

Iterator Zen



Global Knowledge®

Objectives

- Add custom iteration to your classes
- Use generator methods to efficiently write custom iteration
- Filter and transform data with list comprehensions
- Compare list comprehensions and generator expressions
- Perform advanced iteration with the `itertools` module

Recall [for in loops]

- For loops in Python fundamentally work on iterable sets
 - There is no index-based looping construct
 - Many types are iterable
 - lists, sets, dictionaries, strings, files, classes, ...

```
name = "Jeff"

for ch in name:
    print( ch, end='-' )

# prints J-e-f-f-
```

for in loops [custom classes]

- Some classes (e.g. dict) are iterable
- What happens if we try it with our custom classes?

```
class Cart(object):  
    # ...  
  
cart = Cart()  
cart.add('album', 7.99)  
cart.add('book ', 19.99)  
  
for item in cart:  
    print( item )  
  
# Boom!  
# TypeError: 'Cart' object is not iterable
```


for in loops [implementing iteration]

```
class Cart:
    def __init__(self):
        self.__items = []

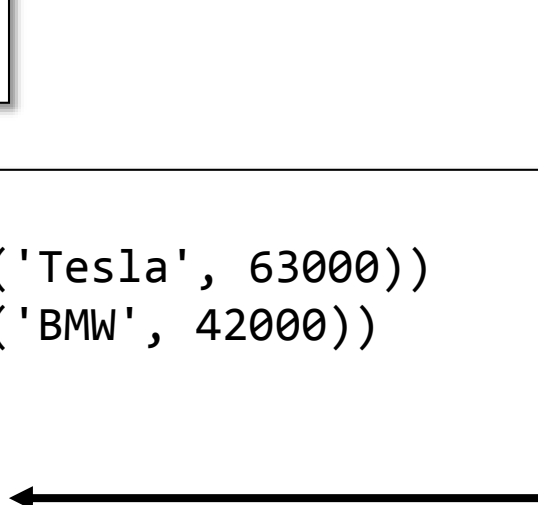
    def add(self, cartItem):
        self.__items.append(cartItem)

    def __iter__(self):
        return self.__items.__iter__()
```

Iterable classes define a
`__iter__` method



We can now iterate
over our class.



```
cart = Cart()
cart.add(CartItem('Tesla', 63000))
cart.add(CartItem('BMW', 42000))

total = 0
for item in cart:
    total += item.price

# total is 105,000
```

for in loops [implementing iteration, directly]

```
class CartIterationHelper:
    def __init__(self, list):
        self.list = list[:]

    def __next__(self):
        if not self.list:
            raise StopIteration()

        return self.list.pop()
```

An object becomes directly iterable (not iterable via delegation as before) by defining `__next__`.

We can then use this class our cart to add iteration.

```
class Cart:
    def __iter__(self):
        return CartIterationHelper(self.__items)
```

Generator methods [motivation]

- Suppose our set has to be computed.

evens returns an iterable set

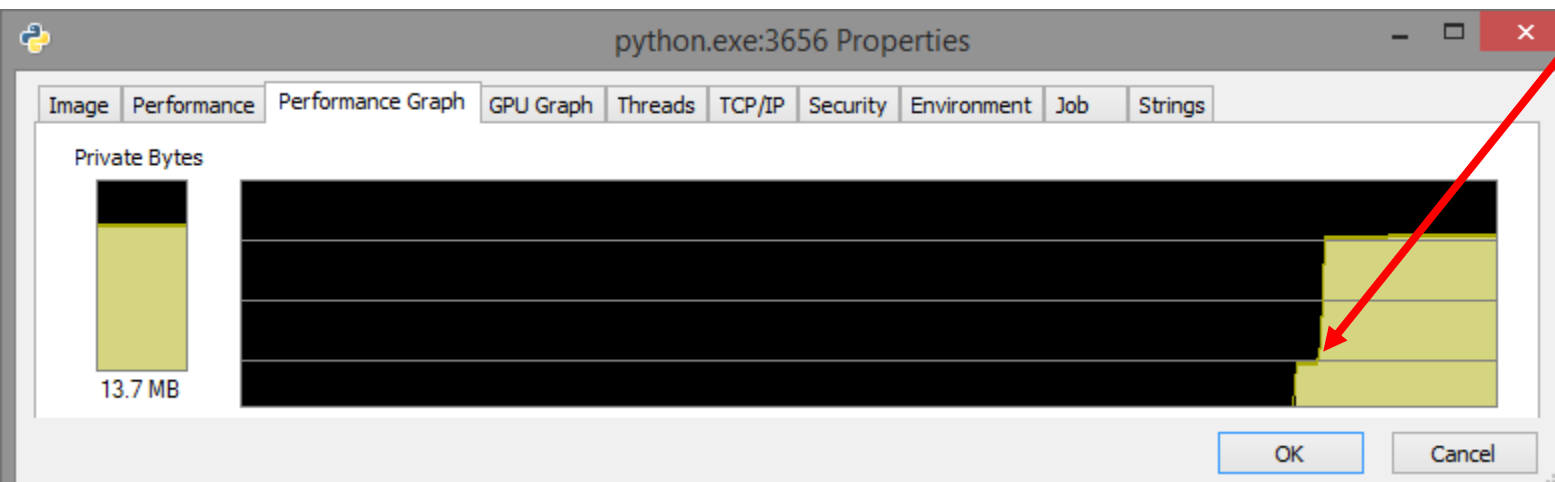


```
def evens(limit):  
    nums = []  
    for n in range(limit):  
        if n % 2 == 0:  
            nums.append(n)  
    return nums
```

What happens if limit is very large?
What if limit is unknown in advance?



