itertools --- 为高效循环而创建迭代器的函数

本模块实现一系列 iterator ， 这些迭代器受到APL， Haskell和SML的启发。为了适用于Python，它们都被重新写过。

本模块标准化了一个快速、高效利用内存的核心工具集，这些工具本身或组合都很有用。它们一起形成了“迭代器代数”，这使得在纯Python中有可能创建简洁又高效的专用工具。

例如，SML有一个制表工具： `tabulate(f)`，它可产生一个序列 `f(0), f(1), ...`。在Python中可以组合 `map()` 和 `count()` 实现： `map(f, count())`。

这些内置工具同时也能很好地与 `operator` 模块中的高效函数配合使用。例如，我们可以将两个向量的点积映射到乘法运算符： `sum(map(operator.mul, vector1, vector2))` 。

无穷迭代器：

迭代器	实参	结果	示例
`count()`	start, [step]	start, start+step, start+2*step, ...	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, ... plast, p0, p1, ...	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, ... 重复无限次或n次	`repeat(10, 3) --> 10 10 10`

根据最短输入序列长度停止的迭代器：

迭代器	实参	结果	示例
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, ...	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
`chain()`	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') --> A B C D E F`
`chain.from_iterable()`	iterable -- 可迭代对象	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
`dropwhile()`	pred, seq	seq[n], seq[n+1], ... 从pred首次真值测试失败开始	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
`filterfalse()`	pred, seq	seq中pred(x)为假值的元素，x是seq中的元素。	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`

迭代器	实参	结果	示例
groupby()	iterable[, key]	根据key(v)值分组的迭代器	
islice()	seq, [start,] stop [, step]	seq[start:stop:step]中的元素	islice('ABCDEFG', 2, None) --> C D E F G
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
takewhile()	pred, seq	seq[0], seq[1], ..., 直到pred真值测试失败	takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
tee()	it, n	it1, it2, ... itn 将一个迭代器拆分为n个迭代器	
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-

排列组合迭代器：

迭代器	实参	结果
product()	p, q, ... [repeat=1]	笛卡尔积，相当于嵌套的for循环
permutations()	p[, r]	长度r元组，所有可能的排列，无重复元素
combinations()	p, r	长度r元组，有序，无重复元素
combinations_with_replacement()	p, r	长度r元组，有序，元素可重复

例子	结果
product('ABCD', repeat=2)	AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
permutations('ABCD', 2)	AB AC AD BA BC BD CA CB CD DA DB DC
combinations('ABCD', 2)	AB AC AD BC BD CD
combinations_with_replacement('ABCD', 2)	AA AB AC AD BB BC BD CC CD DD

Itertool函数

下列模块函数均创建并返回迭代器。有些迭代器不限制输出流长度，所以它们只应在能截断输出流的函数或循环中使用。

itertools.accumulate(iterable[, func, *, initial=None])

创建一个迭代器，返回累积汇总值或其他双目运算函数的累积结果值（通过可选的 *func* 参数指定）。

如果提供了 *func*，它应当为带有两个参数的函数。 输入 *iterable* 的元素可以是能被 *func* 接受为参数的任意类型。 （例如，对于默认的加法运算，元素可以是任何可相加的类型包括 Decimal 或 Fraction。）

通常，输出的元素数量与输入的可迭代对象是一致的。 但是，如果提供了关键字参数 *initial*，则累加会以 *initial* 值开始，这样输出就比输入的可迭代对象多一个元素。

