Python

왜 deque는 list보다 빠른가?

김상훈

강의 내용중...

deque

collections - deque

- deque 는 기존 list보다 효율적인 자료구조를 제공
- 효율적 메모리 구조로 처리 속도 향상

deque

```
from collections import deque
import time

start_time = time.clock()
deque_list = deque()
# Stack
for i in range(10000):
    for i in range(10000):
        deque_list.append(i)
        deque_list.pop()
print(time.clock() - start_time, "seconds")
```

general list

```
import time


start_time = time.clock()
just_list = []
for i in range(10000):
    for i in range(10000):
        just_list.append(i)
        just_list.pop()
print(time.clock() - start_time, "seconds")
```

In [14]:
```
def general_list():
    just_list = []
    for i in range(100):
        for i in range(100):
            just_list.append(i)
            just_list.pop()

%timeit general_list()
```

```
3.6 ms ± 292 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [15]:
```
def deque_list():
    deque_list = deque()

    for i in range(100):
        for i in range(100):
            deque_list.append(i)
            deque_list.pop()

%timeit deque_list()
```

```
1.04 ms ± 115 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

=> Deque를 쓸 경우 1.04ms, list를 쓸 경우 3.6ms로 차이가 많이 난다.

왜 차이가 날까?

That said, the real differences between deques and list in terms of performance are:

- Deques have O(1) speed for *appendleft()* and *popleft()* while lists have O(n) performance for *insert(0, value)* and *pop(0)*.
- List append performance is hit and miss because it uses *realloc()* under the hood. As a result, it tends to have over-optimistic timings in simple code (because the realloc doesn't have to move data) and really slow timings in real code (because fragmentation forces realloc to move all the data). In contrast, deque append performance is consistent because it never reallocs and never moves data.

- **list**는 **realloc**을 사용한다.
- 이론 상으론 사이즈만 늘리면 되기 때문에 최적의 시간을 보장한다.
- 하지만, 실제 사용 시에는 **fragmentation**이 일어나서 데이터를 모두 옮겨야 하는 일이 발생

https://stackoverflow.com/questions/23487307/python-deque-vs-list-performance-comparison

List structure vs Deque structure

```c
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    PyTypeObject *ob_type;
} PyObject;
```

```c
static PyObject *
list_insert_impl(PyListObject *self, Py_ssize_t index, PyObject *object);

static PyObject *
list_insert(PyListObject *self, PyObject *const *args, Py_ssize_t nargs)
{
    PyObject *return_value = NULL;
    Py_ssize_t index;
    PyObject *object;

    if (!_PyArg_CheckPositional("insert", nargs, 2, 2)) {
        goto exit;
    }
    {
        Py_ssize_t ival = -1;
        PyObject *iobj = _PyNumber_Index(args[0]);
        if (iobj != NULL) {
            ival = PyLong_AsSsize_t(iobj);
            Py_DECREF(iobj);
        }
        if (ival == -1 && PyErr_Occurred()) {
            goto exit;
        }
        index = ival;
    }
    object = args[1];
    return_value = list_insert_impl(self, index, object);

exit:
    return return_value;
}
```

```c
typedef struct BLOCK {
    struct BLOCK *leftlink;
    PyObject *data[BLOCKLEN];
    struct BLOCK *rightlink;
} block;

typedef struct {
    PyObject_VAR_HEAD
    block *leftblock;
    block *rightblock;
    Py_ssize_t leftindex;       /* 0 <= leftindex < BLOCKLEN */
    Py_ssize_t rightindex;      /* 0 <= rightindex < BLOCKLEN */
    size_t state;               /* incremented whenever the indices move */
    Py_ssize_t maxlen;          /* maxlen is -1 for unbounded deques */
    PyObject *weakreflist;
} dequeobject;

static PyTypeObject deque_type;
```

https://github.com/python/cpython/blob/master/Objects/clinic/listobject.c.h

https://github.com/python/cpython/blob/master/Modules/_collectionsmodule.c

그래서 realloc은요?

```c
static int
list_resize(PyListObject *self, Py_ssize_t newsize)
{
    PyObject **items;
    size_t new_allocated, num_allocated_bytes;
    Py_ssize_t allocated = self->allocated;

    /* Bypass realloc() when a previous overallocation is large enough
       to accommodate the newsize.  If the newsize falls lower than half
       the allocated size, then proceed with the realloc() to shrink the list.
    */
    if (allocated >= newsize && newsize >= (allocated >> 1)) {
        assert(self->ob_item != NULL || newsize == 0);
        Py_SET_SIZE(self, newsize);
        return 0;
    }
```

```c
    new_allocated = ((size_t)newsize + (newsize >> 3) + 6) & ~(size_t)3;
    /* Do not overallocate if the new size is closer to overallocated size
     * than to the old size.
     */
    if (newsize - Py_SIZE(self) > (Py_ssize_t)(new_allocated - newsize))
        new_allocated = ((size_t)newsize + 3) & ~(size_t)3;

    if (newsize == 0)
        new_allocated = 0;
    num_allocated_bytes = new_allocated * sizeof(PyObject *);
    items = (PyObject **)PyMem_Realloc(self->ob_item, num_allocated_bytes);
    if (items == NULL) {
        PyErr_NoMemory();
        return -1;
    }
    self->ob_item = items;
    Py_SET_SIZE(self, newsize);
    self->allocated = new_allocated;
    return 0;
}
```

