

Iterables vs. Iterators vs. Generators

Let's explore the differences between...

- a container
- an iterable
- an iterator
- a generator
- a generator expression
- a {list, set, dict} comprehension



Containers

- data structures which hold elements
- support membership tests
- live in memory
- typically hold all their values in memory
- e.g., string, list, tuple, set, dict
- an object is a container when it can be asked whether it *contains* a certain element

```
In [53]: 1 in [1, 2, 3], 0 in [1, 2, 3]
```

```
Out[53]: (True, False)
```

```
In [54]: 4 in {4, 5, 6}, 1 in {4, 5, 6}
```

```
Out[54]: (True, False)
```

```
In [55]: 44 in ('Obama', 'Barack', 44, 2008, 'left')
```

```
True
```

Out[55]:

In [56]:

```
# for dicts, membership checks the keys, not the values
44 in { 43: 'Bush', 44: 'Obama'}, 'Bush' in { 43: 'Bush', 44: 'Obama' }
```

Out[56]: (True, False)

In [57]:

```
'J' in 'Steve Jobs', 'Job' in 'Steve Jobs', 'Jobs' not in 'Carlos Jobim'
```

Out[57]: (True, True, True)

Iterables

- any object, not necessarily a data structure, that can return an iterator (with the purpose of returning all of its elements)
- the `__iter__()` function returns an iterator
 - ...therefore, any object which has the `__iter__()` method is an iterable
- most containers are also iterable
- many more things are iterable as well (e.g., open files, open sockets, etc.)

In [58]:

```
mylist = [1, 2, 3]
iter1 = iter(mylist)
print(type(iter1))
iter2 = mylist.__iter__() # iter() maps to __iter__()
print(type(iter2))
next(iter1)
```

```
<class 'list_iterator'>
<class 'list_iterator'>
```

Out[58]: 1

In [59]:

```
iter1.__next__() # next() maps to __next__()
#next(iter1)
```

Out[59]: 2

In [60]:

```
next(iter1)
```

```
Out[60]: 3
```

```
In [61]: next(iter1)
```

```
-----  
StopIteration                                Traceback (most recent call last)  
/tmp/ipykernel_10454/3389250325.py in <module>  
----> 1 next(iter1)
```

```
StopIteration:
```

```
In [62]: next(iter2)
```

```
Out[62]: 1
```

```
In [63]: type(mylist), type(iter1), type(iter2)
```

```
Out[63]: (list, list_iterator, list_iterator)
```

```
In [64]: # a list is iterable, but it is not its own iterator  
next(mylist)
```

```
-----  
TypeError                                 Traceback (most recent call last)  
/tmp/ipykernel_10454/12956728.py in <module>  
      1 # a list is iterable, but it is not its own iterator  
----> 2 next(mylist)
```

```
TypeError: 'list' object is not an iterator
```

```
In [65]: iter(mylist) is mylist
```

```
Out[65]: False
```

```
In [66]: mylistiter = iter(mylist)  
print('%x %x' % (id(mylist), id(mylistiter)))
```

```
7f3cab21f580 7f3ca80eaf70
```

```
In [67]: iter(mylistiter) is mylistiter
```

```
Out[67]: True
```

When we write...

```
mylist = [1, 2, 3]
for x in mylist:
    ...
```

...this is what happens



We can see this by disassembling the Python code...

```
In [68]: import dis
mylist = [1, 2, 3]
total = 0
dis.dis('for item in mylist: total += item')
```

```
 1      0 LOAD_NAME              0 (mylist)
        2 GET_ITER
    >>  4 FOR_ITER               12 (to 18)
        6 STORE_NAME              1 (item)
        8 LOAD_NAME                2 (total)
       10 LOAD_NAME                1 (item)
       12 INPLACE_ADD
       14 STORE_NAME              2 (total)
       16 JUMP_ABSOLUTE           4
    >> 18 LOAD_CONST              0 (None)
       20 RETURN_VALUE
```

So what is an iterator?

- a stateful object that produces the next value when you call `next()` on it

- any object that has a `__next__()` method is therefore an iterator
- how it produces a value is irrelevant
- in other words, an iterator is a value factory
 - each time you ask it for "the next" value, it knows how to compute it because it holds internal state

In [69]:

```
# Let's see how an iterator works...
mylist = [13, 46, -3, 'Go!']
myiter = iter(mylist) # get the list iterator

try:
    while True:
        val = next(myiter)
        print(val, end=' ')
except StopIteration:
    print('Stop!')
```

13 46 -3 Go! Stop!

Let's build our own iterator!

