INTRODUCING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 12: ITERATORS

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

12: ITERATORS

BY FRED ANNEXSTEIN, PHD

CHAPTER 12 OVERVIEW

In this module we will be covering the concepts associated with programming with iterators in Python. An iterator is an special object that contains a countable number of values. An iterator can be iterated upon, meaning that you can traverse through every value of the iterator with a simple for-loop. As we will see, in Python, an iterator is an object which implements the iterator protocol, which consist of the two simple methods called iter() and next().

CHAPTER 12 OBJECTIVES

By the end of this chapter, you will be able to...

1. Understand the concept and use cases of programming with iterators in Python.
2. Understand and apply the construction of iterators, and their use in several examples.
3. Understand and apply generator functions to the construction of iterators.
4. Understand the yield statement and the goals of lazy evaluation.
5. Apply the use of generators in support of lazy evaluation programming style.

ITERATORS IN PYTHON

Python provides a unified way to process the values or elements within a container sequentially; this special container is called an iterator. An iterator is an object that provides sequential access to values, one by one, and can be used by any for-loop.

By convention, an iterator has two components: a) a mechanism for retrieving the next element in the sequence being processed, and b) a mechanism for signaling that the end of the sequence has been reached and no further elements remain.

For any container, such as a list or range, an iterator can be obtained by calling the built-in iter() function. The contents of the iterator can be accessed by calling the built-in next() function. See the following example:

```
>>> primes = [23, 37, 51]

>>> type(primes)
list

>>> piter = iter(primes)

>>> type(piter)
list_iterator

>>> next(piter)
23

>>> next(piter)
37
```

```
>>> next(piter)
51

>>> next(piter)
Traceback (most recent call last):

  Input In [9] in <cell line: 1>
    next(piter)

StopIteration
```

We see from the above example that the next() function can be used to iterate through all the values in the list_iterator, and when there are no more elements in the container to iterate, the system throws a StopIteration exception.

An iterator can always be used with a 'for-in loop'. Python lists, tuples, dicts and sets are all examples of inbuilt iterators. These types are iterators because they implement two special methods. In fact, any container object that wants to be an iterator must implement the following two methods.

__iter__ is a method that is called for initialization of an iterator. This methods should return an object that has a next or __next__ method.

__next__ is a method that when called should return the next value for the collection.

When an iterator is used with a 'for-in' loop

construct, the for loop first initializes and then implicitly calls the next() function on the iterator object. This method should raise a StopIteration exception to signal the end of the iteration.

An iterator works by maintaining sufficient information for a local state to represent the current position in a sequence. Each time next is called, that position advances. Two separate iterators can track two different positions in the same sequence. However, two names for the same iterator will share a position, because as aliases they reference the same object.

```
>>> r = range(30, 100, 10)
>>> s = iter(r)  # 1st iterator over r
>>> next(s)
30
>>> next(s)
40
>>> t = iter(r)  # 2nd iterator over r
>>> next(t)
30
>>> next(t)
40
>>> u = t    #  u is alias for iterator t
>>> next(u)
50
>>> next(u)
60
```

Advancing the second iterator t does not

affect the first s. Since the last value returned from the first iterator was 40, it is positioned to return 50 next. On the other hand, the second iterator u=t is positioned to return 80 next.

```
>>> next(s)
50
>>> next(t)
70
```

Calling the iter() method on an iterator will return that iterator, not a copy. This behavior is included in Python so that a programmer can call iter() on a value to get an iterator without having to worry about whether it is an iterator or a simple container.

```
>>> v = iter(t)
# Another alias for the t iterator
>>> next(v)
80
>>> next(u)
90
```

The usefulness of iterators is derived from the fact that the underlying series of data for an iterator does not have to be represented and stored explicitly in memory. An iterator

provides a mechanism for considering each of a series of values in turn, but all of those elements do not need to be stored simultaneously. Instead, when the next element is requested from an iterator, that element may be computed on demand instead of being retrieved from an existing memory source.