https://github.com/python/cpython/blob/master/Objects/listobject.c

PyMem_Realloc을 찾아서

```
PyAPI_FUNC(void *) PyMem_Malloc(size_t size);
PyAPI_FUNC(void *) PyMem_Realloc(void *ptr, size_t new_size);
PyAPI_FUNC(void) PyMem_Free(void *ptr);
```

https://github.com/python/cpython/blob/master/Include/pymem.h

PyAPI_FUNC type으로 return 해주는데.. PyMem_Realloc에 대한 정확한 함수 정의는 찾을 수가 없었다...

```
/* only get special linkage if built as shared or platform is Cygwin */
#if defined(Py_ENABLE_SHARED) || defined(__CYGWIN__)
#       if defined(HAVE_DECLSPEC_DLL)
#               if defined(Py_BUILD_CORE) && !defined(Py_BUILD_CORE_MODULE)
#                       define PyAPI_FUNC(RTYPE) Py_EXPORTED_SYMBOL RTYPE
#                       define PyAPI_DATA(RTYPE) extern Py_EXPORTED_SYMBOL RTYPE
        /* module init functions inside the core need no external linkage */
        /* except for Cygwin to handle embedding */
#                       if defined(__CYGWIN__)
#                               define PyMODINIT_FUNC Py_EXPORTED_SYMBOL PyObject*
#                       else /* __CYGWIN__ */
#                               define PyMODINIT_FUNC PyObject*
#                       endif /* __CYGWIN__ */
#               else /* Py_BUILD_CORE */
```

~~(제보바람)~~

https://github.com/python/cpython/blob/master/Include/pyport.h

일단 이걸로 보자 (python 2.4로 추정)

```
#undef PyObject_Realloc
void *
PyObject_Realloc(void *p, size_t nbytes)
{
        void *bp;
        poolp pool;
        uint size;

        if (p == NULL)
                return PyObject_Malloc(nbytes);

        pool = POOL_ADDR(p);
        if (Py_ADDRESS_IN_RANGE(p, pool)) {
                /* We're in charge of this block */
                size = INDEX2SIZE(pool->szidx);
                if (nbytes <= size) {
                        /* The block is staying the same or shrinking.  If
                         * it's shrinking, there's a tradeoff:  it costs
                         * cycles to copy the block to a smaller size class,
                         * but it wastes memory not to copy it.  The
                         * compromise here is to copy on shrink only if at
                         * least 25% of size can be shaved off.
                         */
                        if (4 * nbytes > 3 * size) {
                                /* It's the same,
                                 * or shrinking and new/old > 3/4.
                                 */
                                return p;
                        }
                        size = nbytes;
                }
                bp = PyObject_Malloc(nbytes);1
                if (bp != NULL) {   2
                        memcpy(bp, p, size);3
                        PyObject_Free(p);
                }
                return bp;
        }
        /* We're not managing this block.  If nbytes <=
```

Realloc 할 때

1. malloc을 하고
2. 더 이상 메모리가 없으면
3. Memcpy가 일어남

https://svn.red-bean.com/bob/python24-fat/Objects/obmalloc.c

Iterator

Iterable란

- 사전적 의미: 순회가능한
- an object capable of **returning** its **members one by one**
- Python에서는 순회가 가능한 모든 객체를 가리킵니다
- `for 변수 in `뒤에 올 수 있는 것이 모두 iterable 한 객체(iterable object)
- `dir(object)` 에서 __iter__가 존재하면 iterable object

Iterable의 예시

- Sequence types
  - Lists, strings and tuples
  - Support efficient element access using integer indices
- Non seqeunce
  - Dictionaries, file objects, sets

For loop

```
# javascript
let numbers = [10, 12, 15, 18, 20];
for (let i = 0; i < numbers.length; i += 1) {
    console.log(numbers[i])
}

# python
numbers = [10, 12, 15, 18, 20]
for number in numbers:
    print(number)
```

Python의 loop - indices 를 사용하지 않는다

```
index = 0
numbers = [1, 2, 3, 4, 5]
while index < len(numbers):
    print(numbers[index], end=' ')
    index += 1

# 1 2 3 4 5

index = 0
numbers = {1, 2, 3, 4, 5}
while index < len(numbers):
    print(numbers[index])
    index += 1

# TypeError: 'set' object does not support indexing
```

```
numbers = {1, 2, 3, 4, 5}
for number in numbers:
    print(number)
```

Iterator란 - __iter__

- Iterator = iterable object
- `__iter__` method로 iterator를 만들 수 있습니다!

```
numbers = [10, 12, 15, 18, 20]
fruits = ("apple", "pineapple", "blueberry")
message = "I love Python ❤"

print(iter(numbers))
print(iter(fruits))
print(iter(message))
```

```
<list_iterator object at 0x000001DBCEC33B70>
<tuple_iterator object at 0x000001DBCEC33B00>
<str_iterator object at 0x000001DBCEC33C18>
```

Iterator란 -- __next__

- `__next__`를 호출하면 for문의 동작처럼 하나씩 값을 꺼내올 수 있습니다.
  - next() returns successive items in the stream.
- iterator의 다음 item을 반환해주고, 다음 데이터가 없으면 StopIteration exception을 raise

```
values = [10, 20, 30]
iterator = iter(values)
print(next(iterator)) # 10
print(next(iterator)) # 20
print(next(iterator)) # 30
print(next(iterator)) # StopIteration exception
```

Python for loop의 구현

```
def custom_for_loop(iterable, action_to_do):
    iterator = iter(iterable)
    done_looping = False
    while not done_looping:
        try:
            item = next(iterator)
        except StopIteration:
            done_looping = True
        else:
            action_to_do(item)
numbers = {1, 2, 3, 4, 5}
custom_for_loop(numbers, print)
```

Iterator Protocol(규약)

- - The iterator protocol is a fancy way of saying "how looping over iterables works in Python." It's essentially the definition of the way the iter and next functions work in Python. All forms of iteration in Python are powered by the iterator protocol.

