PES
UNIVERSITY

Department of Computer Science and Engineering
PES University, Bangalore, India
Python For Computational Problem Solving (UE22CS151A)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Iterators in Python

What are Iterables?

- Iterable is an object that can be iterated over and capable of returning their members one at a time by associating with an iterator.
- Objects like lists, tuples, sets, dictionaries, strings, etc. are called iterables.
- In short, anything you can loop over is an iterable.

In [ ]:
```
* Python iterables implement a special method called __iter__().
* The __iter__() method returns an iterator object.
```

In [4]:
```
print(iter('Hello Python!') )
print(iter( ['Hi', 'Hello', 'How are you?'] ))
```

```
<str_iterator object at 0x000002D626DF8610>
<list_iterator object at 0x000002D625AC1B80>
```

print(iter(25)) # TypeError: 'int' object is not iterable

Let’s use a Python built-in function dir() to find out all the associated attributes of the iterable.

In [ ]:
```
lst = ['hi','hello','how r you?','how do you do?']
print(dir(lst))
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

In [ ]:
```
print(help(list))
```

```
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
```

What are Iterators?

- An iterator is essentially a value producer that yields successive values from its associated iterable object. The built-in function next() is used with an iterator to obtain the next value from an associated iterable object.
- An iterator allows programmers to access or traverse through all the elements of its associated iterable without any deeper understanding of its structure.

```
Python iterators implement iterator protocol which consists of two special
methods __iter__() and  __next__().

The __iter__() method returns an iterator object whereas __next__() method
returns the next element from the sequence.
```

Let's use a Python built-in function dir() to find out all the associated attributes of the iterators.

In [ ]:
```
lst = ['hi','hello','how r you?','how do you do?']

lst_iterobj = lst.__iter__()

print(dir(lst_iterobj))
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format
__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__length_hint__', '__lt_
_', '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '_
_repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__s
ubclasshook__']
```

In [2]:
```
lst = ['hi','hello','how r you?','how do you do?']
lst_iterobj=lst.__iter__()          # or lst_iterobj=iter(lst)
print(lst_iterobj.__next__())      # or print(next(lst_iterobj))
print(lst_iterobj.__next__())
print(lst_iterobj.__next__())
print(lst_iterobj.__next__())
```

```
hi
hello
how r you?
how do you do?
```

Notice how an iterator retains its state internally. It knows which values have been obtained already, so when you call next(), it knows what value to return next.

What happens when the iterator runs out of values? Let's make one more next() call on the iterator above:

```
In [ ]: print(lst_iterobj.__next__()) # Raises StopIteration Exception
```

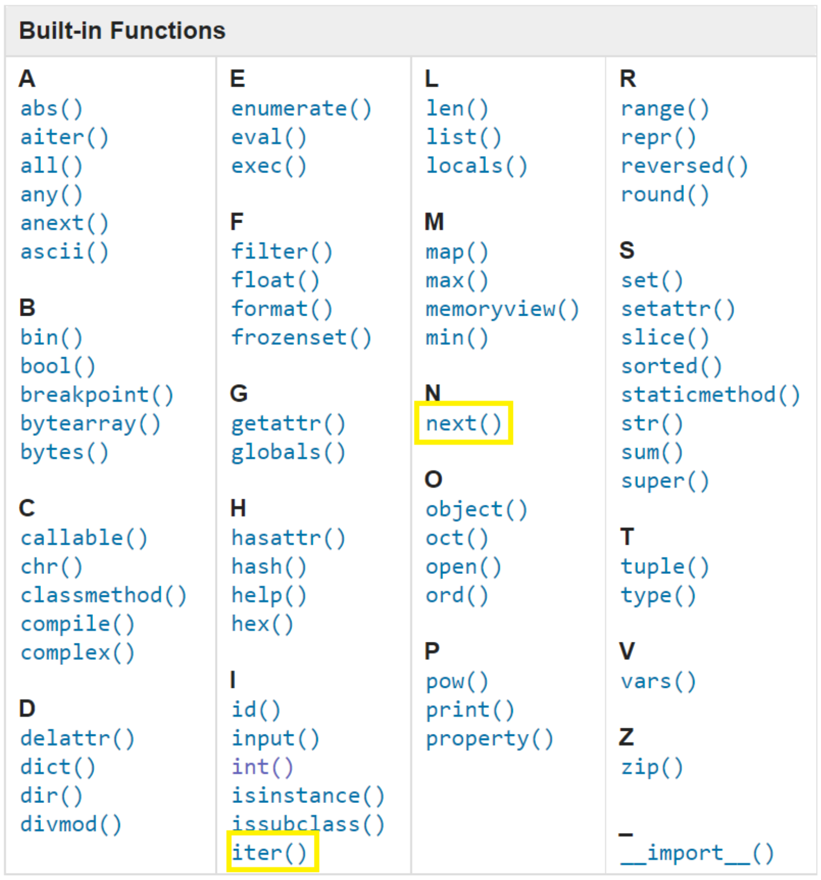
Built-in functions iter() and next()

iter(iterable_object)

- The built-in function used to obtain an iterator from an iterable object.

next(iterator_object)

- The built-in function used to obtain the next value from an associated iterable object.



Built-in Functions			
A	**E**	**L**	**R**
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
any()			round()
anext()	**F**	**M**	
ascii()	filter()	map()	**S**
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	**G**	**N**	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	**H**	object()	
callable()	hasattr()	oct()	**T**
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		**P**	**V**
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	**Z**
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		**_**
	iter()		__import__()

How for loops actually work?

Let's take a list (an iterable) and iterate through it.

