

Data Structures in Python

Chapter 3

- Stack Concept and ADT
- Stack Example - Matching
- Stack Example - Postfix
- Queue
- Deque
- **Deque Profiling**
- Circular Queue
- Linked list
- Unordered List
- Ordered List and Iterator

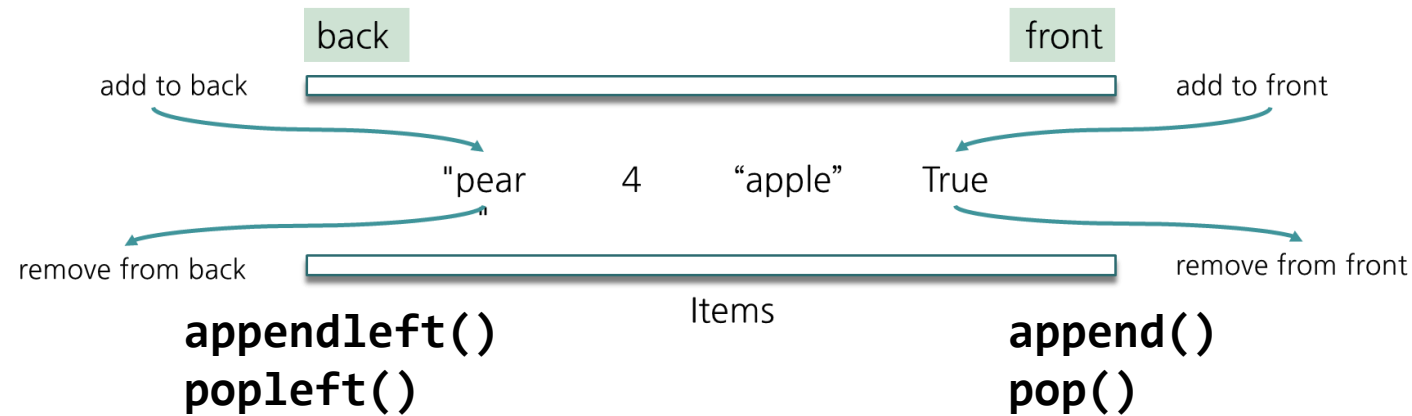
Agenda

- Deque Profiling
 - **list insert(0, i) vs deque appendleft(i)**
 - Using perf_counter()
 - Lambda Function
 - Profiling

Time Complexity: deque vs list

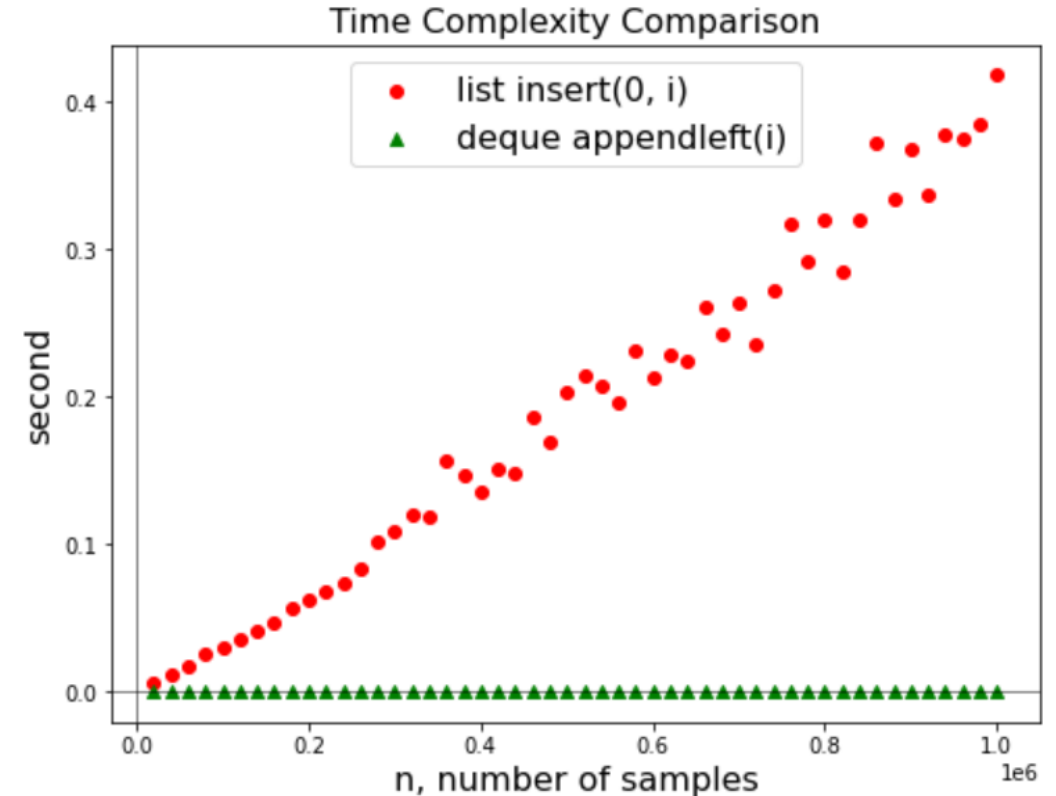
- Deque - Double Ended Queue (pronounced like 'deck')

