CS61A NOTE6 Iterators, Generators

Iterators 迭代器

An iterable is an object where we can go through its elements one at a time. Specifically, we define an **iterable** 迭代对象 as any object 对象 where calling the built-in `iter` function 用 iter 函数 on it returns an *iterator*. An **iterator** is another type of object 也是对象 which can iterate over an iterable by keeping track of which element is next 追踪下一个元素 in the iterable 迭代对象，类似 for elem in iterable

迭代对象是可以一个个地浏览元素的对象，可迭代对象可是任何对象，在其上调用内置的 iter 函数会返回一个 iterator。iterator 是另一种对象，它可以通过跟踪迭代器中的下一个元素来迭代下一个可迭代对象。

```
纯文本
>>> lst = [1, 2, 3]
>>> lst_iter = iter(lst)
>>> lst_iter
<list_iterator object ...>
```

With an iterator, we can call `next` on it to get the next element in the iterator. If calling `next` on an iterator raises a `StopIteration` exception, this signals to us that the iterator has no more elements to go through. 对于一个 iterator，我们可以对它调用 next 来获得迭代器中的下一个元素。若在迭代器上调用 next 产生 StopIteration 异常，表明迭代器没有更多的元素。

Calling `iter` on an iterable 迭代对象 multiple times returns a new iterator each time with distinct states 每次返回具有不同状态的新迭代器(otherwise, you'd never be able to iterate through a iterable more than once). You can also call `iter` on the **iterator** itself, which will just return the **same iterator** without changing its state. However, note that you cannot call `next` directly on an iterable.迭代对象

对于 itrable 迭代对象：

用 iter 在 itrable 迭代对象上，再用 next，每次返回不同状态新 iterator，不可直接 next 会报错。

对于每个 iterator（itrable 被 iter 后）：

也可 iter 在 iterator 上返回相同 iterator

Here's a breakdown of what's happening:

- First, the built-in `iter` function is called on the iterable to create a corresponding *iterator*.
- To get the next element in the sequence, the built-in `next` function is called on this iterator.
- When `next` is called but there are no elements left in the iterator, a `StopIteration` error is raised. In the for loop construct, this exception is caught and execution can continue.

纯文本

```
>>> lst = [1, 2, 3]
>>> next(lst)             # iterable上用next报错
TypeError: 'list' object is not an iterator #list是itrable迭代对直接next报错
>>> list_iter = iter(lst) # 语法iter新建iterator
>>> next(list_iter)       # iterator上用next
1
>>> next(iter(list_iter)) # iterator上用iter返回自身，与上同，再next
2
>>> for e in list_iter:   # 打印list_iter剩下的部分
...     print(e)
3
>>> next(list_iter)       # No elements left!不剩元素了
StopIteration
>>> lst      # Original iterable is unaffected原始iterable不被影响
[1, 2, 3]
```

What Would Python Display?

```
>>> obj = SomeObj()  # obj does not have __iter__
>>> i = iter(obj)
Error (obj is not iterable)
>>> next(obj)
Error (obj is not an iterator)
```

iterable 类型：字典，list，string

Note that we can also call the function `list` on finite iterators, and it will list out the remaining items in that iterator.可在有限 iterator 上用 list 函数显示出剩下的有限元素 list

纯文本

```
>>> lst = [1, 2, 3, 4]
>>> list_iter = iter(lst)  #iter()命为list_iter，名字其实随意
>>> next(list_iter)
1
>>> list(list_iter)       # 用list返回iter后的iterator剩余的元素list
[2, 3, 4]
```

纯文本

```
>>> s = "cs61a"
>>> s_iter = iter(s)
>>> next(s_iter)
'c'     #注意字符串带引号

>>> next(s_iter)
's'

>>> list(s_iter)
['6','1','a']     #注意list要带[],字符串要分开且带''
```

纯文本

```
>>> lst = [1, 2, 3, 4]
>>> next(lst)          # 直接next，TypeError:list不是iterator
>>> list_iter = iter(lst) # 建立iterator
>>> list_iter
```

```
<list_iterator object ...>
>>> next(list_iter)    #注意是从第一个开始
1
>>> next(list_iter)
2
>>> next(iter(list_iter)) # iter在iterator返回自己
3
>>> list_iter2 = iter(lst) #新建了iterator
>>> next(list_iter2)      # 新起点
1
>>> next(list_iter)       # 注意是第一个
4
>>> next(list_iter)       # 没了
StopIteration
>>> lst                   # Original iterable不被影响
[1, 2, 3, 4]
```

纯文本

```
>>> s = [[1, 2, 3, 4]]  #s自身不改变，但注意是两个[]
>>> i = iter(s)
>>> j = iter(next(i)) #iterator上用iter返回自身[1,2,3,4]，格式仍为iterator
>>> next(j)
1

>>> s.append(5)
>>> next(i) #s变i也跟着变
5

>>> next(j) #看清i：[[1, 2, 3, 4]，5]和j：[1,2,3,4]
2

>>> list(j) #用list函数输出剩下的元素
[3, 4]

>>> next(i)
StopIteration
```

Since you can call `iter` on iterators, this tells us that that they are also iterables! Note that while all iterators are iterables, the converse is not true – that is, not all iterables are iterators. You can use iterators wherever you can use iterables, but note that since iterators keep their state, they're only good to iterate through an iterable once:可以调用 `iter` 迭代器，这告诉我们迭代器也是可迭代的！虽然所有迭代器都是可迭代对象，但反之则不然——并非所有可迭代对象都是迭代器。可以在任何可以使用可迭代对象的地方使用迭代器，但由于迭代器会保持其状态，因此它们只能迭代一次可迭代对象：

纯文本

```
>>> list_iter = iter([4, 3, 2, 1])
>>> for e in list_iter:
...     print(e)
4
3
2
1
>>> for e in list_iter:
...     print(e)
```

类比：一个可迭代对象就像一本书（可以翻页），而一本书的迭代器就是一个书签（保存位置并可以定位到下一页）。调用 `iter` 一本书会为您提供一个独立于其他书签的新书签，但调用 `iter` 书签会为您提供书签本身，而根本不会改变其位置。调用 `next` 书签会将其移动到下一页，但不会更改书中的页面。调用 `next` 这本书在语义上没有意义。我们也可以有多个书签，彼此独立。

`range(start, end)` function, which creates an iterable of ascending integers from start (inclusive) to end (exclusive).