```
In [15]: sample = ['data science', 'social network analytics', 'machine learning']
         for ele in sample:
             print(ele)
```

```
data science
social network analytics
machine learning
```

So basically, the process of the for loop going through each element is called iteration and the object sample through which the for loop is iterating is called iterable.

What actually is happening in above for-loop?

- Well, behind the scenes actually the for-loop is using built-in methods
  - iter() on iterable to get an iterator object that produce successive items from its associated iterable and
  - next() on iterator to get next element in the iterable.

- For loop stops when a subsequent next() call raises a StopIteration exception.

In [14]:
```
lst = ['hi', 'hello', 'how r you?', 'how do you do?']
lst_iterobj = iter(lst)  #using iter() function for container_object lst

print(next(lst_iterobj)) #Iteration-1 using next function on iterator object gives lst e
print(next(lst_iterobj)) #Iteration-2 gives 2nd element
print(next(lst_iterobj)) #Iteration-3 gives 3rd element
print(next(lst_iterobj)) #Iteration-4 gives 4th element
```

```
hi
hello
how r you?
how do you do?
```

In [11]:
```
lst = ['hi','hello','how r you?','how do you do?']
lst_iterobj = lst.__iter__()  #using __iter__ function for container_object lst

print(lst_iterobj.__next__()) #Iteration-1 using next function on iterator object gives
print(lst_iterobj.__next__()) #Iteration-2 gives 2nd element
print(lst_iterobj.__next__()) #Iteration-3 gives 3rd element
print(lst_iterobj.__next__()) #Iteration-4 gives 4th element
```

```
hi
hello
how r you?
how do you do?
```

So, here is how things actually work behind the iteration in for loop or any iterable in Python.

In [4]:
```
lst = ['hi','hello','how r you?','how do you do?']
obj = iter(lst) #using iter function for container object lst
print(next(obj)) #Iteration 1 using next function on iterator object gives ist element
```

```
hi
```

An iterator is essentially a value producer that yields successive values from its associated iterable object. The built-in function next() is used to obtain the next value from in iterator.

In [5]:
```
print(next(obj)) #Iteration 2
```

```
hello
```

Notice how an iterator retains its state internally. It knows which values have been obtained already, so when you call next(), it knows what value to return next.

In [6]:
```
print(next(obj)) #Iteration 3
```

```
how r you?
```

In [7]:
```
print(next(obj)) #Iteration 4
```

```
how do you do?
```

What happens when the iterator runs out of values?

If all the values from an iterator have been returned already, a subsequent next() call raises a StopIteration exception. Any further attempts to obtain values from the iterator will fail.

In [12]:
```
print(next(obj)) #Iteration 5
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-12-078ebaa4d5c3> in <module>
```

```
----> 1 print(next(obj)) #Iteration 5

StopIteration:
```

Note:

- You can only obtain values from an iterator in one direction. You can’t go backward. There is no prev() function.
- But you can define two independent iterators on the same iterable object:

In [18]:
```
lst = ['hi','hello','how r you?','how do you do?']
lst_iterator_obj1 = iter(lst)
lst_iterator_obj2 = iter(lst)

print(next(lst_iterator_obj1))
print(next(lst_iterator_obj1))
print('-----')
print(next(lst_iterator_obj2))
print(next(lst_iterator_obj2))
```

```
hi
hello
-----
hi
hello
```

We can summarize above process in following points.

