GKTCS INNOVATIONS

Iterator, Generator Collection framework

What is a Iterator?

GKTCS INNOVATIONS

An iterator is an object that contains a countable number of values.

Python Iterators

GKTCS INNOVATIONS

- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().

Iterator vs Iterable

GKTCS INNOVATIONS

- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.

- All these objects have a iter() method which is used to get an iterator:

Example:-

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
print(next(myit))
print(next(myit))
print(next(myit))
```

Output

GKTCS INNOVATIONS

```
apple
banana
cherry
```

Looping Through an Iterator

GKTCS INNOVATIONS

We can also use a for loop to iterate through an iterable object:

Example:

Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
  print(x)
```

Output:

```
apple
banana
cherry
```

Example:

GKTCS INNOVATIONS

Iterate the characters of a string:

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
  print(x)
```

Output:

```
b
a
n
a
n
a
```

Create an Iterator

GKTCS INNOVATIONS

- To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.
- The __iter__() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.
- The __next__() method also allows you to do operations, and must return the next item in the sequence.

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

GKTCS INNOVATIONS

Example:

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Output:

GKTCS INNOVATIONS

```
1
2
3
4
5
```

StopIteration

GKTCS INNOVATIONS

- To prevent the iteration to go on forever, we can use the **StopIteration** statement.

- In the **__next__()** method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example:

Stop after 20 iterations:

GKTCS INNOVATIONS

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration.

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

Output:


```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

What is a Generator?

GKTCS INNOVATIONS

Generator is used for simplifying the creation of iterators.

Are defined with the def keyword

- Use the yield keyword
- May use several yield keywords
- Return an iterator

Generators

GKTCS INNOVATIONS

- Generators simplifies creation of iterators.
- A generator is a function that produces a sequence of results instead of a single value.
- A *generator* is a special routine that can be used to control the iteration behavior of a loop.
- A generator is similar to a function returning an array.
- A generator has parameters, it can be called and it generates a sequence of result.
- But unlike functions, which *return* a whole array, a generator *yields* one value at a time.

Example:

GKTCS INNOVATIONS

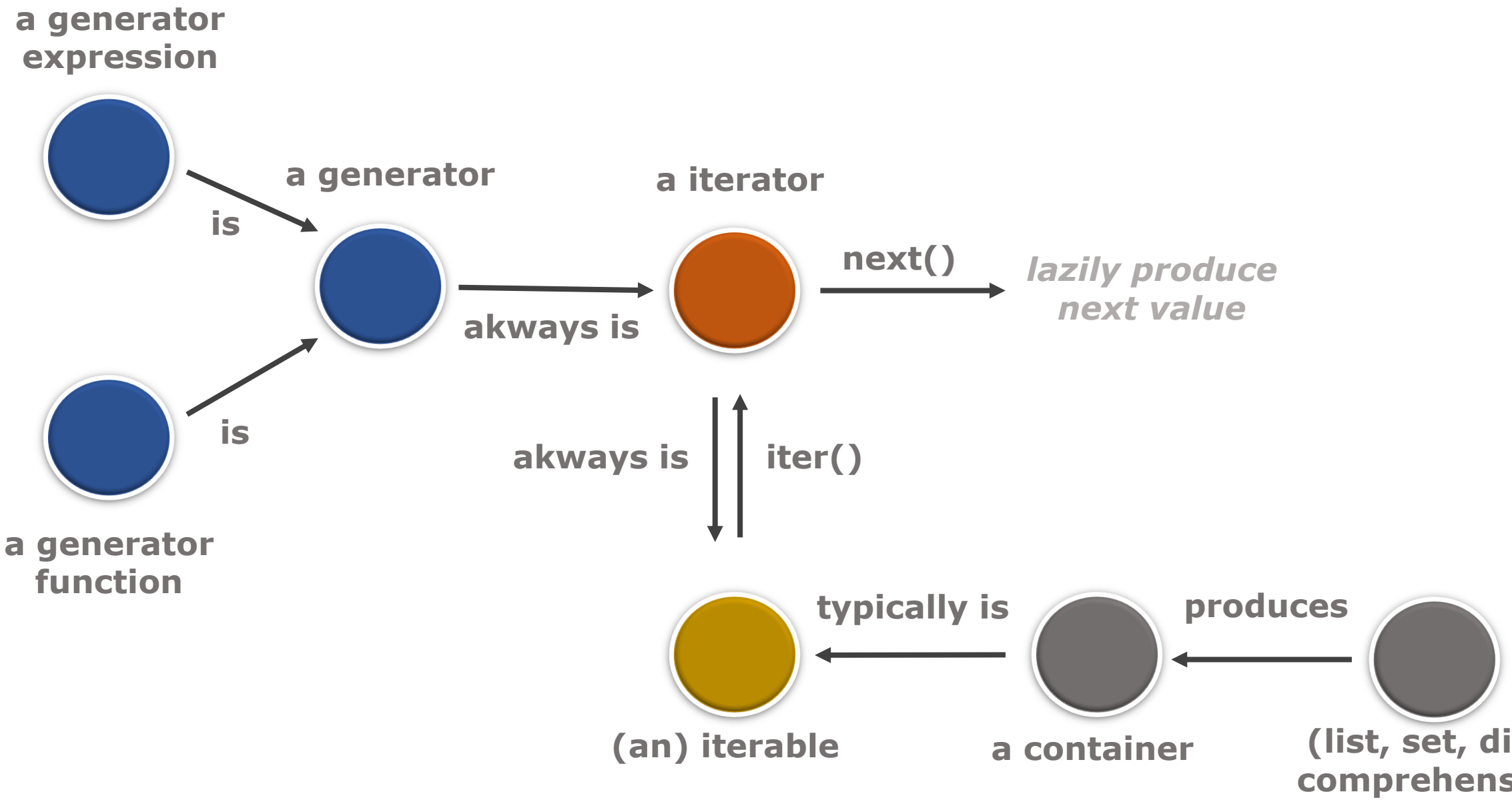
```
def yrange(n):
i = 0
while i < n:
   yield i
   i += 1
```