纯文本

```
>>> for x in range(2, 6):
...     print(x)
2
3
4
5
```

类似列表表达式：

- `map(f, iterable)` – Creates an iterator over `f(x)` for `x` in `iterable`. 类似 [`func(x)` for `x` in `iterable`]. iterators 可以有 infinite values, lists 不能有 infinite elements.
- `filter(f, iterable)` – Creates an iterator over `x` for each `x` in `iterable` if `f(x)`
- `zip(iterables*)` – Creates an iterator over co–indexed tuples with elements from each of the `iterables`
- `reversed(iterable)` – Creates an iterator over all the elements in the input iterable in reverse order
- `list(iterable)` – Creates a list containing all the elements in the input `iterable`
- `tuple(iterable)` – Creates a tuple containing all the elements in the input `iterable`
- `sorted(iterable)` –排序 Creates a sorted list containing all the elements in the input `iterable`
- `reduce(f, iterable)` – Must be imported with `functools`. Apply function of two arguments `f` cumulatively to the items of `iterable`, from left to right, so as to reduce the sequence to a single value.将两个参数 f 的函数从左到右累积应用于 iterable 的 irem，直到一个值。

Lab6 WWPD

```
纯文本

>>> s = [1, 2, 3, 4]
>>> t = iter(s)
>>> next(s) #
Error

>>> next(t)
1

>>> next(t)
2

>>> iter(s)
Iterator
```

```
>>> next(iter(s))
1

>>> next(iter(t)) #追随上面的2
3

>>> next(iter(s)) #？？？神奇，一直在新建，不像t
1

>>> next(iter(t))
4

>>> next(t)
StopIteration
```

添加图片 ⓘ

纯文本

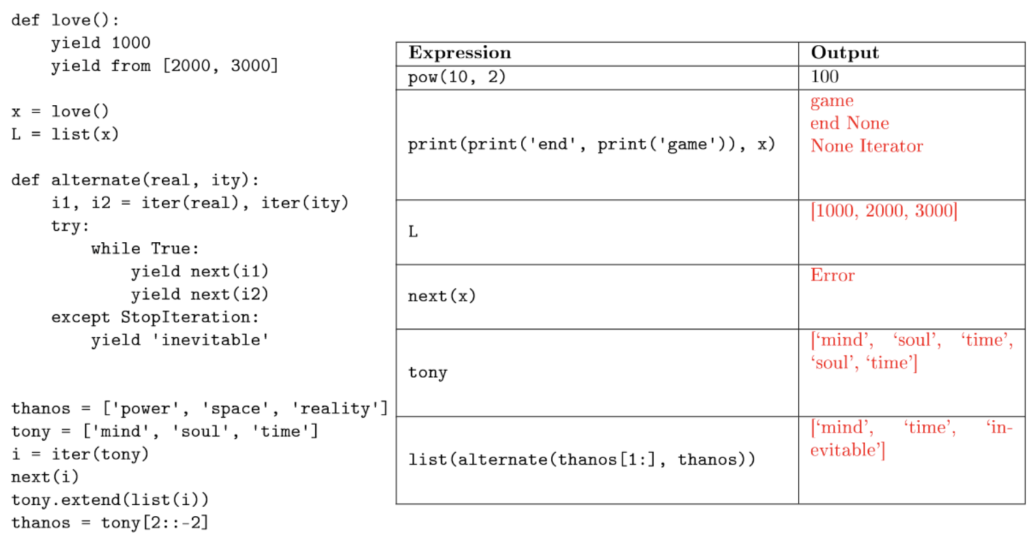
```
>>> r = range(6)
>>> r_iter = iter(r)
>>> next(r_iter)
0

>>> [x + 1 for x in r]
[1,2,3,4,5,6]

>>> [x + 1 for x in r_iter]
[2,3,4,5,6]

>>> next(r_iter)
StopIteration

>>> list(range(-2, 4))   # Converts转换iterable into list
[-2,-1,0,1,2,3]
```

纯文本

```
>>> map_iter = map(lambda x : x + 10, range(5)) #好用！
>>> next(map_iter)
10

>>> next(map_iter)
11
```

```
>>> list(map_iter)
[12,13,14]

>>> for e in filter(lambda x : x % 2 == 0, range(1000, 1008)):
...     print(e)
1000
1002
1004
1006

>>> [x + y for x, y in zip([1, 2, 3], [4, 5, 6])]
[5,7,9]

>>> for e in zip([10, 9, 8], range(3)):
...   print(tuple(map(lambda x: x + 2, e)))
(12,2) #外面是(),e是两个元素都要lambda
(11,3)
(10,4)
```

WWPD (SP19 Final Q1)



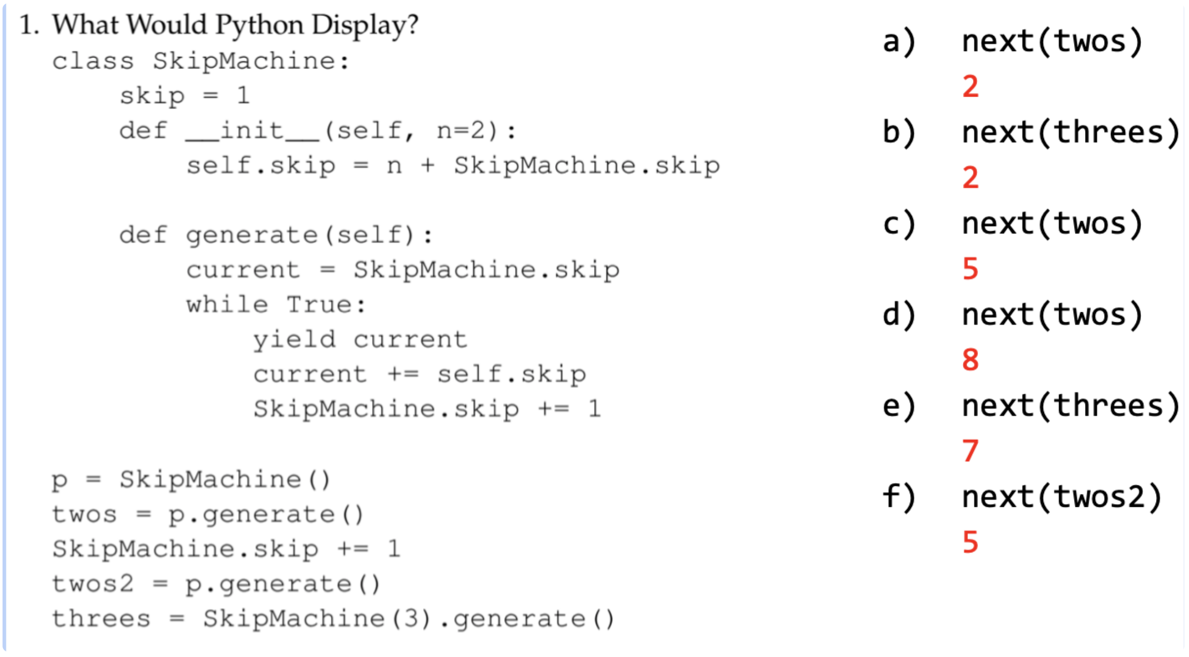
```
def love():
    yield 1000
    yield from [2000, 3000]

x = love()
L = list(x)

def alternate(real, ity):
    i1, i2 = iter(real), iter(ity)
    try:
        while True:
            yield next(i1)
            yield next(i2)
    except StopIteration:
        yield 'inevitable'


thanos = ['power', 'space', 'reality']
tony = ['mind', 'soul', 'time']
i = iter(tony)
next(i)
tony.extend(list(i))
thanos = tony[2::-2]
```

