

덱

Deque

주요 내용

- 선형 자료구조
- 덱 자료구조
- 덱 활용

선형 자료구조

- 여러 개의 항목을 순서에 맞춰 추가하고 삭제할 수 있는 자료구조
- 항목의 추가/삭제 방식에 따라 서로 다르게 작동하며 일반적으로 다음 선형 자료구조가 활용됨.
 - 리스트 list
 - 덱 deque
 - 큐 queue
 - 스택 stack
 - 연결 리스트 linked list
 - 정렬 리스트 sorted list

머리와 꼬리

- 선형 자료구조의 양 끝인 머리와 꼬리를 부르는 이름 다를 수 있음.

양 끝 별칭

머리 헤드(head), 위(top), 앞(front)

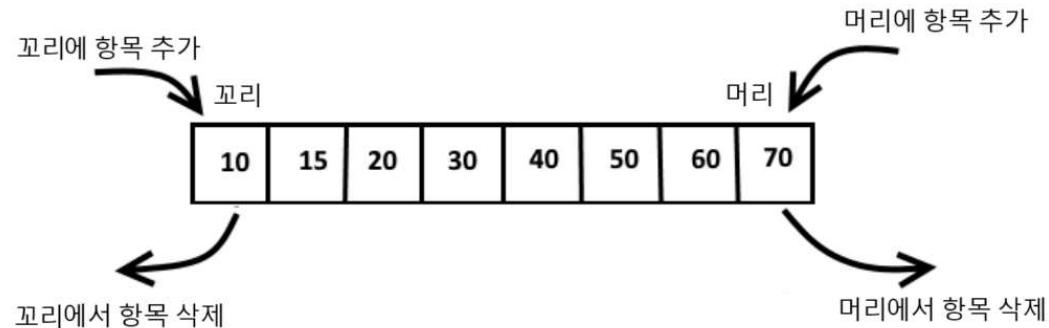
꼬리 테일(tail), 아래(bottom), 뒤(rear)

덱, 큐, 스택 자료구조 구현

- 파이썬은 큐, 스택, 덱 자료구조 이미 제공
 - 덱: [collections](#) 모듈의 `deque` 클래스
 - 큐와 스택: [queue](#) 모듈의 `Queue` 클래스와 `LifoQueue` 클래스
- 덱, 큐, 스택을 추상 자료형으로 선언한 뒤에 직접 파이썬 클래스로 구현해보기

덱 추상 자료형

- 덱 deque: double-ended queue의 줄임말.
- 항목의 빠른 추가와 삭제 지원
- 리스트와 유사한 자료구조이지만 다음 측면에서 다름
 - 항목 추가와 삭제가 기본적으로 머리 head와 꼬리 tail에서만 이루어짐.
 - 항목의 추가와 삭제가 리스트의 `append()`, `pop()` 메서드보다 빠름



Deque 추상 클래스

정의	기능
Deque	덱 추상 클래스
append()	머리에 새로운 항목 추가. 반환값 없음.
appendleft()	꼬리에 새로운 항목 추가. 반환값 없음.
pop()	머리 항목 삭제. 삭제된 항목 반환
popleft()	꼬리 항목 삭제. 삭제된 항목 반환.

Deque 추상 클래스 구현

```
In [1]: class Deque:  
    def __init__(self, items=[]):  
        self._items = items  
    def __len__(self):  
        return len(self._items)  
    def append(self, item):  
        self._items.append(item)  
    def appendleft(self, item):  
        raise NotImplementedError  
    def pop(self):  
        return self._items.pop()  
    def popleft(self):  
        raise NotImplementedError
```

추상 클래스 활용의 한계

In [2]:

```
deque_adt = Deque()
deque_adt.appendleft("덱 항목")
```

```
-----
NotImplementedError                                     Traceback (most recent call last)
          t)
Cell In[2], line 2
    1 deque_adt = Deque()
----> 2 deque_adt.appendleft("덱 항목")

Cell In[1], line 9, in Deque.appendleft(self, item)
    8     def appendleft(self, item):
----> 9         ... raise NotImplementedError

NotImplementedError:
```

구상 클래스

- 추상 클래스의 인스턴스는 제대로 활용할 수 없음
- 구상 클래스_{concrete class}
 - 포함된 모든 메서드의 본문이 제대로 구현된 클래스
 - 즉, 추상 자료형을 자료구조로 제대로 활용할 수 있도록 구현된 클래스

