Jawaharlal Nehru Technological University Hyderabad

SCDE

Kukatpally, Hyderabad - 500 085, Telangana, India

Collections-

Counter, OrderedDict, defaultdict, namedtuple, ChainMap

Session 12 , 25 May 23

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

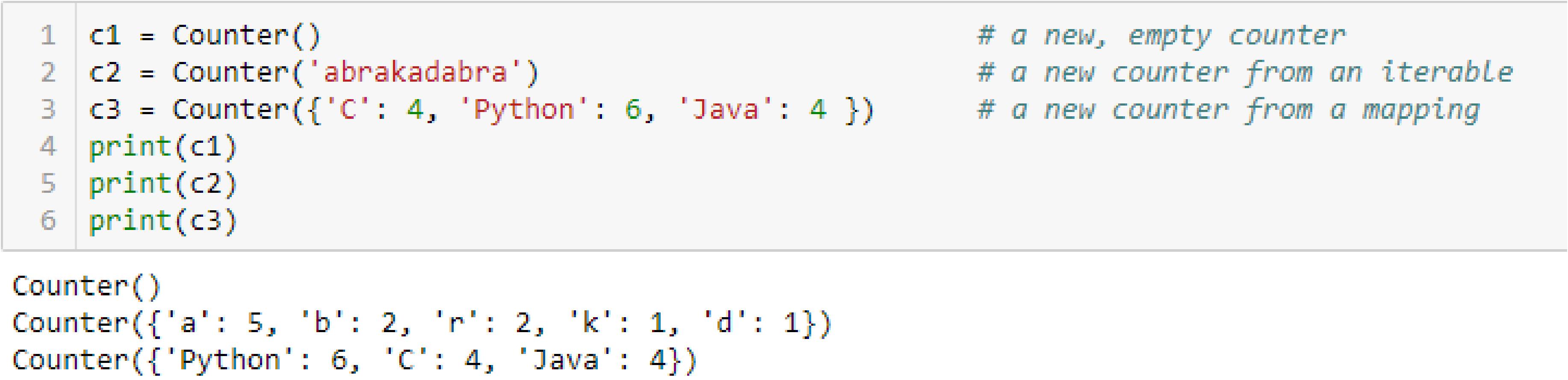
Collections-Counter

- 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

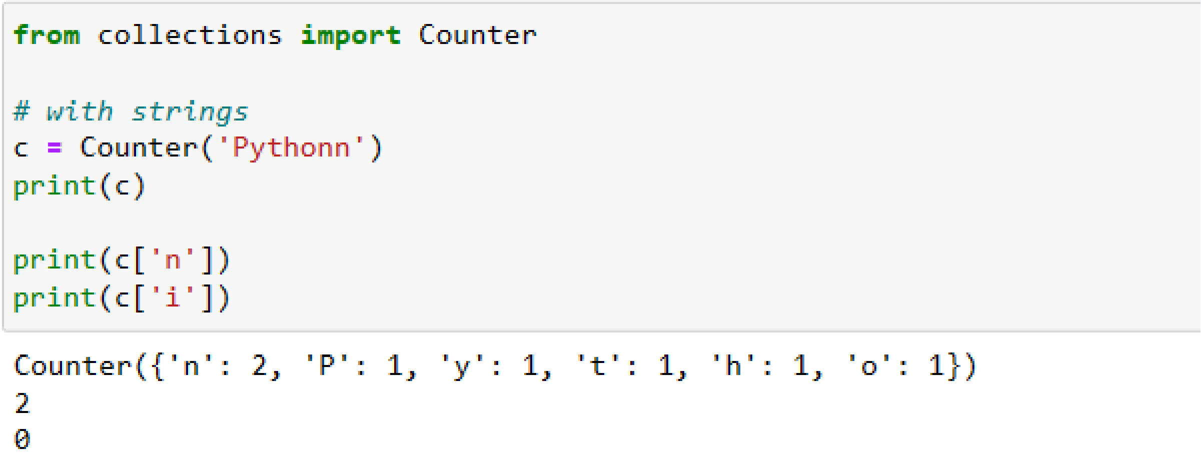
from collections import Counter

c = Counter(‘Pythonn')

print(c)

print(c['n'])

print(c['i'])

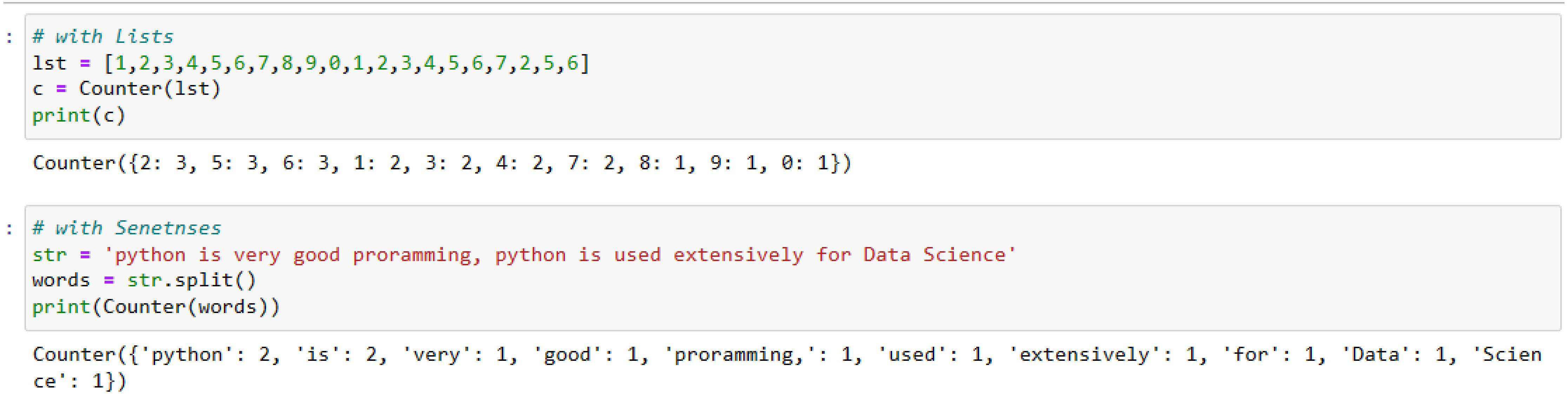


```
from collections import Counter

# 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))
```

