itertools — Iterator Functions

Purpose: The itertools module includes a set of functions for working with sequence data sets.

The functions provided by `itertools` are inspired by similar features of functional programming languages such as Clojure, Haskell, APL, and SML. They are intended to be fast and use memory efficiently, and also to be hooked together to express more complicated iteration-based algorithms.

Iterator-based code offers better memory consumption characteristics than code that uses lists. Since data is not produced from the iterator until it is needed, all of the data does not need to be stored in memory at the same time. This “lazy” processing model can reduce swapping and other side-effects of large data sets, improving performance.

In addition to the functions defined in `itertools`, the examples in this section also rely on some of the built-in functions for iteration.

Merging and Splitting Iterators

The `chain()` function takes several iterators as arguments and returns a single iterator that produces the contents of all of the inputs as though they came from a single iterator.

```
# itertools_chain.py

from itertools import *

for i in chain([1, 2, 3], ['a', 'b', 'c']):
    print(i, end=' ')
print()
```

`chain()` makes it easy to process several sequences without constructing one large list.

```
$ python3 itertools_chain.py

1 2 3 a b c
```

If the iterables to be combined are not all known in advance, or need to be evaluated lazily, `chain.from_iterable()` can be used to construct the chain instead.

```
# itertools_chain_from_iterable.py

from itertools import *


def make_iterables_to_chain():
    yield [1, 2, 3]
    yield ['a', 'b', 'c']


for i in chain.from_iterable(make_iterables_to_chain()):
    print(i, end=' ')
print()
```

```
$ python3 itertools_chain_from_iterable.py

1 2 3 a b c
```

The built-in function `zip()` returns an iterator that combines the elements of several iterators into tuples.

```
# itertools_zip.py

for i in zip([1, 2, 3], ['a', 'b', 'c']):
    print(i)
```

As with the other functions in this module, the return value is an iterable object that produces values one at a time.

```
$ python3 itertools_zip.py

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

`zip()` stops when the first input iterator is exhausted. To process all of the inputs, even if the iterators produce different numbers of values, use `zip_longest()`.

```
# itertools_zip_longest.py

from itertools import *

r1 = range(3)
r2 = range(2)

print('zip stops early:')
print(list(zip(r1, r2)))

r1 = range(3)
r2 = range(2)

print('\nzip_longest processes all of the values:')
print(list(zip_longest(r1, r2)))
```

By default, `zip_longest()` substitutes `None` for any missing values. Use the `fillvalue` argument to use a different substitute value.

```
$ python3 itertools_zip_longest.py

zip stops early:
[(0, 0), (1, 1)]

zip_longest processes all of the values:
[(0, 0), (1, 1), (2, None)]
```

The `islice()` function returns an iterator which returns selected items from the input iterator, by index.

```
# itertools_islice.py

from itertools import *

print('Stop at 5:')
for i in islice(range(100), 5):
    print(i, end=' ')
print('\n')

print('Start at 5, Stop at 10:')
for i in islice(range(100), 5, 10):
    print(i, end=' ')
print('\n')

print('By tens to 100:')
for i in islice(range(100), 0, 100, 10):
    print(i, end=' ')
print('\n')
```

`islice()` takes the same arguments as the slice operator for lists: `start`, `stop`, and `step`. The start and step arguments are optional.

```
$ python3 itertools_islice.py

Stop at 5:
0 1 2 3 4

Start at 5, Stop at 10:
```

```
5 6 7 8 9

By tens to 100:
0 10 20 30 40 50 60 70 80 90
```

The `tee()` function returns several independent iterators (defaults to 2) based on a single original input.

```
# itertools_tee.py

from itertools import *

r = islice(count(), 5)
i1, i2 = tee(r)

print('i1:', list(i1))
print('i2:', list(i2))
```

`tee()` has semantics similar to the Unix `tee` utility, which repeats the values it reads from its input and writes them to a named file and standard output. The iterators returned by `tee()` can be used to feed the same set of data into multiple algorithms to be processed in parallel.

```
$ python3 itertools_tee.py

i1: [0, 1, 2, 3, 4]
i2: [0, 1, 2, 3, 4]
```

The new iterators created by `tee()` share their input, so the original iterator should not be used after the new ones are created.