大致相当于:

```
def accumulate(iterable, func=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) --> 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(it)
        except StopIteration:
            return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

func 参数有几种用法。它可以被设为 min() 最终得到一个最小值，或者设为 max() 最终得到一个最大值，或设为 operator.mul() 最终得到一个乘积。摊销表可通过累加利息和支付款项得到。给iterable设置初始值并只将参数 *func* 设为累加总数可以对一阶 递归关系 建模。

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]

# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

# Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _:  r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> inputs = repeat(x0, 36)     # only the initial value is used
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
```

```
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
 '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
 '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

参考一个类似函数 `functools.reduce()` ，它只返回一个最终累积值。

3.2 新版功能.

在 *3.3* 版更改: 增加可选参数 *func* 。

在 *3.8* 版更改: 添加了可选的 *initial* 形参。

itertools.chain(*iterables)

创建一个迭代器，它首先返回第一个可迭代对象中所有元素，接着返回下一个可迭代对象中所有元素，直到耗尽所有可迭代对象中的元素。可将多个序列处理为单个序列。大致相当于:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

classmethod chain.from_iterable(iterable)

构建类似 `chain()` 迭代器的另一个选择。从一个单独的可迭代参数中得到链式输入，该参数是延迟计算的。大致相当于:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

itertools.combinations(iterable, r)

返回由输入 *iterable* 中元素组成长度为 *r* 的子序列。

组合元组会以字典顺序根据所输入 *iterable* 的顺序发出。 因此，如果所输入 *iterable* 是已排序的，组合元组也将按已排序的顺序生成。

即使元素的值相同，不同位置的元素也被认为是不同的。如果元素各自不同，那么每个组合中没有重复元素。

大致相当于:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
```

```
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

`combinations()` 的代码可被改写为 `permutations()` 过滤后的子序列，（相对于元素在输入中的位置）元素不是有序的。

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

当 `0 <= r <= n` 时，返回项的个数是 `n! / r! / (n-r)!`；当 `r > n` 时，返回项个数为0。

itertools.combinations_with_replacement(iterable, r)

返回由输入 *iterable* 中元素组成的长度为 *r* 的子序列，允许每个元素可重复出现。

组合元组会以字典顺序根据所输入 *iterable* 的顺序发出。 因此，如果所输入 *iterable* 是已排序的，组合元组也将按已排序的顺序生成。

不同位置的元素是不同的，即使它们的值相同。因此如果输入中的元素都是不同的话，返回的组合中元素也都会不同。

大致相当于：

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

`combinations_with_replacement()` 的代码可被改写为 `production()` 过滤后的子序列，（相对于元素在输入中的位置）元素不是有序的。

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

当 `n > 0` 时，返回项个数为 `(n+r-1)! / r! / (n-1)!`.

3.1 新版功能.

itertools. compress(data, selectors)

创建一个迭代器，它返回 *data* 中经 *selectors* 真值测试为 `True` 的元素。迭代器在两者较短的长度处停止。大致相当于：

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

3.1 新版功能.

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

创建一个迭代器，它从 *start* 值开始，返回均匀间隔的值。常用于 `map()` 中的实参来生成连续的数据点。此外，还用于 `zip()` 来添加序列号。大致相当于：

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

当对浮点数计数时，替换为乘法代码有时精度会更好，例如：`(start + step * i for i in count())`。

在 *3.1* 版更改: 增加参数 *step*，允许非整型。

itertools. cycle(iterable)

创建一个迭代器，返回 *iterable* 中所有元素并保存一个副本。当取完 *iterable* 中所有元素，返回副本中的所有元素。无限重复。大致相当于：

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
```

```
            saved.append(element)
        while saved:
            for element in saved:
                  yield element
```

注意，该函数可能需要相当大的辅助空间（取决于 *iterable* 的长度）。

itertools.dropwhile(predicate, iterable)

创建一个迭代器，如果 *predicate* 为true，迭代器丢弃这些元素，然后返回其他元素。注意，迭代器在 *predicate* 首次为false之前不会产生任何输出，所以可能需要一定长度的启动时间。大致相当于：

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.filterfalse(predicate, iterable)

创建一个迭代器，只返回 *iterable* 中 *predicate* 为 `False` 的元素。如果 *predicate* 是 `None`，返回真值测试为false的元素。大致相当于：

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.groupby(iterable, key=None)

创建一个迭代器，返回 *iterable* 中连续的键和组。*key* 是一个计算元素键值函数。如果未指定或为 `None`，*key* 缺省为恒等函数（identity function），返回元素不变。一般来说，*iterable* 需用同一个键值函数预先排序。

`groupby()` 操作类似于Unix中的 `uniq`。当每次 *key* 函数产生的键值改变时，迭代器会分组或生成一个新组（这就是为什么通常需要使用同一个键值函数先对数据进行排序）。这种行为与SQL的GROUP BY操作不同，SQL的操作会忽略输入的顺序将相同键值的元素分在同组中。