Expression	Output
`pow(10, 2)`	100
`print(print('end', print('game')), x)`	game end None None Iterator
`L`	[1000, 2000, 3000]
`next(x)`	Error
`tony`	['mind', 'soul', 'time', 'soul', 'time']
`list(alternate(thanos[1:], thanos))`	['mind', 'time', 'in-evitable']

注意 generator 输出写为 Iterator

list(i)还剩 soul 和 time

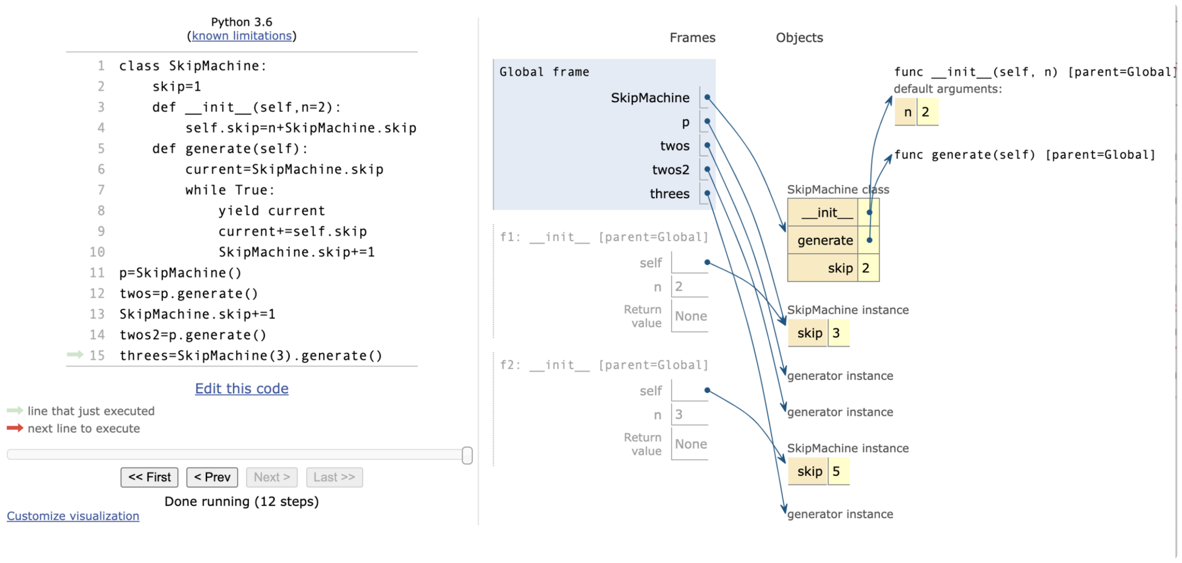
real 和 ity 交错 yield，最后一行后 thanos 是 time mind，前者输入 mind 后者输入 time mind，一轮得 mind time 但之后 i1 没了 yield inevitable



1. What Would Python Display?

```
class SkipMachine:
    skip = 1
    def __init__(self, n=2):
        self.skip = n + SkipMachine.skip

    def generate(self):
        current = SkipMachine.skip
        while True:
            yield current
            current += self.skip
            SkipMachine.skip += 1

p = SkipMachine()
twos = p.generate()
SkipMachine.skip += 1
twos2 = p.generate()
threes = SkipMachine(3).generate()
```

```
a)  next(twos)
    2
b)  next(threes)
    2
c)  next(twos)
    5
d)  next(twos)
    8
e)  next(threes)
    7
f)  next(twos2)
    5
```



Python 3.6
(known limitations)

```
class SkipMachine:
    skip=1
    def __init__(self,n=2):
        self.skip=n+SkipMachine.skip
    def generate(self):
        current=SkipMachine.skip
        while True:
            yield current
            current+=self.skip
            SkipMachine.skip+=1
p=SkipMachine()
twos=p.generate()
SkipMachine.skip+=1
twos2=p.generate()
threes=SkipMachine(3).generate()
```

p.generate()，先不执行，类似 review session 讲的：next 才调用才执行 while 循环下的 yield，到时 skip 已经变为 2

twos 的 step 是 3，threes 的 step 是 n3+skip2=5

twos 调用三次，twos2 被影响的 skip 为 2+3=5，一开始输出 5

最后一行传入 n=3，新鲜的东西。

lab6 Q4 Count Occurrences

Implement `count_occurrences`, which takes in an iterator `t` and returns the number of times the value `x` appears in the first `n` elements of `t`. A value appears in a sequence of elements if it is equal to an entry in the sequence.输入迭代器 t,返回 t 的 n 项前 x 出现的个数

纯文本

```
def count_occurrences(t, n, x):
    k=0
    for i in range(n):#操作n次
        if next(t)==x: #用到迭代器！
            k+=1
    return k
```

Lab Q5 Repeated 两道题

Implement `repeated`, which takes in an iterator `t` and returns the first value in `t` that appears `k` times in a row.输入迭代器 t，返回 t 中第一个连续出现 k 次的值

纯文本

```
def repeated(t, k):
    last=None
    for i in t: #操作次数，可能更早返回
        if i!=last or last==None:
            n=1
            last=i
        else:
            n+=1
            last=i
            if n==k:
                return i

    #法二用next
    assert k > 1
    n = 1
    last_item = None
    while True:
        item = next(t)  #不断执行此行，用到next，自动传入，类似a1=next(a),a1=nex
        if item == last_item:
            count += 1
        else:
            last_item = item #更新item
            count = 1
        if count == k:
            return item
```