Operation	deque	list
Pop and append items on the left end	$O(1)$	$O(n)$
Pop and append items on the right end	$O(1)$	$O(1)$ + reallocation
Insert and delete items in the middle	$O(n)$	$O(n)$
Access arbitrary items through indexing	$O(n)$	$O(1)$



Deque Profiling

- Performance Analysis - Time Complexity
 - `list`의 `insert(0, i)`, `pop(0)` - $O(n)$
 - `deque`의 `appendleft(i)`, `popleft()`, `append()`, `pop()` - $O(1)$
- Using `perf_counter()` in Python
 - works like a stopwatch.
 - returns the float value of time in seconds.



Deque Profiling: Step 1

- Using built-in performance counter, **perf_counter()**, in Python:

```
#!/usr/bin/env python
from collections import deque
from time import perf_counter # performance counter

NSIZE = 100_000
TIMES = 100_000
alist = list(range(NSIZE))
t_start = perf_counter()
for i in range(TIMES):
    alist.insert(0, i)
list_time = perf_counter() - t_start
print(f"list.insert() {list_time:>12.6} sec")

adeq = deque(range(NSIZE))
t_start = perf_counter()
for i in range(TIMES):
    adeq.appendleft(i)
deq_time = perf_counter() - t_start
print(f"deque.appendleft() {deq_time:>12.6} sec")

ratio = list_time / deq_time
print(f"list/deque ratio {ratio:>12.6} x faster")
```

```
PS C:\GitHub\DSpyx\jupyter> python perf_counter1.py
list.insert()          2.52245 sec
deque.appendleft()     0.0058442 sec
list/deque ratio       431.616 x faster
PS C:\GitHub\DSpyx\jupyter> █
```

Deque Profiling: Step 1

- Using built-in performance counter, **perf_counter()**, in Python:

```
#!/usr/bin/env python
from collections import deque
from time import perf_counter # performance counter
```

```
NSIZE = 100_000
TIMES = 100_000
```

```
alist = list(range(NSIZE))
t_start = perf_counter()
for i in range(TIMES):
    alist.insert(0, i)
list_time = perf_counter() - t_start
print(f"list.insert()      {list_time:>12.6} sec")
```

```
adeq = deque(range(NSIZE))
t_start = perf_counter()
for i in range(TIMES):
    adeq.appendleft(i)
deq_time = perf_counter() - t_start
print(f"deque.appendleft() {deq_time:>12.6} sec")
```

```
ratio = list_time / deq_time
print(f"  list/deque ratio {ratio:>12.6} x faster")
```

```
PS C:\GitHub\DSpyx\jupyter> python perf_counter1.py
list.insert()      2.52245 sec
deque.appendleft() 0.0058442 sec
  list/deque ratio  431.616 x faster
PS C:\GitHub\DSpyx\jupyter> █
```