- Iterable object가 __iter__함수와 __next__함수를 지원하면.. Iterator다.

- 다시 정리하면
  - An **iterable** is something you can **loop over**.
  - An **iterator** is an object representing a **stream of data**. It does the **iterating** over an iterable.

Iterable protocol & iterator

- Iterable protocol

```
for n in numbers: # for loop
    print(n)
x, y, z = coordinates # multiple assignment

# star expressions
a, b, *rest = numbers
print(*numbers)

unique_numbers = set(numbers) # built-in functions
```

- Iterator
  - enumerate, zip, reversed, map, filter, file objects, dictinoary
  - next(iterator)호출해보면 다음 값이 나와요

Object	Iterable?	Iterator?
Iterable	✓	?
Iterator	✓	✓
Generator	✓	✓
List	✓	✗

Custom iterator 만들기 (Generator)

- Iterator를 쉽게 생성하게 해주는 것이 generator의 역할
- python의 함수는 보통 return 후 종료가 되지만, generator는 yield(산출)한다는 특징이 있다

```
class square_all:
    def __init__(self, numbers):
        self.numbers = iter(numbers)
    def __next__(self):
        return next(self.numbers) ** 2
    def __iter__(self):
        return self

# gernerate function
def square_all(numbers):
    for n in numbers:
        yield n**2

# generator expression
def square_all(numbers):
    return (n**2 for n in numbers)
```

Lazy evaluation 그리고 일반 함수와의 차이

- Next()를 호출하기 전까지 대기, 호출되면 연산 수행
- 2번 답변 https://www.edwith.org/bcaitech1/forum/46122

참고자료

```cpp
// Helper Node class
class Node {
    public:
        int value;
        Node* next;
};

// Linked List class
class LinkedList {
    public:
        Node* root; // root node

        // Iterator class
        class iterator : public std::iterator<std::forward_iterator_tag, int> {
            public:
                friend class LinkedList; // declare Linked List class as a friend class
                Node* curr;              // the Node this iterator is pointing to

                // the following typedefs are needed for the iterator to play nicely with C++ STL
                typedef int value_type;
                typedef int& reference;
                typedef int* pointer;
                typedef int difference_type;
                typedef std::forward_iterator_tag iterator_category;

                // iterator constructor
                iterator(Node* x=0):curr(x){}

                // overload the == operator of the iterator class
                bool operator==(const iterator& x) const {
                    return curr == x.curr; // compare curr pointers for equality
                }

                // overload the != operator of the iterator class
                bool operator!=(const iterator& x) const {
                    return curr != x.curr; // compare curr pointers for inequality
                }

                // overload the * operator of the iterator class
                reference operator*() const {
                    return curr->value;    // return curr's value
                }

                // overload the ++ (pre-increment) operator of the iterator class
                iterator& operator++() {
                    curr = curr->next;     // move to next element
                    return *this;          // return after the move
                }

                // overload the ++ (post-increment) operator of the iterator class
                iterator operator++(int) {
                    iterator tmp(curr);    // create a temporary iterator to current element
                    curr = curr->next;     // move to next element
                    return tmp;            // return iterator to previous element
                }
        };

        // return iterator to first element
        iterator begin() {
            return iterator(root);
        }

        // return iterator to JUST AFTER the last element
        iterator end() {
            return iterator(NULL);
        }
};
```

```cpp
// Array List class
class ArrayList {
    public:
        int arr[10]; // backing array
        int size;    // number of elements that have been inserted

        // Iterator class
        class iterator : public std::iterator<std::forward_iterator_tag, int> {
            public:
                friend class ArrayList;  // declare Array List class as a friend class
                int* curr;               // the element this iterator is pointing to

                // the following typedefs are needed for the iterator to play nicely with C++ STL
                typedef int value_type;
                typedef int& reference;
                typedef int* pointer;
                typedef int difference_type;
                typedef std::forward_iterator_tag iterator_category;

                // iterator constructor
                iterator(int* x=0):curr(x){}

                // overload the == operator of the iterator class
                bool operator==(const iterator& x) const {
                    return curr == x.curr; // compare curr pointers for equality
                }

                // overload the != operator of the iterator class
                bool operator!=(const iterator& x) const {
                    return curr != x.curr; // compare curr pointers for inequality
                }

                // overload the * operator of the iterator class
                reference operator*() const {
                    return *curr;          // return curr's value
                }

                // overload the ++ (pre-increment) operator of the iterator class
                iterator& operator++() {
                    curr++;                // move to next slot of array
                    return *this;          // return after the move
                }

                // overload the ++ (post-increment) operator of the iterator class
                iterator operator++(int) {
                    iterator tmp(curr);    // create a temporary iterator to current element
                    curr++;                // move to next slot of array
                    return tmp;            // return iterator to previous element
                }
        };

        // return iterator to first element
        iterator begin() {
            return iterator(&arr[0]);
        }

        // return iterator to JUST AFTER the last element
        iterator end() {
            return iterator(&arr[size]);
        }
};
```

```cpp
for(auto it = l.begin(); it != l.end(); it++) {
    cout << *it << endl;
}
```