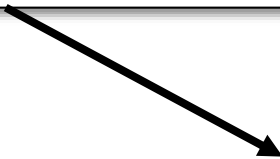
Calling **evens**(1,000,000)



Generator methods [motivation]

- Python has a much easier way to implement iteration *efficiently*.

```
def evens_improved():  
    n = 0  
    while True:  
        yield n  
        n+=2
```

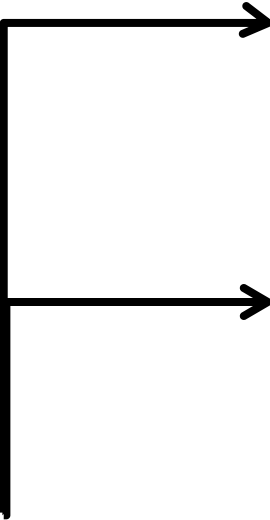


```
for n in evens_improved():  
    if n >= 10:  
        break  
    print(n, end=', ')
```

Note: the return value is not just computed more efficiently, it's an infinite set.

Generator delegation: yield from [motivation]

- When combining generator methods, **yield from** keyword can simplify nested iteration.
 - requires Python 3.3+



```
Class OrderedTree:
    def items_in_order(self, node = None):
        node = self.root

        for d in self.items_in_order(node.left):
            yield d

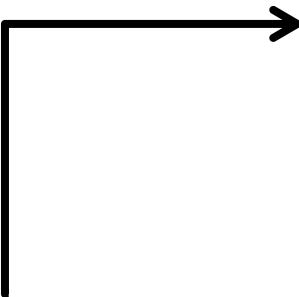
        yield node.data

        for d in self.items_in_order(node.right):
            yield d
```

Why do we need to loop over these recursive calls?
Because yield works on **single items only**.

Generator delegation: yield from

- When combining generator methods, **yield from** keyword can simplify nested iteration.
 - requires Python 3.3+



```
Class OrderedTree:  
    def items_in_order(self, node = self.root):  
  
        yield from self.items_in_order(node.left)  
        yield node.data  
        yield from self.items_in_order(node.right)
```

This is much cleaner!

List comprehensions

- Python has a concise and local mechanism to
 - transform iterables
 - filter iterables

```
# working data for the next few examples
```

```
class Person:
```

```
    def __init__(self, name, age, hobbies):  
        self.name = name  
        self.age = age  
        self.hobbies = hobbies
```

```
people = [  
    Person("Jeff", 42, ['tennis', 'hockey', 'football']),  
    Person("Michael", 40, ['biking', 'hiking', 'motocross']),  
    Person("Pierre", 39, ['biking', 'kite boarding']),  
    Person("Stacey", 32, ['skiing']),  
]
```

List comprehensions [definition]

- How would we find people who:
 - have biking as a hobby
 - retrieve their names uppercased

```
bikers = [  
    p.name.upper()  
    for p in people  
    if 'biking' in p.hobbies  
]  
  
# bikers: MICHAEL, PIERRE
```

First line is the select or projection
(can be separate method).

Second line names 'loop' variable
and source iterable.

Third line is the filter or where clause.

The type of bikers is a **list**.

Generator expressions [defining]

- Generator expressions are a more efficient form of comprehensions
 - list comprehensions are full computed then returned
 - generator expressions are lazily evaluated

```
bikers = (  
    p.name.upper()  
    for p in people  
    if 'biking' in p.hobbies  
)  
  
# bikers: MICHAEL, PIERRE
```

Generator expressions are only executed as they are consumed

The type of bikers is a **<class 'generator'>**.

