Jawaharlal Nehru Technological University Hyderabad

S C D E

Kukatpally, Hyderabad - 500 085, Telangana, India

Iterators, Generators, Decorators

Session 10 , 23 May 23

Iterators and generators are used to implement custom iterable objects in python, and decorators are used to modify the behavior of functions.

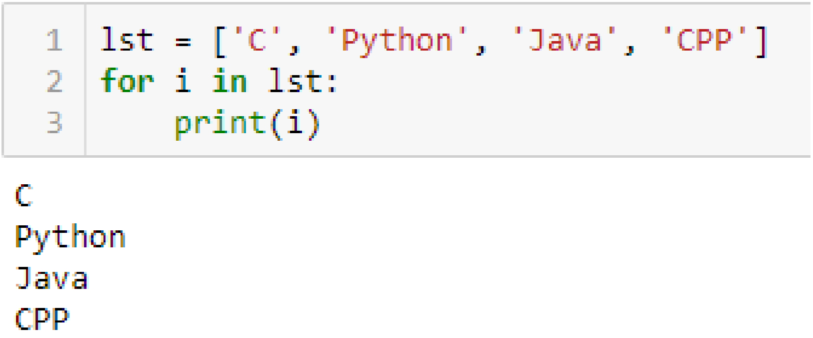
Dr N V Ganapathi Raju
Professor and HOD of IT
Gokaraju Rangaraju Institute of Eng and Tech

Iterables

- *Iterables are objects that are capable of returning their members one at a time, generally will be done using a for-loop.*

- *Objects like lists, tuples, sets, dictionaries, strings, etc. are called iterables. In short, anything you can loop over is an iterable.*

```
lst = ['C', 'Python', 'Java', 'CPP']

for i in lst:

    print(i)
```



```
lst = ['C', 'Python', 'Java', 'CPP']
for i in lst:
    print(i)
```

```
C
Python
Java
CPP
```

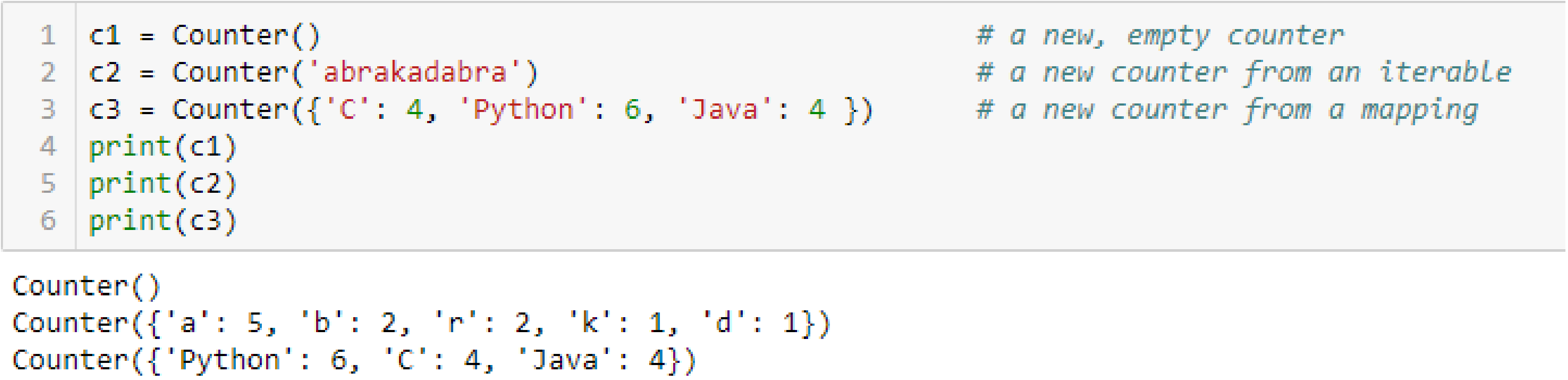
Collections-Counter

- A Counter is a dict subclass for counting hashable objects.
- It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values.
- Counts are allowed to be any integer value including zero or negative counts.
- We should import Counter using the following statement

```
from collections import Counter
```

Creating counter objects

```
c1 = Counter()                                   # a new, empty counter
c2 = Counter('abrakadabra')                      # a new counter from an iterable
c3 = Counter({'C': 4, 'Python': 6, 'Java': 4 })  # a new counter from a mapping
print(c1)
print(c2)
print(c3)
```

