# Python Itertools Module - Python Cheatsheet

*Python Cheatsheet*

The *itertools* module is a collection of tools intended to be fast and use memory efficiently when handling iterators (like lists or dictionaries).

From the Python 3 documentation

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

The *itertools* module comes in the standard library and must be imported.

The operator module will also be used. This module is not necessary when using itertools, but needed for some of the examples below.

```
import itertools
import operator
```

## accumulate()

Makes an iterator that returns the results of a function.

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

Example:

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

Passing a function is optional:

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

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

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

Example:

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

## combinations_with_replacement()

Just like combinations(), but allows individual elements to be repeated more than once.

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

Example:

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

## count()

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

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

Example:

```
>>> for i in itertools.count(10,3):
...     print(i)
...     if i > 20:
...         break
...
# 10
# 13
# 16
# 19
# 22
```

## cycle()

This function cycles through an iterator endlessly.

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

Example:

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

When reached the end of the iterable it start over again from the beginning.

## chain()

Take a series of iterables and return them as one long iterable.

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

Example:

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

## compress()

Filters one iterable with another.

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

Example:

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

## dropwhile()

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

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

Example:

```
>>> 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)
...
# 5
# 6
# 7
# 8
# 9
# 10
# 1
```

## filterfalse()

Makes an iterator that filters elements from iterable returning only those for which the predicate is False.

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

Example:

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

## groupby()

Simply put, this function groups things together.

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

Example:

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

## islice()

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

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

Example:

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

## permutations()

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

Example:

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

## product()

Creates the cartesian products from a series of iterables.

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

## repeat()

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

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

Example:

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

## starmap()

Makes an iterator that computes the function using arguments obtained from the iterable.

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

Example:

```
>>> data = [(2, 6), (8, 4), (7, 3)]
>>> result = itertools.starmap(operator.mul, data)
>>> for each in result:
...     print(each)
...
# 12
# 32
# 21
```

## takewhile()

The opposite of dropwhile(). Makes an iterator and returns elements from the iterable as long as the predicate is true.

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

Example:

```
>>> 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)
...
# 1
# 2
# 3
# 4
```

## tee()

Return n independent iterators from a single iterable.

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

Example:

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

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

## zip_longest()

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.

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

Example:

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