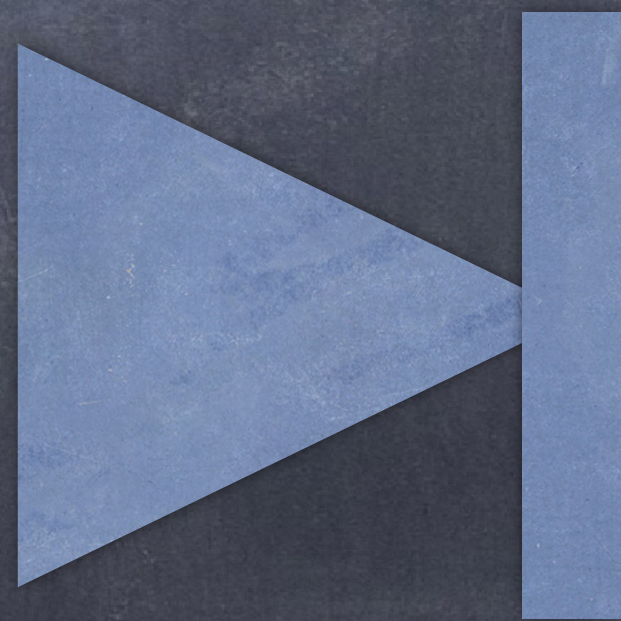


迭代器和生成器



迭代器 (Iterator)

可迭代 (Iterables)

● 列表、元组、字典、Ranges、还有字符串 (还有集合) 都是可迭代的对象

```
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]
```

```
ranked_chocolates = ("Dark", "Milk", "White")
```

```
prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen": 19.99}
```

```
best_topping = "pineapple"
```


迭代

```
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]  
for item in my_order:  
    print(item)  
lowered = [item.lower() for item in my_order]
```

```
ranked_chocolates = ("Dark", "Milk", "White")  
for chocolate in ranked_chocolates:  
    print(chocolate)
```

```
prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen":  
19.99}  
for product in prices:  
    print(product, " costs ", prices[product])  
discounted = { item: prices[item] * 0.75 for item in prices }
```

```
best_topping = "pineapple"  
for letter in best_topping:  
    print(letter)
```

我们可以对
可迭代的对
象迭代:

迭代子

● 一个迭代子是一个可以提供序列化访问值的对象，其一次访问一个值！

◆ `iter(iterable)` 返回一个在 `iterable` 对象之上的迭代器

◆ `next(iterator)` 返回迭代器的下一个元素

迭代子

```
toppings = ["pineapple", "pepper", "mushroom", "roasted red pepper"]  
  
topperator = iter(toppings)  
next(iter) # 'pineapple'  
next(iter) # 'pepper'  
next(iter) # 'mushroom'  
next(iter) # 'roasted red pepper'  
next(iter) # ❌ StopIteration exception
```

迭代子是可变的吗? 🤔

demo

处理 StopIteration

- StopIteration 是一个会终止程序正常运行的“异常”
(Exception)
- 处理异常应该使用 try/except

处理 StopIteration

```
ranked_chocolates = ("Dark", "Milk", "White")
```

```
chocolaterator = iter(ranked_chocolates)
```

```
print(next(chocolaterator))
```

```
print(next(chocolaterator))
```

```
print(next(chocolaterator))
```

```
try:
```

```
    print(next(chocolaterator))
```

```
except StopIteration:
```

```
    print("No more left!")
```


处理 StopIteration

配合 while 来处理迭代

```
ranked_chocolates = ("Dark", "Milk", "White")  
chocolaterator = iter(ranked_chocolates)
```

```
try:  
    while True:  
        choco = next(chocolaterator)  
        print(choco)  
except StopIteration:  
    print("No more left!")
```


Iterators vs. For Loops

```
ranked_chocolates = ("Dark", "Milk", "White")  
chocorator = iter(ranked_chocolates)
```

```
try:  
    while True:  
        choco = next(chocorator)  
        print(choco)  
except StopIteration:  
    print("No more left!")
```

```
ranked_chocolates = ("Dark", "Milk", "White")  
for chocolate in ranked_chocolates:  
    print(chocolate)
```