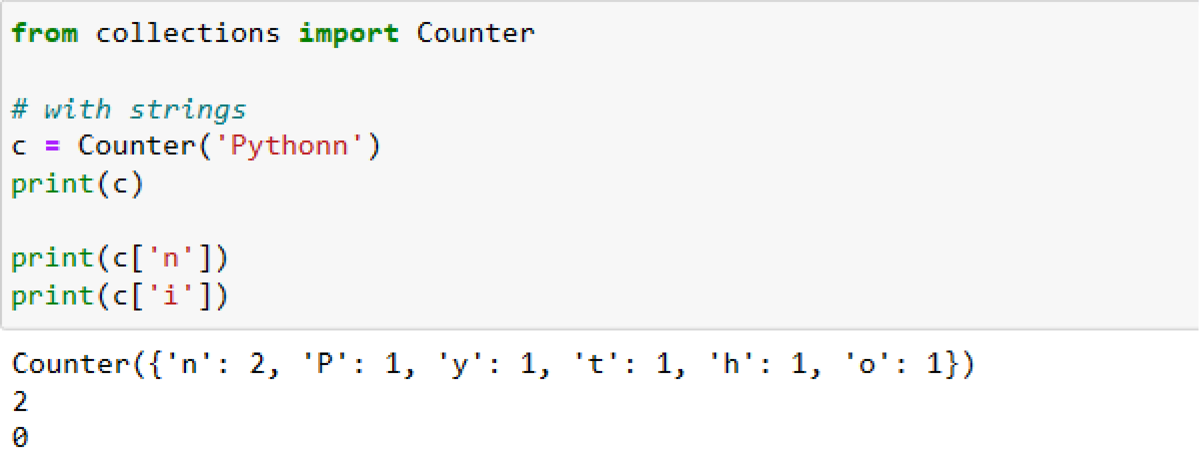


```
c1 = Counter()                                   # a new, empty counter
c2 = Counter('abrakadabra')                      # a new counter from an iterable
c3 = Counter({'C': 4, 'Python': 6, 'Java': 4 })  # a new counter from a mapping
print(c1)
print(c2)
print(c3)
```

```
Counter()
Counter({'a': 5, 'b': 2, 'r': 2, 'k': 1, 'd': 1})
Counter({'Python': 6, 'C': 4, 'Java': 4})
```

Counter object with strings

```
from collections import Counter

c = Counter(‘Pythonn')

print(c)


print(c['n'])

print(c['i'])
```



```
from collections import Counter

# with strings
c = Counter('Pythonn')
print(c)

print(c['n'])
print(c['i'])
```

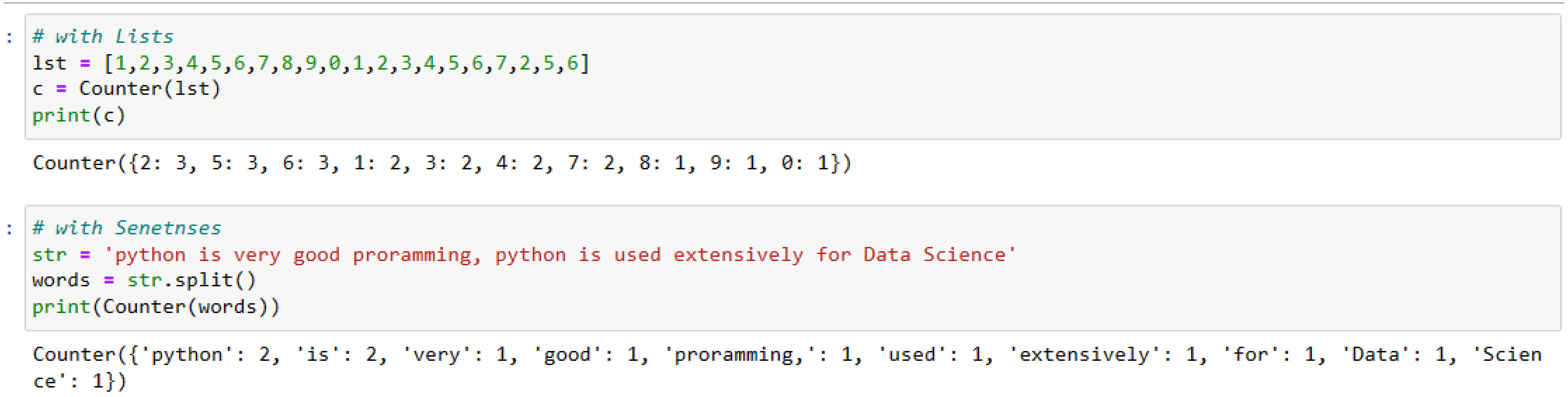
```
Counter({'n': 2, 'P': 1, 'y': 1, 't': 1, 'h': 1, 'o': 1})
2
0
```

Creating Counter with List, Sentences

lst = [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,2,5,6]

c = Counter(lst)

print(c)



```
# with Lists
lst = [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,2,5,6]
c = Counter(lst)
print(c)
```

```
Counter({2: 3, 5: 3, 6: 3, 1: 2, 3: 2, 4: 2, 7: 2, 8: 1, 9: 1, 0: 1})
```

```
# with Senetnses
str = 'python is very good proramming, python is used extensively for Data Science'
words = str.split()
print(Counter(words))
```

