21 Iterators and Generators

"The modern way..."

Contents

Iterables and Iterators

- An object is called iterable if it is capable of returning its members one at a time. Basic types like string and containers like list and tuple are iterables.
- Iterator is an object which is used to iterate over an iterable. An iterable provides an iterator object.
- Iterators are implemented in for loops, comprehensions, generators etc.

zip( ) Function

- **zip( )** function typically receives multiple iterable objects and returns an iterator of tuples based on them. This iterator can be used in a **for** loop as shown below.

```
words = ['A', 'coddle', 'called', 'Molly']
numbers = [10, 20, 30, 40]

for ele in zip(words, numbers) :
   print(ele[0], ele[1])

for ele in zip(words, numbers) :
   print(*ele)

for w, n in zip(words, numbers) :
   print(w, n)
```

All three **for** loops will output:

```
A 10
coddle 20
called 30
Molly 40
```

- If two iterables are passed to **zip( )**, one containing 4 and other containing 6 elements, the returned iterator has 4 (shorter iterable) tuples.

- A list/tuple/set can be generated from the iterator of tuples returned by **zip( )**.

```
words = ['A', 'coddle', 'called', 'Molly']
numbers = [10, 20, 30, 40]
it = zip(words, numbers)
lst = list(it)
print(lst)  # prints [('A', 10), ('coddle', 20), ('called', 30), ('Molly', 40)]

it = zip(words, numbers)    # necessary to zip again
tpl = tuple(it)
print(tpl)  # prints (('A', 10), ('coddle', 20), ('called', 30), ('Molly', 40))

it = zip(words, numbers)    # necessary to zip again
s = set(it)
print(s)    # prints {('coddle', 20), ('Molly', 40), ('A', 10), ('called', 30)}
```

- The values can be unzipped from the list into tuples using *.

```
words = ['A', 'coddle', 'called', 'Molly']
numbers = [10, 20, 30, 40]
it = zip(words, numbers)
lst = list(it)
w, n = zip(*lst)
print(w)            # prints ('A', 'coddle', 'called', 'Molly')
print(n)            # print (10, 20, 30, 40)
```

Iterators

- We know that a string and container objects like list, tuple, set, dictionary etc. can be iterated through using a **for** loop as in

```
for ch in 'Good Afternoon' :
    print(ch)

for num in [10, 20, 30, 40, 50] :
    print(num)
```

Both these **for** loops call **__iter__( )** method of **str**/**list**. This method returns an iterator object. The iterator object has a method **__next__( )** which returns the next item in the **str**/**list** container.

When all items have been iterated, next call to **__next__( )** raises a **StopIteration** exception which tells the **for** loop to terminate. Exceptions have been discussed in Chapter 22.

- We too can call **__iter__( )** and **__next__( )** and get the same results.

```
lst = [10, 20, 30, 40]
i = lst.__iter__( )
print(i.__next__( ))
print(i.__next__( ))
print(i.__next__( ))
```

- Instead of calling **__iter__( )** and **__next__( )**, we can call the more convenient built-in functions **iter( )** and **next( )**. These functions in turn call **__iter__( )** and **__next__( )** respectively.

```
lst = [10, 20, 30, 40]
i = iter(lst)
print(next(i))
print(next(i))
print(next(i))
```

Note than once we have iterated a container, if we wish to iterate it again we have to obtain an iterator object afresh.

- An iterable is an object capable of returning its members one at a time. Programmatically, it is an object that has implemented **__iter__( )** in it.
- An iterator is an object that has implemented both **__iter__( )** and **__next__( )** in it.
- As a proof that an iterable contains **__iter__( )**, whereas an iterator contains both **__iter__( )** and **__next__( )**, we can check it using the **hasattr( )** built-in function.

```
s = 'Hello'
lst = ['Focussed', 'bursts', 'of', 'activity']
print(hasattr(s, '__iter__'))
print(hasattr(s, '__next__'))
print(hasattr(lst, '__iter__'))
print(hasattr(lst, '__next__'))
i = iter(s)
```

```
j = iter(lst)
print(hasattr(i, '__iter__'))
print(hasattr(i, '__next__'))
print(hasattr(j, '__iter__'))
print(hasattr(j, '__next__'))
```