Generators

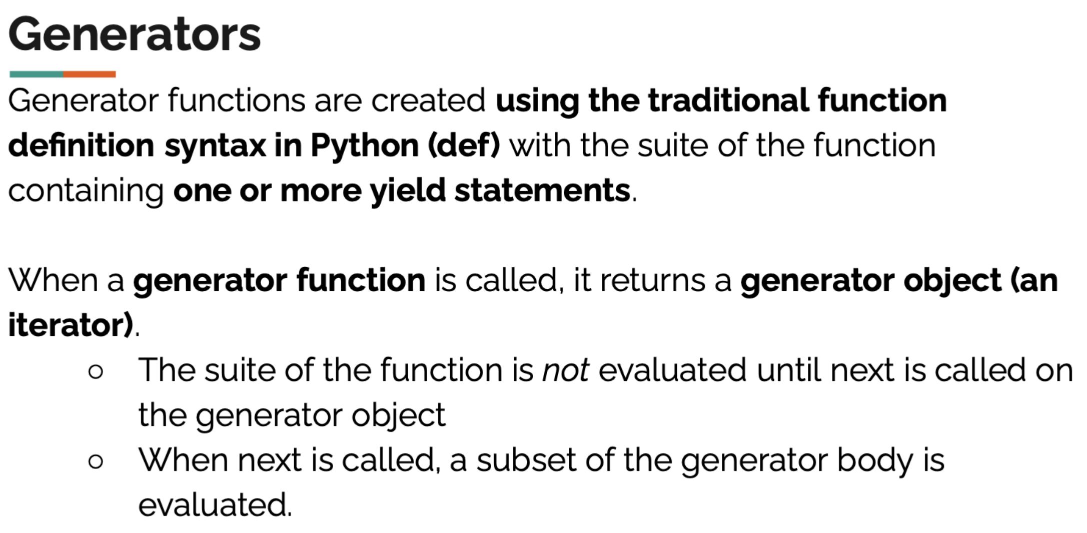
We can define custom iterators by writing a *generator function*, which returns a special type of iterator called a **generator**. 想要自定义迭代器的时候，写一个生成函数，返回的特别的 iterator 称为 generator

A generator function has at least one `yield` statement and returns a ***generator** object* when we call it, without evaluating the body of the generator function itself.生成函数至少有一个 yield 语句，调用时返回生成器对象而非计算生成函数本体?

When we first call `next` on the returned generator, then we will begin evaluating the body of the generator function until an element is yielded or the function otherwise stops (such as if we `return`). The generator remembers where we stopped, and will continue evaluating from that stopping point on the next time we call `next`.返回 generator 上用 next 才开始算函数主题的值，一直算到 yield 一个元素或者函数停止(如 return)。**生成器会记住停止的位置**，并会在我们下次调用时**从该点**继续计算 next

As with other iterators, if there are no more elements to be generated, then calling `next` on the generator will give us a `StopIteration`.没有元素了还 next 显示 StopIteration

To create a new generator object, we can call the generator function. Each returned generator object from a function call will separately keep track of where it is in terms of evaluating the body of the function. Notice that calling `iter` on a generator object doesn't create a new bookmark, but simply returns the existing generator object! 要创建一个新的生成器对象（类似 iterator），我们可以调用生成器函数（类似 iter）。从函数调用返回的每个生成器对象（类似 iter 后的 iterator）将跟踪它在评估函数体方面的位置。不过在生成器对象（iterator）上 iter 不会创建新的仍只返回本身



Generators

Generator functions are created **using the traditional function definition syntax in Python (def)** with the suite of the function containing **one or more yield statements**.

When a **generator function** is called, it returns a **generator object (an iterator)**.

- The suite of the function is *not* evaluated until next is called on the generator object
- When next is called, a subset of the generator body is evaluated.

纯文本

```
def countdown(n): #定义generator自定义迭代器（类似iter生成iterator）
    print("Beginning countdown!")
    while n >= 0:
        yield n #可以记忆位置类似iterator的next，generator必须用到yield
        n -= 1
    print("Blastoff!")

>>> c1, c2 = countdown(2), countdown(2) #生成了generator
>>> c1 is iter(c1)  # generator也是iterator
True #对generator用iter不变返回自身

>>> c1 is c2
False #countdown是创建新的不同的generator

>>> next(c1) #用next开始算函数主题值直到yield一个元素或函数停止不继续往下
Beginning countdown!
2

>>> next(c2) #这是2 另一个
```

```
Beginning countdown!
2
```

In a generator function, we can also have a `yield from` statement, which will **yield** each element **from** an iterator or iterable.在 generator 函数中，也可有 yield from 语句，它将 yield iterator 或 iterable 的每个元素。

纯文本

```
>>> def gen_list(lst):
...     yield from lst   #lst中迭代

>>> g = gen_list([1, 2])
>>> next(g)
1

>>> next(g)
2

>>> next(g)
StopIteration
```

Since generators are themselves iterators, this means we can use `yield from` to create recursive generators! 可以用在 generator 上因为 generator 属于 iterator，用 yield from 来创建递归 generators。类似用 next，每次返回不同状态新 iterator，下面例子用了 next 所以返回新的 rec_countdown(n–1)

WWPD

纯文本

```
>>> def rec_countdown(n): #也是一个生成器函数(类似iter)
...     if n < 0:
...         print("Blastoff!)
...     else:
...         yield n #类似return一个值，但可记位置，每次都出一个值
...         yield from rec_countdown(n-1) #无for，需此行递归下一个迭代器

>>> r = rec_countdown(2) #创建新的generator(类似iterator)
>>> next(r)
2

>>> next(r)
1
```

```
>>> next(r)
0

>>> next(r)
Blastoff!
StopIteration   #不会再rec_countdown(n-1)
```

Q4: Partition Generator

Construct the generator function `partition_gen`, which takes in a number `n` and returns an *n-partition iterator*. An *n-partition iterator* yields partitions of `n`, where a partition of `n` is a list of integers whose sum is `n`. The iterator should only return unique partitions; the order of numbers within a partition and the order in which partitions are returned does not matter.输入 n 返回 n-partition 的迭代器, *n-partition* 是和为 n 的 list, *n-partition* 内顺序和 partitions 顺序都无关紧要

Important: The skeleton code is only a suggestion; feel free to add or remove lines as you see fit.