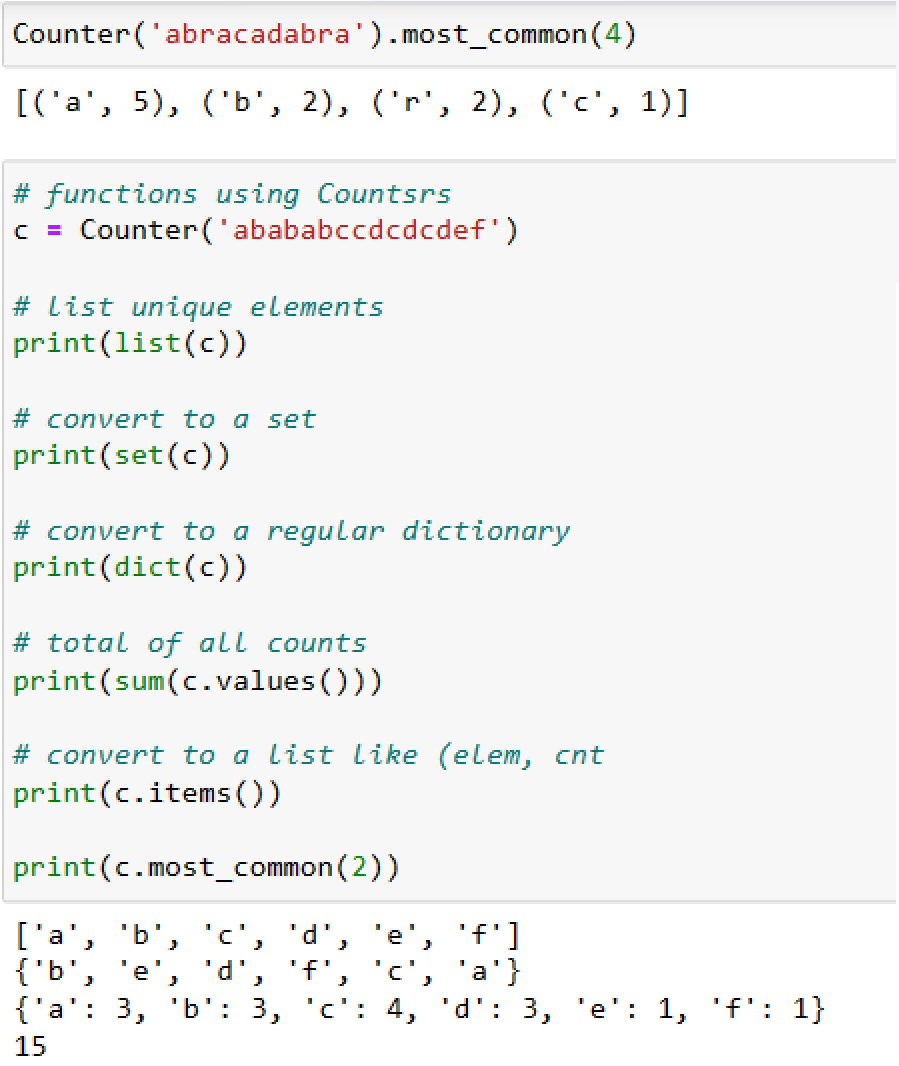
```
Counter({'python': 2, 'is': 2, 'very': 1, 'good': 1, 'proramming,': 1, 'used': 1, 'extensively': 1, 'for': 1, 'Data': 1, 'Scien
ce': 1})
```

Most common words

- Function : most_common([n])
- Returns a list of the most common elements with their counts.

  The number of elements has to be specified as n.

  If none is specified it returns the count of all the elements



```
Counter('abracadabra').most_common(4)
```

