Collections Module

`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

deque

A list-like data structure with fast appends and pops on either end

- Deques are generalization of stacks and queues (the name is pronounced "deck" and is short for "double-ended queue").
- Deques support thread-safe, memory efficient appends and from either side of the deque with approximately the same O(1) performance in either direction.
- Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movements for pop(O) and insert(O, v) operations which change both the size and position of the underlying data representation.
- Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

https://docs.python.org/3/library/collections.html#collections.deque (https://docs.python.org/3/library/collections.html#collections.deque)

In [1]:
```python
# Queue using list

lst= []
def add_person(name):
    lst.append(name)
    print(f"{name} has been added to the queue")
    print(f"new list is {lst}\n")

def service_person():
    lst.pop(0)
    print("first person has been serviced")
    print(f"new list is {lst}\n")

def add_vip(name):
    lst.insert(0, name)
    print(f"{name} : vip is added to the queue")
    print(f"new list is {lst}\n")

add_person("A")
add_person("B")
add_person("C")
service_person()
add_vip("VIP")
```

```
A has been added to the queue
new list is ['A']

B has been added to the queue
new list is ['A', 'B']

C has been added to the queue
new list is ['A', 'B', 'C']

first person has been serviced
new list is ['B', 'C']

VIP : vip is added to the queue
new list is ['VIP', 'B', 'C']
```

https://stackoverflow.com/questions/14668769/does-python-have-built-in-linkedlist-data-

structure#:~:text=Yes%2C%20Python's%20collections%20module%20provides,list%20of%20BLOCK%20s%20internally.&text=See%2C
(https://stackoverflow.com/questions/14668769/does-python-have-built-in-linkedlist-data-
structure#:~:text=Yes%2C%20Python's%20collections%20module%20provides,list%20of%20BLOCK%20s%20internally.&text=See%2C

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one). To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends.

In [2]:
```python
from collections import deque

lst = deque()
def add_person(name):
    lst.append(name)
    print(f"{name} has been added to the queue")
    print(f"new list is {lst}\n")

def service_person():
    lst.popleft()
    print("first person has been serviced")
    print(f"new list is {lst}\n")

def add_vip(name):
    lst.appendleft(name)
    print(f"{name} has been added to the queue")
    print(f"new list is {lst}")

add_person("A")
add_person("B")
add_person("C")
service_person()
add_vip("VIP")
```

```
A has been added to the queue
new list is deque(['A'])

B has been added to the queue
new list is deque(['A', 'B'])

C has been added to the queue
new list is deque(['A', 'B', 'C'])

first person has been serviced
new list is deque(['B', 'C'])

VIP has been added to the queue
new list is deque(['VIP', 'B', 'C'])
```

Default dict

Usually, a Python dictionary throws a KeyError if you try to get an item with a key that is not currently in the dictionary.

The defaultdict in contrast will simply create any items that you try to access (provided of course they do not exist yet).

To create such a "default" item, it calls the function object that you pass to the constructsor (more precisely, it's an arbitrary "callable" object, which includes function and type objects).

defaultdict means that if a key is not found in the dictionary, then instead of a KeyError being thrown, a new entry is created. The type of this new entry is given by the argument of defaultdict.

The functionality of both dictionaries and defualtdict are almost same except for the fact that defualtdict never raises a KeyError. It provides a default value for the key that does not exists.

Syntax: defaultdict(default_factory)

default_factory is a function returning the default value for the dictionary defined. If this argument is absent then the dictionary raises a KeyError.

In [3]:
```
1 d = {1:"hi", 2:"bye"}
2 print(d[1])
3 print(d[3])
```

```
hi

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-3-b7bc426e108d> in <module>
      1 d = {1:"hi", 2:"bye"}
      2 print(d[1])
----> 3 print(d[3])

KeyError: 3
```

In [4]:
```
1 from collections import defaultdict
```

In [5]:
```
def def_value():
    return "Not Present"

d = defaultdict(def_value)
d["a"] = 1
d["b"] = 2

print(d["a"])
print(d["b"])
print(d["c"])
```

```
1
2
Not Present
```

In [6]:
```
# Default items are created using int(), which will return the integer object 0.
d = {1:"hi", 2:"bye"}

new = defaultdict(int, d)
print(new[3])
new
```

```
0
```

Out[6]:
```
defaultdict(int, {1: 'hi', 2: 'bye', 3: 0})
```

In [7]:
```
# Default items are created using list(), which returns a new empty list object.
d = {1: [10, 9, 8], 2:[12, 10, 15]}

new = defaultdict(list, d)
print(new[3])
new
```

```
[]
```

Out[7]:
```
defaultdict(list, {1: [10, 9, 8], 2: [12, 10, 15], 3: []})
```

In [8]:
```
def my_fun():
    return("Not Present")

d = {1:"hi", 2:"bye"}

new = defaultdict(my_fun, d)
print(new[3])
new
```

```
Not Present
```

Out[8]:
```
defaultdict(<function __main__.my_fun()>,
            {1: 'hi', 2: 'bye', 3: 'Not Present'})
```