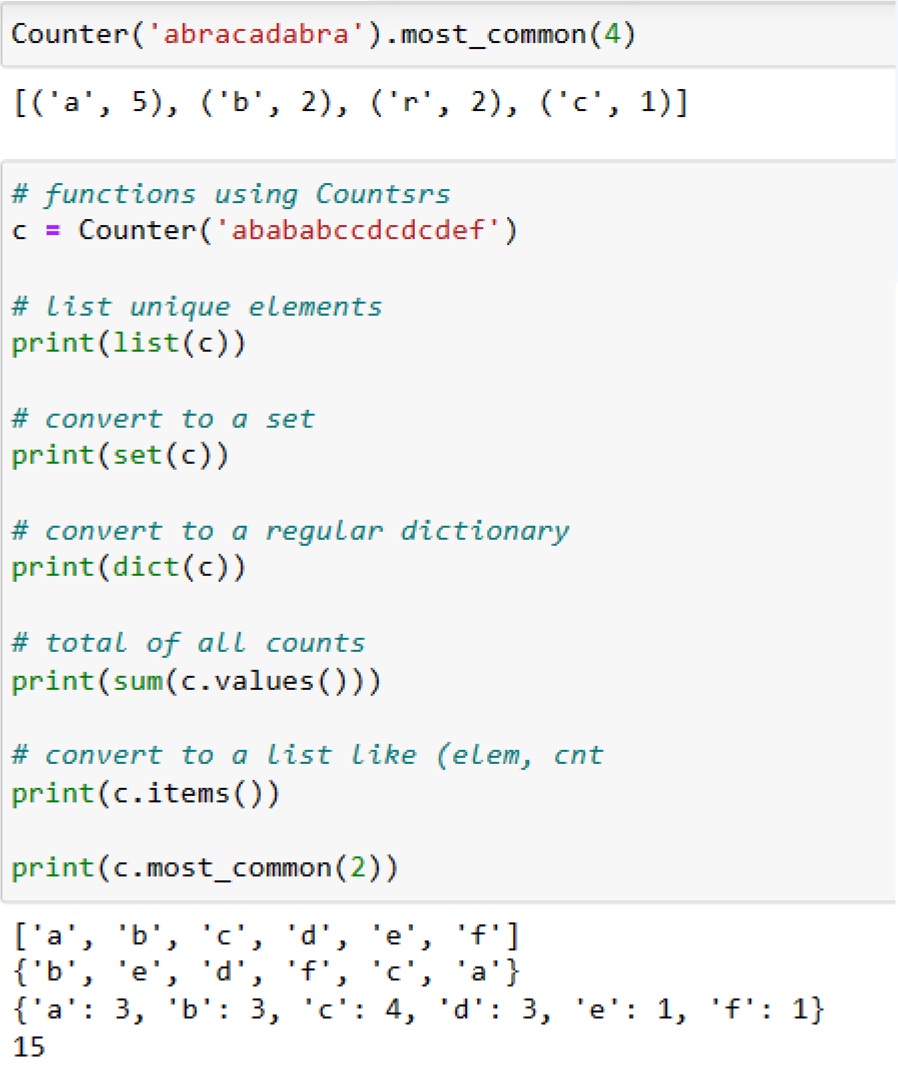
```
Counter({'python': 2, 'is': 2, 'very': 1, 'good': 1, 'proramming,': 1, 'used': 1, 'extensively': 1, 'for': 1, 'Data': 1, 'Scien
ce': 1})
```

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)
```

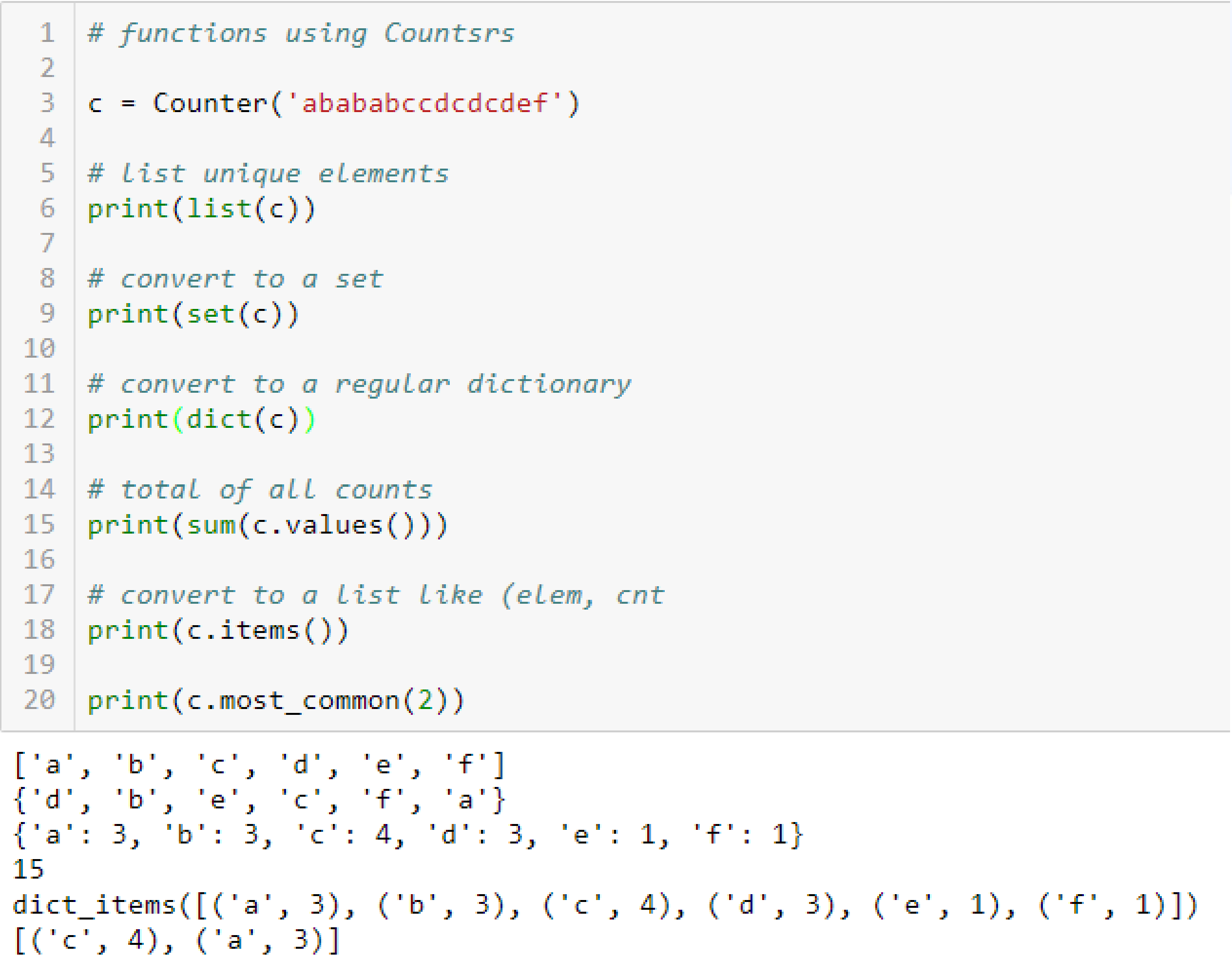
```
[('a', 5), ('b', 2), ('r', 2), ('c', 1)]
```

