A Guide to Python Itertools

medium.com/@jasonrigden/a-guide-to-python-itertools-82e5a306cdf8

Jason Rigden August 1, 2018

Those iterables are more powerful than you can possibly imagine.

> Check out my podcast, "Talking Cryptocurrency" where I interview with the people making the cryptocurrency and blockchain revolution happen. Guests have ranged from solo devs to CEOs. These are quick 15–20 minutes episodes. Read my Podcast Manifesto or listen to the show now.

What are iterables?

The Python `itertools` module is a collection of tools for handling iterators. Simply put, iterators are data types that can be used in a `for` loop. The most common iterator in Python is the list.

```
colors = ['red', 'orange', 'yellow', 'green']
```

In this example we have created a list of strings. We have named this list `colors`.

We can use a `for` loop to iterate, or step through this list. The code below will print out each item in the list.

```
for each in colors:    print(each)
```

There are many ifferent kinds of iterables but for this tutorial we will be using lists.

Requirements

We must import the `itertools` module before we can use it. We will also import the `operator` module. This module is not necessary when using `itertools`, it is only needed for some of the examples below.

All the following examples will have these imports implied.

```
import itertoolsimport operator
```

The Itertools Module

This module is a collection of functions. We are going to explore each one of these function. If you have any questions, suggestion, or correction, please put them in the comments below. I address them as soon as possible.

accumulate()

```
itertools.accumulate(iterable[, func])
```

This function makes an iterator that returns the results of a function. Functions can be passed around very much like variables. The `accumulate()` function takes a function as an argument. It also takes an iterable. It returns the accumulated results. The results are themselves contained in an iterable. This may all sound very confusing. I assure you that, when you play with the code it will make sense.

Code

```
data = [1, 2, 3, 4, 5]result = itertools.accumulate(data, operator.mul)
for each in result:
    print(each)
```

Output

```
1
2
6
24
120
```

The operator.mul takes two numbers and multiplies them.

```
operator.mul(1, 2)
2
operator.mul(2, 3)
6
operator.mul(6, 4)
24
operator.mul(24, 5)
120
```

In this next example will will use the `max` function.

Code

```
data = [5, 2, 6, 4, 5, 9, 1]result = itertools.accumulate(data, max)
for each in result:
    print(each)
```

Output

```
5
5
6
6
6
9
9
```

The `max` function returns the largest item.

```
5
max(5, 2)
5
max(5, 6)
6
max(6, 4)
6
max(6, 5)
6
max(6, 9)
9
max(9, 1)
9
```

Passing a function is optional.

Code

```
data = [5, 2, 6, 4, 5, 9, 1]result = itertools.accumulate(data)
for each in result:
    print(each)
```

Output

```
5
7
13
17
22
31
32
```

If no function is designated the items will be summed.

```
5
5 + 2 = 7
7 + 6 = 13
13 + 4 = 17
17 + 5 = 22
22 + 9 = 31
31 + 1 = 32
```

combinations()

```
itertools.combinations(iterable, r)
```

This function takes an iterable and a integer. This will create all the unique combination that have `r` members.

Code

```
shapes = ['circle', 'triangle', 'square',]result = itertools.combinations(shapes,
2)for each in result:
    print(each)
```

In this code we make all combos with 2 members.

Output

```
('circle', 'triangle')
('circle', 'square')
('triangle', 'square')
```

Code

```
shapes = ['circle', 'triangle', 'square',]result = itertools.combinations(shapes,
3)for each in result:
    print(each)
```

In this code we make all combos with 3members. It is a bit less exciting.

Output

```
('circle', 'triangle', 'square')
```

combinations_with_replacement()

```
itertools.combinations_with_replacement(iterable, r)
```

This one is just like the `combinations()` function, but this one allows individual elements to be repeated more than once.