덱 자료구조 구현

```
In [3]: class myDeque(Deque):
    def __init__(self, items=[]):
        super().__init__(items)

    def __repr__(self):
        return f"deque({self._items})"

    def appendleft(self, item):
        self._items.insert(0, item)

    def popleft(self):
        return self._items.pop(0)
```

덱 객체 기초 활용법

```
In [4]: d=myDeque([])
d.append(4)
d.appendleft('dog')
d.append('cat')
d.append(True)
print(d)
d.appendleft(8.4)
print(d.popleft())
print(d.pop())
print(d)
```

```
deque(['dog', 4, 'cat', True])
```

```
8.4
```

```
True
```

```
deque(['dog', 4, 'cat'])
```

`collections.deque` 클래스

- `collections` 모듈이 덱 추상 자료형을 자료구조로 구현한 `deque` 클래스를 지원
- `myDeque` 보다 항목의 추가와 삭제가 빠름.
- `appendleft()`는 20 배 정도, `popleft()`는 2천배 이상

append() 메서드 속도 비교

```
In [5]: from collections import deque
```

```
In [6]: %%time
n = 100_000
dec1 = myDeque([])
for k in range(n):
    dec1.append(k)
```

```
CPU times: user 7.84 ms, sys: 0 ns, total: 7.84 ms
Wall time: 7.83 ms
```

```
In [7]: %%time
n = 100_000
dec1 = deque([])
for k in range(n):
    dec1.append(k)
```

```
CPU times: user 4.96 ms, sys: 0 ns, total: 4.96 ms
Wall time: 4.93 ms
```

appendleft() 메서드 속도 비교

In [8]:

```
%%time
n = 100_000
dec1 = myDeque([])
for k in range(n):
    dec1.appendleft(k)
```

```
CPU times: user 801 ms, sys: 1.65 ms, total: 803 ms
Wall time: 803 ms
```

In [9]:

```
%%time
n = 100_000
dec1 = deque([])
for k in range(n):
    dec1.appendleft(k)
```

```
CPU times: user 5.35 ms, sys: 0 ns, total: 5.35 ms
Wall time: 5.25 ms
```

pop() 메서드 속도 비교

In [10]:

```
%%time
n = 100_000
dec1 = myDeque(list(range(n)))
for _ in range(n):
    dec1.pop()
```

```
CPU times: user 8.08 ms, sys: 0 ns, total: 8.08 ms
Wall time: 8 ms
```

In [11]:

```
%%time
n = 100_000
dec1 = deque(list(range(n)))
for _ in range(n):
    dec1.pop()
```

```
CPU times: user 5.1 ms, sys: 0 ns, total: 5.1 ms
Wall time: 5.03 ms
```

popleft() 메서드 속도 비교

In [12]:

```
%%time
n = 100_000
dec1 = myDeque(list(range(n)))
for _ in range(n):
    dec1.popleft()
```

```
CPU times: user 484 ms, sys: 319 µs, total: 484 ms
Wall time: 484 ms
```

In [13]:

```
%%time
n = 100_000
dec1 = deque(list(range(n)))
for _ in range(n):
    dec1.popleft()
```

```
CPU times: user 4.94 ms, sys: 0 ns, total: 4.94 ms
Wall time: 4.89 ms
```

myDeque 와 collections.deque 의 차이

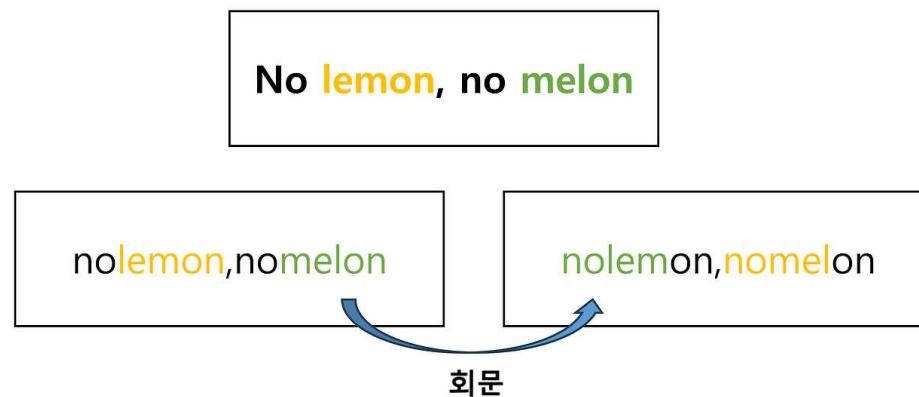
- myDeque 클래스의 appendleft() 와 popleft() 는 리스트의 0번 인덱스에서 항목을 추가하거나 삭제함.
- 이를 위해 사용되는 리스트의 insert(0, item) 와 pop(0) 은 실행시간이 리스트의 길이에 의존함.
- 반면에 collections.deque 클래스는 항목을 저장할 때 리스트가 아닌 다른 방식으로 항목을 저장하며,

