파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

리스트와 딕셔너리 자료형 메소드들의 성능을
Big-O로 비교할 수 있다

Data Structures in Python
Chapter 2 - 2

- Performance Analysis
- Big-O Notation
- Big-O Properties
- Growth Rates
- **Growth Rates Examples**

Agenda & Reading

- Growth Rate
  - Comparison
  - Profiling and Prediction
- **Growth Rate Examples**
  - **Python List & Dictionary**

- References:
  - Textbook: Problem Solving with Algorithms and Data Structures
    - Chapter 3. Analysis
  - Textbook: www.github.idebtor/DSpy
    - Chapter 2.1 ~ 3

1 Performance of Python Lists

- We have a general idea of the performance analysis and big-O notation.
- It is important to **understand the efficiency** of these Python data structures.
  - Now, we will investigate the Big-O performance for the operations on Python **lists** and **dictionaries.**

1 Performance of Python Lists - Review

- Python lists are **ordered sequences** of items.
- Specific values in the sequence can be referenced using subscripts.
- Python lists are:
  - **dynamic**: They can grow and shrink on demand.
  - **heterogeneous**: a single list can hold arbitrary data types.
  - **mutable** sequences of arbitrary objects

1 Performance of Python Lists - Operations

- Using operators:

```
my_list = [1,2,3,4]
print(2 in my_list)

zeroes = [0] * 10
print(zeroes)
```

```
True
```

```
[0,0,0,0,0,0,0,0,0,0]
```

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

1 Performance of Python Lists - Operations

- Using Methods:

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

1 Performance of Python Lists - Examples

```
my_list = [3, 1, 4, 1, 5, 9]
my_list.append(2)
my_list.sort()
my_list.reverse()
```

```
[3, 1, 4, 1, 5, 9, 2]
[1, 1, 2, 3, 4, 5, 9]
[9, 5, 4, 3, 2, 1, 1]
```

```
print(my_list.index(4))
```

```
2
```

Index of the first occurrence of the parameter

```
my_list.insert(4, "Hello")
print(my_list)
```

```
[9, 5, 4, 3, 'Hello', 2, 1, 1]
```

```
print(my_list.count(1))
```

```
2
```

The number of occurrence of the parameter

```
my_list.remove(1)
print(my_list)
```

```
[9, 5, 4, 3, 'Hello', 2, 1]
```

```
print(my_list.pop(3))
print (my_list)
```

```
3
[9, 5, 4, 'Hello', 2, 1]
```

1 Performance of Python Lists - Operations

- The **del** statement
  - Removes an item from a list given its index instead of its value.
  - Used to remove slices from a list or clear the entire list.

- Sample Run:

```
my_list = [1, 2, 3, 4]
ur_list = [4, 3, 2, 1]

total, max = sum(my_list), max(ur_list)
print(total, max)

total, max = sum(ur_list), max(ur_list)
print(total, max)
```

```
10, 4
```

```
TypeError: 'int' object is not callable
```

1 Performance of Python Lists - Big-O Efficiency of List Operators

index[]	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O($n + k$)
reverse	O(n)
concatenate	O(k)
sort	O($n \log n$)
multiply	O(nk)

1 Performance of Python Lists - Big-O Efficiency of List Operators

index[]	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O($n + k$)
reverse	O(n)
concatenate	O(k)
sort	O($n \log n$)
multiply	O(nk)

```
from timeit import Timer

t1 = Timer('a = ["a"] * 100; a.pop(0)')
t2 = Timer('b = ["b"] * 100; b[1:]')
t3 = Timer('c = ["c"] * 100; del c[0]')
t4 = Timer('d = ["d"] * 100; d.remove("d")')

print(t1.timeit())
print(t2.timeit())
print(t3.timeit())
print(t4.timeit())
```

```
0.6552486000000499
0.8781033000000207
0.6186867000001257
0.6430327999999008
```