참조

- https://opensource.com/article/18/3/loop-better-deeper-look-iteration-python
- https://towardsdatascience.com/python-basics-iteration-and-looping-6ca63b30835c
- https://livetodaykono.tistory.com/25

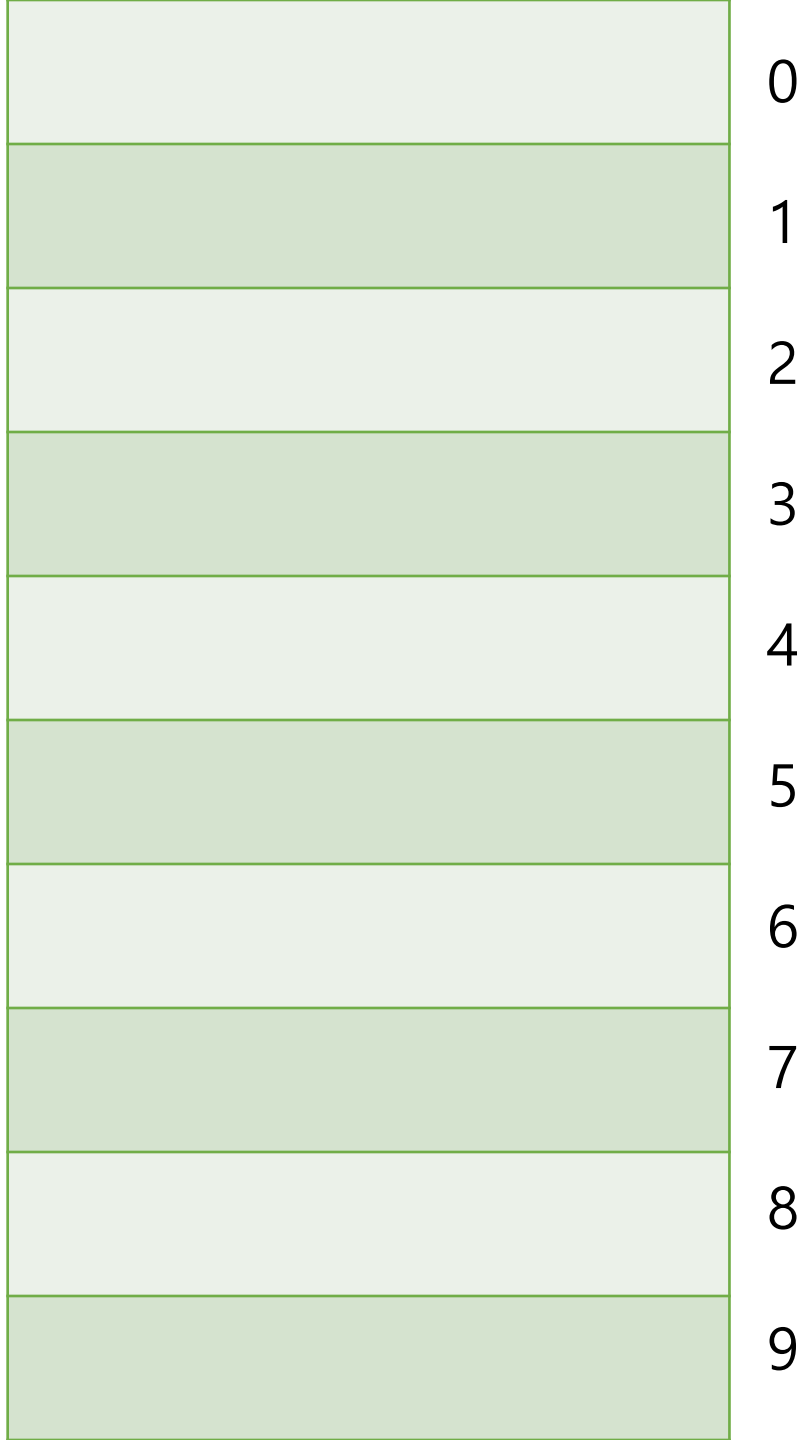
Hash Table

해쉬 구조란?

- 키(Key)와 값(Value)쌍으로 이루어진 데이터 구조를 의미합니다. Key를 이용하여 데이터를 찾으므로, 속도를 빠르게 만드는 구조입니다.
- 파이썬에서는 딕셔너리(Dictionary) 타입이 해쉬 테이블과 같은 구조입니다.
- 기본적으로는, 배열로 미리 Hash Table 크기만큼 생성해서 사용합니다. 공간은 많이 사용하지만, 시간은 빠르다는 장점이 있습니다.
- 검색이 많이 필요한 경우, 저장, 삭제, 읽기가 많은 경우, 캐쉬를 구현할 때 주로 사용됩니다.
- 장점
    - 데이터 저장/검색 속도가 빠릅니다.
    - 해쉬는 키에 대한 데이터가 있는지(중복) 확인이 쉽습니다.
- 시간 복잡도
    - 일반적인 경우(충돌이 없는 경우): O(1)
    - 최악의 경우(모든 경우에 충돌이 발생하는 경우): O(n)
- 단점
    - 일반적으로 저장공간이 좀더 많이 필요합니다.
    - 여러 키에 해당하는 주소가 동일할 경우 충돌을 해결하기 위한 별도 자료구조가 필요합니다. (충돌 해결 알고리즘)

리스트
삽입
삭제
탐색

Ex) 2020019111 : `배창은`
2020019222 : `주찬형`
2020018333 : `김상훈`
2020018444 : `최보미`
2020019555 : `최길희`
2020019666 : `유지훈`