```
# itertools_tee_error.py

from itertools import *

r = islice(count(), 5)
i1, i2 = tee(r)

print('r:', end=' ')
for i in r:
    print(i, end=' ')
    if i > 1:
        break
print()

print('i1:', list(i1))
print('i2:', list(i2))
```

If values are consumed from the original input, the new iterators will not produce those values:

```
$ python3 itertools_tee_error.py

r: 0 1 2
i1: [3, 4]
i2: [3, 4]
```

Converting Inputs

The built-in `map()` function returns an iterator that calls a function on the values in the input iterators, and returns the results. It stops when any input iterator is exhausted.

```
# itertools_map.py


def times_two(x):
    return 2 * x


def multiply(x, y):
```

```
    return (x, y, x * y)


print('Doubles:')
for i in map(times_two, range(5)):
    print(i)

print('\nMultiples:')
r1 = range(5)
r2 = range(5, 10)
for i in map(multiply, r1, r2):
    print('{:d} * {:d} = {:d}'.format(*i))

print('\nStopping:')
r1 = range(5)
r2 = range(2)
for i in map(multiply, r1, r2):
    print(i)
```

In the first example, the lambda function multiplies the input values by 2. In a second example, the lambda function multiplies two arguments, taken from separate iterators, and returns a tuple with the original arguments and the computed value. The third example stops after producing two tuples because the second range is exhausted.

```
$ python3 itertools_map.py

Doubles:
0
2
4
6
8

Multiples:
0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36

Stopping:
(0, 0, 0)
(1, 1, 1)
```

The `starmap()` function is similar to `map()`, but instead of constructing a tuple from multiple iterators, it splits up the items in a single iterator as arguments to the mapping function using the * syntax.

```
# itertools_starmap.py

from itertools import *

values = [(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]

for i in starmap(lambda x, y: (x, y, x * y), values):
    print('{} * {} = {}'.format(*i))
```

Where the mapping function to `map()` is called `f(i1, i2)`, the mapping function passed to `starmap()` is called `f(*i)`.

```
$ python3 itertools_starmap.py

0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36
```

Producing New Values

The `count()` function returns an iterator that produces consecutive integers, indefinitely. The first number can be passed as an argument (the default is zero). There is no upper bound argument (see the built-in `range()` for more control over the result

set).

```
# itertools_count.py

from itertools import *

for i in zip(count(1), ['a', 'b', 'c']):
    print(i)
```

This example stops because the list argument is consumed.

```
$ python3 itertools_count.py

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

The start and step arguments to `count()` can be any numerical values that can be added together.

```
# itertools_count_step.py
```

```
import fractions
from itertools import *

start = fractions.Fraction(1, 3)
step = fractions.Fraction(1, 3)

for i in zip(count(start, step), ['a', 'b', 'c']):
    print('{}: {}'.format(*i))
```

In this example, the start point and steps are `Fraction` objects from the `fraction` module.

```
$ python3 itertools_count_step.py

1/3: a
2/3: b
1: c
```

The `cycle()` function returns an iterator that repeats the contents of the arguments it is given indefinitely. Since it has to remember the entire contents of the input iterator, it may consume quite a bit of memory if the iterator is long.

```
# itertools_cycle.py

from itertools import *

for i in zip(range(7), cycle(['a', 'b', 'c'])):
    print(i)
```

A counter variable is used to break out of the loop after a few cycles in this example.

```
$ python3 itertools_cycle.py

(0, 'a')
(1, 'b')
(2, 'c')
(3, 'a')
(4, 'b')
(5, 'c')
(6, 'a')
```

The `repeat()` function returns an iterator that produces the same value each time it is accessed.

```
# itertools_repeat.py

from itertools import *

for i in repeat('over-and-over', 5):
```

```
    print(i)
```

The iterator returned by `repeat()` keeps returning data forever, unless the optional `times` argument is provided to limit it.

```
$ python3 itertools_repeat.py

over-and-over
over-and-over
over-and-over
over-and-over
over-and-over
```

It is useful to combine `repeat()` with `zip()` or `map()` when invariant values need to be included with the values from the other iterators.

```
# itertools_repeat_zip.py
```

```
from itertools import *

for i, s in zip(count(), repeat('over-and-over', 5)):
    print(i, s)
```

A counter value is combined with the constant returned by `repeat()` in this example.

```
$ python3 itertools_repeat_zip.py

0 over-and-over
1 over-and-over
2 over-and-over
3 over-and-over
4 over-and-over
```

This example uses `map()` to multiply the numbers in the range 0 through 4 by 2.

```
# itertools_repeat_map.py
```

```
from itertools import *

for i in map(lambda x, y: (x, y, x * y), repeat(2), range(5)):
    print('{:d} * {:d} = {:d}'.format(*i))
```

The `repeat()` iterator does not need to be explicitly limited, since `map()` stops processing when any of its inputs ends, and the `range()` returns only five elements.

```
$ python3 itertools_repeat_map.py

2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

Filtering

The `dropwhile()` function returns an iterator that produces elements of the input iterator after a condition becomes false for the first time.