返回的组本身也是一个迭代器，它与 `groupby()` 共享底层的可迭代对象。因为源是共享的，当 `groupby()` 对象向后迭代时，前一个组将消失。因此如果稍后还需要返回结果，可保存为列表：

```
groups = []
uniquekeys = []
```

```
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

groupby() 大致相当于：

```
class groupby:
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def __next__(self):
        self.id = object()
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey, self.id))
    def _grouper(self, tgtkey, id):
        while self.id is id and self.currkey == tgtkey:
            yield self.currvalue
            try:
                self.currvalue = next(self.it)
            except StopIteration:
                return
            self.currkey = self.keyfunc(self.currvalue)
```

itertools.islice(iterable, stop)

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

创建一个迭代器，返回从 *iterable* 里选中的元素。如果 *start* 不是0，跳过 *iterable* 中的元素，直到到达 *start* 这个位置。之后迭代器连续返回元素，除非 *step* 设置的值很高导致被跳过。如果 *stop* 为 None，迭代器耗光为止；否则，在指定的位置停止。与普通的切片不同，islice() 不支持将 *start*， *stop*，或 *step* 设为负值。可用来从内部数据结构被压平的数据中提取相关字段（例如一个多行报告，它的名称字段出现在每三行上）。大致相当于：

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
```

```
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass
```

如果 *start* 为 `None`，迭代从0开始。如果 *step* 为 `None` ，步长缺省为1。

itertools.permutations(iterable, r=None)

连续返回由 *iterable* 元素生成长度为 *r* 的排列。

如果 *r* 未指定或为 `None` ， *r* 默认设置为 *iterable* 的长度，这种情况下，生成所有全长排列。

排列元组会以字典顺序根据所输入 *iterable* 的顺序发出。 因此，如果所输入 *iterable* 是已排序的，组合元组也将按已排序的顺序生成。

即使元素的值相同，不同位置的元素也被认为是不同的。如果元素值都不同，每个排列中的元素值不会重复。

大致相当于:

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
```

```
                break
        else:
            return
```

`permutations()` 的代码也可被改写为 `product()` 的子序列，只要将含有重复元素（来自输入中同一位置的）的项排除。

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

当 `0 <= r <= n` ，返回项个数为 `n! / (n-r)!` ；当 `r > n` ，返回项个数为0。

itertools. product(*iterables, repeat=1)

可迭代对象输入的笛卡儿积。

大致相当于生成器表达式中的嵌套循环。例如， `product(A, B)` 和 `((x,y) for x in A for y in B)` 返回结果一样。

嵌套循环像里程表那样循环变动，每次迭代时将最右侧的元素向后迭代。这种模式形成了一种字典序，因此如果输入的可迭代对象是已排序的，笛卡尔积元组依次序发出。

要计算可迭代对象自身的笛卡尔积，将可选参数 *repeat* 设定为要重复的次数。例如，`product(A, repeat=4)` 和 `product(A, A, A, A)` 是一样的。

该函数大致相当于下面的代码，只不过实际实现方案不会在内存中创建中间结果。

```
def product(*args, repeat=1):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

itertools. repeat(object[, times])

创建一个迭代器，不断重复 *object* 。除非设定参数 *times* ，否则将无限重复。可用于 `map()` 函数中的参数，被调用函数可得到一个不变参数。也可用于 `zip()` 的参数以在元组记录中创建一个不变的部分。

大致相当于:

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
```

```
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

repeat 最常见的用途就是在 *map* 或 *zip* 提供一个常量流：

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools. starmap(function, iterable)

创建一个迭代器，使用从可迭代对象中获取的参数来计算该函数。当参数对应的形参已从一个单独可迭代对象组合为元组时（数据已被“预组对”）可用此函数代替 `map()`。`map()` 与 `starmap()` 之间的区别可以类比 `function(a,b)` 与 `function(*c)` 的区别。大致相当于:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools. takewhile(predicate, iterable)

创建一个迭代器，只要 predicate 为真就从可迭代对象中返回元素。大致相当于:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

itertools. tee(iterable, n=2)