Hash Function

어떻게 mapping을 할 것인가?
Less collision
Fast computation

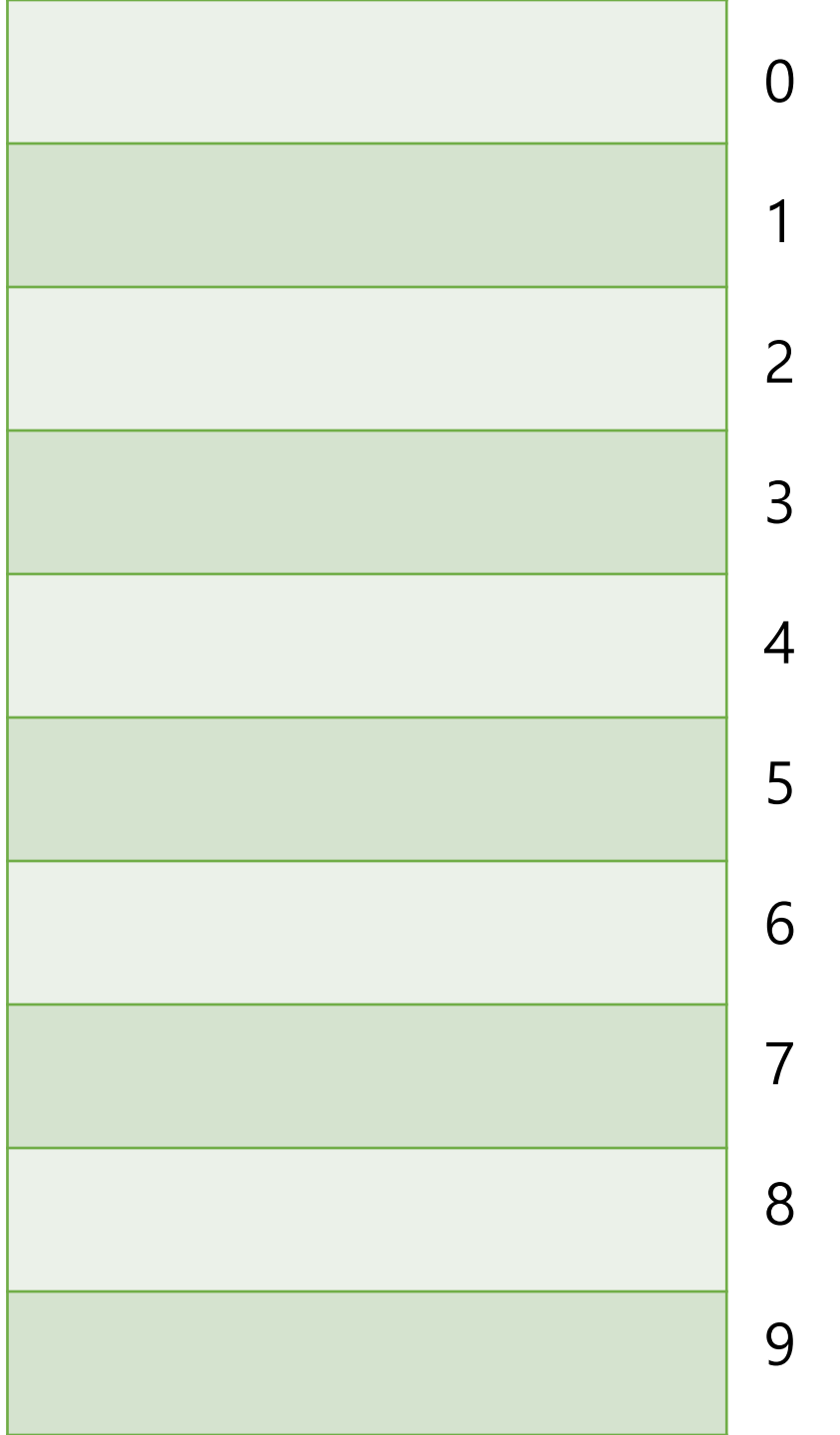
Perfect hash Function : 이상적, 비현실적
Universal hash Function : 1/m
C-Universal hash function : c/m

Ex) 나머지
스트링일 경우에는 아스키코드

Collision Resolution Method

Open addressing : linear probing

chaining:



레퍼런스

- 파이썬으로 구현하는 자료구조 요약 정리 - 해쉬 테이블(Hash Table) (tistory.com)
- 자료구조 해시테이블 - 소개, 해시 함수 – YouTube
- 자료구조 해시테이블 - open addressing (linear probing) – YouTube
- SHA - 위키백과, 우리 모두의 백과사전 (wikipedia.org)

Deep Copy

Class, Object, Instance, Variable...

Mutable vs Immutable Objects

Copy Problem & Method

Class, Object, Instance, Variable…

- Example : INT
- Class로 정의되어 있으나, built in method __int__ 를 구현하기 위한 object frame.
- 흔히 알려진 32비트, 4바이트에 해당하는 수 표현 범위를 가진 정수를 나타내기 위함
- __int__, 우리가 흔히 알고 있는 int() 메소드는 int형 object, 정확히는 int class의 instance를 반환한다.

class **int**([*x*])

class **int**(*x*, *base=10*)

Return an integer object constructed from a number or string *x*, or return `0` if no arguments are given. If *x* defines `__int__()`, `int(x)` returns `x.__int__()`. If *x* defines `__index__()`, it returns `x.__index__()`. If *x* defines `__trunc__()`, it returns `x.__trunc__()`. For floating point numbers, this truncates towards zero.