```
# itertools_dropwhile.py
```

```
from itertools import *


def should_drop(x):
    print('Testing:', x)
    return x < 1
```

```
for i in dropwhile(should_drop, [-1, 0, 1, 2, -2]):
    print('Yielding:', i)
```

`dropwhile()` does not filter every item of the input; after the condition is false the first time, all of the remaining items in the input are returned.

```
$ python3 itertools_dropwhile.py

Testing: -1
Testing: 0
Testing: 1
Yielding: 1
Yielding: 2
Yielding: -2
```

The opposite of `dropwhile()` is `takewhile()`. It returns an iterator that returns items from the input iterator as long as the test function returns true.

```
# itertools_takewhile.py

from itertools import *


def should_take(x):
    print('Testing:', x)
    return x < 2


for i in takewhile(should_take, [-1, 0, 1, 2, -2]):
    print('Yielding:', i)
```

As soon as `should_take()` returns `False`, `takewhile()` stops processing the input.

```
$ python3 itertools_takewhile.py

Testing: -1
Yielding: -1
Testing: 0
Yielding: 0
Testing: 1
Yielding: 1
Testing: 2
```

The built-in function `filter()` returns an iterator that includes only items for which the test function returns true.

```
# itertools_filter.py

from itertools import *


def check_item(x):
    print('Testing:', x)
    return x < 1


for i in filter(check_item, [-1, 0, 1, 2, -2]):
    print('Yielding:', i)
```

`filter()` is different from `dropwhile()` and `takewhile()` in that every item is tested before it is returned.

```
$ python3 itertools_filter.py

Testing: -1
Yielding: -1
Testing: 0
Yielding: 0
Testing: 1
Testing: 2
```

```
Testing: -2
Yielding: -2
```

`filterfalse()` returns an iterator that includes only items where the test function returns false.

```
# itertools_filterfalse.py

from itertools import *


def check_item(x):
    print('Testing:', x)
    return x < 1


for i in filterfalse(check_item, [-1, 0, 1, 2, -2]):
    print('Yielding:', i)
```

The test expression in `check_item()` is the same, so the results in this example with `filterfalse()` are the opposite of the results from the previous example.

```
$ python3 itertools_filterfalse.py

Testing: -1
Testing: 0
Testing: 1
Yielding: 1
Testing: 2
Yielding: 2
Testing: -2
```

`compress()` offers another way to filter the contents of an iterable. Instead of calling a function, it uses the values in another iterable to indicate when to accept a value and when to ignore it.

```
# itertools_compress.py

from itertools import *

every_third = cycle([False, False, True])
data = range(1, 10)

for i in compress(data, every_third):
    print(i, end=' ')
print()
```

The first argument is the data iterable to process and the second is a selector iterable producing Boolean values indicating which elements to take from the data input (a true value causes the value to be produced, a false value causes it to be ignored).

```
$ python3 itertools_compress.py

3 6 9
```

Grouping Data

The `groupby()` function returns an iterator that produces sets of values organized by a common key. This example illustrates grouping related values based on an attribute.

```
# itertools_groupby_seq.py

import functools
from itertools import *
import operator
import pprint


@functools.total_ordering
class Point:
```

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return '({}, {})'.format(self.x, self.y)

    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)

    def __gt__(self, other):
        return (self.x, self.y) > (other.x, other.y)


# Create a dataset of Point instances
data = list(map(Point,
                cycle(islice(count(), 3)),
                islice(count(), 7)))
print('Data:')
pprint.pprint(data, width=35)
print()

# Try to group the unsorted data based on X values
print('Grouped, unsorted:')
for k, g in groupby(data, operator.attrgetter('x')):
    print(k, list(g))
print()

# Sort the data
data.sort()
print('Sorted:')
pprint.pprint(data, width=35)
print()

# Group the sorted data based on X values
print('Grouped, sorted:')
for k, g in groupby(data, operator.attrgetter('x')):
    print(k, list(g))
print()
```