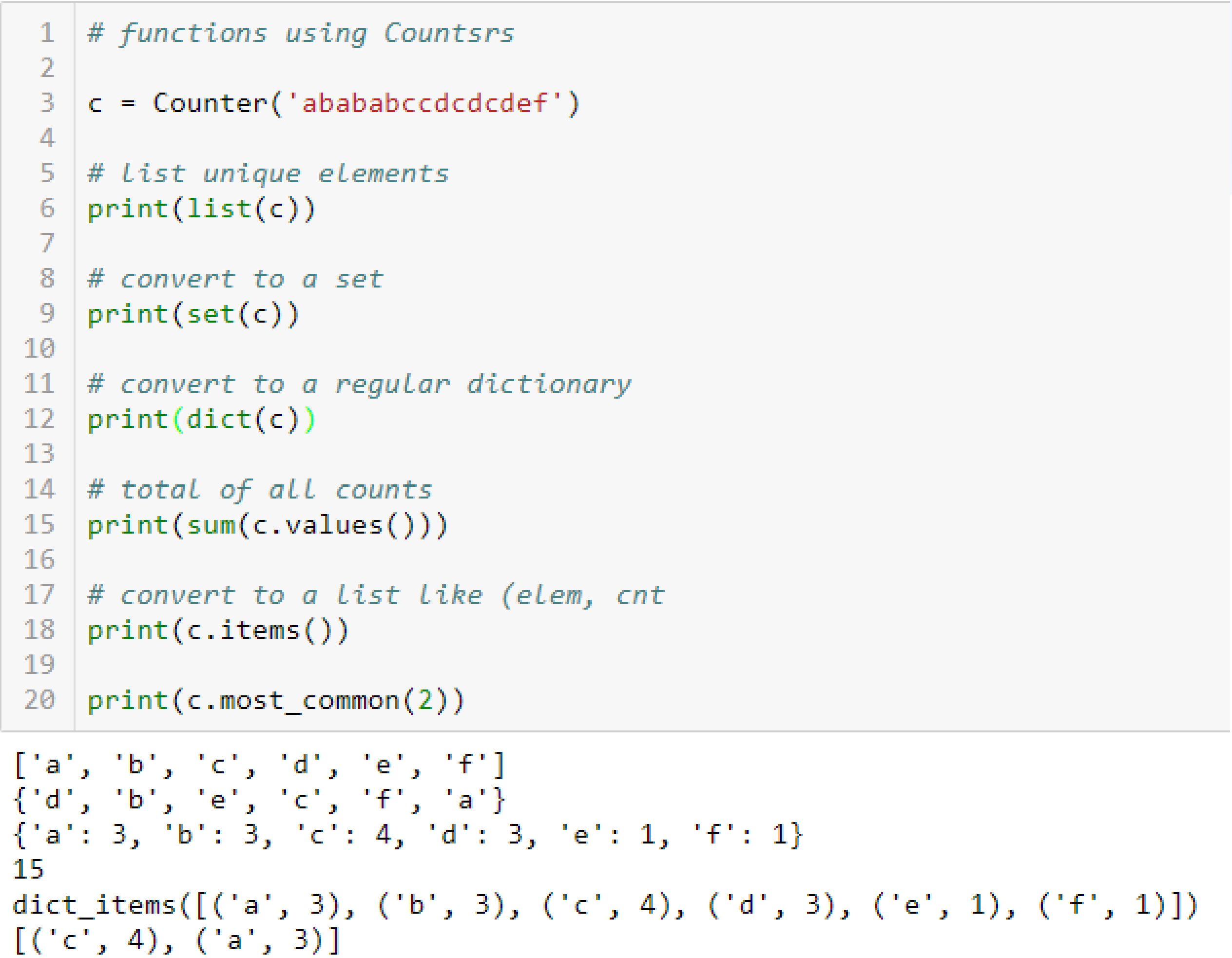
```
[('a', 5), ('b', 2), ('r', 2), ('c', 1)]
```

```
# functions using Countsrs
c = Counter('ababaabccdcdcdef')

# list unique elements
print(list(c))

# convert to a set
print(set(c))

# convert to a regular dictionary
print(dict(c))

# total of all counts
print(sum(c.values()))

# convert to a list like (elem, cnt
print(c.items())

print(c.most_common(2))
```

```
['a', 'b', 'c', 'd', 'e', 'f']
{'b', 'e', 'd', 'f', 'c', 'a'}
{'a': 3, 'b': 3, 'c': 4, 'd': 3, 'e': 1, 'f': 1}
15
```

Common patterns when using the Counter() object

```
c = Counter('abababccdcdcdef')

# list unique elements
print(list(c))

# convert to a set
print(set(c))

# convert to a regular dictionary
print(dict(c))

# total of all counts
print(sum(c.values()))

# convert to a list like (elem, cnt
print(c.items())

print(c.most_common(2))
```



```
# functions using Countsrs

c = Counter('abababccdcdcdef')

# list unique elements
print(list(c))

# convert to a set
print(set(c))

# convert to a regular dictionary
print(dict(c))

# total of all counts
print(sum(c.values()))

# convert to a list like (elem, cnt
print(c.items())

print(c.most_common(2))
```

```
['a', 'b', 'c', 'd', 'e', 'f']
{'d', 'b', 'e', 'c', 'f', 'a'}
{'a': 3, 'b': 3, 'c': 4, 'd': 3, 'e': 1, 'f': 1}
15
dict_items([('a', 3), ('b', 3), ('c', 4), ('d', 3), ('e', 1), ('f', 1)])
[('c', 4), ('a', 3)]
```

Mathematical operations with Counter

c1 = Counter (a=5,b=4, e=2)