If *x* is not a number or if *base* is given, then *x* must be a string, `bytes`, or `bytearray` instance representing an integer literal in radix *base*. Optionally, the literal can be preceded by `+` or `-` (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with `a` to `z` (or `A` to `Z`) having values 10 to 35. The default *base* is 10. The allowed values are 0 and 2–36. Base-2, -8, and -16 literals can be optionally prefixed with `0b`/`0B`, `0o`/`0O`, or `0x`/`0X`, as with integer literals in code. Base 0 means to interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, and so that `int('010', 0)` is not legal, while `int('010')` is, as well as `int('010', 8)`.

The integer type is described in Numeric Types — int, float, complex.

Changed in version 3.4: If *base* is not an instance of `int` and the *base* object has a `base.__index__` method, that method is called to obtain an integer for the base. Previous versions used `base.__int__` instead of `base.__index__`.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: *x* is now a positional-only parameter.

Changed in version 3.8: Falls back to `__index__()` if `__int__()` is not defined.

isinstance(*object*, *classinfo*)

Return `True` if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect or virtual)

Class, Object, Instance, Variable…

- Python은 interpreter 언어, 기존의 컴파일러 언어(C,C++,..)와 달리 프로그램 단위(object file 및 기계어 -> exe)가 아닌 script, 한 줄 한 줄 단위로 excutable file 생성을 목적으로 읽어 들인다.

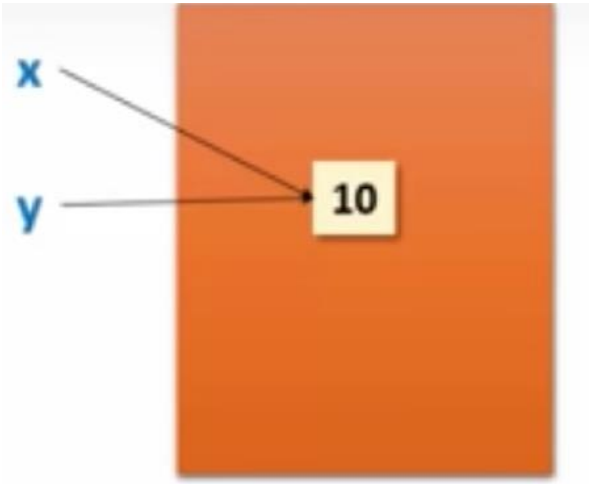
- Example : A = 5 , 위 특징에 의해 A는 어떤 타입의 object를 가리키고 할당 받는지는 해당 statement에 인터프리터가 도달해야만 비로소 정해진다.

→5 ? Int **class**에 기술된 내용, 특징을 두루 갖춘 **object** initiate. 이어서 5라는 값을 object를 구현하는 attribute에 할당. 결과적으로 '5'의 int-numeric-type **instance**가 생성. 동시에 object id 할당.(heap)

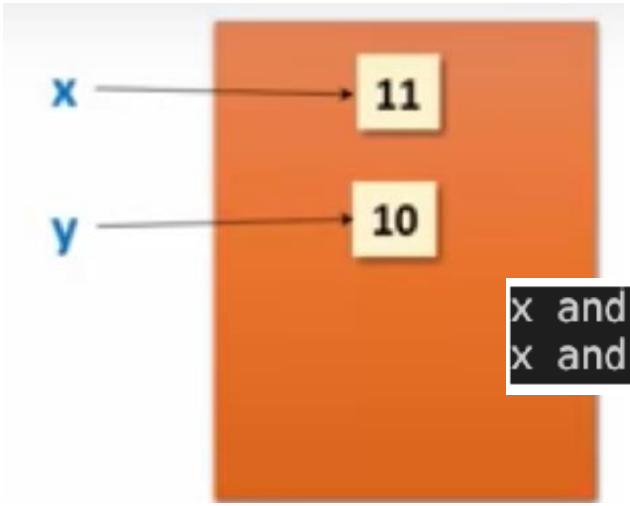
→A = ? A라는 **variable**이 stack memory load, 5 instance address pointing.

Don't confuse, Everything is Object in Python (int class도 object에 불과).

Mutable VS Immutable Objects




```
x = x + 1
```




```
x and y refer to the same object
x and y refer to DIFFERENT object
```

Mutable: 수정가능 / Immutable: 수정불가능

So, Int 클래스 타입의 오브젝트는 Mutable ? Immutable?

Conclusion : 변경 후 객체 id가 달라진다면 immutable, 동일하다면 mutable

Mutable VS Immutable Objects

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Mutable VS Immutable Objects

- 바꾸는 Cost 효율? Mutable > Immutable : 전자는 copy본 필요x , 후자는 새로운 object 생성을 요구할 수 있다.
- 참조 및 접근 속도? Mutable < Immutable : 전자는 크기, 값의 변화에 따른 대처가 필요. 후자는 불변의 객체에 접근만 하면 끝.
- 끔찍한 혼종,

→temp_tuple = (“it is string” , [1,2,3]) : 2번째 요소인 list object는 변경이 가능할까?

→가능하다. [1,2,3]이 위치한 1번째 index에 다른 object로 갈아치울순 없어도, [1,2,3] 자체는 mutable하기에 list 내부 원소의 값을 추가,삭제,갱신이 가능하다.