RANGES ARE LAZY

Ranges are able to compute the elements of a sequence lazily because the sequence represented is uniform, and any element is easy to compute from the starting and ending bounds of the range. Thus in Python it is possible to compute functions such as len(), getitem(), and the in() methods extremely fast, as shown in the following:

```
r = range(1,10000000,7)

>>> 100004 in r
False
>>> 100003 in r
True
>>> len(r)
1428572
>>> r[987654]
6913579
```

Iterators allow for lazy generation of a much broader class of underlying sequential

datasets, because they do not need to provide access to arbitrary elements of the underlying series. Instead, iterators are only required to compute the next element of the series, in order, each time another element is requested. While not as flexible as accessing arbitrary elements of a sequence (called random access), sequential access to sequential data is often sufficient for data processing applications.

ITERABLES

Any object that can produce an iterator is called an *iterable* object. In Python, an iterable is anything that can be passed to the built-in iter() function. Iterables include sequence values such as strings and tuples, as well as other containers such as sets and dictionaries. Iterators are also iterables, because they have an iter() function attribute by definition.

Even unordered collections such as dictionaries must define an ordering over their contents when they produce iterators. Dictionaries and sets are unordered because the programmer has no control over the order of iteration. Python does guarantee certain properties about their order based on the hash function used in its specification.

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d
{'one': 1, 'three': 3, 'two': 2}
>>> k = iter(d)
>>> next(k)
'one'
>>> next(k)
'three'
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
3
>>> next(v)
2
>>> next(v)
Traceback (most recent call last):

  Input In [26] in <cell line: 1>
    next(v)

StopIteration
```

If a dictionary changes in structure because a key is added or removed, then all iterators become invalid and future iterators may exhibit arbitrary changes to the order their contents. On the other hand, changing the value of an existing key does not change the order of the contents or invalidate iterators.

```
>>> d.pop('two')
2
>>> next(k)
```

```
RuntimeError: dictionary changed size during
iteration
Traceback (most recent call last):
```

FOR-LOOP STATEMENTS

The for-in statement in Python operates on any iterator. Objects are iterable (an interface) if they have an __iter__ method that returns an iterator. Iterable objects can be the value of the <expression> in the header of a for statement:

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

To execute a for-in statement, Python evaluates the header <expression>, which must yield an iterable object. Then, the __iter__ method is invoked on that value.

How does iter work? It calls the __iter__ magic method on its argument, so iter(something_iterable) is equivalent to something_iterable.__iter__(). So, any object is iterable if it implements this method, and this method must return an iterator.

How does next work? It calls the __next__ magic method on its argument, so next(the_iterator) is equivalent

to the_iterator.__next__(). So, to implement the iterator protocol, we must implement this method. This method eithers returns the next element in our iteration, or raises the StopIteration exception when there are no elements left.

```
>>> counts = [1, 2, 3]

>>> for item in counts:
        print(item)
1
2
3
```

In the above example, the counts list returns an iterator from its __iter__() method. The for statement then calls that iterator's __next__() method repeatedly, and assigns the returned value to item each time. This process continues until the iterator raises a StopIteration exception, at which point execution of the for statement concludes.

With our knowledge of iterators, we can implement the execution rule of a for statement in terms of basic while, assignment, and try statements.

```
>>> items = counts.__iter__()
>>> try:
        while True:
            item = items.__next__()
            print(item)
```

```
    except StopIteration:
        pass

1
2
3
```

Above, the iterator returned by invoking the __iter__ method of counts is bound to the variable name items so that it can be queried for each element in turn. The handling clause for the StopIteration exception does nothing, but handling the exception provides a control mechanism for exiting the while loop.

Python docs suggest that an iterator have an __iter__ method that returns the iterator itself, so that all iterators are iterable.

AN ITERABLE USER DEFINED TYPE

We illustrate using OOP classes to implement several iterators. Any class that will produce an iterable object must include the iter() and next() magic methods.

In this first example the Start10 class will construct an iterator that iterates starting from the number 10 and will iterate to any given input value, which is an argument to the object initializer.

