PYTHON FEATURES

Contents

Python Zip Function—Real Python 3

Passing Arguments of Unequal Length 3

Comparing zip() in Python 3 and 2 3

Looping Over Multiple Iterables 4

Unzipping a Sequence 4

Calculating in Pairs 5

Building Dictionaries 5

Threading in Python—Real Python 6

Python Decorators—Real Python 7

Heading 8

References 10

Python Zip Function—Real Python

Zip function is a bulitin. It aggregates elements from each of the iterables passed to it. It returns a tuple of the elements of the iterables. Stops bundling when the shortest length iterables is exhausted. For lists of length 5,7,8 the tuple will be of length 5.

Python's zip() function is defined as zip(*iterables). The function takes in iterables as arguments and returns an iterator. This iterator generates a series of tuples containing elements from each iterable. zip() can accept any type of iterable, such as files, lists, tuples, dictionaries, sets, and so on.

If you use zip() with n arguments, then the function will return an iterator that generates tuples of length n.

Zip itself returns an iterator, which must be consumed before being used.

```
>>> numbers = [1, 2, 3]
>>> letters = ['a', 'b', 'c']
>>> zipped = zip(numbers, letters)
>>> zipped  # Holds an iterator object
<zip object at 0x7fa4831153c8>
>>> type(zipped)
<class 'zip'>
>>> list(zipped)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Related links: zip(*iterables),

(python.org),

If you're working with sequences like lists, tuples, or strings, then your iterables are guaranteed to be evaluated from left to right. However, for other types of iterables (like sets), you might see some weird results. Set objects don't keep their elements in any particular order. This means that the tuples returned by zip() will have elements that are paired up randomly. If you're going to use the Python zip() function with unordered iterables like sets, then this is something to keep in mind.

Zips are generator objects, you can call next() on them to retrieve items. They raise StopIteration at the end.

```
>>> a = [1,2,3]
>>> b = [5,6,7]
>>> c = zip(a,b)
>>> c
<zip object at 0x000002434111B180>
>>> next(c)
(1, 5)
```

Passing Arguments of Unequal Length

When you're working with the Python zip() function, it's important to pay attention to the length of your iterables. It's possible that the iterables you pass in as arguments aren't the same length. In these cases, the number of elements that zip() puts out will be equal to the length of the shortest iterable. The remaining elements in any longer iterables will be totally ignored by zip()

```
>>> list(zip(range(5), range(100)))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

Since 5 is the length of the first (and shortest) range() object, zip() outputs a list of five tuples. There are still 95 unmatched elements from the second range() object. These are all ignored by zip() since there are no more elements from the first range() object to complete the pairs.

If trailing or unmatched values are important to you, then you can use itertools.zip_longest() instead of zip(). With this function, the missing values will be replaced with whatever you pass to the fillvalue argument (defaults to None). The iteration will continue until the longest iterable is exhausted: Here, you use itertools.zip_longest() to yield five tuples with elements from letters, numbers, and longest. The iteration only stops when longest is exhausted. The missing elements from numbers and letters are filled with a question mark ?, which is what you specified with fillvalue.

```
>>> from itertools import zip_longest
>>> numbers = [1, 2, 3]
>>> letters = ['a', 'b', 'c']
>>> longest = range(5)
>>> zipped = zip_longest(numbers, letters, longest, fillvalue='?')
>>> list(zipped)
[(1, 'a', 0), (2, 'b', 1), (3, 'c', 2), ('?', '?', 3), ('?', '?', 4)]
```

Comparing zip() in Python 3 and 2

In Python 2 zip returns a list and in 3 it returns a zip object, which is an iterator.

Looping Over Multiple Iterables

Looping over multiple iterables is one of the most common use cases for Python's zip() function. If you need to iterate through multiple lists, tuples, or any other sequence, then it's likely that you'll fall back on zip().Python's zip() function allows you to iterate in parallel over two or more iterables. Since zip() generates tuples, you can unpack these in the header of a for loop.

```
>>> letters = ['a', 'b', 'c']
>>> numbers = [0, 1, 2]
>>> operators = ['*', '/', '+']
>>> for l, n, o in zip(letters, numbers, operators):
...     print(f'Letter: {l}')
...     print(f'Number: {n}')
...     print(f'Operator: {o}')
...
Letter: a
Number: 0
Operator: *
Letter: b
Number: 1
Operator: /
Letter: c
Number: 2
Operator: +
```

Here, you iterate through the series of tuples returned by zip() and unpack the elements into l, o, and n. When you combine zip(), for loops, and tuple unpacking, you can get a useful and Pythonic idiom for traversing two or more iterables at once.