```
# functions using Countsrs
c = Counter('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']
{'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))
```

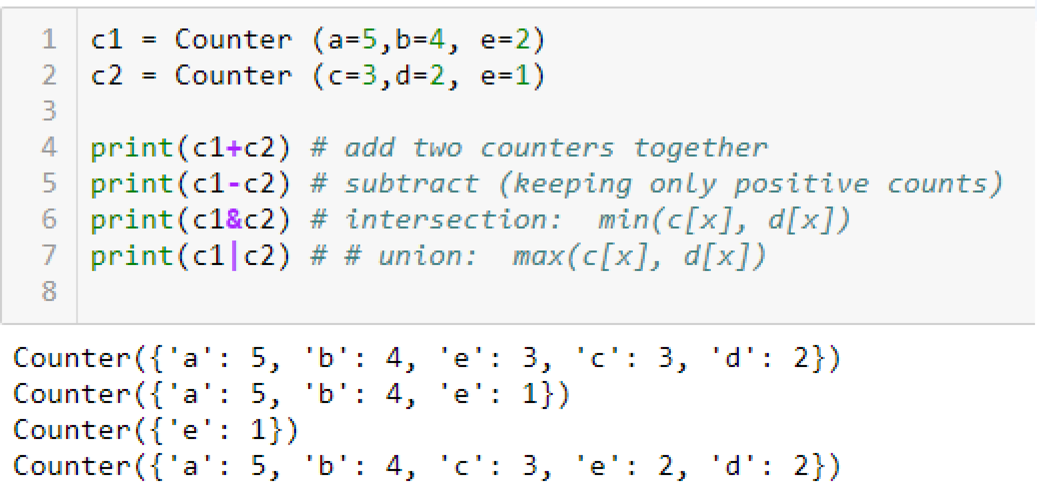


```
# functions using Countsrs

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])

```

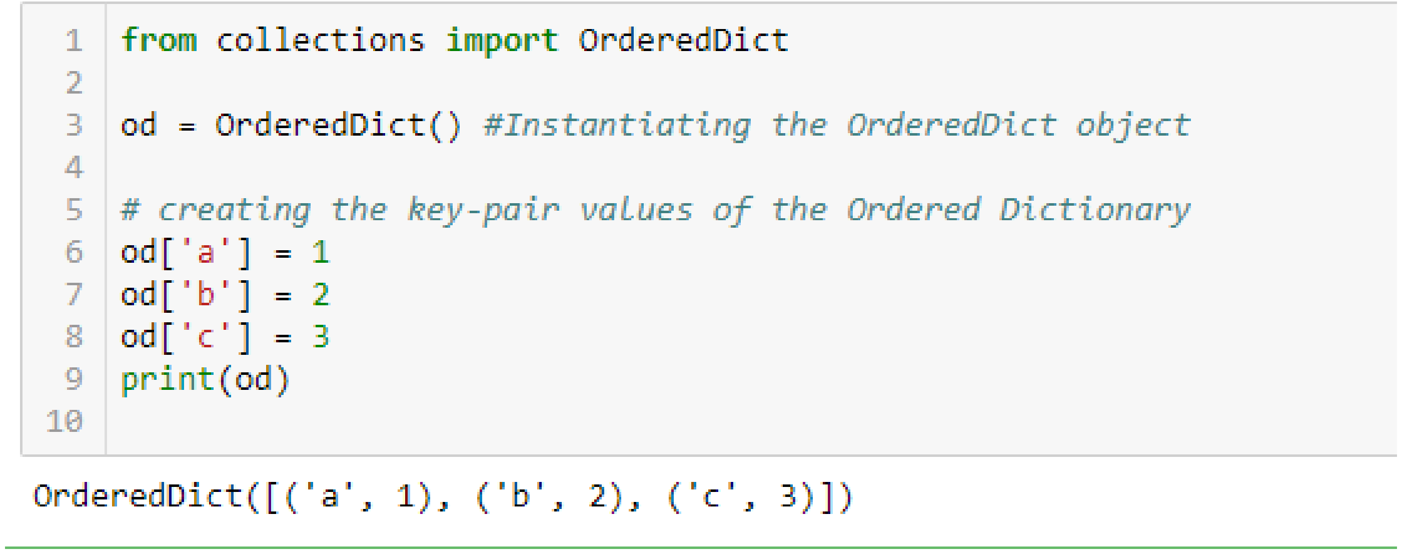
```
Counter({'a': 5, 'b': 4, 'e': 3, 'c': 3, 'd': 2})
Counter({'a': 5, 'b': 4, 'e': 1})
Counter({'e': 1})
Counter({'a': 5, 'b': 4, 'c': 3, 'e': 2, 'd': 2})
```

Collections-OrderedDict

- An *OrderedDict* is a dictionary subclass that **remembers the order in which that keys were first inserted.**
- When iterating over an ordered dictionary, the items are returned in the order their keys were first added.
- Since an ordered dictionary remembers its insertion order, it can be used in conjunction with sorting to make a sorted dictionary.

Creating a ordered dictionary

```
from collections import OrderedDict

od = OrderedDict()