1 Performance of Python Lists - Big-O Efficiency of List Operators

index[]	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O(n + k)
reverse	O(n)
concatenate	O(k)
sort	O(n log n)
multiply	O(nk)

```
from timeit import Timer

t1 = Timer('a = ["a"] * 100; a.pop(0)')
t2 = Timer('b = ["b"] * 100; b[1:]')
t3 = Timer('c = ["c"] * 100; del c[0]')
t4 = Timer('d = ["d"] * 100; d.remove("d")')

print(t1.timeit())
print(t2.timeit())
print(t3.timeit())
print(t4.timeit())
```

```
0.6552486000000499
0.8781033000000207
0.6186867000001257
0.6430327999999008
```

```
from timeit import Timer

t1 = Timer('a = ["a"] * 100; a.pop()')
t2 = Timer('b = ["b"] * 100; b[1:]')
t3 = Timer('c = ["c"] * 100; del c[0]')
t4 = Timer('d = ["d"] * 100; d.remove("d")')

print(t1.timeit())
print(t2.timeit())
print(t3.timeit())
print(t4.timeit())
```

```
0.49597530000005463
0.8824276999998801
0.6347200000000157
0.6409297999998671
```

1 Performance of Python Lists - O(1) - Constant

- Operations for **indexing and assigning** to an index position
  - Big-O = O(1)
  - It takes the same amount of time no matter how large the list becomes.
  - i.e., independent of the size of the list

1 Performance of Python Lists - Inserting elements to a List

- There are two ways to create a longer list.
  - Use the **append** method or the **concatenation** operator

- Big-O for the append method is O(1) .
- Big-O for the concatenation operator is O(1) where is the size of the list that is being concatenated

1 Performance of Python Lists - 4 Experiments

- Four different ways to generate a list of n numbers starting with 0.
  - Use the **append** method or the **concatenation** operator

- Example 1:
  - Using a for loop and create the list by **concatenation**
- Example 2:
  - Using a for loop and the **append** method
- Example 3:
  - Using list comprehension
- Example 4:
  - Using the range function wrapped by a call to the list constructor.

```
for i in range(n):
    my_list = my_list + [i]
```

```
for i in range(n):
    my_list.append(i)
```

```
my_list = [i for i in range(n)]
```

```
my_list = list(range(n))
```

1 Performance of Python Lists - 4 Experiments Result

- From the results of our experiment:
  1. Using for loop
     - The **append** operation is much faster than **concatenation**
     - (note) Append: Big-O is O(1), Concatenation: Big-O is O(k)
  2. Two additional methods for creating a list
     - Using the **list constructor** with a call to range is much faster than a **list comprehension**
- It is interesting to note that the list comprehension is twice as fast as a for loop with an append operation.

```
for i in range(n):
    my_list = my_list + [i]
```