Actually, a for loop is just syntactic sugar! 🍬

再次回顾for语句

```
for <name> in <expression>:  
    <suite>
```

语义：

- 1. Python 首先求值头部的<expression>, 确保其产生一个Iterable
- 2. Python 得到iterable的迭代器
- 3. Python 利用iterable得到其next value, 并绑定到当前帧的name
- 4. Python 执行<suite>中的语句
- 5. Python 重复上述操作直到 StopIteration error

```
iterator = iter(<expression>)  
try:  
    while True:  
        <name> = next(iterator)  
        <suite>  
except StopIteration:  
    pass
```


内部的 `__next__()` 和 `__iter__()`

- `iter()` 函数本质上调用该对象“自己”的 `__iter__()`

```
ranked_chocolates = ("Dark", "Milk", "White")  
chocorator1 = iter(ranked_chocolates)  
chocorator2 = ranked_chocolates.__iter__()
```

什么叫自己的？

- `next()` 函数本质上调用该迭代器“自己”的 `__next__()`

```
ranked_chocolates = ("Dark", "Milk", "White")  
chocolate1 = next(chocorator1)  
chocolate2 = chocorator2.__next__()
```


比较两种迭代

for

```
ranked_chocolates = ("Dark", "Milk", "White")  
for chocolate in ranked_chocolates:  
    print(chocolate)
```

Iterator

```
ranked_chocolates = ("Dark", "Milk", "White")  
chocorator = iter(ranked_chocolates)  
try:  
    while True:  
        print(next(chocorator))  
except StopIteration:  
    pass
```


行为相同不等于实现相同

👁 For循环和迭代器的行为是一样的，但是Python实现是不同的

	10,000 runs	1,000,000 runs
For loop	3.2 milliseconds	336 milliseconds
Iterator	8.3 milliseconds	798 milliseconds

具体实现的不同

用dis模块来查看具体的不同

```
import dis
```

```
def for_version():
```

```
    y = 0
```

```
    for x in [1, 2, 3]:
```

```
        y += x * 2
```

```
dis.dis(for_version)
```

```
dis.dis(iter_version)
```

```
def iter_version():
```

```
    _gen_ = iter([1, 2, 3])
```

```
    y = 0
```

```
    try:
```

```
        while True:
```

```
            y += next(_gen_) * 2
```

```
    except StopIteration:
```

```
        pass
```

1	>>	8 FOR_ITER	16 (to 26)	1	15 >>	20 LOAD_FAST	1 (y)
2		10 STORE_FAST	1 (x)	2		22 LOAD_GLOBAL	1 (next)
3	7	12 LOAD_FAST	0 (y)	3		24 LOAD_FAST	0 (_gen_)
4		14 LOAD_FAST	1 (x)	4		26 CALL_FUNCTION	1
5		16 LOAD_CONST	3 (2)	5		28 LOAD_CONST	2 (2)
6		18 BINARY_MULTIPLY		6		30 BINARY_MULTIPLY	
7		20 INPLACE_ADD		7		32 INPLACE_ADD	
8		22 STORE_FAST	0 (y)	8		34 STORE_FAST	1 (y)
9		24 JUMP_ABSOLUTE	8	9		36 JUMP_ABSOLUTE	20

一些返回迭代子的函数

函数	描述
<code>reversed(sequence)</code>	返回 <code>sequence</code> 的一个逆向的迭代子
<code>zip(*iterables)</code>	返回一个迭代器，其元素是一个个元组，每个元组中的元素是每个 <code>iterable</code> 的元素。
<code>map(func, iterable, ...)</code>	返回一个迭代器，每个元素是 <code>func(x)</code> ，其中 <code>x</code> 是 <code>iterable</code> 中的元素
<code>filter(func, iterable)</code>	返回 <code>iterable</code> 的一个迭代器，但是过滤掉其中使得 <code>func</code> 返回为 <code>False</code> 的元素

一个有用的小细节

① 对一个迭代器上调用 `iter()` 会返回该迭代器

```
numbers = ["一", "二", "三"]  
num_iter = iter(numbers)  
num_iter2 = iter(num_iter)  
num_iter is num_iter2
```