# creating the key-pair values of the Ordered Dictionary

od['a'] = 1

od['b'] = 2

od['c'] = 3

print(od)
```

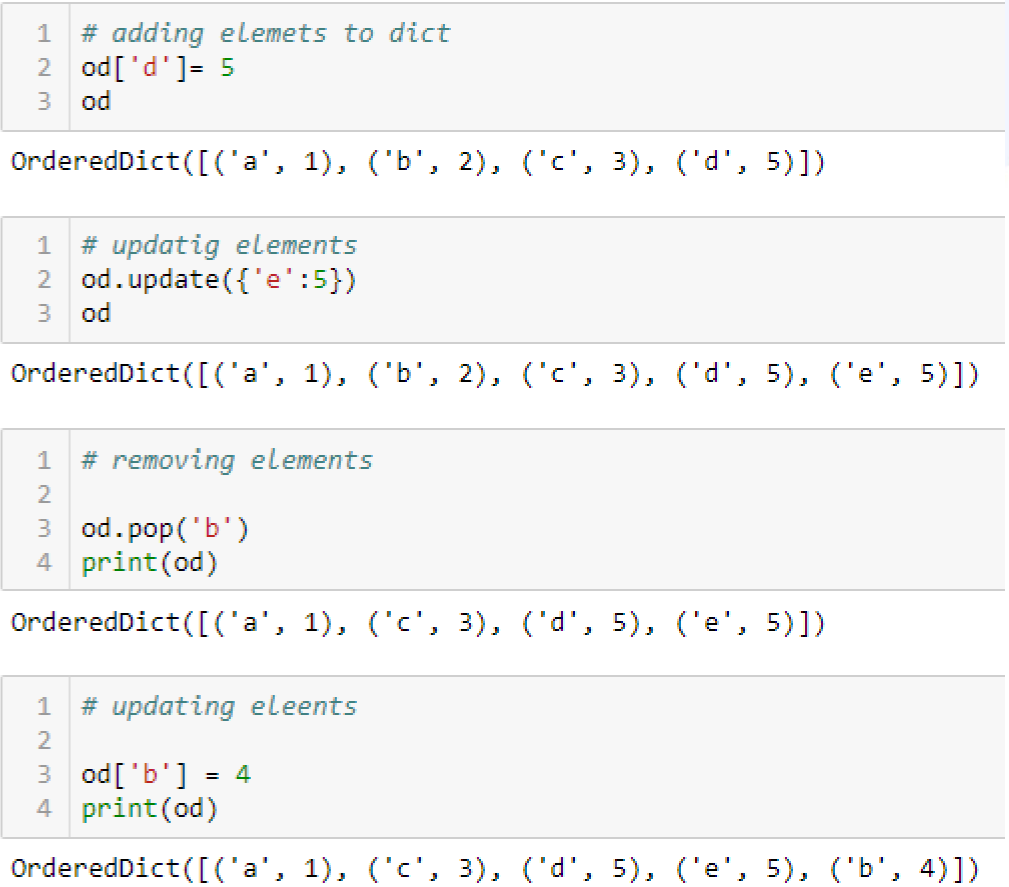


```
from collections import OrderedDict

od = OrderedDict() #Instantiating the OrderedDict object

# creating the key-pair values of the Ordered Dictionary
od['a'] = 1
od['b'] = 2
od['c'] = 3
print(od)

```

```
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

Adding, updating and removing elements

```
# OrderedDict([('a', 1), ('b', 2), ('c', 3)])
# adding elements to dict
od['d']= 5
print(od)

# updatig elements
od.update({'e':5})
print(od)

# removing elements
od.pop('b')
print(od)

# updating elements
od['b'] = 4
print(od)
```



```
# adding elemets to dict
od['d']= 5
od
```

```
OrderedDict([('a', 1), ('b', 2), ('c', 3), ('d', 5)])
```

```
# updatig elements
od.update({'e':5})
od
```

```
OrderedDict([('a', 1), ('b', 2), ('c', 3), ('d', 5), ('e', 5)])
```

```
# removing elements

od.pop('b')
print(od)
```

```
OrderedDict([('a', 1), ('c', 3), ('d', 5), ('e', 5)])
```

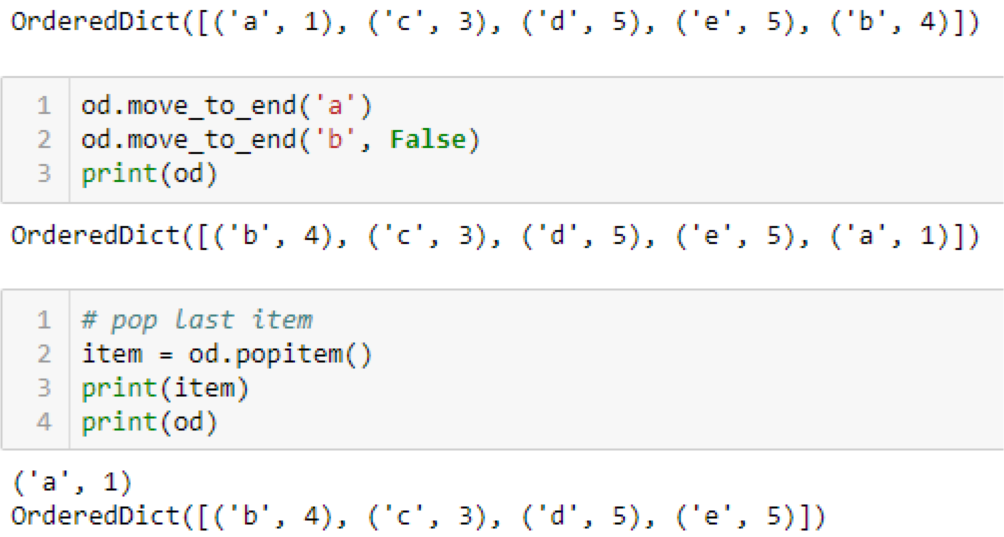
```
# updating eleents

od['b'] = 4
print(od)
```

```
OrderedDict([('a', 1), ('c', 3), ('d', 5), ('e', 5), ('b', 4)])
```

popitem() and move_to_end()

- The popitem() method for ordered dictionaries returns and removes a (key, value) pair.
- move_to_end() : Move an existing key to either end of an ordered dictionary.
- The item is moved to the right end if last is true (the default) or to the beginning if last is false.

```
od.move_to_end('a')
od.move_to_end('b', False)
print(od)    # OrderedDict([('b', 4), ('c', 3), ('d', 5), ('e', 5), ('a', 1)])

# pop last item
item = od.popitem()
print(item)   # ('a', 1)
print(od)      # OrderedDict([('b', 4), ('c', 3), ('d', 5), ('e', 5)])
```



```
OrderedDict([('a', 1), ('c', 3), ('d', 5), ('e', 5), ('b', 4)])
```

```
od.move_to_end('a')
od.move_to_end('b', False)
print(od)
```