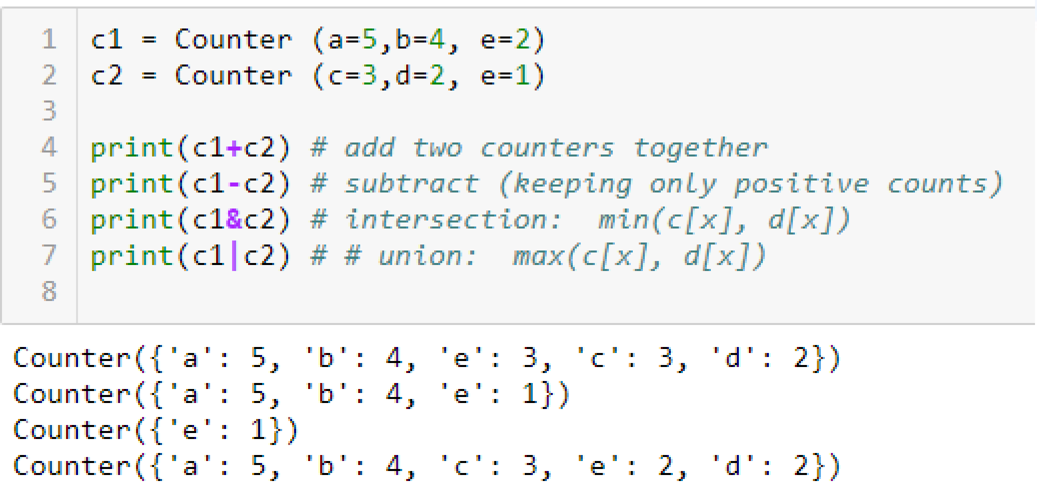
c2 = Counter (c=3,d=2, e=1)

print(c1+c2) # add two counters together

print(c1-c2) # subtract (keeping only positive counts)

print(c1&c2) # intersection: min(c[x], d[x])

print(c1|c2) # union: max(c[x], d[x])



```
c1 = Counter (a=5,b=4, e=2)
c2 = Counter (c=3,d=2, e=1)

print(c1+c2) # add two counters together
print(c1-c2) # subtract (keeping only positive counts)
print(c1&c2) # intersection:  min(c[x], d[x])
print(c1|c2) # # union:  max(c[x], d[x])

```

```
Counter({'a': 5, 'b': 4, 'e': 3, 'c': 3, 'd': 2})
Counter({'a': 5, 'b': 4, 'e': 1})
Counter({'e': 1})
Counter({'a': 5, 'b': 4, 'c': 3, 'e': 2, 'd': 2})
```

Iterators

- Iterators are objects that allow us to **traverse** through **collection**, return one element at a time.

- Iterator, implemented in constructs like for-loops, comprehensions, and python generators.

- An iterator keeps track of the **current state** of an iterable.

Iterator methods

- Python **iterator object** implement two special methods

__iter__() : returns the iterator object itself.

__next__() : must return the next item in the sequence.

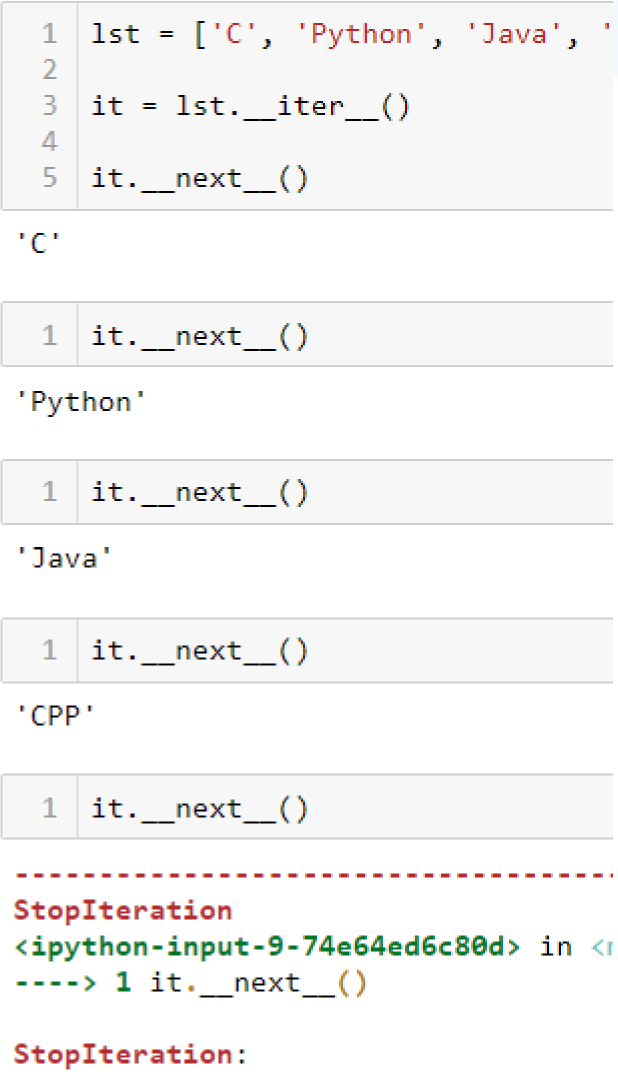
On reaching the end, raise StopIteration exception

*instead of using the **__iter__()** and **__next__()** methods, you can use the **iter()** and **next()** methods*

- next() : to get the next element
- Iter() function (which in turn calls the__iter__()) returns an iterator from them.