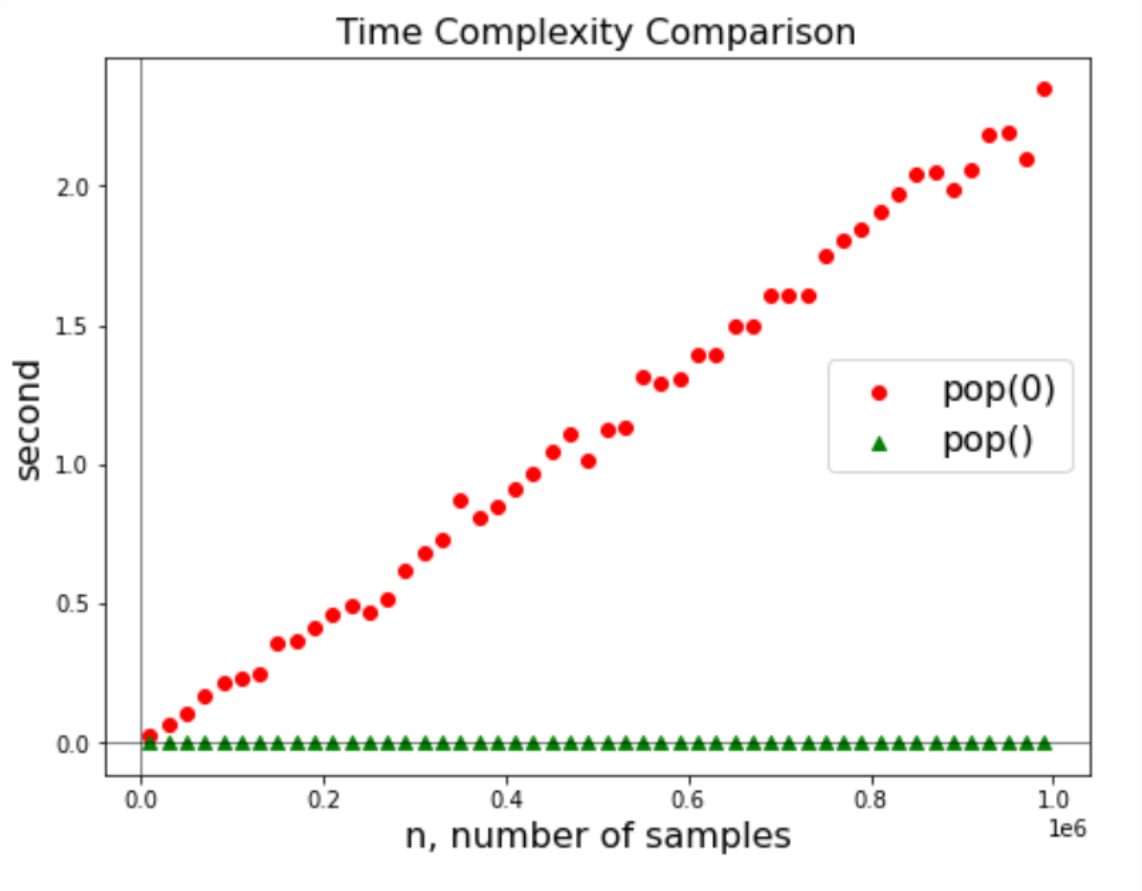
```
for i in range(n):
    my_list.append(i)
```

```
my_list = [i for i in range(n)]
```

```
my_list = list(range(n))
```

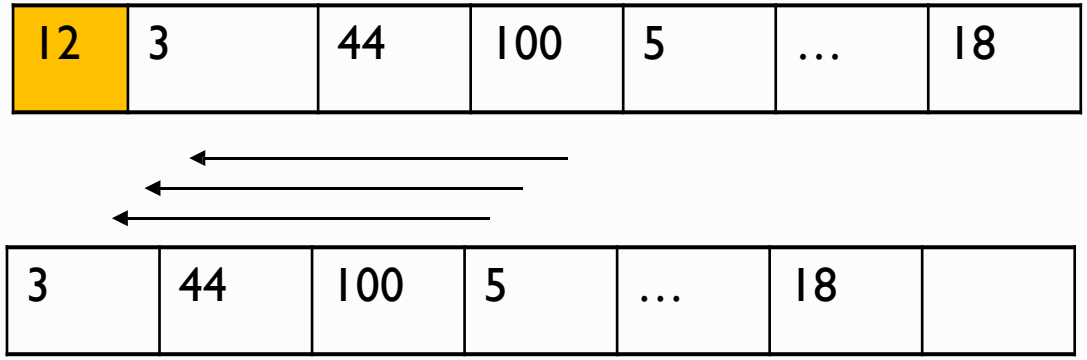
1 Performance of Python Lists - Pop() vs Pop(0)

- From the results of our experiment:
  - As the list gets longer and longer the time it takes to pop(0) also increases
  - the time for pop stays very flat.
  - pop(0): Big-O is O(n)
  - pop(): Big-O is O(1)
  - Why?



1 Performance of Python Lists - Pop() vs Pop(0)

- pop():
  - Removes element from the **end of the list**
- pop(0)
  - Removes from the **beginning of the list.**
  - Big-O is O(n) as we will need to shift all elements from space to the beginning of the list



2 Performance of Python Dictionaries

- Dictionaries store a mapping between a set of **keys** and a set of **values**
  - Keys can be any immutable type.
  - Values can be any type
  - A single dictionary can store values of different types
- You can define, modify, view, lookup or delete the key-value pairs in the dictionary
- Dictionaries are unordered
- Note:
  - Dictionaries differ from lists in that you can access items in a dictionary by a **key** rather than a position.