In Python 3.6 and beyond, dictionaries are ordered collections, meaning they keep their elements in the same order in which they were introduced.

```
>>> dict_one = {'name': 'John', 'last_name': 'Doe', 'job': 'Python Consultant'}
>>> dict_two = {'name': 'Jane', 'last_name': 'Doe', 'job': 'Community Manager'}
>>> for (k1, v1), (k2, v2) in zip(dict_one.items(), dict_two.items()):
...     print(k1, '->', v1)
...     print(k2, '->', v2)
...
name -> John
name -> Jane
last_name -> Doe
last_name -> Doe
job -> Python Consultant
job -> Community Manager
```

Here, you iterate through dict_one and dict_two in parallel. In this case, zip() generates tuples with the items from both dictionaries. Then, you can unpack each tuple and gain access to the items of both dictionaries at the same time.

Unzipping a Sequence

The opposite of zip() is zip(). Zip can be used to unzip with * because it zips things into separate iterables by aggregating them in the same order they were packed in. Zip is a bijective function, and much like a packed up box, things get out the way they were packed in.

```
>>> pairs = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
>>> numbers, letters = zip(*pairs)
>>> numbers
(1, 2, 3, 4)
>>> letters
('a', 'b', 'c', 'd')
```

Sorting in Parallel

Sorting is a common operation in programming. Suppose you want to combine two lists and sort them at the same time. To do this, you can use zip() along with .sort().

```
>>> letters = ['b', 'a', 'd', 'c']
>>> numbers = [2, 4, 3, 1]
>>> data1 = list(zip(letters, numbers))
```

```
>>> data1
[('b', 2), ('a', 4), ('d', 3), ('c', 1)]
>>> data1.sort()  # Sort by letters
>>> data1
[('a', 4), ('b', 2), ('c', 1), ('d', 3)]
>>> data2 = list(zip(numbers, letters))
>>> data2
[(2, 'b'), (4, 'a'), (3, 'd'), (1, 'c')]
>>> data2.sort()  # Sort by numbers
>>> data2
[(1, 'c'), (2, 'b'), (3, 'd'), (4, 'a')]
```

In this example, you first combine two lists with zip() and sort them. Notice how data1 is sorted by letters and data2 is sorted by numbers. List of tuples get sorted by the first item in the tuple.

```
>>> letters = ['b', 'a', 'd', 'c']
>>> numbers = [2, 4, 3, 1]
>>> data = sorted(zip(letters, numbers))  # Sort by letters
>>> data
[('a', 4), ('b', 2), ('c', 1), ('d', 3)]
```

You can also use sorted() and zip() together to achieve a similar result. In this case, sorted() runs through the iterator generated by zip() and sorts the items by letters, all in one go. This approach can be a little bit faster since you'll need only two function calls: zip() and sorted(). With sorted(), you're also writing a more general piece of code.

Calculating in Pairs

You can use the Python `zip()` function to make some quick calculations. Suppose you have the following data in a spreadsheet:

Element/Month	January	February	March
Total Sales	52,000.00	51,000.00	48,000.00
Production Cost	46,800.00	45,900.00	43,200.00

```
>>> total_sales = [52000.00, 51000.00, 48000.00]
>>> prod_cost = [46800.00, 45900.00, 43200.00]
>>> for sales, costs in zip(total_sales, prod_cost):
...     profit = sales - costs
...     print(f'Total profit: {profit}')
...
Total profit: 5200.0
Total profit: 5100.0
Total profit: 4800.0
```

Here, you calculate the profit for each month by subtracting costs from sales. Python's zip() function combines the right pairs of data to make the calculations. You can generalize this logic to make any kind of complex calculation with the pairs returned by zip().

Building Dictionaries

To build a dictionary from two different but closely related sequences, a convenient way is to use dict() and zip() together.

```
>>> fields = ['name', 'last_name', 'age', 'job']
>>> values = ['John', 'Doe', '45', 'Python Developer']
>>> a_dict = dict(zip(fields, values))
>>> a_dict
{'name': 'John', 'last_name': 'Doe', 'age': '45', 'job': 'Python Developer'}
>>> new_job = ['Python Consultant']
>>> field = ['job']
>>> a_dict.update(zip(field, new_job))
>>> a_dict
{'name': 'John', 'last_name': 'Doe', 'age': '45', 'job': 'Python Consultant'}
```

Here, you create a dictionary that combines the two lists. zip(fields, values) returns an iterator that generates 2-items tuples. If you call dict() on that iterator, then you'll be building the dictionary you need. The elements of fields become the dictionary's keys, and the elements of values represent the values in the dictionary. You can also update an existing dictionary by combining zip() with dict.update().

Threading in Python—Real Python

Python Decorators—Real Python

Heading

Some text

References

There are no sources in the current document.