The input sequence needs to be sorted on the key value in order for the groupings to work out as expected.

```
$ python3 itertools_groupby_seq.py

Data:
[(0, 0),
 (1, 1),
 (2, 2),
 (0, 3),
 (1, 4),
 (2, 5),
 (0, 6)]

Grouped, unsorted:
0 [(0, 0)]
1 [(1, 1)]
2 [(2, 2)]
0 [(0, 3)]
1 [(1, 4)]
2 [(2, 5)]
0 [(0, 6)]

Sorted:
[(0, 0),
 (0, 3),
 (0, 6),
 (1, 1),
 (1, 4),
 (2, 2),
 (2, 5)]
```

```
 (2, 5)]

Grouped, sorted:
0 [(0, 0), (0, 3), (0, 6)]
1 [(1, 1), (1, 4)]
2 [(2, 2), (2, 5)]
```

Combining Inputs

The `accumulate()` function processes the input iterable, passing the nth and n+1st item to a function and producing the return value instead of either input. The default function used to combine the two values adds them, so `accumulate()` can be used to produce the cumulative sum of a series of numerical inputs.

```
# itertools_accumulate.py

from itertools import *

print(list(accumulate(range(5))))
print(list(accumulate('abcde')))
```

When used with a sequence of non-integer values, the results depend on what it means to “add” two items together. The second example in this script shows that when `accumulate()` receives a string input each response is a progressively longer prefix of that string.

```
$ python3 itertools_accumulate.py

[0, 1, 3, 6, 10]
['a', 'ab', 'abc', 'abcd', 'abcde']
```

It is possible to combine `accumulate()` with any other function that takes two input values to achieve different results.

```
# itertools_accumulate_custom.py

from itertools import *


def f(a, b):
    print(a, b)
    return b + a + b


print(list(accumulate('abcde', f)))
```

This example combines the string values in a way that makes a series of (nonsensical) palindromes. Each step of the way when `f()` is called, it prints the input values passed to it by `accumulate()`.

```
$ python3 itertools_accumulate_custom.py

a b
bab c
cbabc d
dcbabcd e
['a', 'bab', 'cbabc', 'dcbabcd', 'edcbabcde']
```

Nested `for` loops that iterate over multiple sequences can often be replaced with `product()`, which produces a single iterable whose values are the Cartesian product of the set of input values.

```
# itertools_product.py

from itertools import *
import pprint

FACE_CARDS = ('J', 'Q', 'K', 'A')
SUITS = ('H', 'D', 'C', 'S')

DECK = list(
    product(
        chain(range(2, 11), FACE_CARDS),
```

```
        SUITS,
    )
)

for card in DECK:
    print('{:>2}{}'.format(*card), end=' ')
    if card[1] == SUITS[-1]:
        print()
```

The values produced by `product()` are tuples, with the members taken from each of the iterables passed in as arguments in the order they are passed. The first tuple returned includes the first value from each iterable. The *last* iterable passed to `product()` is processed first, followed by the next to last, and so on. The result is that the return values are in order based on the first iterable, then the next iterable, etc.

In this example, the cards are ordered by value and then by suit.

```
$ python3 itertools_product.py

 2H  2D  2C  2S
 3H  3D  3C  3S
 4H  4D  4C  4S
 5H  5D  5C  5S
 6H  6D  6C  6S
 7H  7D  7C  7S
 8H  8D  8C  8S
 9H  9D  9C  9S
10H 10D 10C 10S
 JH  JD  JC  JS
 QH  QD  QC  QS
 KH  KD  KC  KS
 AH  AD  AC  AS
```

To change the order of the cards, change the order of the arguments to `product()`.

```
# itertools_product_ordering.py

from itertools import *
import pprint

FACE_CARDS = ('J', 'Q', 'K', 'A')
SUITS = ('H', 'D', 'C', 'S')

DECK = list(
    product(
        SUITS,
        chain(range(2, 11), FACE_CARDS),
    )
)

for card in DECK:
    print('{:>2}{}'.format(card[1], card[0]), end=' ')
    if card[1] == FACE_CARDS[-1]:
        print()
```

The print loop in this example looks for an Ace card, instead of the spade suit, and then adds a newline to break up the output.

```
$ python3 itertools_product_ordering.py

 2H  3H  4H  5H  6H  7H  8H  9H 10H  JH  QH  KH  AH
 2D  3D  4D  5D  6D  7D  8D  9D 10D  JD  QD  KD  AD
 2C  3C  4C  5C  6C  7C  8C  9C 10C  JC  QC  KC  AC
 2S  3S  4S  5S  6S  7S  8S  9S 10S  JS  QS  KS  AS
```