② 这也是为什么下面代码能够运行的原因

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
for num in filter(lambda x: x % 2 == 0, nums):  
    print(num)
```


一些返回可迭代对象的函数

函数	描述
<code>list(iterable)</code>	返回一个 <code>list</code> 包含 <code>iterable</code> 里的所有元素
<code>tuple(iterable)</code>	返回一个 <code>tuple</code> 包含 <code>iterable</code> 里的所有元素
<code>sorted(iterable)</code>	返回一个排序好的 <code>list</code> ，包含 <code>iterable</code> 里的所有元素

生成器 (Generator)

生成器 (Generator)

- 一个生成器就是一种迭代器，其产生由生成器函数决定的元素
- 一个生成器函数使用 **yield** 而不是 **return** :

generator function		generator
<pre>def evens(): num = 0 while num < 10: yield num num += 2</pre>		<pre>evengen = evens() next(evengen) # 0 next(evengen) # 2 next(evengen) # 4 next(evengen) # 6 next(evengen) # 8 next(evengen) # ❌ StopIteration exception</pre>

生成器工作流程

```
def evens():  
    num = 0  
    while num < 2:  
        yield num  
        num += 2
```

```
gen = evens()
```

```
next(gen)
```

```
next(gen)
```

- 当生成器函数被调用时，其直接返回一个迭代器（没有进入函数体）。
- 当 `next()` 被调用时，其进入函数体，从上一次的终止点（初始就是函数体的第一句），运行到下一次的 `yield` 语句。
- 如果找到了 `yield` 语句，那么其在下一句停止（这一次的终止点，作为下一次的起点），并返回 `yield` 语句中的表达式的值。
- 如果找不到 `yield` 语句，则在函数的末端终止，并产生一个 `StopIteration` 的异常。

对生成器进行循环

● 我们可以对生成器进行循环，因为其本质也是一个迭代器

```
def evens(start, end):  
    num = start + (start % 2)  
    while num < end:  
        yield num  
        num += 2
```

```
for num in evens(12, 60):  
    print(num)
```


对生成器进行循环

```
def evens(start, end):  
    num = start + (start % 2)  
    while num < end:  
        yield num  
        num += 2  
for num in evens(12, 60):  
    print(num)
```

对比

```
evens = [num for num in range(12, 60) if num % 2 == 0]  
# Or = filter(lambda x: x % 2 == 0, range(12, 60))  
for num in evens:  
    print(num)
```


为什么需要生成器

- 生成器是惰性的 (Lazy) : 他们只在需要时才产生下一个项
- 很多时候你不需要一次生成整个序列, 而是只需要部分。

为什么需要生成器

```
def find_matches(filename, match):  
    for line in open(filename):  
        if line.find(match) > -1:  
            yield line
```

```
line_iter = find_matches('frankenstein.txt', "!")  
next(line_iter)  
next(line_iter)
```

```
def find_matches(filename, match):  
    matched = []  
    for line in open(filename):  
        if line.find(match) > -1:  
            matched.append(line)  
    return matched
```

```
matched_lines = find_matches('frankenstein.txt', "!")  
matched_lines[0]  
matched_lines[1]
```

有了generator, 可以在使用时才获得想要的项。而很多时候并不事先知道到底要使用“多少”，因此如果没有generator的话，需要整个都生成，这显然是低效的！而且会ran out of memeroy！

对可迭代对象直接yield

```
def a_then_b(a, b):  
    for item in a:  
        yield item  
    for item in b:  
        yield item
```

```
list(a_then_b(["Apples", "Aardvarks"], ["Bananas", "BEARS"]))
```

对比

another syntactic sugar! 🍬

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