Example for iterators

```
lst = ['C', 'Python', 'Java', 'CPP']

it = lst.__iter__()

it.__next__()
```



```
lst = ['C', 'Python', 'Java', '

it = lst.__iter__()

it.__next__()
```

```
'C'
```

```
it.__next__()
```

```
'Python'
```

```
it.__next__()
```

```
'Java'
```

```
it.__next__()
```

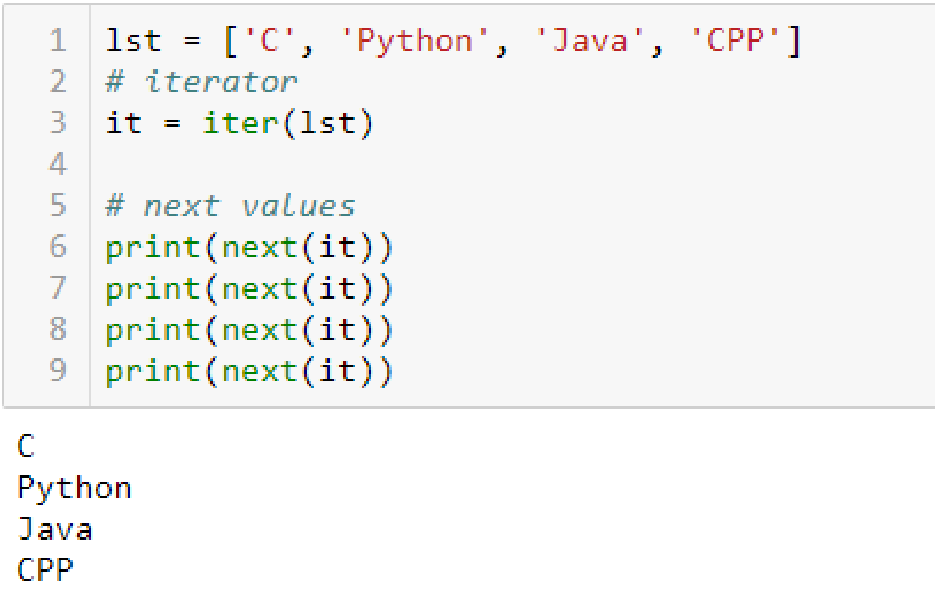
```
'CPP'
```

```
it.__next__()
```

```
-----------------------------------
StopIteration
<ipython-input-9-74e64ed6c80d> in <
----> 1 it.__next__()

StopIteration:
```

Example for iterators

```
lst = ['C', 'Python', 'Java', 'CPP']

# iterator

it = iter(lst)


# next values

print(next(it))

print(next(it))

print(next(it))

print(next(it))
```

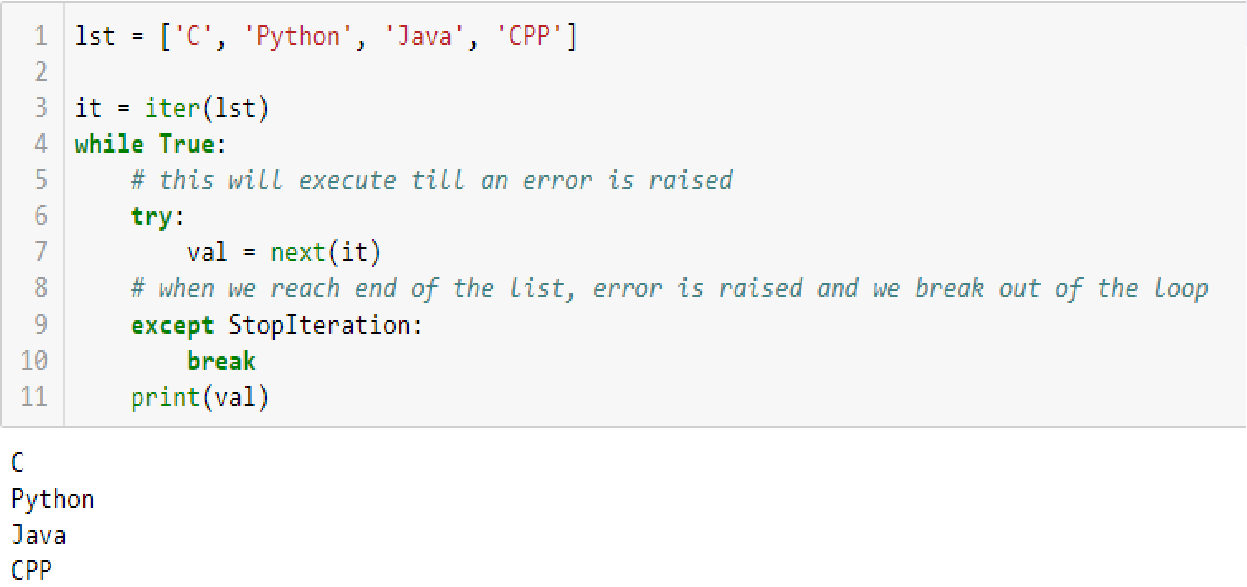


```
lst = ['C', 'Python', 'Java', 'CPP']
# iterator
it = iter(lst)

# next values
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