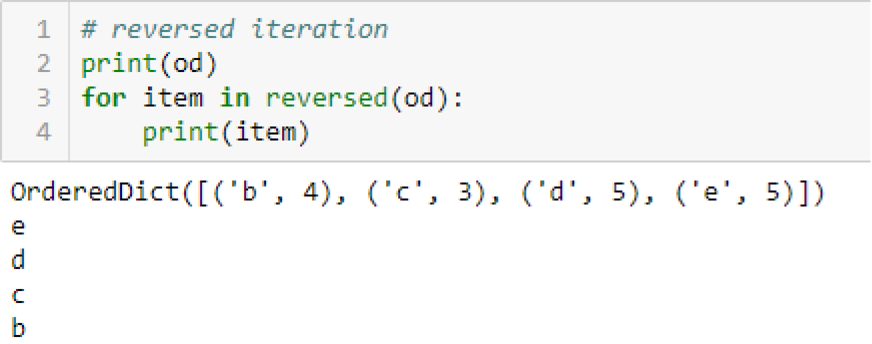
```
OrderedDict([('b', 4), ('c', 3), ('d', 5), ('e', 5), ('a', 1)])
```

```
# pop last item
item = od.popitem()
print(item)
print(od)
```

```
('a', 1)
OrderedDict([('b', 4), ('c', 3), ('d', 5), ('e', 5)])
```

Reverse ordered dictionary

```
# reversed iteration
print(od)
for item in reversed(od):
    print(item)
```



```
# reversed iteration
print(od)
for item in reversed(od):
    print(item)
```

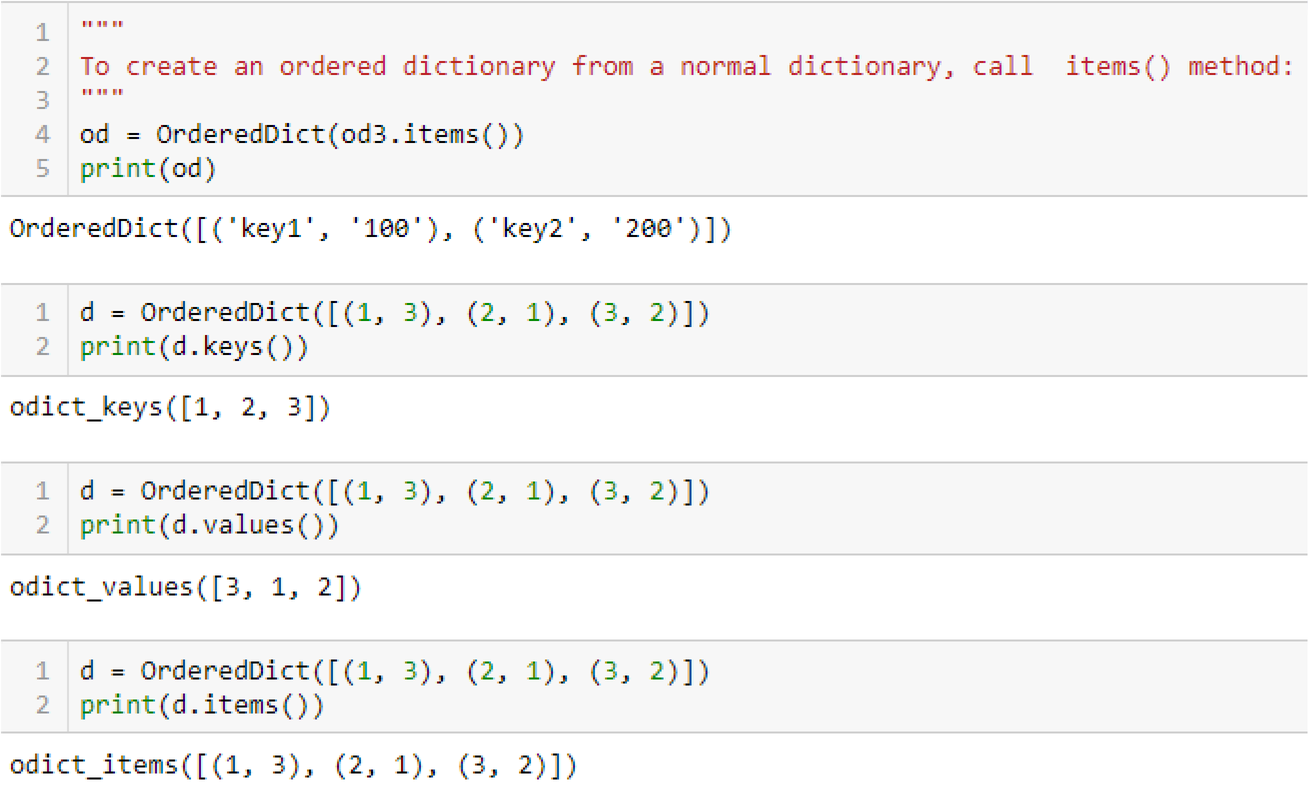
```
OrderedDict([('b', 4), ('c', 3), ('d', 5), ('e', 5)])
e
d
c
b
```

Basic operations on ordered dictionary

```
od = OrderedDict()
print(od)   # OrderedDict()


od['key1'] = '100'
od['key2'] = '200'
print(od)  # OrderedDict([('key1', '100'), ('key2', '200')])

od2 = OrderedDict(od)
print(od==od2) # True
```

Basic operations on ordered dictionary

```
"""
To create an ordered dictionary from a normal dictionary, call  items() method:
"""
od = OrderedDict(od3.items())
print(od)  # OrderedDict([('key1', '100'), ('key2', '200')])

d = OrderedDict([(1, 3), (2, 1), (3, 2)])
print(d.keys()) # odict_keys([1, 2, 3])

d = OrderedDict([(1, 3), (2, 1), (3, 2)])
print(d.values()) # odict_values([3, 1, 2])

d = OrderedDict([(1, 3), (2, 1), (3, 2)])
print(d.items()) # odict_items([(1, 3), (2, 1), (3, 2)])
```



```
"""
To create an ordered dictionary from a normal dictionary, call  items() method:
"""
od = OrderedDict(od3.items())
print(od)
```

```
OrderedDict([('key1', '100'), ('key2', '200')])
```

```
d = OrderedDict([(1, 3), (2, 1), (3, 2)])
print(d.keys())
```

```
odict_keys([1, 2, 3])
```

```
d = OrderedDict([(1, 3), (2, 1), (3, 2)])
print(d.values())
```

```
odict_values([3, 1, 2])
```

```
d = OrderedDict([(1, 3), (2, 1), (3, 2)])
print(d.items())
```

```
odict_items([(1, 3), (2, 1), (3, 2)])
```

Collections - defaultdict

- **defaultdict** is a subclass of the dictionary data structure that allows for default values if the key does not exist in the dictionary.
- The main difference between defaultdict and dict is that when you try to access or modify a key that’s not present in the dictionary, a default value is automatically given to that key.