On execution of this program we get the following output:

```
True
False
True
False
True
True
True
True
```

User-defined Iterators

- Suppose we wish our class to behave like an iterator. To do this we need to define **__iter__( )** and **__next__( )** in it.
- Our iterator class **AvgAdj** should maintain a list. When it is iterated upon it should return average of two adjacent numbers in the list.

```
class AvgAdj :
    def __init__(self, data) :
        self.__data = data
        self.__len = len(data)
        self.__first = 0
        self.__sec = 1

    def __iter__(self) :
        return self

    def __next__(self) :
        if self.__sec == self.__len :
            raise StopIteration        # raises exception (runtime error)
        self.__avg = (self.__data[self.__first] +
                      self.__data[self.__sec]) / 2
        self.__first += 1
        self.__sec += 1
        return self.__avg
```

```
lst = [10, 20, 30, 40, 50, 60, 70]
coll = AvgAdj(lst)
for val in coll :
    print(val)
```

On execution of this program, we get the following output:

```
15.0
25.0
35.0
45.0
55.0
65.0
```

- **__iter__( )** is supposed to return an object which has implemented **__next__( )** in it. Since we have defined **__next__( )** in **AvgAdj** class, we have returned **self** from **__iter__( )**.
- Length of **lst** is 7, whereas elements in it are indexed from 0 to 6.
- When **self._sec** becomes 7 it means that we have reached the end of list and further iteration is not possible. In this situation we have raised an exception **StopIteration**.

Generators

- Generators are very efficient functions that create iterators. They use **yield** statement instead of **return** whenever they wish to return data from the function.
- Specialty of a generator is that, it remembers the state of the function and the last statement it had executed when **yield** was executed.
- So each time **next( )** is called, it resumes where it had left off last time.
- Generators can be used in place of class-based iterator that we saw in the last section.
- Generators are very compact because the **__iter__( )**, **__next__( )** and **StopIteration** code is created automatically for them.
- Given below is an example of a generator that returns average of next two adjacent numbers in the list every time.

```
def AvgAdj(data) :
    for i in range(0, len(data) - 1) :
        yield (data[i] + data[i + 1]) / 2

lst = [10, 20, 30, 40, 50, 60, 70]
for i in AvgAdj(lst) :
    print(i)
```

On execution of this program, we get the following output:

```
15.0
25.0
35.0
45.0
55.0
65.0
```

Which to use When?

- Suppose from a list of 100 integers we are to return an entity which contains elements which are prime numbers. In this case we will return an 'iterable' which contains a list of prime numbers.
- Suppose we wish to add all prime numbers below three million. In this case, first creating a list of all prime numbers and then adding them will consume lot of memory. So we should write an iterator class or a generator function which generates next prime number on the fly and adds it to the running sum.

Generator Expressions

- Like list/set/dictionary comprehensions, to make the code more compact as well as succinct, we can write compact generator expressions.
- A generator expression creates a generator on the fly without being required to use the **yield** statement.
- Some sample generator expressions are given below.

```
# generate 20 random numbers in the range 10 to 100 and obtain
# maximum out of them
```

```
print(max(random.randint(10, 100) for n in range(20)))
# print sum of cubes of all numbers less than 20
print(sum(n * n * n for n in range(20)))
```

- List comprehensions are enclosed within **[ ]**, set/dictionary comprehensions are enclosed within **{ }**, whereas generator expressions are enclosed within **( )**.
- Since a list comprehension returns a list, it consumes more memory than a generator expression. Generator expression takes less memory since it generates the next element on demand, rather than generating all elements upfront.

```
import sys
lst = [i * i for i in range(15)]
gen = (i * i for i in range(15))
print(lst)
print(gen)
print(sys.getsizeof(lst))
print(sys.getsizeof(gen))
```

