**chunked(iterable, n)**  
**def** take(n, iterable):  
 """Return first \*n\* items of the iterable as a list.  
 >>> take(3, range(10))  
 [0, 1, 2]  
 """**return** list(itertools.islice(iterable, n))

**def** chunked(iterable, n):  
 """Break \*iterable\* into lists of length \*n\*:  
 >>> list(chunked([1, 2, 3, 4, 5, 6], 3))  
 [[1, 2, 3], [4, 5, 6]]  
 >>> list(chunked([1, 2, 3, 4, 5, 6, 7, 8], 3))  
 [[1, 2, 3], [4, 5, 6], [7, 8]]  
 """**return** iter(functools.partial(take, n, iter(iterable)), []) #iter(callable, sentinel)

#等同于 **zip(\*[iter(iterable)]\*n)**

#将list中的每12项的平均值组成一个新的list

[sum(x)/len(x) for x in chunked(L,12)]

[sum(x)/len(x) for x in zip(\*[iter(L)]\*12)]

**collate(\*iterables) #按序整合**

#Return a sorted merge of the items from each of several already-sorted  
>>> list(collate('ACDZ', 'AZ', 'JKL'))  
 ['A', 'A', 'C', 'D', 'J', 'K', 'L', 'Z', 'Z']

**def** flatten(listOfLists):  
 *"""Return an iterator flattening one level of nesting in a list of lists.  
 >>> list(flatten([[0, 1], [2, 3]]))  
 [0, 1, 2, 3]  
 """* return chain.from\_iterable(listOfLists)

**def intersperse**(e,iterable): **点缀** *"""Intersperse object \*e\* between the items of \*iterable\*.  
 >>> list(intersperse('x', 'ABCD'))  
 ['A', 'x', 'B', 'x', 'C', 'x', 'D']  
 """* it = iter(iterable)  
 filler = itertools.repeat(e)  
 zipped = flatten(zip(it,filler))  
 **return** zipped

**def unique\_to\_each**(\*iterables): **寻找特异值**  
*"""Return the elements from each of the input iterables that aren't in the other input iterables.  
>>> unique\_to\_each({'A', 'B'}, {'B', 'C'}, {'B', 'D'})  
 [['A'], ['C'], ['D']]  
>>> unique\_to\_each("mississippi", "missouri")  
 [['p', 'p'], ['o', 'u', 'r']]  
"""* pool = [list(it) **for** it **in** iterables]  
 counts = collections.Counter(itertools.chain.from\_iterable(map(set, pool)))  
 uniques = {element **for** element **in** counts **if** counts[element] == 1}  
 **return** [list(filter(uniques.\_\_contains\_\_, it)) **for** it **in** pool]

**windowed**(seq,n,fillvalue=None,step=1) **窗口滑动式输出**  
*"""*

*Return a sliding window of width \*n\* over the given iterable.  
 >>> all\_windows = windowed([1, 2, 3, 4, 5], 3)  
>>> list(all\_windows)  
 [(1, 2, 3), (2, 3, 4), (3, 4, 5)]  
When the window is larger than the iterable, \*fillvalue\* is used in place of missing values  
>>> list(windowed([1, 2, 3], 4))  
 [(1, 2, 3, None)]  
Each window will advance in increments of \*step\*:  
>>> list(windowed([1, 2, 3, 4, 5, 6], 3, fillvalue='!', step=2))  
 [(1, 2, 3), (3, 4, 5), (5, 6, '!')]  
"""*

**def interleave**(\*iterables): **交错**  
*"""*

*Return a new iterable yielding from each iterable in turn,until the shortest is exhausted.  
>>> list(interleave([1, 2, 3], [4, 5], [6, 7, 8]))  
 [1, 4, 6, 2, 5, 7]  
For a version that doesn't terminate after the shortest iterable is exhausted,   
"""* **return** itertools**.**chain.from\_iterable(zip(\*iterables))

**def interleave\_longest**(\*iterables):  
*"""*

*Return a new iterable yielding from each iterable in turn,skipping any that are exhausted.  
>>> list(interleave\_longest([1, 2, 3], [4, 5], [6, 7, 8]))  
 [1, 4, 6, 2, 5, 7, 3, 8]  
"""* i = chain.from\_iterable(zip\_longest(\*iterables, fillvalue=\_marker))  
 **return** filter(**lambda** x: x **is not** \_marker, i)

**def collapse**(iterable, base\_type=**None**, levels=**None**):  
*"""Flatten an iterable with multiple levels of nesting into non-iterable types.  
>>> iterable = [(1, 2), ([3, 4], [[5], [6]])]  
>>> list(collapse(iterable))  
 [1, 2, 3, 4, 5, 6]  
String types are not considered iterable and will not be collapsed.  
To avoid collapsing other types, specify \*base\_type\*:  
>>> iterable = ['ab', ('cd', 'ef'), ['gh', 'ij']]  
>>> list(collapse(iterable, base\_type=tuple))  
 ['ab', ('cd', 'ef'), 'gh', 'ij']  
Specify \*levels\* to stop flattening after a certain level:  
>>> iterable = [('a', ['b']), ('c', ['d'])]  
>>> list(collapse(iterable, levels=1)) # Only one level flattened  
 ['a', ['b'], 'c', ['d']]  
"""* **def** walk(node, level):  
 **if** (  
 ((levels **is not None**) **and** (level > levels)) **or** isinstance(node, string\_types) **or** ((base\_type **is not None**) **and** isinstance(node, base\_type))  
 ):  
 **yield** node  
 **return  
  
 try**:  
 tree = iter(node)  
 **except** TypeError:  
 **yield** node  
 **return  
 else**:  
 **for** child **in** tree:  
 **for** x **in** walk(child, level + 1):  
 **yield** x  
  
 **for** x **in** walk(iterable, 0):  
 **yield** x