zip(*iterables)

Make an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator. Equivalent to:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n)`. This repeats the *same* iterator `n` times so that each output tuple has the result of `n` calls to the iterator. This has the effect of dividing the input into n-length chunks.

`zip()` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use `itertools.zip_longest()` instead.

`zip()` in conjunction with the `*` operator can be used to unzip a list:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

enumerate(*iterable*, *start=0*)

Return an enumerate object. *iterable* must be a sequence, an iterator, or some other object which supports iteration. The `__next__()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

4.7.3. Arbitrary Argument Lists ¶

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see Tuples and Sequences). Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Variable Length of Parameters

We will introduce now functions, which can take an arbitrary number of arguments. Those who have some programming background in C or C++ know this from the varargs feature of these languages.

Some definitions, which are not really necessary for the following: A function with an arbitrary number of arguments is usually called a variadic function in computer science. To use another special term: A variadic function is a function of indefinite arity. The arity of a function or an operation is the number of arguments or operands that the function or operation takes. The term war derived from words like "unary", "binary", "ternary", all ending in "ary".

The asterisk "*" is used in Python to define a variable number of arguments. The asterisk character has to precede a variable identifier in the parameter list.

```
>>> def varpafu(*x): print(x)
...
>>> varpafu()
()
>>> varpafu(34,"Do you like Python?", "Of course")
(34, 'Do you like Python?', 'Of course')
>>>
```

We learn from the previous example that the arguments passed to the function call of varpafu() are collected in a tuple, which can be accessed as a "normal" variable x within the body of the function. If the function is called without any arguments, the value of x is an empty tuple.

Sometimes, it's necessary to use positional parameters followed by an arbitrary number of parameters in a function definition. This is possible, but the positional parameters always have to precede the arbitrary parameters. In the following example, we have a positional parameter "city", - the main location, - which always have to be given, followed by an arbitrary number of other locations:

```
>>> def locations(city, *other_cities): print(city, other_cities)
...
>>> locations("Paris")
Paris ()
>>> locations("Paris", "Strasbourg", "Lyon", "Dijon", "Bordeaux", "Marseille")
Paris ('Strasbourg', 'Lyon', 'Dijon', 'Bordeaux', 'Marseille')
>>>
```