```
list(a_then_b(["Apples", "Aardvarks"], ["Bananas", "BEARS"]))
```

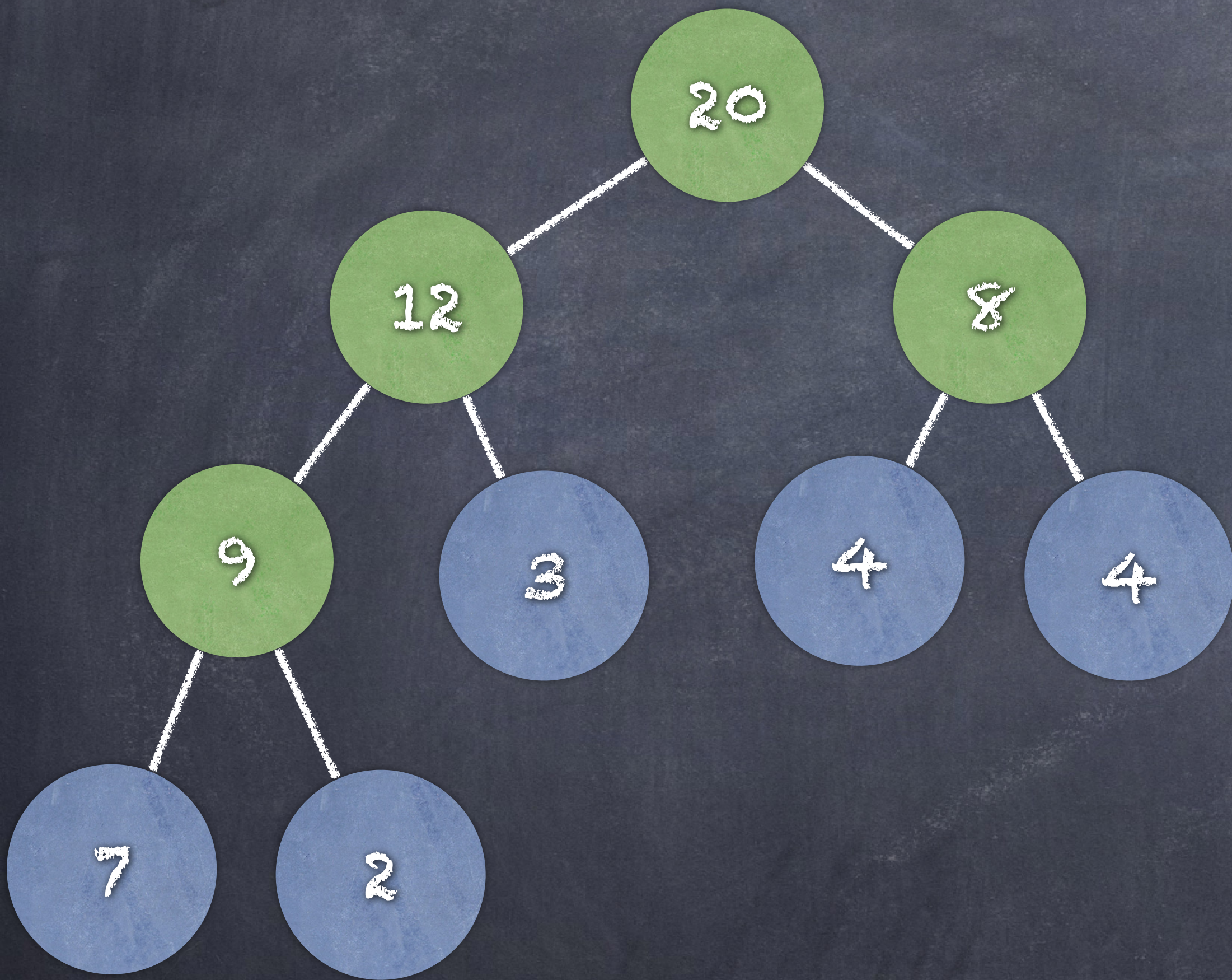

递归的yield

```
def factorial(n, accum):  
    if n == 0:  
        yield accum  
    else:  
        for result in factorial(n - 1, n * accum):  
            yield result  
  
for num in factorial(3, 1):  
    print(num)
```

yield from还可以从generator
function中获得generator:

```
def factorial(n, accum):  
    if n == 0:  
        yield accum  
    else:  
        yield from factorial(n - 1, n * accum)  
  
for num in factorial(3, 1):  
    print(num)
```


树的递归生成器



A pre-order traversal of the tree:

```
def nodes(t):
```

```
    yield label(t)
```

```
    for c in branches(t):
```

```
        yield from nodes(c)
```

```
t = tree(20, [tree(12,
                    [tree(9,
                        [tree(7), tree(2)]),
                     tree(3)]),
               tree(8,
                    [tree(4), tree(4)])])
```

```
node_gen = nodes(t)
```

```
next(node_gen)
```


Any questions ?