```
* iterables have function __iter__() as we saw using dir()
* __iter__() functions returns an iterator object
* using iterator and __next__() function we traverse through all the items in
the list
* once there are no items left to iterate through, the function __next__()
raises an exception StopIteration and the iteration ends there.
```

Additional info:

The Guts of the Python for Loop

You now have been introduced to all the concepts you need to fully understand how Python’s for loop works. Before proceeding, let’s review the relevant terms:

Term	Meaning
Iteration	The process of looping through the objects or items in a collection
Iterable	An object (or the adjective used to describe an object) that can be iterated over
Iterator	The object that produces successive items or values from its associated iterable
`iter()`	The built-in function used to obtain an iterator from an iterable

Now, consider again the simple for loop presented at the start of this tutorial:

In [7]:
```
a = ['foo', 'bar', 'baz']
for i in a:
    print(i)
```

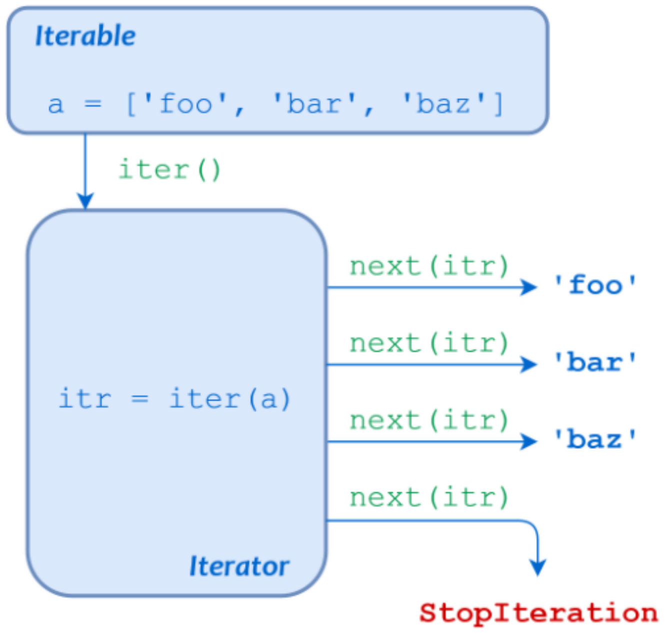
```
foo
bar
baz
```

This loop can be described entirely in terms of the concepts you have just learned about. To carry out the iteration this for loop describes, Python does the following:

- Calls iter() to obtain an iterator for a given iterable
- Calls next() repeatedly to obtain each item from the iterator in turn
- Terminates the loop when next() raises the StopIteration exception

The loop body is executed once for each item next() returns, with loop variable i set to the given item for each iteration.

This sequence of events is summarized in the following diagram:



Schematic Diagram of a Python for Loop

Creating our own Iterator in Python

Building our own Iterator is nothing different than what we explained above. We use the same __iter__() and __next__() functions.

But this time we will define these special functions inside a class as we need.

Example to create our own Python Iterator

Here is an example to build our own iterator to display odd number from 1 to the max number supplied as the argument.

In [16]:
```
class OddNum:
    """Class to implement iterator protocol"""
    def __init__(self, num = 0):
        self.num = num

    def __iter__(self):
        self.x = 1
        return self

    def __next__(self):
        if self.x <= self.num:
            odd_num = self.x
            self.x += 2
            return odd_num
        else:
            raise StopIteration
```

In [ ]:
```
Now we can use directly for loop or use __iter__() and __next__().
```

Using for loop

In [10]:
```
obj = OddNum(10)
for num in obj:
    print(num)
```

```
1
3
5
7
9
```

In [ ]:
```
The for statement will call iter(obj).
This call is changed to OddNum.__iter__(obj).

Using __iter__() and __next__()
```

In [12]:
```
obj = OddNum(10)
i = iter(obj)    # i = OddNum.__iter__(obj)
print(next(i))         # OddNum.__next__(i)
```

```
1
```

In [17]:
```
print(next(i))
```

```
3
```

In [18]:
```
print(next(i))
```

```
5
```

In [19]:
```
print(next(i))
```

```
7
```

In [20]:
```
print(next(i))
```

```
9
```

In [21]:
```
print(next(i))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-21-5bae39b0365f> in <module>
----> 1 print(next(i))

<ipython-input-9-07d15d6386fc> in __next__(self)
     15         return odd_num
     16     else:
---> 17       raise StopIteration

StopIteration:
```

References:

1. 19_gen_iterator.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://www.w3schools.com/python/
3. https://docs.python.org/
4. https://www.geeksforgeeks.org/ generators-in-python/
5. http://www.trytoprogram.com/python-programming/python-iterators/
6. http://www.trytoprogram.com/python-programming/python-generators