Copy problem : Shallow copy

```
>>> a = [1,2,3]
>>> b = a[:]
>>> id(a)
4396179528
>>> id(b)
4393788808
>>> a == b
True
>>> a is b
False
>>> b[0] = 5
>>> a
[1, 2, 3]
>>> b
[5, 2, 3]
```

```
>>> a = [[1,2], [3,4]]
>>> b = a[:]
>>> id(a)
4395624328
>>> id(b)
4396179592
>>> id(a[0])
4396116040
>>> id(b[0])
4396116040
```

```
>>> a[0] = [8,9]
>>> a
[[8, 9], [3, 4]]
>>> b
[[1, 2], [3, 4]]
>>> id(a[0])
4393788808
>>> id(b[0])
4396116040
```

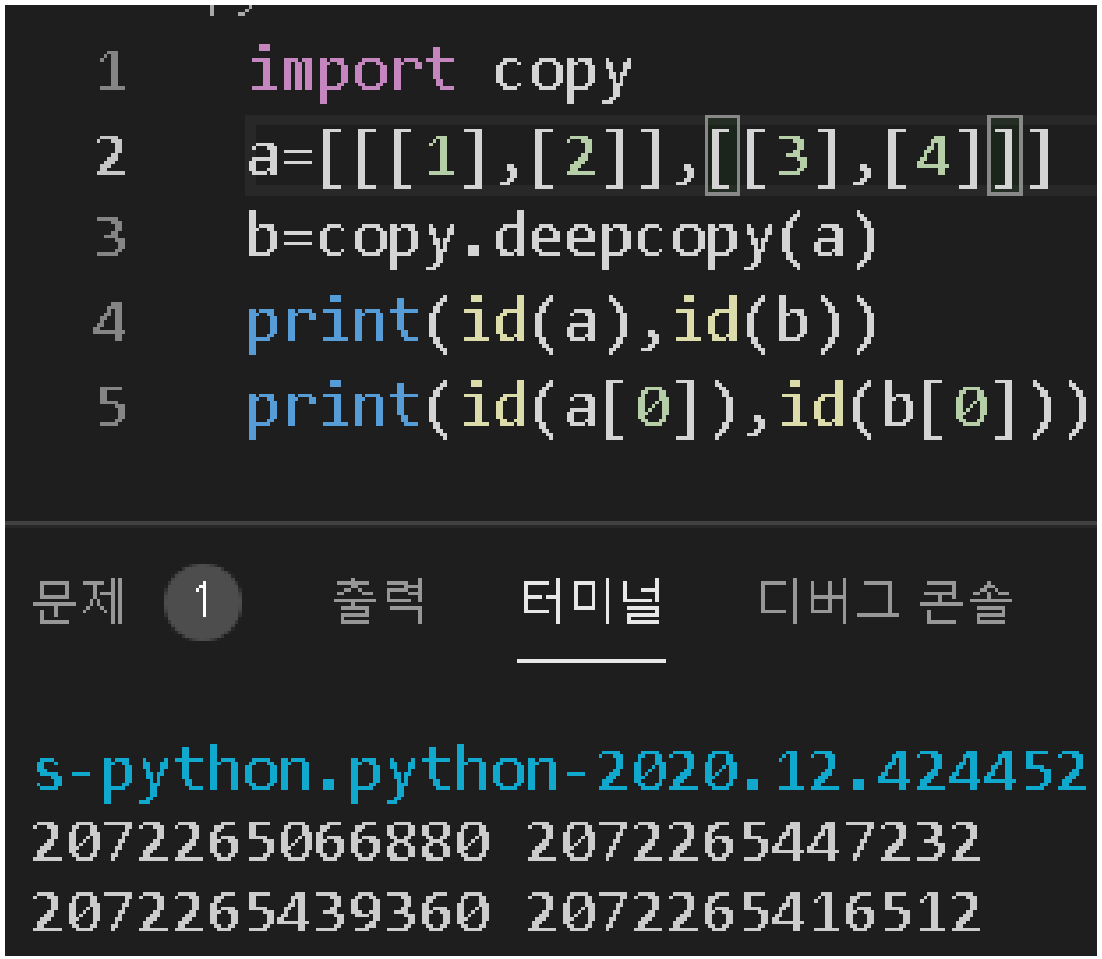
```
1  a=[[1,2],[3,4]]
2  b=a[:][:]
3  print(id(a),id(b))
4  print(id(a[0]),id(b[0]))
```

문제 1 출력 터미널 디버그 콘솔

```
2327048107776 2327048083968
2327048108352 2327048108352
```

- 가장 바깥 axis에 대한 복사본만 만들어주는 얕은 카피.
- [:] assign == copy modul의 copy() method

Method : Deep Copy



Method : Deep Copy

Tradeoff: deepcopy는 무결 성능으로 값만 똑같은 다른 객체를 반환해 원본 데이터 보존을 보장하나, copy 계열의 process 중 속도가 가장 느리다고 한다.

따라서 1차원이라면 앞서 설명한 슬라이싱(가장 빠름), 다차원이고 속도를 고려해야 한다면 다음 그림과 같이 직접 참조하여 값만(값 자체는 immutable한 element일 때) 복사하여 이 문제를 해결하자.

```
a = [1, 2, 3, 4]
b = [i for i in a]
b[1] = 0
print(a,b) # [1, 2, 3, 4] [1, 0, 3, 4]
```

```
a = [1, 2, 3, 4]
b = []
for i in range(len(a)):
    b.append(a[i])
b[1] = 0
print(a,b) # [1, 2, 3, 4] [1, 0, 3, 4]
```

```
a = [1, 2, 3, 4]
b = []
for item in a:
    b.append(item)
b[1] = 0
print(a,b) # [1, 2, 3, 4] [1, 0, 3, 4]
```