Deque Profiling: Step 1

- Using built-in performance counter, **perf_counter()**, in Python:

```
#!/usr/bin/env python
from collections import deque
from time import perf_counter # performance counter

NSIZE = 100_000
TIMES = 100_000

alist = list(range(NSIZE))
t_start = perf_counter()
for i in range(TIMES):
    alist.insert(0, i)
list_time = perf_counter() - t_start
print(f"list.insert()      {list_time:>12.6} sec")
```

```
adeq = deque(range(NSIZE))
t_start = perf_counter()
for i in range(TIMES):
    adeq.appendleft(i)
deq_time = perf_counter() - t_start
print(f"deque.appendleft() {deq_time:>12.6} sec")

ratio = list_time / deq_time
print(f"  list/deque ratio {ratio:>12.6} x faster")
```

These two code snippets are almost identical.
It violates one of coding principles:

```
PS C:\GitHub\DSpyx\jupyter> python perf_counter1.py
list.insert()      2.52245 sec
deque.appendleft() 0.0058442 sec
  list/deque ratio  431.616 x faster
PS C:\GitHub\DSpyx\jupyter>
```

Deque Profiling: Step 2

- Let us make them as functions like `list_timing()` and `deq_timing()`.

```
#!/usr/bin/env python
# %%writefile perf_counter1.py
from collections import deque
from time import perf_counter # performance counter

NSIZE = 100_000
TIMES = 100_000

alist = list(range(NSIZE))
t_start = perf_counter()
for i in range(TIMES):
    alist.insert(0, i)
list_time = perf_counter() - t_start
print(f"list.insert()      {list_time:>12.6} sec")
```

Let us rewrite each part as a function,
`list_timing()`, that returns **`list_time`** and
`deq_timing()` that returns **`deq_time`**.



```
#!/usr/bin/env python
# %%writefile perf_counter2.py
```

```
def list_timing(NSIZE, TIMES):
    alist = list(range(NSIZE))
    start = perf_counter()
    for i in range(TIMES):
        alist.insert(0, i)
    return perf_counter() - start

def deq_timing(NSIZE, TIMES):
    adeq = deque(range(NSIZE))
    start = perf_counter()
    for i in range(TIMES):
        adeq.appendleft(i),
    return perf_counter() - start
```


Deque Profiling: Step 2

- Let us make them as functions like `list_timing()` and `deq_timing()`.

```
#!/usr/bin/env python
from collections import deque
from time import perf_counter # performance counter

def list_timing(NSIZE, TIMES):
    alist = list(range(NSIZE))
    start = perf_counter()
    for i in range(TIMES):
        alist.insert(0, i)
    return perf_counter() - start

def deq_timing(NSIZE, TIMES):
    adeq = deque(range(NSIZE))
    start = perf_counter()
    for i in range(TIMES):
        adeq.appendleft(i)
    return perf_counter() - start
```

```
if __name__ == '__main__':
    NSIZE = 100_000
    TIMES = 100_000
    list_time = list_timing(NSIZE, TIMES)
    print(f"list.insert() {list_time:>12.6} sec")
    deq_time = deq_timing(NSIZE, TIMES)
    print(f"deque.appendleft() {deq_time:>12.6} sec")
    ratio = list_time / deq_time
    print(f"list/deque ratio {ratio:>12.6} x faster")
```

```
PS C:\GitHub\DSpyx\jupyter> python perf_counter2.py
list.insert()          2.49762 sec
deque.appendleft()     0.0051234 sec
list/deque ratio       487.492 x faster
PS C:\GitHub\DSpyx\jupyter>
```