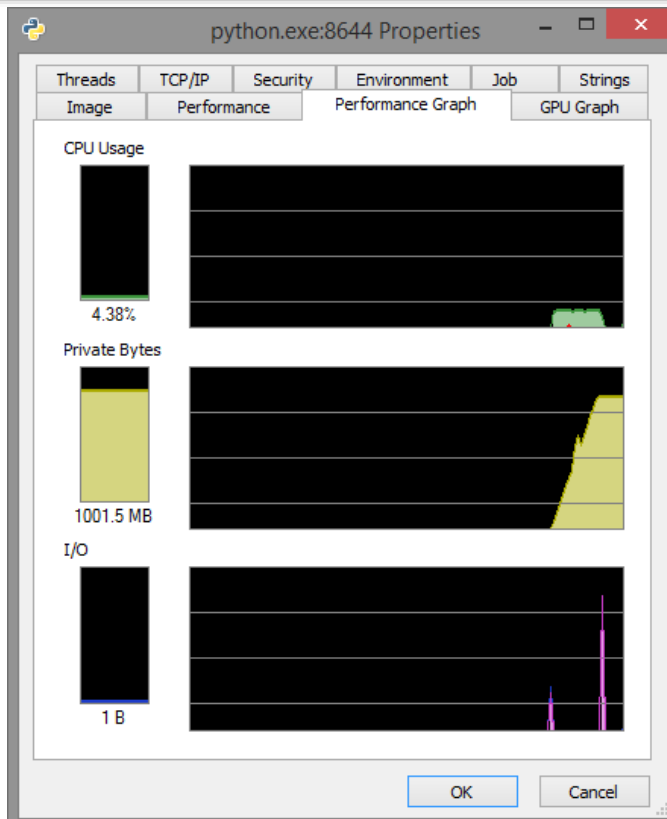
Generator expressions vs. list comprehensions

- Because list comprehensions return lists, they must evaluate all elements at once. This can be expensive.

List comprehension

```
cmp = [x/(x - 100.0)
        for x in range(101, 50000000)]
m = max(cmp)
```

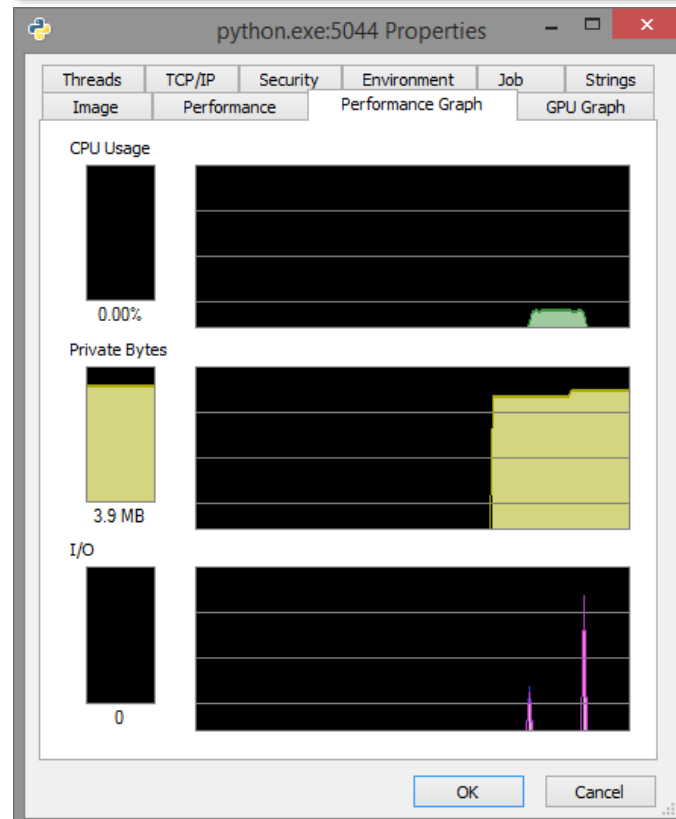
**1 GB
memory**



generator expression

```
gen = (x/(x - 100.0)
        for x in range(101, 50000000))
m = max(gen)
```

**0.004 GB
memory**



Set and Dict Comprehensions

- Python has similar syntax for sets and dictionaries

```
# Calculate the prime numbers to 100
```

```
from math import sqrt
```

```
nonprimes = {  
    j  
    for i in range(2,int(sqrt(100)))  
    for j in range(i*2, 100, i) }
```

```
primes = [  
    i  
    for i in range(2,100)  
    if i not in nonprimes ]
```

```
# Compute checksums for files in current directory
```

```
import os, hashlib
```

```
checksums = {  
    f : hashlib.md5(open(f, 'rb').read()).digest()  
    for f in os.listdir('.')  
    if os.path.isfile(f) }
```

Itertools

- The itertools module has many advanced iteration methods. Here are a few:

Iterator	Example
<code>count()</code>	<code>count(10) --> 10 11 12 13 14 ...</code>
<code>cycle()</code>	<code>cycle('ABCD') --> A B C D A B C D ...</code>
<code>chain()</code>	<code>chain('ABC', 'DEF') --> A B C D E F</code>
<code>dropwhile()</code>	<code>dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1</code>
<code>groupby()</code>	sub-iterators grouped by value of <code>keyfunc(v)</code>
<code>islice()</code>	<code>islice('ABCDEFGH', 2, None) --> C D E F G</code>
<code>takewhile()</code>	<code>takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4</code>
<code>permutations()</code>	r-length tuples, all possible orderings, no repeated elements

Summary

- Use the `__iter__` magic method to enable iteration
- Create efficient and simple iteration via the `yield` keyword
- List comprehensions provide a query language for iterable objects
- Generator expressions can be dramatically more efficient than list comprehensions
- Perform advanced iteration with the `itertools` module