Usage of defaultdict

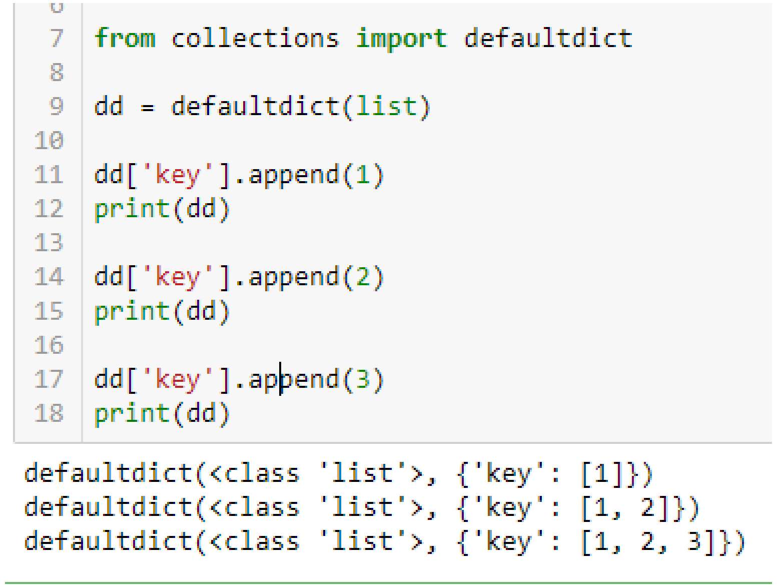
- Sometimes, we will use a mutable built-in collection (a list, dict, or set) as values in your Python dictionaries.
- In these cases, we will need to initialize the keys before first use, or you’ll get a KeyError.
- We can either do this process manually or automate it using a Python defaultdict.
- Python defaultdict type for solving some common programming problems:
    - Grouping the items in a collection
    - Counting the items in a collection
    - Accumulating the values in a collection

Grouping Items

- A typical use of the Python defaultdict type is to set .default_factory to list and then build a dictionary that maps keys to lists of values.

- Steps
  - Call list() to create a new empty list
  - Insert the empty list into the dictionary using the missing key as key
  - Return a reference to that list

```
from collections import defaultdict
dd = defaultdict(list)
dd['key'].append(1)
print(dd)
dd['key'].append(2)
print(dd)
dd['key'].append(3)
print(dd)
```

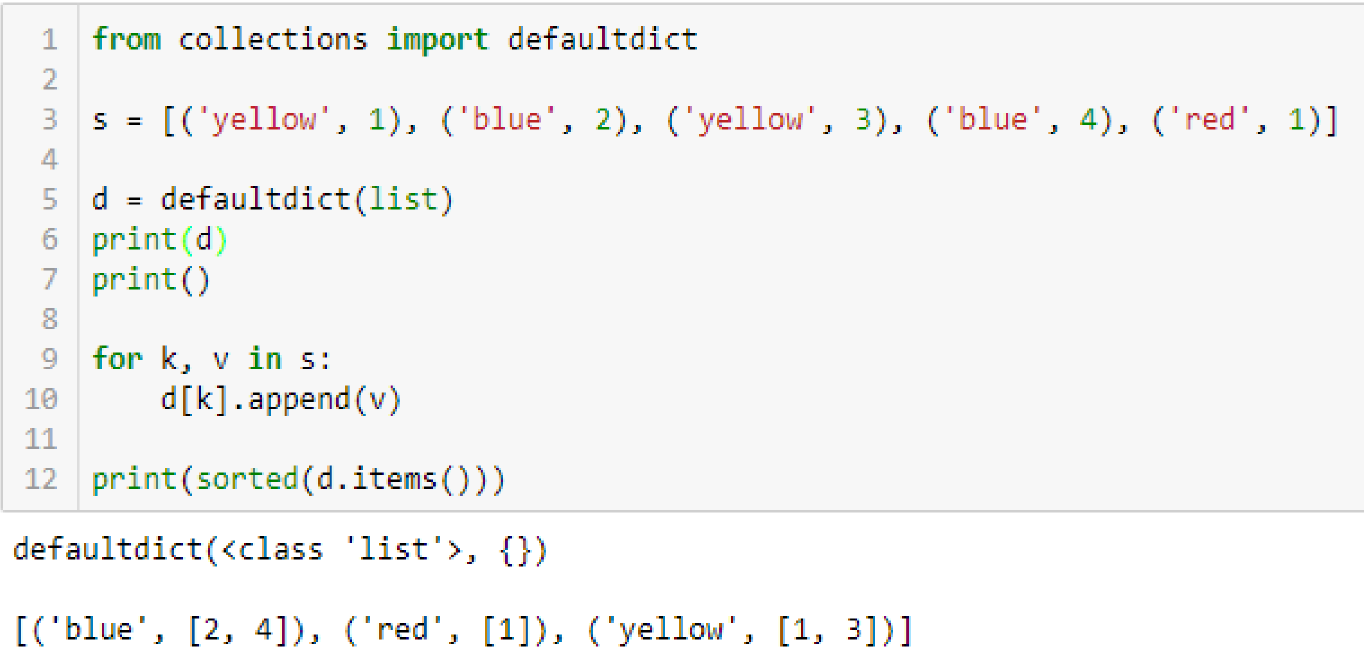


```
from collections import defaultdict

dd = defaultdict(list)

dd['key'].append(1)
print(dd)

dd['key'].append(2)
print(dd)

dd['key'].append(3)
print(dd)
```

```
defaultdict(<class 'list'>, {'key': [1]})
defaultdict(<class 'list'>, {'key': [1, 2]})
defaultdict(<class 'list'>, {'key': [1, 2, 3]})
```