Deque Profiling: Step 2

- Let us make them as functions like `list_timing()` and `deq_timing()`.

```
#!/usr/bin/env python
from collections import deque
from time import perf_counter # performance counter

def list_timing(NSIZE, TIMES):
    alist = list(range(NSIZE))
    start = perf_counter()
    for i in range(TIMES):
        alist.insert(0, i)
    return perf_counter() - start

def deq_timing(NSIZE, TIMES):
    adeq = deque(range(NSIZE))
    start = perf_counter()
    for i in range(TIMES):
        adeq.appendleft(i)
    return perf_counter() - start
```

These two functions are almost identical.
It still violates one of coding principles:

Deque Profiling: Step 3

- Combine two `~_timing()` functions to `perf_timeit()`. Make a helper function `elapsed()` that accepts `list insert()` and `deque appendleft()` as an argument.

```
#!/usr/bin/env python
from collections import deque
from time import perf_counter

def list_timing(NSIZE, TIMES):
    alist = list(range(NSIZE))
    start = perf_counter()
    for i in range(TIMES):
        alist.insert(0, i)
    return perf_counter() - start

def deq_timing(NSIZE, TIMES):
    adeq = deque(range(NSIZE))
    start = perf_counter()
    for i in range(TIMES):
        adeq.appendleft(i)
    return perf_counter() - start
```



```
#!/usr/bin/env python
from collections import deque
from time import perf_counter

def perf_timeit(NSIZE, TIMES):
    alist = list(range(NSIZE))
    adeq = deque(range(NSIZE))

    def elapsed(func) # a helper function
        start = perf_counter()
        for i in range(TIMES):
            func(i)
        return perf_counter() - start

    list_time = elapsed(... alist.insert(0, i))
    deq_time = elapsed(... adeq.appendleft(i))
    return list_time, deq_time
```

Pass this as an argument to the function `elapsed()`.

Deque Profiling: Step 4

- Let us learn about lambda function or define an anonymous function in one line.
- Syntax: **lambda argument(s): expression**

■ Example:

```
def times2plus(x, y):          # regular function
    return x * 2 + y
lambda x, y: x * 2 + y        # lambda function
```

```
(lambda x, y: x * 2 + y)(1, 2) # returns 4
```

```
func = lambda x, y: x * 2 + y
func(3,4)                          # returns 10
```

```
>>> (lambda x, y: x * 2 + y)(1, 2)
4
>>> func = lambda x, y: x * 2 + y
>>> func(3,4)
10
>>>
```

Deque Profiling: Step 4

- Let us learn about lambda function or define an anonymous function in one line.
- Syntax: **lambda argument(s): expression**

■ Example:

```
def times2plus(x, y):           # regular function
    return x * 2 + y
lambda x, y: x * 2 + y         # lambda function
```

```
(lambda x, y: x * 2 + y)(1, 2) # returns 4
```

```
func = lambda x, y: x * 2 + y
func(3,4)                        # returns 10
```

```
>>> (lambda x, y: x * 2 + y)(1, 2)
4
>>> func = lambda x, y: x * 2 + y
>>> func(3,4)
10
>>>
```

■ Example:

```
>>> list(map(lambda x: x.capitalize(), ['cat', 'dog', 'cow']))
['Cat', 'Dog', 'Cow']
```

```
>>> [x.capitalize() for x in ['cat', 'dog', 'cow']]
```

Deque Profiling: Step 4

- Let us learn about lambda function or define an anonymous function in one line.
- Syntax: **lambda argument(s): expression**

Example:

```
def times2plus(x, y):           # regular function
    return x * 2 + y
lambda x, y: x * 2 + y         # lambda function
```

```
(lambda x, y: x * 2 + y)(1, 2) # returns 4
```

```
func = lambda x, y: x * 2 + y
func(3,4)                               # returns 10
```

```
>>> (lambda x, y: x * 2 + y)(1, 2)
4
>>> func = lambda x, y: x * 2 + y
>>> func(3,4)
10
>>>
```

Example:

```
>>> list(map(lambda x: x.capitalize(), ['cat', 'dog', 'cow']))
['Cat', 'Dog', 'Cow']
```

```
>>> [x.capitalize() for x in ['cat', 'dog', 'cow']]
```