```
C
Python
Java
CPP
```

Example for iterators

```
lst = ['C', 'Python', 'Java', 'CPP']

it = iter(lst)

while True:

    # this will execute till an error is raised

    try:

        val = next(it)

    # when we reach end of the list,

    # error is raised and we break out of the loop

    except StopIteration:

        break

    print(val)
```



```
lst = ['C', 'Python', 'Java', 'CPP']

it = iter(lst)
while True:
    # this will execute till an error is raised
    try:
        val = next(it)
    # when we reach end of the list, error is raised and we break out of the loop
    except StopIteration:
        break
    print(val)
```

```
C
Python
Java
CPP
```

Generators

- Generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Creating a generator functions:

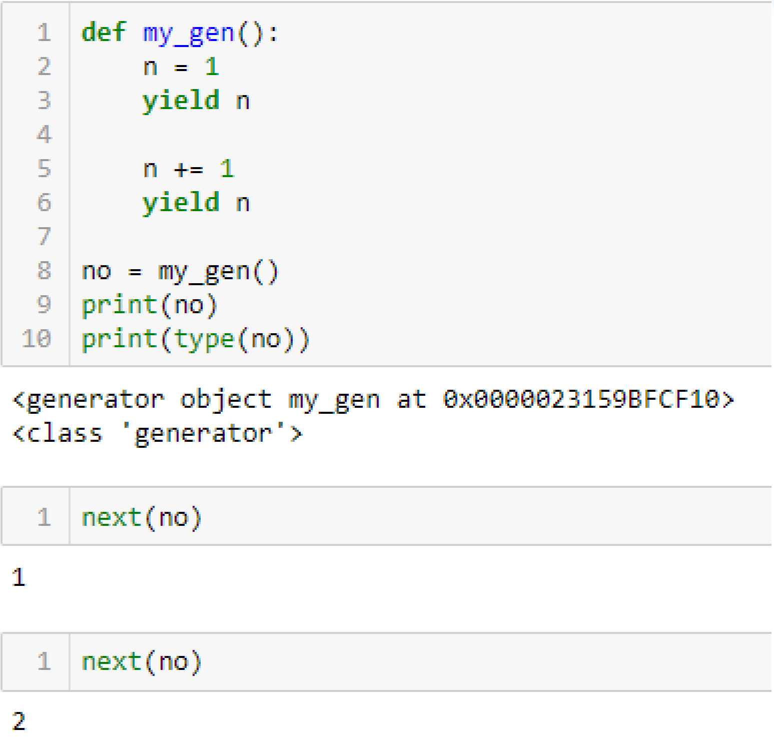
- Define a **normal function**, but with a **yield** statement.
- If a function contains at least one **yield** statement, it becomes a generator function.
  - a **return** statement terminates a function entirely,
  - **yield** statement pauses the function saving all its states and later continues from there on successive calls.

Points to Remember

- Generator function contains **one or more yield** statements.
- When called, it returns an **object (iterator)** but does not start execution immediately.
- Methods like **__iter__() and __next__(**) are implemented automatically. So we can iterate through the items using next().
- Once the function **yields**, the function is paused and the control is transferred to the **caller**.
- **Local variables and their states are remembered between successive calls.**
- Finally, when the function terminates, **StopIteration** is raised automatically on further calls.

Example on generator