Grouping a sequence of key-value pairs into a dictionary of lists using list as the default_factory

```
from collections import defaultdict

s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

d = defaultdict(list)
print()

for k, v in s:
    d[k].append(v)

print(sorted(d.items()))
# [('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```



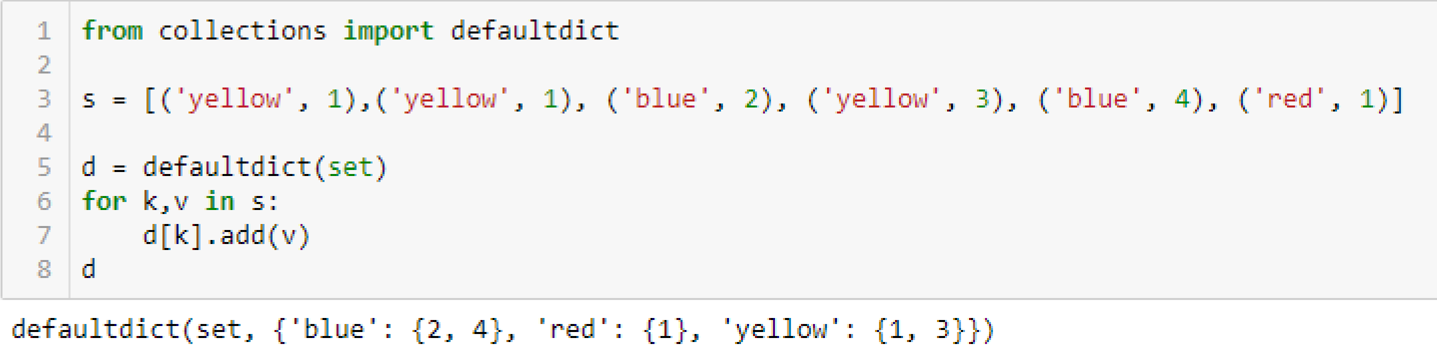
```
from collections import defaultdict

s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

d = defaultdict(list)
print(d)
print()

for k, v in s:
    d[k].append(v)

print(sorted(d.items()))
```

```
defaultdict(<class 'list'>, {})

[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Grouping Unique Items

- We use .default_factory to set. Sets are collections of unique objects, which means that we can’t create a set with repeated items.
- This is a really interesting feature of sets, which guarantees that you won’t have repeated items in our final dictionary.

```
from collections import defaultdict

s = [('yellow', 1),('yellow', 1), ('blue', 2), ('yellow', 3),
('blue', 4), ('red', 1)]

d = defaultdict(set)
for k,v in s:
    d[k].add(v)
Print(d)
```



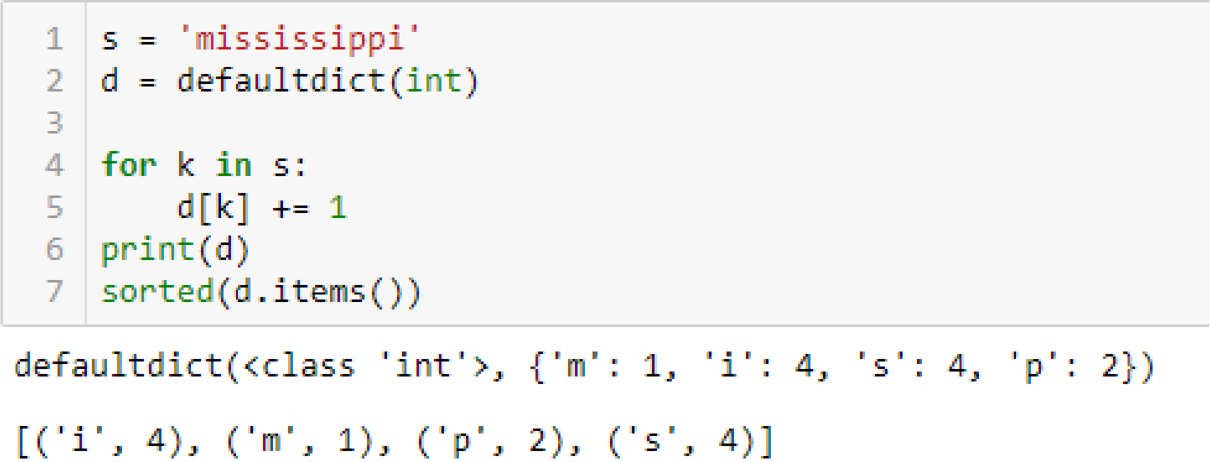
```
from collections import defaultdict

s = [('yellow', 1),('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

d = defaultdict(set)
for k,v in s:
    d[k].add(v)
d
```

```
defaultdict(set, {'blue': {2, 4}, 'red': {1}, 'yellow': {1, 3}})
```