머리와 꼬리에서 항목을 저장하고 삭제하는 일이 리스트의 길이에 무관하게 일정한 시간이 걸림.

- 하지만 collections.deque 의 메서드가 항상 리스트의 메서드보다 빠른 것은 아니기에 필요에 따라 리스트 또는 collections.deque 중에 하나의 자료구조를 선택해야 함.

덱 활용: 회문 판별기

- 회문 palindrome: 앞으로 읽어도, 뒤로 읽어도 동일한 단어 또는 문장
 - 한글 회문 예제: '기러기', '우영우', '토마토', "인싸 의사의 싸인", "여보게, 저기 저게 보여?" 등등
 - 영어 회문 예제: 'mom', 'radar', 'Yo! Banana boy.', 'I did, did I?', 'No lemon, no melon' 등등

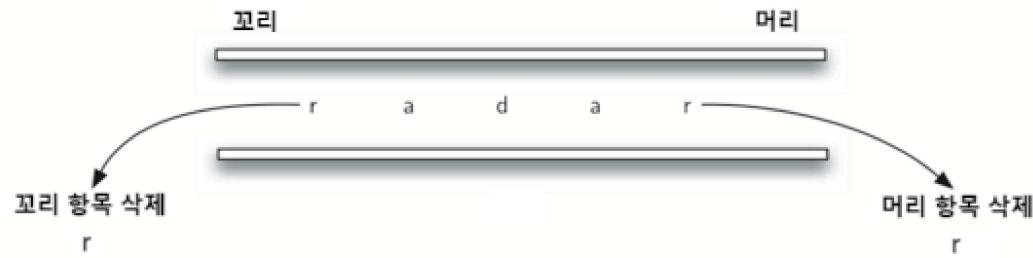
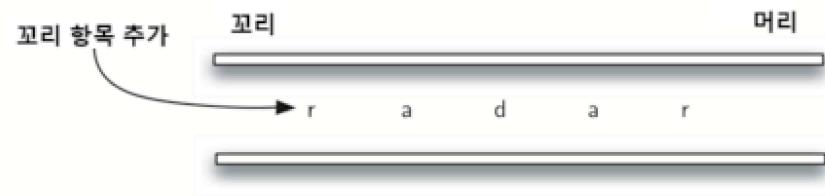


회문 판별기 구현

```
In [14]: def pal_checker(text):
    # 덱 객체 생성
    char_deque = myDeque([])
    for ch in text:
        char_deque.appendleft(ch)

    # 머리와 꼬리 항목 비교
    while len(char_deque) > 1:
        first = char_deque.pop()
        last = char_deque.popleft()
        # 대칭 문자가 다를 경우 회문 아님 판정
        if first != last:
            return False

    # 이전 반복문을 무사히 통과하면 회문 판정
    return True
```



```
In [15]: assert pal_checker("기러기") == True  
assert pal_checker("사이다") == False  
assert pal_checker("radar") == True  
assert pal_checker("tomato") == False
```

```
In [16]: assert pal_checker("Bob") == True
        assert pal_checker("인싸 의사의 싸인") == True
        assert pal_checker("여보게, 저기 저게 보여?") == True
        assert pal_checker('I did, did I?') == True
        assert pal_checker('No lemon, no melon') == True
```

```
-----
-->AssertionError: Traceback (most recent call last)
      Cell In[16], line 1
      ----> 1 assert pal_checker("Bob") == True
            2 assert pal_checker("인싸 의사의 싸인") == True
            3 assert pal_checker("여보게, 저기 저게 보여?") == True

AssertionError:
```