In [70]:

```
class Fibonacci(object):
    def __init__(self):
        self.prev = 0
        self.curr = 1

    def __iter__(self):
        return self

    def __next__(self):
        """
        each call to next() does two important things:

        1. modify its state for the subsequent next() call
        2. produces a result for the current call
        """
        value = self.curr
        self.curr += self.prev
        self.prev = value
        if value > 1000:
            raise StopIteration
        return value
```

```
# Note that this class is both an iterable due to __iter__()
# method and its own iterator, due to __next__() method!

f = Fibonacci()
print(next(f), next(f), 'before the for loop')

for num in f:
    print(num, end=' ')
```

```
1 1 before the for loop
2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Lab: Iterators

Write your own iterator class which takes an iterable and each time it's invoked, it returns a *random* element. The iterator should stop (i.e., `raise` the `StopIteration` exception) when it has returned all elements of the iterable.

Example: `MyRandomIterator([1, 2, 3])` might return

```
2
3
1
...then raise StopIteration
```

In [71]:

```
from collections import namedtuple

card = namedtuple('Card', ['rank', 'suit'])

class DeckOfCards:
    ranks = list(range(2, 11)) + list('JQKA')
    suits = 'clubs diamonds hearts spades'.split()

    def __init__(self):
        self._cards = [card(rank, suit) for suit in self.suits for rank in self.ranks]

    def __str__(self):
        time_to_print = [str(x.rank) + ' of ' + str(x.suit) for x in self._cards]
        return str('\n'.join(time_to_print))

    # use the iterator from the underlying list
    # instead of relying on the "default" Python iterator
    # which makes use of __getitem__ and __len__ (if it exists)
```

```
# __getitem__ should be enough)
def __iter__(self):
    return iter(self._cards)

deck = DeckOfCards()
for card in deck:
    print(card)
```

```
Card(rank=2, suit='clubs')
Card(rank=3, suit='clubs')
Card(rank=4, suit='clubs')
Card(rank=5, suit='clubs')
Card(rank=6, suit='clubs')
Card(rank=7, suit='clubs')
Card(rank=8, suit='clubs')
Card(rank=9, suit='clubs')
Card(rank=10, suit='clubs')
Card(rank='J', suit='clubs')
Card(rank='Q', suit='clubs')
Card(rank='K', suit='clubs')
Card(rank='A', suit='clubs')
Card(rank=2, suit='diamonds')
Card(rank=3, suit='diamonds')
Card(rank=4, suit='diamonds')
Card(rank=5, suit='diamonds')
Card(rank=6, suit='diamonds')
Card(rank=7, suit='diamonds')
Card(rank=8, suit='diamonds')
Card(rank=9, suit='diamonds')
Card(rank=10, suit='diamonds')
Card(rank='J', suit='diamonds')
Card(rank='Q', suit='diamonds')
Card(rank='K', suit='diamonds')
Card(rank='A', suit='diamonds')
Card(rank=2, suit='hearts')
Card(rank=3, suit='hearts')
Card(rank=4, suit='hearts')
Card(rank=5, suit='hearts')
Card(rank=6, suit='hearts')
Card(rank=7, suit='hearts')
Card(rank=8, suit='hearts')
Card(rank=9, suit='hearts')
Card(rank=10, suit='hearts')
Card(rank='J', suit='hearts')
Card(rank='Q', suit='hearts')
Card(rank='K', suit='hearts')
```

```
Card(rank='A', suit='hearts')
Card(rank=2, suit='spades')
Card(rank=3, suit='spades')
Card(rank=4, suit='spades')
Card(rank=5, suit='spades')
Card(rank=6, suit='spades')
Card(rank=7, suit='spades')
Card(rank=8, suit='spades')
Card(rank=9, suit='spades')
Card(rank=10, suit='spades')
Card(rank='J', suit='spades')
Card(rank='Q', suit='spades')
Card(rank='K', suit='spades')
Card(rank='A', suit='spades')
```

Generators

- a generator allows you to write iterators much like the Fibonacci iterator above but in an elegant, succinct syntax that avoids writing classes with `__iter__()` and `__next__()` methods
- every generator is an iterator (but not vice versa!)
- a generator is a factory that lazily produces values
- two types: generator *functions* and generator *expressions*

The `yield` statement

- before we jump into generators, let's take an in-depth look at what makes them possible...
- when a normal Python function is invoked, execution starts at the first line and continues until a `return` statement is encountered or an exception is thrown (remember that "falling off the end of the function" is the same as if we had written `return None`)
 - once a function returns, that's it—any work done by the function and stored in local variables is lost
 - the next call to the function starts everything anew
- there are times when we'd like to have a "function" which yields a series of values, i.e., it would have to save its state so that the next time it's invoked, it picks up where it left off
 - we use the term "yield" here because in fact we are *not returning* to the caller i.e., we are not returning control of execution to the point where the function was called
 - instead of `return`-ing, we are `yield`-ing, which implies that the transfer of control is temporary and voluntary—our function expects to regain control in the future
- functions that use `yield` instead of `return` are generator functions (or *coroutines* in other languages)

- think of `yield` as `return` + "some magic" for generator functions

So what's the magic?

- when `yield` is called the state of the generator function is recorded
 - the value of all variables are saved
 - the next line of code to be executed is also saved
 - i.e., the function simply resumes where it left off

In [72]:

```
def simple_generator():
    yield 1
    yield 'boo!'
    yield 3

for value in simple_generator():
    print(value)
```

```
1
boo!
3
```

Why do we need generator functions?

- initially they gave programmers an easy way to write code that produced a series of values
 - without generator functions, writing something like a random number generator required a class or module that both generated values and kept track of state between calls
 - with generator functions, doing the above is greatly simplified
- suppose we want a function which, given a list of numbers, returns a list of those numbers which are prime
 - straightforward...

```
def get_primes(nums):
    return ([num for num in nums if is_prime(num)])
```

- now suppose we want to use the above function for very large lists of numbers...so large, in fact, that they won't fit in memory
 - so now we want the function to take a starting value, and return all the primes that are greater than that value
 - since functions only return once, they only have one "chance" to return a value (or list of values)
 - what if our function could return the *next* value, rather than a list?

- we wouldn't need to create a list at all!

What is a generator function?

- defined like a normal function, but whenever it needs to generate a value, it does so with the `yield` keyword rather than `return`
 - if the body of a def contains `yield`, the function automatically becomes a generator function (even if it also has a `return` statement)
 - ...there's nothing else we need to do to create one
- generator functions create *generator iterators* (or simply, a *generator*)
 - a generator is a special type of iterator (meaning it has a `__next__()` function)
 - to get the next value from a generator, we use the same built-in function as for iterators: `next()`
- let's return to the more basic notion of a generator function...

Now we can rewrite our `get_primes()` function as a generator...

```
def get_primes(num):
    while True:
        if is_prime(num):
            yield num
        num += 1
```

- note that if a generator function calls `return` (or simply hits the end of the function), then a `StopIteration` exception is raised, signaling the generator is exhausted (just as an iterator does)

In [73]:

```
def fibonacci():
    ...
    defined as a normal function, but...
    ...no return keyword

    The yield keyword returns a value, but the function retains its state
    ...
    prev, curr = 0, 1
    while True:
        yield curr
        prev, curr = curr, prev + curr

f = fibonacci()
```

```

print(next(f), next(f), 'before the for loop', sep='\n')

import random

for num in range(0, random.randint(10, 100)):
    val = next(f)
    print(val, end=' ')

```

```

1
1
before the for loop
2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229
832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 43349443
7 701408733 1134903170 1836311903 2971215073 4807526976 7778742049 12586269025 20365011074 32951280099 53316291173 862675
71272 139583862445 225851433717 365435296162 591286729879 956722026041 1548008755920 2504730781961 4052739537881 65574703
19842 10610209857723 17167680177565 27777890035288 44945570212853 72723460248141 117669030460994 190392490709135 30806152
1170129 498454011879264 806515533049393 1304969544928657 2111485077978050 3416454622906707 5527939700884757 8944394323791
464 14472334024676221 23416728348467685

```

PEP-342: sending values into generators

- PEP-342 added support for passing values *into* generators using the `send()` function
- let's go back to the prime number example but instead of simply printing every prime number greater than some number, we'll find the smallest prime number greater than successive powers of a number (i.e. for 10, we want the smallest prime greater than 10, then 100, then 1000, etc.)