纯文本

```
def partition_gen(n):
    """
    >>> for partition in partition_gen(4): # note: order doesn't matter
    ...     print(partition)
    [4]
    [3, 1]
    [2, 2]
    [2, 1, 1]
    [1, 1, 1, 1]
    """
    def yield_helper(j, k):#k最大可能，yield from情况k - 1,和为j
        if j == 0:
            yield []
        elif k > 0 and j > 0:
            for b in yield_helper(j-k, k):
            #helper里面只可能与自己有关
                yield [k]+b
            yield from yield_helper(j, k - 1)
    yield from yield_helper(n, n)
```

disc Q4:Generators WWPD

纯文本

```
>>> def infinite_generator(n):
...     yield n
...     while True:
...         n += 1
...         yield n
>>> next(infinite_generator) #括号，没输入参数
里面类似iter，没意义
TypeError

>>> gen_obj = infinite_generator(1) #创建了generator（iterator）
>>> next(gen_obj)
1 #只yield一个值

>>> next(gen_obj)
2

>>> list(gen_obj)
Infinite Loop
```

纯文本

```
>>> def rev_str(s): #iter
...     for i in range(len(s)): #无n-1无[1:]，不得不for i
...         yield from s[i::-1] #得到hehyeh,-1语法是从头到尾倒着，先h再eh再yeh，

>>> hey = rev_str("hey") #创建iterator，注意是字符串
>>> next(hey)
'h'    #这里是字符串加引号

>>> next(hey)
'e'

>>> next(hey)
'h'

>>> list(hey)
['y', 'e', 'h']   #注意[],''
```

纯文本

```
>>> def add_prefix(s, pre): #iter
...     if not pre: #pre没了的时候
...         return  #跳出
...     yield pre[0] + s  #出这里的值
...     yield from add_prefix(s, pre[1:])#对pre迭代递归，无n-1但有[1:]
>>> school = add_prefix("schooler", ["pre", "middle", "high"])
>>> next(school)
'preschooler'

>>> list(map(lambda x: x[:-2], school)) #map是映射，前是函数后是参数
#middleschooler, highschooler去掉er
['middleschool', 'highschool']  #这里类似list(school)
```

disc Q5: Filter–Iter 选择迭代器

Implement a generator function called `filter_iter(iterable, f)` that only yields elements of `iterable` for which `f` returns True.只 yield iterable 中使 f 真的元素

纯文本

```
def filter_iter(iterable, f):
    #法一用yield，正常循环return
    for i in iterable:
        if f(i):
            yield i

    #法二用yield from，含有递归思想
    yield from [i for i in iterable if f(i)] #递归yield from，无n-1格式不得不l
```

disc Q6: Primes Generator

Write a function `primes_gen` that takes a single argument n and yields all prime numbers less than or equal to n in decreasing order. Assume n≥1，也是一种选择迭代器

纯文本

```
def is_prime(n): #先判断是不是质数
```

```
    def helper(i): #因为要循环一个新的i参数因此需要helper
        if i > (n ** 0.5): # 数学上，直到此时真了
            return True
        elif n % i == 0:
            return False #一不小心就假了
        return helper(i + 1) #注意缩紧，前面都不执行就执行次行，有别的写法
    return helper(2) #跳过2

def primes_gen(n):
    #法一：yield+for循环
    for i in range(n,1,-1): #range另一种写法，语法与list相似但用括号，可跳过1
        if is_prime(i):
            yield i

    #法二：yield from 不用for循环用递归，天然是正着下降
    if n == 1:
        return      #跳出
    if is_prime(n):
        yield n     #正经输出
    yield from primes_gen(n-1)  #不用写for，有n-1格式，代替for循环的作用
```

disc Q7:Generate Preorder

Part 1:

First define the function `preorder`, which takes in a tree as an argument and returns a list of all the entries 所有条目 in the tree in the order 顺序 that `print_tree` would print them.This ordering of the nodes in a tree is called a preorder traversal.

定义函数 preorder 先序，接收一个树，并按照 print_tree 打印它们的顺序返回一个树中所有条目的列表。树中节点的这种排序称为先序遍历。此时与迭代无关，就是树 + 递归。

The following diagram shows the order that the nodes would get printed 节点打印顺序, with the arrows representing function calls.箭头表示函数调用

纯文本

```
def preorder(t):
    """Return a list of the entries in this tree in the order that they
    would be visited by a preorder traversal (see problem description).
    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [t
    >>> preorder(numbers)
    [1, 2, 3, 4, 5, 6, 7]
    >>> preorder(tree(2, [tree(4, [tree(6)])]))
    [2, 4, 6]
    """
    lst=[]              #天然加list外套
    for i in branches(t):  #i其实也是从0到n-1，branches其实与range类似
        lst+=preorder(i)  #如i==2时78910已经排好直接加后面
    return [label(t)]+lst #别忘了第一个，注意label加[]
```

Part 2:

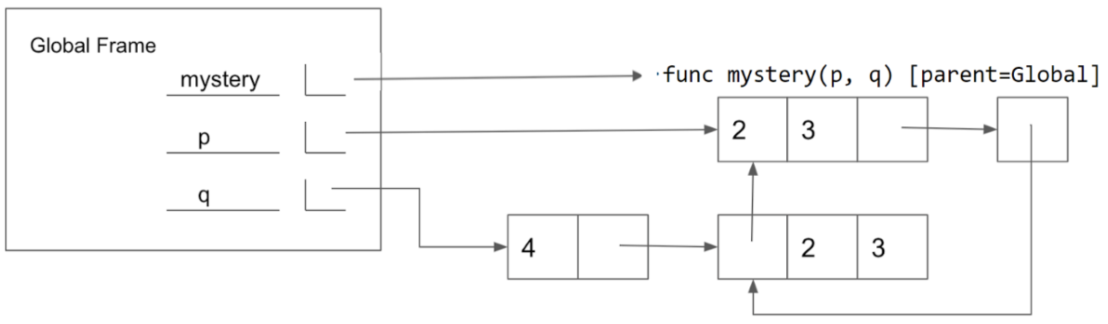
Similarly to `preorder` defined above, define the function `generate_preorder`, which takes in a tree as an argument and now instead yields the entries in the tree in the order that `print_tree` would print them.现在用 generator，不用递归，考虑 yield from 来代替

纯文本

```
def generate_preorder(t):
    yield label(t) #先前的不用递归用yield，yield和yield from都是不用[]
    for i in branches(t): #因为写不出来i-1这样的递归，是branch，用for i
        yield from generate_preorder(i)
```

disc Q8: Mystery Reverse Environment Diagram

Fill in the lines below so that the variables in the **global frame** are bound to the values below. Note that the image does not contain a full environment diagram. **You may only use brackets 方括号 colons 冒号, `p` and `q` in your answer.**



纯文本

纯文本

```
def mystery(p, q):
    p[1].extend(q)   #一个1处添加另一个
    q.append(p[1:])  #一个末尾添加另一个1后的部分，怎么看append还是extend？？？？？

#extend或一列——直接在后面加，因此下行是[4, [p]]，1.extend(上行)
#append或只一个框[]——新建一个框格，前后连接，上行是[2, 3]，后.append(下行1：)

p = [2, 3]
q = [4, [p]]
mystery(q, p)  #这里的反转很别致
```

hw5 Q1 Infinite Hailstone

hailstone sequence is defined:

1. Pick a positive integer `n` as the start.
2. If `n` is even, divide it by 2. 偶数除 2
3. If `n` is odd, multiply it by 3 and add 1. 基数*3+1
4. Continue this process until `n` is 1. 直到 1

Hint: Since `hailstone` returns a generator, you can `yield from` a call to `hailstone`

纯文本

```
def hailstone(n):
    """Yields the elements of the hailstone sequence starting at n.
       At the end of the sequence, yield 1 infinitely.

    >>> hail_gen = hailstone(10)
    >>> [next(hail_gen) for _ in range(10)]
    [10, 5, 16, 8, 4, 2, 1, 1, 1, 1]
    >>> next(hail_gen)
    1
    """
    yield n  #头出n
    if n == 1: #终点yield 1 infinitely
        yield from hailstone(n) #这行容易错写为1,'int' object is not iterable
    elif n % 2 == 0:
        yield from hailstone(n // 2)
    else:
        yield from hailstone(n * 3 + 1)
```

yield from 适当后面是 iterable 的时候用的，yield 则跟扑通 return 差不多

Q2: Merge (有点难)

Write a generator function `merge` that takes in two infinite generators `a` and `b` that are in increasing order without duplicates and returns a generator that has all the elements of both generators, in increasing order, without duplicates.写一个 generator 函数输入两个单增无重复的 generator(提示可以 next，不光是 generator 函数，连参数都是 generator，很罕见)，返回 generator 有所有元素，单增，无重复

纯文本

```
def merge(a, b):
    """
    >>> a = sequence(2, 3) # 2, 5, 8, 11, 14, ...
    >>> b = sequence(3, 2) # 3, 5, 7, 9, 11, 13, 15, ...
    >>> result = merge(a, b) # 2, 3, 5, 7, 8, 9, 11, 13, 14, 15
    """

    a1,b1=next(a),next(b) #注意每次next都会都下一个，害怕if操作中麻烦，另设a1b1
    while True:
        if a1==b1:
            yield a1
            a1,b1=next(a),next(b)   #都近一步
        elif a1<b1:
            yield a1
            a1=next(a)   #a1进一步
        else:
            yield b1
            b1=next(b)
```

hw5 Q3: Generate Permutations

Given a sequence of unique elements, a *permutation* of the sequence is a list containing the elements of the sequence in some arbitrary order. For example, `[2, 1, 3]`, `[1, 3, 2]`, and `[3, 2, 1]` are some of the permutations 变化 of the sequence `[1, 2, 3]`.

Implement `perms`, a generator function that takes in a sequence `seq` and returns a generator that yields all permutations of `seq`. For this question, assume that `seq` will not be empty.输入 seq 返回含有所有变化的生成器，默认 seq 非空,yield from 还是 yield 只需判断生成 generator（一般有生成器函数）还是别的，千万别写 return 这里是要 generator

Permutations may be yielded in any order. Note that the doctests test whether you are yielding all possible permutations, but not in any particular order. The built-in `sorted` function takes in an iterable object and returns a list containing the elements of the iterable in non-decreasing order.变化可以任意排序，内置的 sorted 函数将迭代对象升序排列

纯文本

```
def perms(seq): #好聪明的一道题！类似递归
    if len(seq)==0:
        yield [] #yield from不用管[]？yield要管？
    else:
        for p in perms(seq[1:]):
            for i in range(len(seq)):
                yield p[:i]+[seq[0]]+p[i:]
            #易错：一方面是yield不返回生成器，另一方面[seq[0]]记得带[]
```

hw5 Q4:Yield paths

Define a generator function `yield_paths` which takes in a tree `t`, a value `value`, and returns a generator object which yields each path from the root of `t` to a node that has label `value`.输入一个树一个值，返回所有可能树 generator object，提示用 yield/yield from

Each path should be represented as a list of the labels along that path in the tree. You may yield the paths in any order.

We have provided a skeleton for you. You do not need to use this skeleton, but if your implementation diverges significantly from it, you might want to think about how you can get it to fit the skeleton.

纯文本

```
def yield_paths(t, value):
    """Yields all possible paths from the root of t to a node with the labe
    value as a list.
    """

    if label(t)==value:
        yield [label(t)] #不像return，还可继续 #[[0,2], [0,2,1,2]]

    #else:这里不要else，因为label(t)==value后仍然可以继续找路径
    for b in branches(t):
        for i in yield_paths(b, value):
            yield [label(t)]+i
            #不能是yield [label(t),i]会嵌套，此外是yield因为并不返回generator
```

Q5: Remainder Generator 余数生成器

Like functions, generators can also be *higher-order*. For this problem, we will be writing `remainders_generator`, which yields a series of generator objects. `remainders_generator` takes in an integer `m`, and yields `m` different generators. The first generator is a generator of multiples of `m`, i.e. numbers where the remainder is 0. The second is a generator of natural numbers with remainder 1 when divided by `m`. The last generator yields natural numbers with remainder `m - 1` when divided by `m`.生成器也可以 higher order，yield 一系列生成器对象，输入 m（除数）生成 m 个不同的 generators，第一个余 0，第二个余 1...余 m-1，因此提示了要 helper 因为需要另一个未提及的参数

> *Hint*: To create a generator of infinite natural numbers, you can call the `naturals` function that's provided in the starter code.

纯文本

```
def remainders_generator(m):
    """
    Yields m generators. The ith yielded generator yields natural numbers w
    remainder is i when divided by m.
    """
    def helper(i):
        for x in naturals():
            if x%m==i:
                yield x

    for i in range(m):
        yield helper(i)       #不是remainders_generator，不用yield from
```