从一个可迭代对象中返回 *n* 个独立的迭代器。

下面的Python代码能帮助解释 *tee* 做了什么（尽管实际的实现更复杂，而且仅使用了一个底层的 FIFO 队列）。

大致相当于:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                try:
                    newval = next(it)   # fetch a new value and
                except StopIteration:
                    return
                for d in deques:        # load it to all the deques
```

```
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

一旦 `tee()` 实施了一次分裂，原有的 *iterable* 不应再被使用；否则tee对象无法得知 *iterable* 可能已向后迭代。

`tee` 迭代器不是线程安全的。当同时使用由同一个 `tee()` 调用所返回的迭代器时可能引发 `RuntimeError`，即使原本的 *iterable* 是线程安全的。

该迭代工具可能需要相当大的辅助存储空间（这取决于要保存多少临时数据）。通常，如果一个迭代器在另一个迭代器开始之前就要使用大部份或全部数据，使用 `list()` 会比 `tee()` 更快。

itertools.zip_longest(*iterables, fillvalue=None)

创建一个迭代器，从每个可迭代对象中收集元素。如果可迭代对象的长度未对齐，将根据 *fillvalue* 填充缺失值。迭代持续到耗光最长的可迭代对象。大致相当于：

```
def zip_longest(*args, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

如果其中一个可迭代对象有无限长度，`zip_longest()` 函数应封装在限制调用次数的场景中（例如 `islice()` 或 `takewhile()`）。除非指定，*fillvalue* 默认为 `None`。

itertools 配方

本节将展示如何使用现有的 itertools 作为基础构件来创建扩展的工具集。

基本上所有这些西方和许许多多其他的配方都可以通过 Python Package Index 上的 more-itertools 项目 来安装:

```
pip install more-itertools
```

扩展的工具提供了与底层工具集相同的高性能。保持了超棒的内存利用率，因为一次只处理一个元素，而不是将整个可迭代对象加载到内存。代码量保持得很小，以函数式风格将这些工具连接在一起，有助于消除临时变量。速度依然很快，因为倾向于使用“矢量化”构件来取代解释器开销大的 for 循环和 generator 。

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterator)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
    "Return an iterator over the last n items"
    # tail(3, 'ABCDEFG') --> E F G
    return iter(collections.deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(map(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
```

```
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def flatten(list_of_lists):
    "Flatten one level of nesting"
    return chain.from_iterable(list_of_lists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            # Remove the iterator we just exhausted from the cycle.
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))

def partition(pred, iterable):
    'Use a predicate to partition entries into false entries and true entries'
    # partition(is_odd, range(10)) --> 0 2 4 6 8   and  1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))
```

```
def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return map(next, map(operator.itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like builtins.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        iter_except(functools.partial(heappop, h), IndexError)   # priority queue i
        iter_except(d.popitem, KeyError)                         # non-blocking dic
        iter_except(d.popleft, IndexError)                       # non-blocking dec
        iter_except(q.get_nowait, Queue.Empty)                   # loop over a proc
        iter_except(s.pop, KeyError)                             # non-blocking se

    """
    try:
        if first is not None:
            yield first()           # For database APIs needing an initial cast t
        while True:
            yield func()
    except exception:
        pass

def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*

    If *pred* is not None, returns the first item
    for which pred(item) is true.
```

```
    """
    # first_true([a,b,c], x) --> a or b or c or x
    # first_true([a,b], x, f) --> a if f(a) else b if f(b) else x
    return next(filter(pred, iterable), default)

def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in range(r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    'Equivalent to list(combinations(iterable, r))[index]'
    pool = tuple(iterable)
    n = len(pool)
    if r < 0 or r > n:
        raise ValueError
    c = 1
    k = min(r, n-r)
    for i in range(1, k+1):
        c = c * (n - k + i) // i
    if index < 0:
        index += c
    if index < 0 or index >= c:
        raise IndexError
    result = []
    while r:
        c, n, r = c*r//n, n-1, r-1
        while index >= c:
            index -= c
            c, n = c*(n-r)//n, n-1
        result.append(pool[-1-n])
    return tuple(result)
```