Example:

```
>>> from timeit import timeit
>>> from math import factorial
>>> timeit("factorial(999)", "from math import factorial", number=10)
0.0013087529951008037
```

← setup

```
>>> timeit(lambda: factorial(999), number=10)
```

Deque Profiling: Step 5

A performance test code using `perf_counter()` and lambda function:

```
#!/usr/bin/env python3
from collections import deque
from time import perf_counter # performance counter

def perf_timeit(NSIZE, TIMES):
    alist = list(range(NSIZE))
    adeq = deque(range(NSIZE))

    def elapsed(func): # a helper function
        start = perf_counter()
        for i in range(TIMES):
            func(i)
        return perf_counter() - start

    list_time = elapsed(None) # your code here
    deq_time = elapsed(None)
    return list_time, deq_time
```

```
if __name__ == '__main__':
    NSIZE = 100_000
    TIMES = 100_000
    list_time, deq_time = perf_timeit(NSIZE, TIMES)
    print(f"list.insert() {list_time:>12.6} sec")
    print(f"deque.appendleft() {deq_time:>12.6} sec")
    ratio = list_time / deq_time
    print(f"list/deque ratio {ratio:>12.6} x faster")
```

Deque Profiling: Step 5

A performance test code using `perf_counter()` and lambda function:

```
#!/usr/bin/env python
from collections import deque
from time import perf_counter # performance counter

def perf_timeit(NSIZE, TIMES):
    alist = list(range(NSIZE))
    adeq = deque(range(NSIZE))

    def elapsed(func): # a helper function
        start = perf_counter()
        for i in range(TIMES):
            func(i)
        return perf_counter() - start

    list_time = elapsed(None) # your code here
    deq_time = elapsed(None)
    return list_time, deq_time
```

```
if __name__ == '__main__':
    NSIZE = 100_000
    TIMES = 100_000
    list_time, deq_time = perf_timeit(NSIZE, TIMES)
    print(f"list.insert() {list_time:>12.6} sec")
    print(f"deque.appendleft() {deq_time:>12.6} sec")
    ratio = list_time / deq_time
    print(f"list/deque ratio {ratio:>12.6} x faster")
```

```
PS C:\GitHub\DSpyx\jupyter> python perf_counter3.py
list.insert() 2.51812 sec
deque.appendleft() 0.0065251 sec
list/deque ratio 385.913 x faster
PS C:\GitHub\DSpyx\jupyter> 
```


Deque Profiling: Step 6 - Exercise

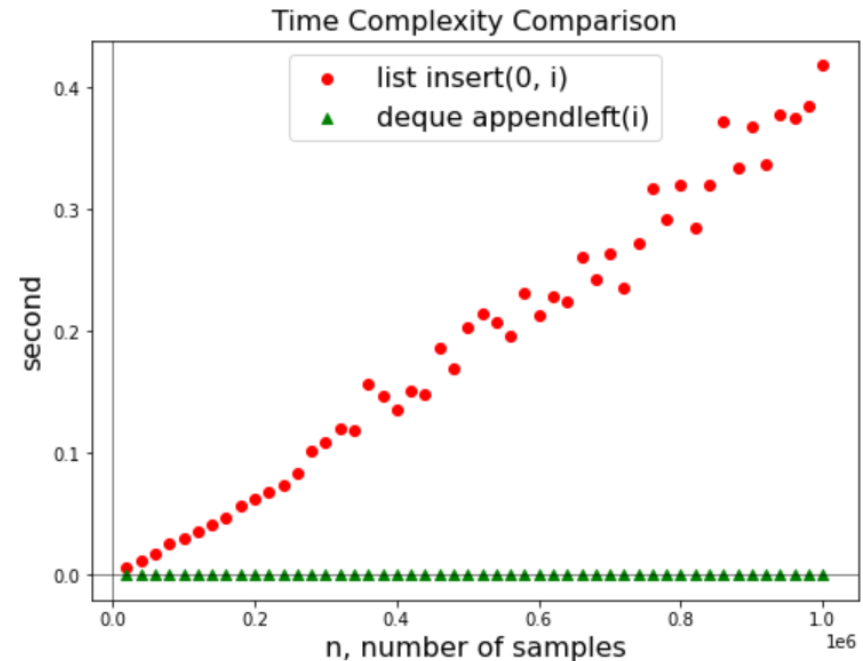
Use `perf_timeit()` instead of `timeit.timeit()` for the performance analysis.

```
#!/usr/bin/env python
from collections import deque
from time import perf_counter # performance counter

def perf_timeit(NSIZE, TIMES):
    alist = list(range(NSIZE))
    adeq = deque(range(NSIZE))

    def elapsed(func): # a helper function
        start = perf_counter()
        for i in range(TIMES):
            func(i)
        return perf_counter() - start

    list_time = elapsed(None) # your code here
    deq_time = elapsed(None)
    return list_time, deq_time
```