Each time the yield statement is executed the function generates a new value.

```
y = yrange(3)
o/p:  y

y.next()
o/p: 0

y.next()
o/p: 1

y.next()
o/p: 2

y.next()
TraceBack (most recent call last):
File<stdin>, line 1, in <module>
StopIterartions
```

Relationship between Generator & Iterator

GKTCS INNOVATIONS



Advantages

GKTCS INNOVATIONS

1. Iterators can work with infinite sequences
2. Memory and CPU efficient
3. Generator allows to write streaming code with fewer intermediate variables and data structures.
4. Iterators save resources
5. Cleaner code

Collections Module

GKTCS INNOVATIONS

The collections module provides alternatives to built-in container data types such as list, tuple and dict.

- namedtuple()
- OrderedDict()
- deque()

namedtuple()

GKTCS INNOVATIONS

The namedtuple() function returns a tuple-like object with named fields. These field attributes are accessible by lookup as well as by index.

General usage of this function is:

Signature:

collections.namedtuple(type_name, field-list)

Example:-

GKTCS INNOVATIONS

The following statement declares a student class having name, age and marks as fields.

```
>>> import collections
>>> student = collections.namedtuple('student', [name, age, marks])
```

To create a new object of this namedtuple, do the following:

```
>>> s1=student("Imran", 21, 98)
```

The values of the field can be accessible by attribute lookup:

```
>>> s1.name
o/p: 'Imran'
```

Or by index:

```
>>>s1[0]
o/p: 'Imran'
```

OrderedDict()

GKTCS INNOVATIONS

The OrderedDict() function is similar to a normal dictionary object in Python. However, it remembers the order of the keys in which they were first inserted.

```
import collections
    d1=collections.OrderedDict()
    d1['A']=65
    d1['C']=67
    d1['B']=66
    d1['D']=68
    for k,v in d1.items():
print (k,v)
```

Output:

GKTCS INNOVATIONS

```
A 65
C 67
B 66
D 68
```

Upon traversing the dictionary, pairs will appear in the order of their insertion.

deque()

GKTCS INNOVATIONS

A deque object support appends and pops from either ends of a list. It is more memory efficient than a normal list object. In a normal list object, the removal of any item causes all items to the right to be shifted towards left by one index. Hence, it is very slow.

```
>>> q=collections.deque([10,20,30,40])
>>>q.appendleft(0)
>>>q
deque([0, 10, 20, 30, 40])
>>>q.append(50)
>>>q
deque([0, 10, 20, 30, 40, 50])
>>>q.pop()
50
>>>q
deque([0, 10, 20, 30, 40])
>>>q.popleft()
0
>>>q
deque([10, 20, 30, 40])
```