On execution of this program, we get the following output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
<generator object <genexpr> at 0x003BD570>
100
48
```

- Though useful, generator expressions do not have the same power of a full-fledged generator function.

P</> Programs

Problem 21.1

Write a program that proves that a list is an iterable and not an iterator.

Program

```
lst = [10, 20, 30, 40, 50]
print(dir(lst))
```

```
i = iter(lst)
print(dir(i))
```

Output

```
['__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__',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
['__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__']
```

Tips

- **lst** is an iterable since **dir(lst)** shows **__iter__** but no **__next__**.
- **iter(lst)** returns an iterator object, which is collected in **i**.
- **dir(i)** shows **__iter__** as well as **__next__**. This shows that it is an iterator object.

Problem 21.2

Write a program that generates prime numbers below 3 million. Print sum of these prime numbers.

Program

```
def generate_primes( ) :
    num = 1
    while True :
        if isprime(num) :
            yield num
        num += 1
```

```
def isprime( n ) :
    if n > 1 :
        if n == 2 :
            return True
        if n % 2 == 0 :
            return False
        for i in range(2, n // 2) :
            if n % i == 0 :
                return False
        else :
            return True
    else :
        return False

total = 0
for next_prime in generate_primes( ) :
    if next_prime < 300000 :
        total += next_prime
    else :
        print(total)
        exit( )
```

Output

```
3709507114
```

Tips

- **exit( )** terminates the execution of the program.

Problem 21.3

Write a program that uses dictionary comprehension to print sin, cos and tan tables for angles ranging from 0 to 360 in steps of 15 degrees. Write generator expressions to find the maximum value of sine and cos.

Program

```
import math
pi = 3.14
sine_table = {ang : math.sin(ang * pi / 180) for ang in range(0, 360, 90)}
```

```
cos_table = {ang : math.cos(ang * pi / 180) for ang in range(0, 360, 90)}
tan_table = {ang : math.tan(ang * pi / 180) for ang in range(0, 360, 90)}
print(sine_table)
print(cos_table)
print(tan_table)
maxsin = max((math.sin(ang * pi / 180) for ang in range(0, 360, 90)))
maxcos = max((math.cos(ang * pi / 180) for ang in range(0, 360, 90)))
print(maxsin)
print(maxcos)
```

Output

```
{0: 0.0, 90: 0.9999996829318346, 180: 0.0015926529164868282, 270: -
0.999997146387718}
{0: 1.0, 90: 0.0007963267107332633, 180: -0.9999987317275395, 270: -
0.0023889781122815386}
{0: 0.0, 90: 1255.7655915007897, 180: -0.001592654936407223, 270:
418.58782265388515}
0.9999996829318346
1.0
```

Problem 21.4

Create 3 lists—a list of names, a list of ages and a list of salaries. Generate and print a list of tuples containing name, age and salary from the 3 lists. From this list generate 3 tuples—one containing all names, another containing all ages and third containing all salaries.

Program

```
names = ['Amol', 'Anil', 'Akash']
ages = [25, 23, 27]
salaries= [34555.50, 40000.00, 450000.00]
# create iterator of tuples
it = zip(names, ages, salaries)

# build list by iterating the iterator object
lst = list(it)
print(lst)

# unzip the list into tuples
```

```
n, a, s = zip(*lst)
print(n)
print(a)
print(s)
```

Output

```
[('Amol', 25, 34555.5), ('Anil', 23, 40000.0), ('Akash', 27, 450000.0)]
('Amol', 'Anil', 'Akash')
(25, 23, 27)
(34555.5, 40000.0, 450000.0)
```

Problem 21.5

Write a program to obtain transpose of a 3 x 4 matrix.

Program

```
mat = [[1, 2, 3, 4], [5, 6, 7, 8],  [9, 10, 11, 12]]
ti = zip(*mat)
lst = [[ ] for i in range(4)]
i = 0
for t in ti :
      lst[i] = list(t)
      i += 1
print(lst)
```

Output

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Tips

- **mat** contains a list of lists. These can be accessed using either **mat[0]**, **mat[1]** and **mat[2]** or simply ***mat**.
- **zip(*mat)** receives three lists and returns an iterator of tuples, each tuple containing 3 elements.
- **lst** is intitialized as a list of 4 empty lists.

- The iterator returned by **zip( )** is iterated upon and a list is generated using the **list( )** function. Each generated list is inserted in the list of lists at an appropriate index.

Problem 21.6

Write a program to multiply two matrices x(2 x 3) and y(2, 2) using list comprehension.

Program