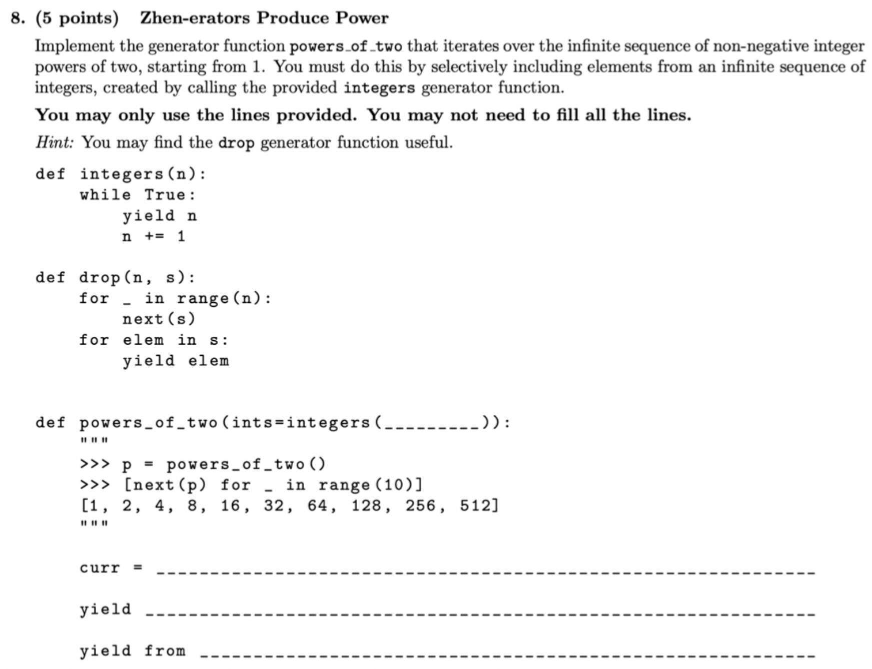
Lab7 Q1: Apply That Again

Implement `amplify`, a generator function that takes a one-argument function `f` and a starting value `x`. The element at index *k* that it yields (starting at 0) is the result of applying `f` *k* times to `x`. It terminates 终止当 x 为 False 时 whenever the next value it would yield is a falsy value, such as `0`, `""`, `[]`, `False`, etc.

纯文本

```
def amplify(f, x):
    """Yield the longest sequence x, f(x), f(f(x)), ... that are all true v

    >>> list(amplify(lambda s: s[1:], 'boxes'))
    ['boxes', 'oxes', 'xes', 'es', 's']
    >>> list(amplify(lambda x: x//2-1, 14))
    [14, 6, 2]
    """

    while x:
        yield x
        x = f(x)
```

Zhen-erators (Su16 Final)



8. (5 points) **Zhen-erators Produce Power**

Implement the generator function `powers_of_two` that iterates over the infinite sequence of non-negative integer powers of two, starting from 1. You must do this by selectively including elements from an infinite sequence of integers, created by calling the provided `integers` generator function.

You may only use the lines provided. You may not need to fill all the lines.

Hint: You may find the `drop` generator function useful.

```
def integers(n):
    while True:
        yield n
        n += 1

def drop(n, s):
    for _ in range(n):
        next(s)
    for elem in s:
        yield elem


def powers_of_two(ints=integers(_________)):
    """
    >>> p = powers_of_two()
    >>> [next(p) for _ in range(10)]
    [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
    """

    curr = ______________________________________________________________

    yield _______________________________________________________________

    yield from __________________________________________________________
```

Zhen-erators (Su16 Final)

```
def powers_of_two(ints=integers(1)):
    """
    >>> p = powers_of_two()
    >>> [next(p) for _ in range(10)]
    [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
    """

    curr = next(ints)

    yield curr

    yield from powers_of_two(drop(curr-1, ints))
```

有点神奇，drop 是对 s 跳过 n 格再 yield，如 4 到 8 之间 yield3，8 到 16 之间 yield7，

curr 是书签，ints 是正整数书，curr 一开始是 1，之后因为 for 循环不断前进，

返回所有上升路径的叶子

Big Leaf (Fa 19 Final)

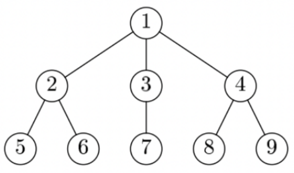
```
def tops(t):
    if t.is_leaf():                # base case: if t is a leaf, it is
        yield t.label              # definitely strictly increasing
    else:
        for b in t.branches:       # for each branch, consider strictly increasing leaves
            if t.label < b.label:  # only keep ones where t.label -> b.label is also increasing
                yield from tops(b)
```

`yield` vs. `yield from`
We want to yield individual labels
tops(b) is a generator. We want to yield the elements that tops(b) generates.
This suggests that we should use `yield from`

给树，生成 label 的 list，一层 level 从左到右，helper 函数先解决 k 层，因为要记录之前，所以用 yield

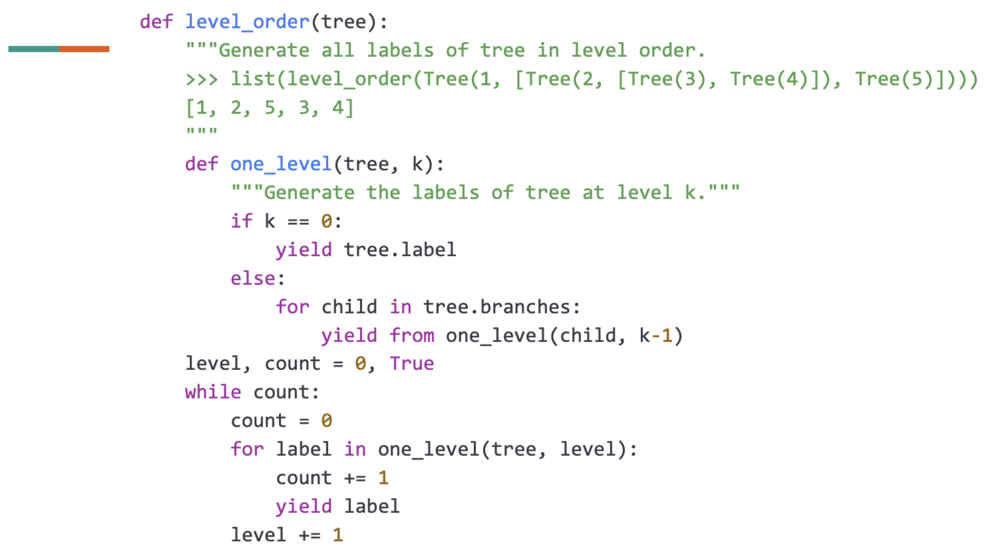
Level-Headed Trees (Fa17 Final)

5. (13 points) Level-Headed Trees A *level-order traversal* of a tree, T, traverses the root of T (level 0), then the roots of all the branches of T (level 1) left to right, then all the roots of the branches of the nodes traversed in level 1, (level 2) and so forth. Thus, a level-order traversal of the tree