```
class Start10:
     def __init__( self , limit):
         self.limit  = limit

     def __iter__( self ):
         self.x  = 9
         return self

     def __next__( self ):
         if self.x >=  self.limit:
             raise StopIteration
         self.x += 1 ;
         return self.x

>>> for i  in Start10(21):
       print (i, end = " ")
10 11 12 13 14 15 16 17 18 19 20 21

# will print nothing since 9 < 10
>>> for i  in Start10(9):
        print(i)
```

In this next example the Rev class will construct an iterator that iterates in reverse for any starting sequence given as an argument to the initializer. The reverse iterator works by keeping a pointer ptr that starts at the end of the sequence and moves backwards through the sequence.

```
class Rev:
  def __init__(self,seq):
    self.data = seq

  def __iter__(self):
    self.ptr = len(self.data)
```

```
    return self

  def __next__(self):
    if self.ptr == 0:
      raise StopIteration
    self.ptr -= 1
    return self.data[self.ptr]


>>> for i  in Rev(range(10)):
       print (i, end= " ")
9 8 7 6 5 4 3 2 1 0
```

In this next example the DoubleIt class will construct an iterator that takes another iterator as an argument, and then doubles (multiples by 2) any value iterated by the original iterator. The DoubleIt iterator works by using the iter() and next() methods of the original iterator.

```
class DoubleIt:
    def __init__(self, it):
      self.it = it
    def __iter__(self):
      iter(self.it)
      return self

    def __next__(self):
      return next(self.it) * 2
```

```
>>> for i in DoubleIt(iter([1, 2, 3, 4])):
        print(i, end= " ")

2 4 6 8

>>> for i in DoubleIt(iter(range(1,10,3))):
        print(i, end= " ")
2 8 14
```

GENERATORS AND YIELD STATEMENTS

With complex sequences, it can be quite difficult to program the __next__() method to save its place in the calculation. Python allows for an object called a generator that makes defining more complicated iterations easier by leveraging the features of the Python interpreter.

A generator is a special iterator that is returned by a special class of functions called a generator functions. Generator functions are distinguished from regular functions in that rather than containing return statements in their body, they use yield statements to return elements of a series.

Generators do not use attributes of an object to track their progress through a series. Instead, they control the execution of the generator function, which runs until the next yield statement is executed each time

the generator's __next__ method is invoked.

In the following example, we define a simple generator that mimics the operation of the Start10 iterable from above

```
def generator10(limit):
  print("Start10")
  x = 9
  while x < limit:
    x += 1
    yield x


>>> for i in generator10(21):
      print(i, end= " ")
Start10
10 11 12 13 14 15 16 17 18 19 20 21
```

In the following example we define a letters_generator, producing an iterator that contains all the letters. This example iterator shows that the implementation is much more compact using generators.

```
def letters_generator():
        current = 'a'
        while current <= 'z':
            yield current
            current = chr(ord(current)+1)

>>> for letter in letters_generator():
        print(letter, end=“ “)
a b c d e f g h i j k l m n o p q r s t u v
w x y z
```

Even though we never explicitly defined __iter__ or __next__ methods, the yield statement indicates to the interpretor that we are defining a generator function.

A generator function is different from a generator object. The generator function doesn't return a particular yielded value, but instead returns a generator object (which is a type of iterator).

The generator object has __iter__ and __next__ attribute methods, and each call to __next__ continues execution of the generator function from wherever it left off previously until another yield statement is executed.

The first time __next__ is called, the program executes statements from the body of the generator function until it encounters the yield statement. Then, it pauses and returns the value specifiec. The yield statements do not destroy the newly created environment, they preserve it for later. When __next__ is called again, execution resumes where it left off. The values of any bound names in the scope of generator function are preserved across subsequent calls to __next__.

We can walk through the generator by manually calling ____next__(). Here is an

example using the letters_generator function.

```
>>> letters = letters_generator()
>>> type(letters)
<class 'generator'>
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
‘d'
```

The generator does not start executing any of the body statements of its generator function until the first time __next__ is invoked. The generator raises a StopIteration exception whenever its generator function returns.

GENERATOR FOR NATURAL NUMBERS

In the next example we show how a generator can define a container with an infinite number of values. We use the example of the natural numbers 1,2,3,…

It is easy to define this infinite container by using a generator that only needs to store its current value i, and then with each call to next() increments the stored value.

```
def naturals():
  i = 1
  while True:
    yield i
    i += 1
```