In [9]:
```
# Example 1
# Input: s = 'mississippi'
# Output: {'m':1, 'i':4, 's':4, 'p':2}
```

In [10]:
```
# Using dictionary
s = 'mississippi'
r = {}
def count(s):
    for i in s:
        if i not in r:                          # Can also use r.keys() insted of r
            r[i] = 1
        else:
            r[i] += 1

count(s)
print(r)
```

```
{'m': 1, 'i': 4, 's': 4, 'p': 2}
```

In [11]:
```
# Using defaultdict
s = 'mississippi'
r = defaultdict(int)
def count(s):
    for i in s:
        r[i] += 1

count(s)
print(r)
```

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

In [12]:
```
# Example 2
# Input: s = [('red',1), ('blue',2), ('red',3), ('blue',5), ('yellow',1)]
# Output: {'red': [1, 3], 'blue': [2, 5], 'yellow': [1]}
```

In [13]:
```
# Using defaultdict
s = [('red',1), ('blue',2), ('red',3), ('blue',5), ('yellow',1)]

a = defaultdict(list)
for key, value in s:
    x = a[key]
    x.append(value)
#     a[key].append(value)

print(a)
```

```
defaultdict(<class 'list'>, {'red': [1, 3], 'blue': [2, 5], 'yellow': [1]})
```

Counter

class collections.Counter(iterable-or-mapping)

- A Counter is a dict sublass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values.
- Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a KeyError

In [14]:
```
from collections import Counter
```

In [15]:
```
# Converting list to counter
l = [1, 2, 1, 3, 4, 2, 2, 3, 4, 2, 3]
c = Counter(l)
print(c)
```

```
Counter({2: 4, 3: 3, 1: 2, 4: 2})
```

In [16]:
```
c[3], c[2]
```

Out[16]:
```
(3, 4)
```

In [17]:
```
# Converting dictionary to counter
d = {2: 4, 3: 3, 1: 2, 4: 2}
c = Counter(d)
c[5]
```

Out[17]:
```
0
```

In [18]:
```
d = {2: [1,2,3], 3: [1,3,3], 1: [1,5,7], 4: [2,5,6]}
c = Counter(d)
c[6]
```

Out[18]:
```
0
```

In [19]:
```
# Deleting key, same as dictionary
del c[4]
c
```

Out[19]:
```
Counter({2: [1, 2, 3], 3: [1, 3, 3], 1: [1, 5, 7]})
```

Counter Methods: Counter support 3 additional methods apart from available dictionary methods.

In [20]:
```
# elements(): Returns an iterator over elements repeating each as many times as its count. Eleme
# in arbitrary order. If an element's count is less than one, elements() will ignore it.

c = Counter(a=4, b=2, d=-5, g =7)
c.elements()
# list(c.elements())
```

Out[20]:
```
<itertools.chain at 0x7f79d97b7490>
```

In [21]:
```
# most_common(n): Returns a list of the n most commmon elements and their counts from the most c
# least. If n is omitted or None, most_common() returns all elernents in the counter.
# Elements with equal counts are ordered arbitrarily.

c = Counter('mississippiippimriver')
print(c)
c.most_common()       # give all count
# c.most_common(4)      # top 4
```

```
Counter({'i': 7, 's': 4, 'p': 4, 'm': 2, 'r': 2, 'v': 1, 'e': 1})
```

Out[21]:
```
[('i', 7), ('s', 4), ('p', 4), ('m', 2), ('r', 2), ('v', 1), ('e', 1)]
```

In [22]:
```
# Same problem using dictionary
s = 'mississippiippimriver'
d = {}
for char in s:
    if char not in d:                  # This case is omitted for defaultdict
        d[char] = 1
    else:
        d[char] += 1

new_d = sorted(d.items(), key=lambda x: x[1], reverse= True)
new_d

# Can you try this using defaultdict?
```

Out[22]:
```
[('i', 7), ('s', 4), ('p', 4), ('m', 2), ('r', 2), ('v', 1), ('e', 1)]
```

In [23]:
```
# subtract([iterable-or-mapping]): Elements are subtracted from an iterable or from another mapp
# counter). Like dict.update() but subtracts instead of replacing them. Both inputs and outputs
# or negative

c = Counter({2: 4, 3: 3, 1: 2, 4: 2})
d = Counter({2: 5, 1: 1, 4: 0, 3: 2})

c.subtract(d)
c
```

Out[23]:
```
Counter({2: -1, 3: 1, 1: 1, 4: 2})
```

OrderedDict

An OrderedDict is a dictionary subclass that remembers the order that keys were first inserted. The only difference between dict() and OrderedDict() is that:

OrderedDict preserves the order in which the keys are inserted. A regular dict doesn't track the insertion order, and iterating it gives the values in an arbitrary order. By contrast, the order the items are inserted is remembered by OrderedDict.

https://www.geeksforgeeks.org/ordereddict-in-python/ (https://www.geeksforgeeks.org/ordereddict-in-python/)

In [ ]:
```
1
```