23), Number(44.44), Number(0), Number(-9)]
```

```
d = "Hello"
result = zip(a, b, c, d)

while True:
    try:
        p, q, r, s = next(result)
        print("A:", p)
        print("B:", q)
        print("C:", r)
        print("D:", s)
    except StopIteration:
        break
```

Output:

```
A: 11
B: 1.1
C: Number(1)
D: H
A: 22
B: 2.2
C: Number(23)
D: e
A: 33
B: 3.3
C: Number(44.44)
D: l
A: 44
B: 4.4
C: Number(0)
D: l
A: 55
B: 5.5
C: Number(-9)
D: o
```

When no values are available inside an iterator, it raises a `StopIteration` exception. Using a `try-except` block, we catch this exception and exit the infinite `while` loop.

Iterable objects of different lengths

In the following Python code, a tuple of integers, a list of floating values, a list of class objects, and a string of different lengths will be passed to the `zip()` method.

```
class Number:
    def __init__(self, number):
        self.number = number

    def square(self):
```

```
        return number ** 2

    def __repr__(self):
        return f"Number({self.number})"

a = (11, 22, 33)
b = [1.1, 2.2, 3.3, 4.4]
c = [Number(1), Number(23), Number(44.44), Number(0), Number(-9)]
d = "HelloWorld"
result = zip(a, b, c, d)

for p, q, r, s in result:
    print("A:", p)
    print("B:", q)
    print("C:", r)
    print("D:", s)
```

Output:

```
A: 11
B: 1.1
C: Number(1)
D: H
A: 22
B: 2.2
C: Number(23)
D: e
A: 33
B: 3.3
C: Number(44.44)
D: l
```

All the iterable objects have different lengths. The first iterable object or the tuple of integers has the smallest length, that is, `3`. Hence, the output only prints the first `3` values from all the iterable objects.

Creating a dictionary

We can create a dictionary of key-value pairs with the help of the `zip()` method. The idea is to create an iterator of two arrays of the same lengths, containing keys and their respective values, and mapping them to each other inside a dictionary while iterating over the returned iterable object. Refer to the following code for the same.

```
import json

a = ["W", "O", "R", "L", "D"]
b = [1.1, True, "Hello", None, 5]
result = zip(a, b)
mapping = {x : y for x, y in result}
print(json.dumps(mapping, indent = 4))
```

Output:

```
{
    "W": 1.1,
    "O": true,
    "R": "Hello",
    "L": null,
    "D": 5
}
```

The above code only uses the `json` module to beautify the dictionary's output. Note that using it is entirely optional.

Using the `zip()` function along with the `enumerate()` function

The `enumerate()` function is used to get the index and the value at the same time while iterating over an iterable object. Since the `zip()` function returns an iterator, we can club the two methods together and have access to not only the indexes but also the values. Refer to the following Python code for the same.

```
class Number:
    def __init__(self, number):
        self.number = number

    def square(self):
        return number ** 2

    def __repr__(self):
        return f"Number({self.number})"

a = (11, 22, 33)
b = [1.1, 2.2, 3.3, 4.4]
c = [Number(1), Number(23), Number(44.44), Number(0), Number(-9)]
d = "HelloWorld"
result = zip(a, b, c, d)

for i, (p, q, r, s) in enumerate(result):
    print(f"A{i + 1}:", p)
    print(f"B{i + 1}:", q)
    print(f"C{i + 1}:", r)
    print(f"D{i + 1}:", s)
```

Output:

```
A1: 11
B1: 1.1
C1: Number(1)
```

```
D1: H
A2: 22
B2: 2.2
C2: Number(23)
D2: e
A3: 33
B3: 3.3
C3: Number(44.44)
D3: l
```

In the above Python code, inside the `for` loop, `i, (p, q, r, s)` unpack the values returned by the `enumerate()` function, and `(p, q, r, s)` unpack the values returned by the `zip()` function.

The values returned by the `enumerate()` function are in the following format.

```
(0, (11, 1.1, Number(1), 'H'))
(1, (22, 2.2, Number(23), 'e'))
(2, (33, 3.3, Number(44.44), 'l'))
```

This will make it more clear why `i, (p, q, r, s)` was used to unpack all the values.