To compute the product of a sequence with itself, specify how many times the input should be repeated.

```
# itertools_product_repeat.py

from itertools import *
```

```
def show(iterable):
    for i, item in enumerate(iterable, 1):
        print(item, end=' ')
        if (i % 3) == 0:
            print()
    print()


print('Repeat 2:\n')
show(list(product(range(3), repeat=2)))

print('Repeat 3:\n')
show(list(product(range(3), repeat=3)))
```

Since repeating a single iterable is like passing the same iterable multiple times, each tuple produced by `product()` will contain a number of items equal to the repeat counter.

```
$ python3 itertools_product_repeat.py

Repeat 2:

(0, 0) (0, 1) (0, 2)
(1, 0) (1, 1) (1, 2)
(2, 0) (2, 1) (2, 2)

Repeat 3:

(0, 0, 0) (0, 0, 1) (0, 0, 2)
(0, 1, 0) (0, 1, 1) (0, 1, 2)
(0, 2, 0) (0, 2, 1) (0, 2, 2)
(1, 0, 0) (1, 0, 1) (1, 0, 2)
(1, 1, 0) (1, 1, 1) (1, 1, 2)
(1, 2, 0) (1, 2, 1) (1, 2, 2)
(2, 0, 0) (2, 0, 1) (2, 0, 2)
(2, 1, 0) (2, 1, 1) (2, 1, 2)
(2, 2, 0) (2, 2, 1) (2, 2, 2)
```

The `permutations()` function produces items from the input iterable combined in the possible permutations of the given length. It defaults to producing the full set of all permutations.

```
# itertools_permutations.py

from itertools import *


def show(iterable):
    first = None
    for i, item in enumerate(iterable, 1):
        if first != item[0]:
            if first is not None:
                print()
            first = item[0]
        print(''.join(item), end=' ')
    print()


print('All permutations:\n')
show(permutations('abcd'))

print('\nPairs:\n')
show(permutations('abcd', r=2))
```

Use the `r` argument to limit the length and number of the individual permutations returned.

```
$ python3 itertools_permutations.py

All permutations:

abcd abdc acbd acdb adbc adcb
```

```
bacd badc bcad bcda bdac bdca
cabd cadb cbad cbda cdab cdba
dabc dacb dbac dbca dcab dcba

Pairs:

ab ac ad
ba bc bd
ca cb cd
da db dc
```

To limit the values to unique combinations rather than permutations, use `combinations()`. As long as the members of the input are unique, the output will not include any repeated values.

```
# itertools_combinations.py

from itertools import *


def show(iterable):
    first = None
    for i, item in enumerate(iterable, 1):
        if first != item[0]:
            if first is not None:
                print()
            first = item[0]
        print(''.join(item), end=' ')
    print()


print('Unique pairs:\n')
show(combinations('abcd', r=2))
```

Unlike with permutations, the `r` argument to `combinations()` is required.

```
$ python3 itertools_combinations.py

Unique pairs:

ab ac ad
bc bd
cd
```

While `combinations()` does not repeat individual input elements, sometimes it is useful to consider combinations that do include repeated elements. For those cases, use `combinations_with_replacement()`.

```
# itertools_combinations_with_replacement.py

from itertools import *


def show(iterable):
    first = None
    for i, item in enumerate(iterable, 1):
        if first != item[0]:
            if first is not None:
                print()
            first = item[0]
        print(''.join(item), end=' ')
    print()


print('Unique pairs:\n')
show(combinations_with_replacement('abcd', r=2))
```

In this output, each input item is paired with itself as well as all of the other members of the input sequence.

```
$ python3 itertools_combinations_with_replacement.py
```

```
Unique pairs:

aa ab ac ad
bb bc bd
cc cd
dd
```

See also

- Standard library documentation for itertools
- Python 2 to 3 porting notes for itertools
- The Standard ML Basis Library) – The library for SML.
- Definition of Haskell and the Standard Libraries – Standard library specification for the functional language Haskell.
- Clojure – Clojure is a dynamic functional language that runs on the Java Virtual Machine.
- tee – Unix command line tool for splitting one input into multiple identical output streams.
- Cartesian product – Mathematical definition of the Cartesian product of two sequences.

functools — Tools for Manipulating Functions

operator — Functional Interface to Built-in Operators

Quick Links

Merging and Splitting Iterators
Converting Inputs
Producing New Values
Filtering
Grouping Data
Combining Inputs

This page was last updated 2016-12-28.

Navigation

functools — Tools for Manipulating Functions
operator — Functional Interface to Built-in Operators

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

- Blog
- The Python Standard Library By Example