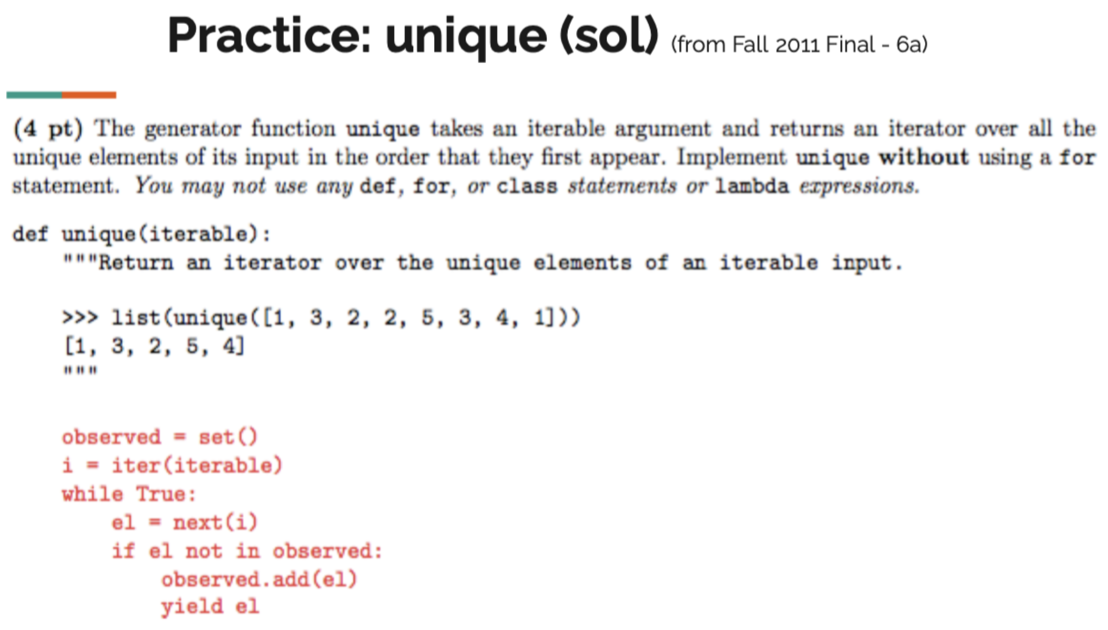
visits nodes with labels 1, 2, 3, 4, 5, 6, 7, 8, 9 in that order.

Level-Headed Trees (Fa17 Final)



```python
def level_order(tree):
    """Generate all labels of tree in level order.
    >>> list(level_order(Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])))
    [1, 2, 5, 3, 4]
    """
    def one_level(tree, k):
        """Generate the labels of tree at level k."""
        if k == 0:
            yield tree.label
        else:
            for child in tree.branches:
                yield from one_level(child, k-1)
    level, count = 0, True
    while count:
        count = 0
        for label in one_level(tree, level):
            count += 1
            yield label
        level += 1
```

count 用途是什么？若这一层没有 label 即叶子了，不会 count+1，也不执行 while 循环了

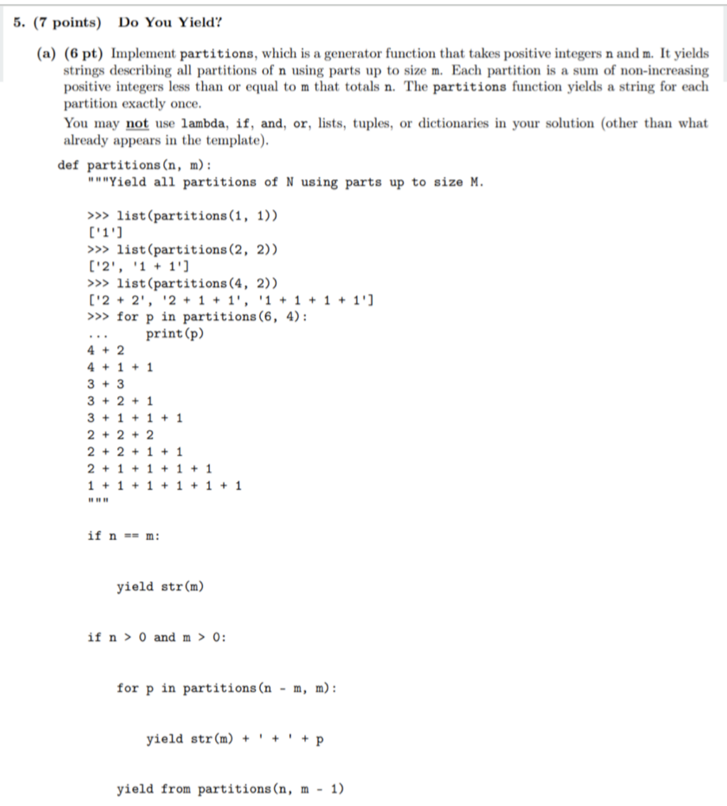


Practice: unique (sol) (from Fall 2011 Final - 6a)

(4 pt) The generator function `unique` takes an iterable argument and returns an iterator over all the unique elements of its input in the order that they first appear. Implement `unique` **without** using a `for` statement. *You may not use any* `def`, `for`, *or* `class` *statements or* `lambda` *expressions.*

```python
def unique(iterable):
    """Return an iterator over the unique elements of an iterable input.

    >>> list(unique([1, 3, 2, 2, 5, 3, 4, 1]))
    [1, 3, 2, 5, 4]
    """

    observed = set()
    i = iter(iterable)
    while True:
        el = next(i)
        if el not in observed:
            observed.add(el)
            yield el
```

Fa19 MT2 5(a)



5. (7 points) Do You Yield?

(a) **(6 pt)** Implement `partitions`, which is a generator function that takes positive integers `n` and `m`. It yields strings describing all partitions of `n` using parts up to size `m`. Each partition is a sum of non-increasing positive integers less than or equal to `m` that totals `n`. The `partitions` function yields a string for each partition exactly once.

You may not use `lambda`, `if`, `and`, `or`, lists, tuples, or dictionaries in your solution (other than what already appears in the template).

```
def partitions(n, m):
    """Yield all partitions of N using parts up to size M.

    >>> list(partitions(1, 1))
    ['1']
    >>> list(partitions(2, 2))
    ['2', '1 + 1']
    >>> list(partitions(4, 2))
    ['2 + 2', '2 + 1 + 1', '1 + 1 + 1 + 1']
    >>> for p in partitions(6, 4):
    ...     print(p)
    4 + 2
    4 + 1 + 1
    3 + 3
    3 + 2 + 1
    3 + 1 + 1 + 1
    2 + 2 + 2
    2 + 2 + 1 + 1
    2 + 1 + 1 + 1 + 1
    1 + 1 + 1 + 1 + 1 + 1
    """

    if n == m:

        yield str(m)

    if n > 0 and m > 0:

        for p in partitions(n - m, m):

            yield str(m) + ' + ' + p

        yield from partitions(n, m - 1)
```

第一个 n==m 不好想，后面有一个 for+yield，和一个 yield from（–1）

str(m)