Code

```
shapes = ['circle', 'triangle', 'square',]result =
itertools.combinations_with_replacement(shapes, 2)
for each in result:
    print(each)
```

Output

```
('circle', 'circle')
('circle', 'triangle')
('circle', 'square')
('triangle', 'triangle')
('triangle', 'square')
('square', 'square')
```

count()

```
itertools.count(start=0, step=1)
```

Makes an iterator that returns evenly spaced values starting with number start.

Code

```
for i in itertools.count(10,3):
    print(i)
    if i > 20:
        break
```

In the above code, we iterate or loop over a function. We tell the function to start at 10 and step 3.

Output

```
10
13
16
19
22
```

This first iteration has the value 10. In the next step we step or add 3. This has the value of 13. We do the same thing for the next iteration and get a value of 16. This would continue forever but, we have added a break.

cycle()

```
itertools.cycle(iterable)
```

This function cycles through an iterator endlessly.

Code

```
colors = ['red', 'orange', 'yellow', 'green', 'blue', 'violet']for color in
itertools.cycle(colors):
    print(color)
```

In the above code, we create a list. Then we cycle or loop through this list endlessly. Normally, a `for` loop steps through an iterable until it reached the end. If a list has 3 items, the loop will iterate 3 times. But not if we use the `cycle()` function. With this function, when we reach the end of the iterable we start over again from the beginning.

Output

```
red
orange
yellow
green
blue
indigo
violet
red
orange
yellow
green
...
```

I have truncated the endless output above with ellipses.

chain()

```
itertools.chain(*iterables)
```

This function takes a series of iterables and return them as one long iterable.

Code

```
colors = ['red', 'orange', 'yellow', 'green', 'blue']
shapes = ['circle', 'triangle', 'square', 'pentagon']result =
itertools.chain(colors, shapes)
for each in result:
    print(each)
```

Output

```
red
orange
yellow
green
blue
circle
triangle
square
pentagon
```

compress()

```
itertools.compress(data, selectors)
```

This function filters one iterable with another.

Code

```
shapes = ['circle', 'triangle', 'square', 'pentagon']
selections = [True, False, True, False]result = itertools.compress(shapes,
selections)
for each in result:
    print(each)
```

Output

```
circle
square
```

dropwhile()

```
itertools.dropwhile(predicate, iterable)
```

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element.

Code

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]result = itertools.dropwhile(lambda x:
x<5, data)
for each in result:
    print(each)
```

Output

```
5
6
7
8
9
10
1
```

Ok. This is can be confusing. The code says to drop each item while the item is less than 5. After it encounters an item that is not less than 5, it returns the rest. That is why that last one is returned.

Step Through It

```
1 < 5:  True,  drop
2 < 5:  True,  drop
3 < 5:  True,  drop
4 < 5:  True,  drop
5 < 5:  False, return surviving items
```

filterfalse()

```
itertools.filterfalse(predicate, iterable)
```

This function makes an iterator that filters elements from iterable returning only those for which the predicate is `False`.

Code

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result = itertools.filterfalse(lambda x: x<5, data)
for each in result:
    print(each)
```

Output

```
5
6
7
8
9
10
```

Step Through It

```
1 < 5:  True,  drop
2 < 5:  True,  drop
3 < 5:  True,  drop
4 < 5:  True,  drop
5 < 5:  False, keep
6 < 5:  False, keep
7 < 5:  False, keep
8 < 5:  False, keep
9 < 5:  False, keep
10 < 5:  False, keep
```

groupby()

```
itertools.groupby(iterable, key=None)
```

Simply put, this function groups things together. Ok. This one is complicated. And the example is a bit long.

Code

```
robots = [{
    'name': 'blaster',
    'faction': 'autobot'
}, {
    'name': 'galvatron',
    'faction': 'decepticon'
}, {
    'name': 'jazz',
    'faction': 'autobot'
}, {
    'name': 'metroplex',
    'faction': 'autobot'
}, {
    'name': 'megatron',
    'faction': 'decepticon'
}, {
    'name': 'starcream',
    'faction': 'decepticon'
}]for key, group in itertools.groupby(bots, key=lambda x: x['faction']):
    print(key)
    print(list(group))
```

Output

```
autobot
[{'faction': 'autobot', 'name': 'blaster'}, {'faction': 'autobot', 'name':
'jazz'}, {'faction': 'autobot', 'name': 'metroplex'}]
decepticon
[{'faction': 'decepticon', 'name': 'galvatron'}, {'faction': 'decepticon', 'name':
'megatron'}, {'faction': 'decepticon', 'name': 'starcream'}]
```

islice()

```
itertools.islice(iterable, start, stop[, step])
```

This function is very much like slices. This function allows you to cut out a piece of an iterable.