```

def get_primes(num):
    while True:
        if is_prime(num):
            num = yield num
        num += 1

```

- the `yield` line now says "**yield num, and when a value is sent to me, set num to that value**"
- and we can print the next prime greater than 10, 100, 1000, as follows:

```

def print_successive_primes(iterations, base=10):
    prime_generator = get_primes(base)
    prime_generator.send(None)
    for power in range(iterations):
        print(prime_generator.send(base ** power))

```

- printing `generator.send()` is possible because `send` both sends a value to the generator and returns the value yielded by the generator
- note that the first time we send a value into a generator, it must be `None`

Now let's look at a generator expression

- generator equivalent of a list comprehension

```
In [74]: squares = [num * num for num in range(1, 11)] # List comprehension  
squares
```

```
Out[74]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
In [75]: squares = {num * num for num in range(1, 11)} # set comprehension  
squares
```

```
Out[75]: {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

```
In [76]: squares = {num: num * num for num in range(1, 11)} # dict comprehension  
squares
```

```
Out[76]: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

```
In [77]: squares = (num * num for num in range(1, 11)) # generator expression (NOT a 'tuple comprehension')  
squares
```

```
Out[77]: <generator object <genexpr> at 0x7f3ca8150a50>
```

```
In [78]: next(squares), next(squares)
```

```
Out[78]: (1, 4)
```

```
In [79]: list(squares) # for thing in squares: print(thing)
```

```
Out[79]: [9, 16, 25, 36, 49, 64, 81, 100]
```

Lab: Generators

- modify your random iterator to be a generator function

itertools

- module of functions for efficient looping
- all of its functions return iterators
- some produce finite sequences
- others produce infinite sequences

```
In [80]:
```

```
from itertools import zip_longest
list1 = ['a', 'b', 'c', 'd']
list2 = ['apple', 'banana', 'cherry']

for item1, item2 in zip_longest(list1, list2, fillvalue='***'):
    print(item1, '=>', item2)
```

```
a => apple
b => banana
c => cherry
d => ***
```

```
In [81]:
```

```
from itertools import count
counter = count(start=789)
for _ in range(10):
    print(next(counter))
```

```
789
790
791
792
793
794
795
796
797
798
```

```
In [82]: from itertools import count

counter = count(1, 0.25)
for _ in range(10):
    print(next(counter))
```

```
1
1.25
1.5
1.75
2.0
2.25
2.5
2.75
3.0
3.25
```

```
In [83]: from itertools import cycle
sizes = ['S', 'M', 'L']
sc = cycle(sizes)
#next(sc), next(sc), next(sc), next(sc), next(sc)
for num in range(1, 22):
    print(next(sc), end=' ')
```

```
S M L S M L S M L S M L S M L S M L S M L
```

```
In [84]: from itertools import islice
list(islice(fibonacci(), 50, 60))
```

```
Out[84]: [20365011074,
32951280099,
53316291173,
86267571272,
139583862445,
225851433717,
365435296162,
591286729879,
956722026041,
1548008755920]
```

```
In [85]: from itertools import count, islice
```

```
for num in islice(count(1, 0.25), 13, 17):
    print(num)
```

```
4.25
4.5
4.75
5.0
```

In [86]:

```
# some produce a finite sequence from an infinite sequence
from itertools import islice, cycle
colors = cycle(['red', 'white', 'blue']) # infinite
limited = islice(colors, 0, 5)

for color in limited:
    print(color, end=' ')
```

```
red white blue red white
```

In [87]:

```
from itertools import chain
rank = list(range(2, 11))
#picture = { 'J': 'Jack', 'Q': 'Queen', 'K': 'King', 'A': 'Ace' }
picture = list('JQKA')

#for card in chain(rank, picture):
#    print(card, end=' ')

list(chain(rank, picture))
```

Out[87]: [2, 3, 4, 5, 6, 7, 8, 9, 10, 'J', 'Q', 'K', 'A']

In [88]:

```
from itertools import filterfalse
# filter out items for which predicate is False
numbers = [7, 12, 20, 23, 32, 44]
list(filterfalse(lambda x: x % 2, numbers))
```

Out[88]: [12, 20, 32, 44]

In [89]:

```
# filters elements, returning only those that have a corresponding
# element that evaluates to True
from itertools import compress
words = ['how', 'now', 'brown', 'cow']
```

```
counts = [13, '', 'x', None]
list(compress(words, counts))
```

```
Out[89]: ['how', 'brown']
```

```
In [90]: # accumulate sums, or other binary functions
from itertools import accumulate
list(accumulate([3, 5, 10, 21]))
#help(accumulate)
```

```
Out[90]: [3, 8, 18, 39]
```

```
In [91]: list(accumulate(range(1, 10), lambda x, y: x * y))
```

```
Out[91]: [1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

```
In [92]: # Amortize a 5% Loan of 1000 with 4 annual payments of 250
cashflows = [1000, 250, 250, 250, 250]
list(accumulate(cashflows, lambda bal, pmt: bal * 1.05 - pmt))
```

```
Out[92]: [1000, 800.0, 590.0, 369.5, 137.97500000000002]
```