2 Performance of Python Dictionaries - Examples:

```
capitals = {'Korea':'Seoul','Japan':'Tokyo'}
print(capitals['Korea'])

capitals['Rwanda'] = 'Kigali'
print(capitals)

capitals['Taiwan'] = 'Taipei'

print(len(capitals))
for k in capitals:
    print(capitals[k]," is the capital of ", k)
```

```
Seoul
```

```
{'Korea': 'Seoul', 'Japan': 'Tokyo', 'Rwanda': 'Kigali'}
```

```
4
```

```
Seoul  is the capital of  Korea
Tokyo  is the capital of  Japan
Kigali  is the capital of  Rwanda
Taipei  is the capital of  Taiwan
```

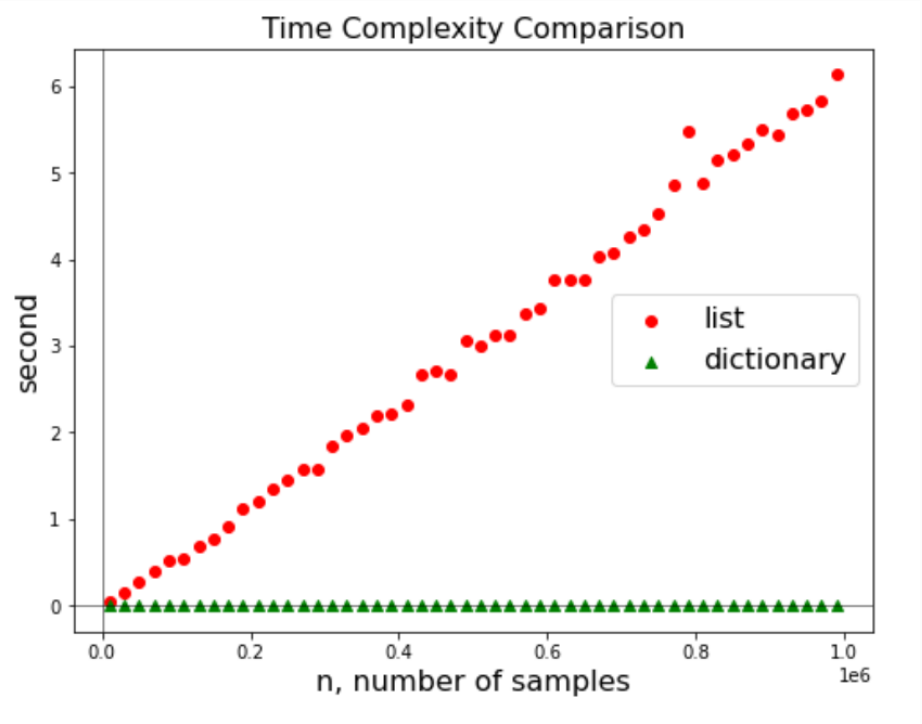
2 Performance of Python Dictionaries - Big-O Efficiency of Operators

- Big-O Efficiency of Operators

Operation	Big-O
copy	O(n)
get item	O(1)
set item	O(1)
delete item	O(1)
containis (in)	O(1)
iteration	O(n)

2 Performance of Python Dictionaries - Big-O Efficiency of Operators

- **Contains** (**in** operator) between lists and dictionaries
- From the results
  - The time it takes for the contains operator on the list grows linearly with the size of the list.
  - The time for the contains operator on a dictionary is constant even as the dictionary size grows
- Lists, Big-O is O(n)
- Dictionaries, big-O is O(1)



Summary

- Performance of Python List and Dictionary Operations

index[]	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O($n + k$)
reverse	O(n)
concatenate	O(k)
sort	O($n \log n$)
multiply	O(nk)

Operation	Big-O
copy	O(n)
get item	O(1)
set item	O(1)
delete item	O(1)
containis (in)	O(1)
iteration	O(n)

학습 정리

1) 리스트 자료형은 인덱싱으로 추가/삭제/수정하고,
딕셔너리 자료형은 key/value로 구성되어 있으며 자료의 순서를
보장하지 않는다

2) 리스트(list) 자료형에서 pop()은 O(1), pop(0)는 O(n)이다

3) 딕셔너리(dict)자료형은 해시(hash)구조이기 때문에
대부분의 연산이 O(1)이다

파이썬으로 배우는 데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니다