Code

```
colors = ['red', 'orange', 'yellow', 'green', 'blue',]
few_colors = itertools.islice(colors, 2)
for each in few_colors:
    print(each)
```

Output

```
red
orange
```

permutations()

```
itertools.permutations(iterable, r=None)
```

Code

```
alpha_data = ['a', 'b', 'c']result = itertools.permutations(alpha_data)
for each in result:
    print(each)
```

Output

```
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
```

product()

This function creates the cartesian products from a series of iterables.

Code

```
num_data = [1, 2, 3]
alpha_data = ['a', 'b', 'c']result = itertools.product(num_data, alpha_data)
for each in result:
    print(each)
```

Output

```
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
```

Imagine a table like so:

```
      a     b      c
1     a1    b1     c1
2     a2    b2     c3
3     a3    b3     b3
```

repeat()

```
itertools.repeat(object[, times])
```

This function will repeat an object over and over again. Unless, there is a `times` argument.

Code

```
for i in itertools.repeat("spam"):
    print(i)
```

In the above code, we create an iterable that just repeats `spam` over and over again. It will do this endlessly. This is how infinite `spam` is made.

Output

```
spam
spam
spam
spam
spam
spam
...
```

I have truncated the endless output above with ellipses.

Code

```
for i in itertools.repeat("spam", 3):
    print(i)
```

If we use the `times` argument, we can limit the number of times it will repeat.

Output

```
spam
spam
spam
```

In this example `spam` only repeats three times.

starmap()

```
itertools.starmap(function, iterable)
```

This function makes an iterator that computes the function using arguments obtained from the iterable. Let us take a looky.

Code

```
data = [(2, 6), (8, 4), (7, 3)]
result = itertools.starmap(operator.mul, data)
for each in result:
    print(each)
```

Output

```
12
32
21
```

Step Through

```
operator.mul(2, 6)
12
operator.mul(8, 4)
32
operator.mul(7, 3)
21
```

takewhile()

```
itertools.takwwhile(predicate, iterable)
```

This is kind of the opposite of `dropwhile()`. This function makes an iterator and returns elements from the iterable as long as the predicate is true.

Code

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]result = itertools.takewhile(lambda x:
x<5, data)
for each in result:
    print(each)
```

Output

```
1
2
3
4
```

Step Through It

```
1 < 5:  True,  keep going
2 < 5:  True,   keep going
3 < 5:  True,   keep going
4 < 5:  True,   keep going
5 < 5:  False, stop and drop
```

tee()

```
itertools.tee(iterable, n=2)
```

Return n independent iterators from a single iterable.

Code

```
colors = ['red', 'orange', 'yellow', 'green', 'blue']
alpha_colors, beta_colors = itertools.tee(colors)for each in alpha_colors:
    print(each)print('..')for each in beta_colors:
     print(each)
```

The default is 2, but you can make as many as needed.

Output

```
red
orange
yellow
green
blue
..
red
orange
yellow
green
blue
```

zip_longest()

```
itertools.zip_longest(*iterables, fillvalue=None)
```

This function makes an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted.

Code

```
colors = ['red', 'orange', 'yellow', 'green', 'blue',]
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,]for each in itertools.zip_longest(colors,
data, fillvalue=None):
    print(each)
```

Output

```
('red', 1)
('orange', 2)
('yellow', 3)
('green', 4)
('blue', 5)
(None, 6)
(None, 7)
(None, 8)
(None, 9)
(None, 10)
```