```
x = [ [1, 2, 3], 4, 5, 6] ]
y = [ [11, 12], [21, 22], [31, 32] ]

l1 = [xrow for xrow in x]
print(l1)
l2 = [(xrow, ycol) for ycol in zip(*y) for xrow in x]
print(l2)
l3 = [[sum(a * b for a, b in zip(xrow,ycol)) for ycol in zip(*y)]for xrow in x]
print(l3)
```

Output

```
[[1, 2, 3], [4, 5, 6]]
[([1, 2, 3], (11, 21, 31)), ([4, 5, 6], (11, 21, 31)), ([1, 2, 3], (12, 22, 32)),
    ([4, 5, 6], (12, 22, 32))]
[[146, 152], [335, 350]]
```

Tips

- To make it easy for you to understand the list comprehension, I have built it in 3 parts. Follow them by checking their output.

Problem 21.7

Suppose we have a list of 5 integers and a tuple of 5 floats. Can we zip them and obtain an iterator. If yes, how?

Program

```
integers = [10, 20, 30, 40, 50]
```

```
floats = (1.1, 2.2, 3.3, 4.4, 5.5)
ti = zip(integers, floats)
lst = list(ti)
for i, f in lst :
    print(i, f)
```

Output

```
10 1.1
20 2.2
30 3.3
40 4.4
50 5.5
```

Tips

- Any type of iterables can be passed to a **zip( )** function.

Problem 21.8

Create two lists **students** and **marks**. Create a dictionary from these two lists using dictionary comprehension. Use names as keys and marks as values.

Program

```
# lists of keys and values
lstnames = ['Sunil', 'Sachin', 'Rahul', 'Kapil', 'Rohit']
lstmarks = [54, 65, 45, 67, 78]

# dictionary comprehension
d = {k:v for (k, v) in zip(lstnames, lstmarks)}
print(d)
```

Output

```
{'Sunil': 54, 'Sachin': 65, 'Rahul': 45, 'Kapil': 67, 'Rohit': 78}
```

Problem 21.9

Create a dictionary containing names of students and marks obtained by them in three subjects. Write a program to print these names in tabular form with sorted names as columns and marks in three subjects listed below each student name as shown below.

Rahul	Rakesh	Sameer
67	59	58
76	70	86
39	81	78

Program

```
d = {'Rahul':[67,76,39],'Sameer':[58,86,78],'Rakesh':[59,70,81]}

lst = [(k, *v) for k, v in d.items( )]
print(lst)

lst = [(k, *v) for k, v in sorted(d.items( ))]
print(lst)

for row in zip(*lst) :
    print(row)

for row in zip(*lst) :
    print(*row, sep = '\t')

for row in zip(*((k, *v) for k, v in sorted(d.items( )))):
    print(*row, sep = '\t')
```

Output

```
[('Rahul', 67, 76, 39), ('Sameer', 58, 86, 78), ('Rakesh', 59, 70, 81)]
[('Rahul', 67, 76, 39), ('Rakesh', 59, 70, 81), ('Sameer', 58, 86, 78)]
('Rahul', 'Rakesh', 'Sameer')
(67, 59, 58)
(76, 70, 86)
(39, 81, 78)
Rahul   Rakesh  Sameer
67      59      58
76      70      86
39      81      78
Rahul   Rakesh  Sameer
67      59      58
```

