Deque Module All Versions

Examples

Basic deque using

The main methods that are useful with this class are `popleft` and `appendleft`

```
from collections import deque

d = deque([1, 2, 3])
p = d.popleft()        # p = 1, d = deque([2, 3])
d.appendleft(5)        # d = deque([5, 2, 3])
```

Available methods in deque

Creating empty deque:

```
dl = deque()  # deque([]) creating empty deque
```

Creating deque with some elements:

```
dl = deque([1, 2, 3, 4])  # deque([1, 2, 3, 4])
```

Adding element to deque:

```
dl.append(5)  # deque([1, 2, 3, 4, 5])
```

Adding element left side of deque:

```
dl.appendleft(0)  # deque([0, 1, 2, 3, 4, 5])
```

Adding list of elements to deque:

```
dl.extend([6, 7])  # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

Adding list of elements to from the left side:

```
dl.extendleft([-2, -1])  # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

Using `.pop()` element will naturally remove an item from the right side:

```
dl.pop()  # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

Using `.popleft()` element to remove an item from the left side:

```
dl.popleft()  # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

Remove element by its value:

```
dl.remove(1)  # deque([-2, 0, 2, 3, 4, 5, 6])
```

Reverse the order of the elements in deque:

```
dl.reverse()  # deque([6, 5, 4, 3, 2, 0, -2])
```

limit deque size

Use the `maxlen` parameter while creating a deque to limit the size of the deque:

```
from collections import deque
d = deque(maxlen=3)  # only holds 3 items
d.append(1)  # deque([1])
d.append(2)  # deque([1, 2])
d.append(3)  # deque([1, 2, 3])
d.append(4)  # deque([2, 3, 4]) (1 is removed because its maxlen is 3)
```

```
d.popleft() # deque([2, 3, 4]) (2 is removed because it's maximum is 3)
```

Breadth First Search

The Deque is the only Python data structure with fast Queue operations . (Note `queue.Queue` isn't normally suitable, since it's meant for communication between threads.) A basic use case of a Queue is the breadth first search .

```
from collections import deque

def bfs(graph, root):
    distances = {}
    distances[root] = 0
    q = deque([root])
    while q:
        # The oldest seen (but not yet visited) node will be the left most one.
        current = q.popleft()
        for neighbor in graph[current]:
            if neighbor not in distances:
                distances[neighbor] = distances[current] + 1
                # When we see a new node, we add it to the right side of the queue.
                q.append(neighbor)
    return distances
```

Say we have a simple directed graph:

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

We can now find the distances from some starting position:

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

Syntax

```
dq = deque() # Creates an empty deque
```

```
dq = deque(iterable) # Creates a deque with some elements
```

```
dq.append(object) # Adds object to the right of the deque
```

```
dq.appendleft(object) # Adds object to the left of the deque
```

```
dq.pop() -> object # Removes and returns the right most object
```

```
dq.popleft() -> object # Removes and returns the left most object
```

```
dq.extend(iterable) # Adds some elements to the right of the deque
```

```
dq.extendleft(iterable) # Adds some elements to the left of the deque
```

Parameters

Parameter	Details
iterable	Creates the deque with initial elements copied from another iterable.
maxlen	Limits how large the deque can be, pushing out old elements as new are added.

Remarks

This class is useful when you need an object similar to a list that allows fast append and pop operations from either side (the name deque stands for " *double-ended queue* ").

The methods provided are indeed very similar, except that some like `pop`, `append`, or `extend` can be suffixed with `left`. The `deque` data structure should be preferred to a list if one needs to frequently insert and delete elements at both ends because it allows to do so in constant time O(1).