```
def my_gen():
    n = 1
    yield n

    n += 1
    yield n

no = my_gen()
print(no)
print(type(no))

next(no)
```



```
def my_gen():
    n = 1
    yield n

    n += 1
    yield n

no = my_gen()
print(no)
print(type(no))
```

```
<generator object my_gen at 0x0000023159BFCF10>
<class 'generator'>
```

```
next(no)
```

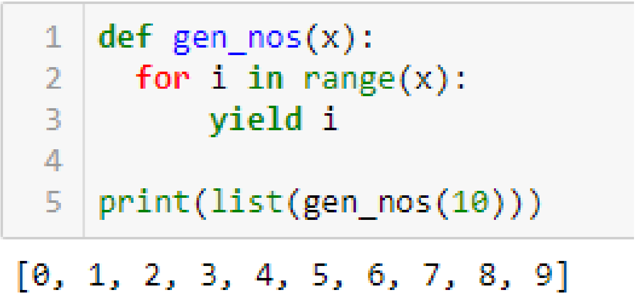
```
1
```

```
next(no)
```

```
2
```

Example on generator

```
def gen_nos(x):
  for i in range(x):
     yield i
print(list(gen_nos(10)))
```



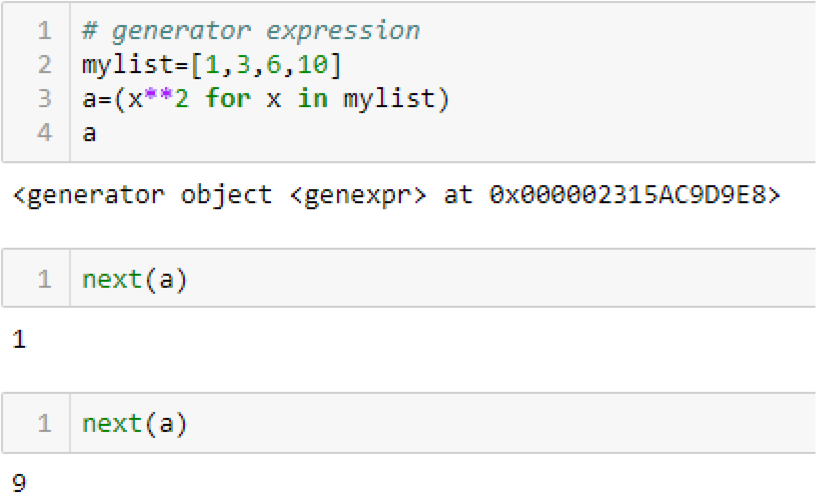
```
def gen_nos(x):
  for i in range(x):
      yield i

print(list(gen_nos(10)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Generator Expressions

- A generator expression, much like list comprehension. The only difference is that unlike a list comprehension, a generator expression is enclosed within parenthesis.

```
# generator expression

mylist=[1,3,6,10]
a=(x**2 for x in mylist)
print(a)

print(next(a))

print(next(a))
```



```
# generator expression
mylist=[1,3,6,10]
a=(x**2 for x in mylist)
a
```

```
<generator object <genexpr> at 0x000002315AC9D9E8>
```

```
next(a)
```

```
1
```

```
next(a)
```

```
9
```

Decorators

- In Python, functions are first-class objects. i.e. functions can be passed around and used as arguments, just like any other object (string, int, float, list, and so on).
- decorators wrap a function, modifying its behavior.
- A decorator takes a function, extends it and returns. a function can return a function.

Case 1. Everything in Python is an object

Case 1. Different names can be bound to the same function object.

```
def fun1(str):
    print(str)

fun1("Hai Data Science")

fun2 = fun1
del fun1

fun2("Hello Machine Learning")
```

Note: fun1 and fun2 refer same function object.

Case 2. Functions can be passed as arguments to another function

Case 2. Functions can be passed as arguments to another function.

```
def inc(x):
    return x + 1

def dec(x):
    return x - 1

def opr(func, x):
    result = func(x)
    return result

opr(inc,3)   # 4

opr(dec,3)  # 2
```

Case 3. A function can return another function

Case 3. A function can return another function.

```
def fun1():
    def fun2():
        print("Hello DS participants")
    return fun2

ret_fun = fun1()

ret_fun()
```

Note: fun2() is a nested function which is defined and returned each time we call fun1().

Case 4: A nested function accessing non local variables

Case 4: A nested function accessing nonlocal variables

```
def print_msg(msg):

    def show():
        print(msg)

    show()

print_msg("Hello DS Students")
```

Note: nested show() function was able to access the non-local msg variable of the enclosing function.

A function defined inside another function is called a nested function.
Nested functions can access variables of the enclosing scope.
In Python, these non-local variables are read-only by default.

Case 5: Python Closures

Case 5: Closures

```
def print_msg(msg):

    def show():
        print(msg)

    return show

res = print_msg("Hello DS")

res()

del print_msg

res()
```

Note:
The print_msg() function was called with the string "Hello DS" and the returned function was bound to the name res.

On calling res(), the message was still remembered although we had already finished executing the print_msg() function.

This technique by which some data ("Hello DS) gets attached to the code is called closure in Python.

This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

Case 6: Decorators

```
def decor(func):
    def wrap():
        print("*******************")
        func()
        print("*******************")
    return wrap

def sayhello():
    print("Hello Data Science")

newfunc=decor(sayhello)

newfunc()
```