```
76      70      86
39      81      78
```

Tips

- Try to understand this program step-by-step:

  lst = [(k, *v) for k, v in d.items( )]

  *v will unpack the marks in v. So a tuple like ('Rahul', 67, 76, 39) will be created. All such tuples will be collected in the list to create:

  [('Rahul', 67, 76, 39), ('Sameer', 58, 86, 78), ('Rakesh', 59, 70, 81)]

- To create a list of tuples sorted by name we have used the **sorted( )** function:

  lst = [(k, *v) for k, v in sorted(d.items( ))]

  This will create the list:

  [('Rahul', 67, 76, 39), ('Rakesh', 59, 70, 81), ('Sameer', 58, 86, 78)]

- The sorted list is then unpacked and submitted to the **zip( )** function

  ```
  for row in zip(*lst) :
      print(row)
  ```

  This will print the tuples

  ```
  ('Rahul', 'Rakesh', 'Sameer')
  (67, 59, 58)
  (76, 70, 86)
  (39, 81, 78)
  ```

- We have then unpacked these tuples before printing and added separator '\t' to properly align the values being printed.

  ```
  for row in zip(*lst) :
      print(*row, sep = '\t')
  ```

- Lastly we have combined all these activities into one loop:

  ```
  for row in zip(*((k, *v) for k, v in sorted(d.items( )))):
      print(*row, sep = '\t')
  ```

Problem 21.10

Write a program that defines a function **pascal_triangle( )** that displays a Pascal Triangle of level received as parameter to the function. A Pascal's Triangle of level 5 is shown below.

```
                1
             1     1
          1     2     1
       1     3     3     1
    1     4     6     4     1
```

Program

```
def pascal_triangle(n) :
    row = [1]
    z = [0]
    for x in range(n) :
        print(row)
        row = [l + r for l, r in zip(row + z, z + row)]
pascal_triangle(5)
```

Output

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
```

Tips

- For **n = 5**, **x** will vary from 0 to 4.
- **row + z** merges two lists.
- For x = 1, row = [1], z = [0], so,
  zip([1, 0], [0, 1]) gives tuples (1, 0), (0, 1)
  l + r gives row = [1, 1]

- For x = 2, row = [1, 1], z = [0], so,

  zip([1, 1, 0], [0, 1, 1]) gives tuples (1, 0), (1, 1), (0, 1)
  l + r gives [1, 2, 1]

- For x = 3, row = [1, 2, 1], z = [0], so,

  zip([1, 2, 1, 0], [0, 1, 2, 1]) gives tuples (1, 0), (2, 1), (1, 2), (0, 1)
  l + r gives [1, 3, 3, 1]

- For x = 4, row = [1, 3, 3, 1], z = [0], so,

  zip([1, 3, 3, 1, 0], [0, 1, 3, 3, 1]) gives (1, 0), (3, 1), (3, 3), (1, 3), (0, 1)
  l + r gives [1, 4, 6, 4, 1]

Problem 21.11

Write a program that defines a class called **Progression** and inherits three classes from it **AP**, **GP** and **FP**, standing for Arithmetic Progression, Geometric Progression and Fibonacci Progression respectively. **Progression** class should act as a user-defined iterator. By default, it should generate integers stating with 0 and advancing in steps of 1. **AP**, **GP** and **FP** should make use of the iteration facility of **Progression** class. They should appropriately adjust themselves to generate numbers in arithmetic progression, geometric progression or Fibonacci progression.

Program

```
class Progression :
    def __init__ (self, start = 0) :
        self._cur = start

    def __iter__ (self):
        return self

    def advance(self):
        self._cur += 1

    def __next__ (self) :
        if self._cur is None :
            raise StopIteration
        else :
            data = self._cur
            self.advance( )
```

```
            return data

    def display(self, n) :
        print('  '.join(str(next(self)) for i in range(n)))

class AP(Progression) :
    def __init__ (self, start = 0, step = 1) :
        super( ).__init__ (start)
        self.__step = step

    def advance(self) :
        self._cur += self.__step

class GP(Progression) :
    def __init__ (self, start = 1, step = 2 ) :
        super( ).__init__(start)
        self.__step = step

    def advance(self) :
        self._cur *= self.__step

class FP(Progression) :
    def __init__ (self, first = 0, second = 1) :
        super( ).__init__(first)
        self.__prev = second - first

    def advance(self) :
        self.__prev, self._cur = self._cur, self.__prev + self._cur

print('Default progression:')
p = Progression( )
p.display(10)
print('AP with step 5:')
a = AP(5)
a.display(10)
print('AP with start 2 and step 4:')
a = AP(2, 4)
a.display(10)
print('GP with default multiple:')
g = GP( )
g.display(10)
```

```
print('GP with start 1 and multiple 3:')
g = GP(1, 3)
g.display(10)
print('FP with default start values:')
f = FP( )
f.display(10)
print('FP with start values 4 and 6:')
f = FP(4, 6)
f.display(10)
```