Counting Items

- If we set .default_factory to int, then our defaultdict will be useful for counting the items in a sequence or collection.

```
s = 'mississippi'
d = defaultdict(int)

for k in s:
   d[k] += 1
print(d)
sorted(d.items())
```



```
s = 'mississippi'
d = defaultdict(int)

for k in s:
    d[k] += 1
print(d)
sorted(d.items())
```

```
defaultdict(<class 'int'>, {'m': 1, 'i': 4, 's': 4, 'p': 2})

[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

Collections-namedtuple

- Named tuple container datatype is an alternative to the built-in tuple.
- This extension type enhances standard tuples so that their elements can be **accessed by both their attribute name and the positional index.**
- Named tuples are available in Python’s standard **library collections module under the namedtuple utility.**
- The type accepts as **parameters the name of the typename and names** of **the fields associated with** it.
- The utility will then return a new tuple sub-class which is named with the given typename.

namedtuple

- A named tuple is an extension and custom data type that enrich built-in tuples with extra utilities.

- **They are very useful in context where we need to create a data structure that can be accessed by both the positional index and the named attribute of the elements**.

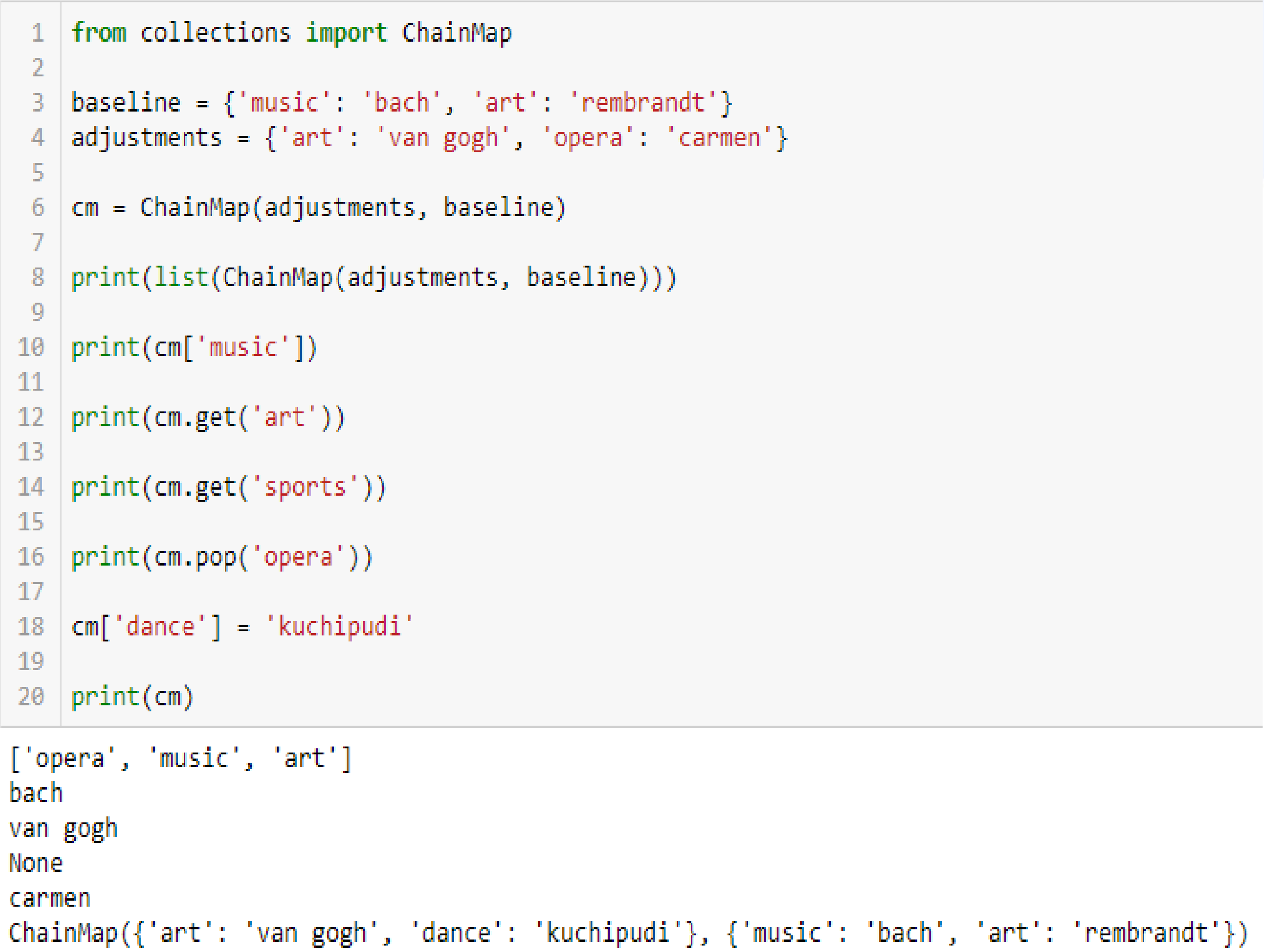
Collections - ChainMap

- A ChainMap is an updatable view over multiple dicts, and it behaves just like a normal dict.
- ChainMap combines a lot of dictionaries together and returns a list of dictionaries.
- ChainMaps basically encapsulates a lot of dictionaries into one single unit with no restriction on the number of dictionaries.

  syntax: ChainMap(dict1, dict2,…..)

Operations on ChainMap

```
from collections import ChainMap

baseline = {'music': 'bach', 'art': 'rembrandt'}
adjustments = {'art': 'van gogh', 'opera': 'carmen'}

cm = ChainMap(adjustments, baseline)

print(list(ChainMap(adjustments, baseline)))

print(cm['music'])

print(cm.get('art'))

print(cm.get('sports'))

print(cm.pop('opera'))

cm['dance'] = 'kuchipudi'

print(cm)
```



```
from collections import ChainMap

baseline = {'music': 'bach', 'art': 'rembrandt'}
adjustments = {'art': 'van gogh', 'opera': 'carmen'}

cm = ChainMap(adjustments, baseline)

print(list(ChainMap(adjustments, baseline)))

print(cm['music'])

print(cm.get('art'))

print(cm.get('sports'))

print(cm.pop('opera'))

cm['dance'] = 'kuchipudi'

print(cm)
```

```
['opera', 'music', 'art']
bach
van gogh
None
carmen
ChainMap({'art': 'van gogh', 'dance': 'kuchipudi'}, {'music': 'bach', 'art': 'rembrandt'})
```

Adding and reversing elements

- We can add a new dictionary at the beginning of a ChainMap using .new_child() method.
- The order in which dictionaries are stored in a ChainMap can be reversed using reversed() function.

```
# Creating a chainmap whose dictionaries do not have unique keys
dic1 = {'red':1,'white':4}
dic2 = {'red':9,'black':8}
chain = ChainMap(dic1,dic2)
print(list(chain.keys()))                    # ['black', 'red', 'white']

new_dic={'blue':10,'yellow':12}
chain=chain.new_child(new_dic)
print(chain)                                   # ChainMap({'blue': 10, 'yellow': 12}, {'red': 1, 'white': 4}, {'red': 9, 'black': 8})

chain.maps = reversed(chain.maps)
print('reversed Chainmap', str(chain))

# reversed Chainmap ChainMap({'red': 9, 'black': 8}, {'red': 1, 'white': 4}, {'blue': 10, 'yellow': 12})
```

Operations on ChainMap.

```
dic1 = {'red':1,'white':4}

dic2 = {'red':9,'black':8}

c = ChainMap(dic1,dic2)

print(c)  # ChainMap({'red': 1, 'white': 4}, {'red': 9, 'black': 8})

print(dict(c)) # {'black': 8, 'red': 1, 'white': 4}          # Flatten into a regular dictionary

print(c.items()) # ItemsView(ChainMap({'red': 1, 'white': 4}, {'red': 9, 'black': 8}))    #  All nested items

print(len(c)) # 3      # Number of nested values

print(list(c)) # ['black', 'red', 'white’]      # All nested values

print('red' in c) # True    # Check all nested values
```

Operations on ChainMap.

```
del c['red’]          # Delete from current context
print(c)    # ChainMap({'white': 4}, {'red': 9, 'black': 8})

c['red'] =1       # Set value in current context
print(c)     # ChainMap({'white': 4, 'red': 1}, {'red': 9, 'black': 8})

print(c['red'])  # 1             # Get first key in the chain of contexts

print(c.parents)  # ChainMap({'red': 9, 'black': 8})      # Enclosing context chain

print(c.maps[0])  # {'white': 4, 'red': 1}    # Current context dictionary

print(c.maps[-1]) # {'red': 9, 'black': 8}    # Root context
```