Deque Profiling: Step 6 - Exercise

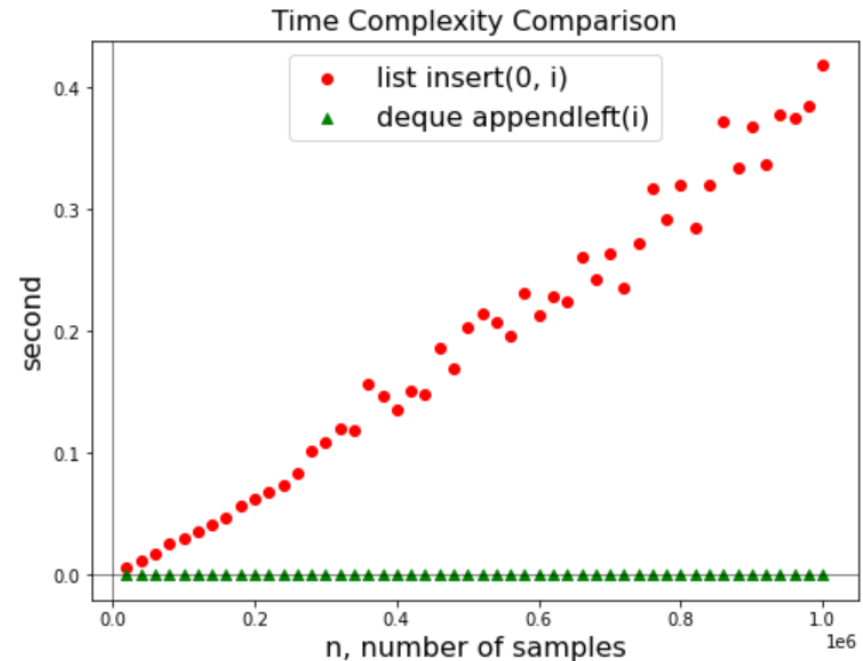
Use `perf_timeit()` instead of `timeit.timeit()` for the performance analysis.

```
#!/usr/bin/env python
# %%writefile perf_counter3.py
from collections import deque
from time import perf_counter # performance counter

def perf_timeit(NSIZE, TIMES):
    alist = list(range(NSIZE))
    adeq = deque(range(NSIZE))

    def elapsed(func): # a helper function
        start = perf_counter()
        for i in range(TIMES):
            func(i)
        return perf_counter() - start

    list_time = elapsed(None) # your code here
    deq_time = elapsed(None)
    return list_time, deq_time
```



```
if __name__ == '__main__':
    list_time = [] # list of list_time values
    deque_time = [] # list of deque_time values
    n = [] # list of samples for plotting

    for i in range(20_000, 1_000_001, 20_000):
        None # your code here
```

Summary

- The **deque** in Python was designed to guarantee efficient **append and pop** operations on either end of the sequence in **$O(1)$** .
 - `perf_counter()` - Performance counter in Python
 - Lambda Function - Anonymous Function or Lambda Expression
- Now, you may decide when to use **deque** instead of **list**.