Iterators and generators

Iterators:

|  |
| --- |
| Python Iterators  An object whose class consists of an “\_\_iter\_\_” method which returns “self” and a “\_\_next\_\_” (next in Python 2) method is called an iterator. It is assumed that the class defines “\_\_next\_\_()”, then \_\_iter\_\_() can just return self:  In general, the “\_\_iter\_\_” function returns an iterator object that defines the method “\_\_next\_\_” which accesses elements in the container one at a time.  When there are no more elements, “\_\_next\_\_()” raises a StopIteration exception which tells the for loop to terminate. |

Note:

List is not a iterator , but it can be iterable over a loop.   
if we use dir() on a iterator or a variable will get a list methods and attribute that can be implement on that particular iterator or variable [can consist of any list, tuple, dict… etc]

If we dir() returns list shows \_\_iter\_\_ and \_\_next\_\_ methods , which are also called special methods.

Then that is a iterator.

**Note:**

Iter() is a method which returns a object which can be iterable.

**Example1:**

Here if you see ‘list’ has no \_\_next\_\_ method but only \_\_iter\_\_ method.

So it is iterable but not a iterator.

|  |
| --- |
| >>> list = [1,2,3,4]  >>>  >>> dir(list)  ['\_\_add\_\_', '\_\_class\_\_', '\_\_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']  >>> |

**Example2:**

So with iter() method we got a object called ‘x’ which has both methods \_\_iter\_\_ and \_\_next\_\_ .

So now we can use next() method on iterator.

|  |
| --- |
| >>> list = [1,2,3,4]  >>> x = iter(list)  >>>  >>> dir(x)  ['\_\_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\_\_', '\_\_subclasshook\_\_']  >>>  >>>  >>> next(x)  1  >>> next(x)  2  >>> next(x)  3  >>> next(x)  4  >>> next(x)  Traceback (most recent call last):  File "<pyshell#15>", line 1, in <module>  next(x)  StopIteration  >>> |

**Note**:

Now we can make a class which consist of special methods, and if for this class we will create any object.

Then that object can be treated as an iterator.

|  |
| --- |
| >>> class demo:  def \_\_init\_\_(self,n):  self.length = n  def \_\_iter\_\_(self):  return self  def \_\_next\_\_():  pass    >>> obj = demo(5)  >>> dir(obj)  ['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_next\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'length']  >>> |

Question:

Implement range() function with custom iterators?

Ans:

|  |
| --- |
| >>> class demo:  def \_\_init\_\_(self, n):  self.cur\_val = 0  self.max\_range = n  def \_\_iter\_\_(self):  return self  def \_\_next\_\_(self):  if self.cur\_val < self.max\_range:  val = self.cur\_val  self.cur\_val = self.cur\_val + 1  return val  else:  raise StopIteration()  >>> for i in mobj: ####here object can be treated as iterator..  print(i)    0  1  2  3  4  >>> |

Question:

|  |
| --- |
| '''Write an iterator class “my\_reverse\_list”, that takes a list and iterates it from the reverse direction'''  class my\_reverse\_list:  def \_\_init\_\_(self, list):  self.list = list  self.length = len(list)  self.cur\_val = 0  def \_\_iter\_\_(self):  return self  def \_\_next\_\_(self):  if self.cur\_val < self.length:  val = self.cur\_val  self.cur\_val = self.cur\_val + 1  index = self.length - (val+1)  return self.list[index]  else:  raise StopIteration()  def main():  list = [1, 2, 3, 4, 5]  obj = my\_reverse\_list(list)  for i in obj:  print(i)  if \_\_name\_\_ == '\_\_main\_\_':  main() |

**Generators**

• Generators are a simple and powerful tool for creating iterators.

• Generators are written like regular functions

• The yield statement is used to return data from generators

• Each time next() is called on it , the generator resumes where it left off (it remembers all the data values and which statement was last executed). If a “for” loop is used, “next()” is called internally.

Example: [if yield keyword will not be In use .. then it can not be a generator] 🡪 no \_\_next\_\_ method

|  |
| --- |
| >>> def sq\_num(nums):  result = []  for i in nums:  result.append(i\*i)  return result  >>> my\_num = sq\_num([1,2,3,4,5])  >>>  >>> print(my\_num)  [1, 4, 9, 16, 25]  >>>  >>> dir (my\_num)  ['\_\_add\_\_', '\_\_class\_\_', '\_\_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']  >>> |

Example: [now will use yield keyword to create generator]🡪 it will have \_\_next\_\_ method

|  |
| --- |
| >>> def sq\_num(nums):  result = []  for i in nums:  yield (i\*i)    >>> my\_num = sq\_num([1,2,3,4,5])  >>> for num in my\_num:  print(num)    1  4  9  16  25  >>> dir(my\_num)  ['\_\_class\_\_', '\_\_del\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_name\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_next\_\_', '\_\_qualname\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'close', 'gi\_code', 'gi\_frame', 'gi\_running', 'gi\_yieldfrom', 'send', 'throw']  >>> |