Output

```
Default progression:
0 1 2 3 4 5 6 7 8 9
AP with step 5:
5 6 7 8 9 10 11 12 13 14
AP with start 2 and step 4:
2 6 10 14 18 22 26 30 34 38
GP with default multiple:
1 2 4 8 16 32 64 128 256 512
GP with start 1 and multiple 3:
1 3 9 27 81 243 729 2187 6561 19683
FP with default start values:
0 1 1 2 3 5 8 13 21 34
FP with start values 4 and 6:
4 6 10 16 26 42 68 110 178 288
```

Tips

- Since **Progression** is an iterator it has to implement **__iter__( )** and **__next__( )** methods.
- **__next__( )** calls **advance( )** method to suitably adjust the value of **self.cur** (and **self.prev** in case of **FP**).
- Each derived class has an **advance( )** method. Depending on which object's address is present in **self**, that object's **advance( )** method gets called.
- The generation of next data value happens one value at a time, when **display( )** method's **for** loop goes into action.
- There are two ways to create an object and call **display( )**. These are:

```
a = AP(5)
a.display(10)
```

or

```
AP(5).display(10)
```

[A] Answer the following:

(a) Write a program to create a list of 5 odd integers. Replace the third element with a list of 4 even integers. Flatten, sort and print the list.

(b) Write a program to flatten the following list:

mat1 = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

(c) Write a program to generate a list of numbers in the range 2 to 50 that are divisible by 2 and 4.

(d) Suppose there are two lists, each holding 5 strings. Write a program to generate a list that consists of strings that are concatenated by picking corresponding elements from the two lists.

(e) Suppose a list contains 20 integers generated randomly. Receive a number from the keyboard and report position of all occurrences of this number in the list.

(f) Suppose there are two lists—one contains questions and another contains lists of 4 possible answers for each question. Write a program to generate a list that contains lists of question and its 4 possible answers.

(g) Suppose a list has 20 numbers. Write a program that removes all duplicates from this list.

(h) Write a program to obtain a median value of a list of numbers, without disturbing the order of the numbers in the list.

(i) A list contains only positive and negative integers. Write a program to obtain the number of negative numbers present in the list.

(j) Write a program to convert a list of tuples

[(10, 20, 30), (150.55, 145.60, 157.65), ('A1', 'B1', 'C1')]

into another list of tuples

[(10, 150.55, 'A1'), (20, 145.60, 'B1'), (30, 157.65, 'C1')]

(k) What will be the output of the following program:

```
x = [[1, 2, 3, 4], [4, 5, 6, 7]]
y = [[1, 1], [2, 2], [3, 3], [4, 4]]
l1 = [xrow for xrow in x]
print(l1)
l2 = [(xrow, ycol) for ycol in zip(*y) for xrow in x]
print(l2)
```

(l) Write a program that uses a generator to create a set of unique words from a line input through the keyboard.

(m) Write a program that uses a generator to find out maximum marks obtained by a student and his name from tuples of multiple students.

(n) Write a program that uses a generator that generates characters from a string in reverse order.

(o) What is the difference between the following statements:

```
sum([x**2 for x in range(20)])
sum(x**2 for x in range(20))
```

(p) Suppose there are two lists, each holding 5 strings. Write a program to generate a list that consists of strings that are concatenated by picking corresponding elements from the two lists.

(q) 36 unique combinations can result from use of two dice. Create a dictionary which stores these combinations as tuples.