Reference

- https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/ , Why python is slow? : Looking under the hood
- https://zzonglove.tistory.com/21 , python 의 변경가능(Mutable) vs 변경불가능(Immutable) 객체들
- https://medium.com/@meghamohan/mutable-and-immutable-side-of-python-c2145cf72747, **Mutable vs Immutable Objects in Python**

Collections_ChainMap

ChainMap

Python 3.3 부터 새롭게 생긴 객체

여러 개의 dictionary의 mapping을 이어주는 역할

```
dict1 = {'1':'one', '2':'two','3':'three'}
dict2 = {'4':'four', '5':'five','6':'six'}

chain_map = ChainMap(dict1, dict2)
print(chain_map)
```

```
ChainMap({'1': 'one', '2': 'two', '3': 'three'}, {'4': 'four', '5': 'five', '6': 'six'})
```

Dict.update()와 차이

```
dict1 = {'1':'one', '2':'two','3':'three'}
dict2 = {'4':'four', '5':'five','6':'six'}

chain_map = ChainMap(dict1, dict2)
print(chain_map)
dict1.update(dict2)
print(dict1)
```

ChainMap은 mapping을 이어주는 역할이라
Dict.dupdate()와는 type이 다름

```
ChainMap({'1': 'one', '2': 'two', '3': 'three'}, {'4': 'four', '5': 'five', '6': 'six'})
{'1': 'one', '2': 'two', '3': 'three', '4': 'four', '5': 'five', '6': 'six'}
```

```
dict1 = {'1':'one', '2':'two','3':'three'}
dict2 = {'4':'four', '5':'five','6':'six'}
dict3 = {'7':'seven','8':'eight', '9':'nine'}

chain_map = ChainMap(dict1, dict2, dict3)
print(chain_map)
dict1.update(dict2)
dict1.update(dict3)
print(dict1)
```

Dict를 새로 생성하거나 여러 번 update() 함수를 호출하는 것보다
Chainmap은 한 번에 여러 개의 dict를 연결할 수 있어 더 빠름

```
ChainMap({'1': 'one', '2': 'two', '3': 'three'}, {'4': 'four', '5': 'five', '6': 'six'}, {'7': 'seven', '8': 'eight', '9': 'nine'})
{'1': 'one', '2': 'two', '3': 'three', '4': 'four', '5': 'five', '6': 'six', '7': 'seven', '8': 'eight', '9': 'nine'}
```

Dict.update()와 차이

```
dict1 = {'color':'red', 'number':'1'}
dict2 = {'size':'small', 'number':'2'}

chain_map = ChainMap(dict1, dict2)
print(chain_map)
dict1.update(dict2)
print(dict1)
```

```
ChainMap({'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
{'color': 'red', 'number': '2', 'size': 'small'}
```

Dict.update()와 차이

```
dict1 = {'color':'red', 'number':'1'}
dict2 = {'size':'small', 'number':'2'}
dict3 = {'animal':'dog', 'number':'3'}
```

```
dict1.update(dict2)
dict1.update(dict3)
print(dict1)
```

```
print(f'dict1.items() datatype : {type(dict1.items())}')
print(dict1.items())
```

```
for k,v in dict1.items():
    print(k,v)
```

```
{'color': 'red', 'number': '3', 'size': 'small', 'animal': 'dog'}
dict1.items() datatype : <class 'dict_items'>
dict_items([('color', 'red'), ('number', '3'), ('size', 'small'), ('animal', 'dog')])
color red
number 3
size small
animal dog
```

dict1

Dict1.items()의 type

Dict1.items()

Dict1.items()의 key, value

Dict.update

Dict.update()와 차이

```
dict1 = {'color':'red', 'number':'1'}
dict2 = {'size':'small', 'number':'2'}
dict3 = {'animal':'dog', 'number':'3'}
```

```
chain_map = ChainMap(dict1, dict2, dict3)
print(chain_map)
```

```
print(f'chain_map.items() datatype : {type(chain_map.items())}')
print(chain_map.items())

for k,v in chain_map.items():
    print(k,v)
```

```
ChainMap({'color': 'red', 'number': '1'}, {'size': 'small', 'number'
: '2'}, {'animal': 'dog', 'number': '3'})
chain_map.items() datatype : <class 'collections.abc.ItemsView'>
ItemsView(ChainMap({'color': 'red', 'number': '3', 'size': 'small',
'animal': 'dog'}, {'size': 'small', 'number': '2'}, {'animal': 'dog'
, 'number': '3'}))
animal dog
number 3
size small
color red
```

Chain_map

Chain_map.items()의 type

Chain_map.items()

Chain_map.items()의 key, value

Chain_map

Dict.update()와 차이

```
for k,v in dict1.items():
    print(k,v)
    print(count)
    count+=1
```

```
color red
0
number 3
1
size small
2
animal dog
3
```

```
for k,v in chain_map.items():
    print(k,v)
    print(count)
    count+=1
```

```
animal dog
0
number 3
1
size small
2
color red
3
```

순서는 섞여 있으나 출력하는 값은 동일
여러 개의 dictionary 데이터를 update하고 key와 value값을 꺼내야 한다면
연결이 더 빠른 chainmap을 사용한 후 chainmap.items를 활용 할 수 있음

.maps

Chainmap의 값을 list로 반환해줌

일반적으로 chainmap data에 접근하고자 할 때 maps를 사용

```
dict1 = {'color':'red', 'number':'1'}
dict2 = {'size':'small', 'number':'2'}

chain_map = ChainMap(dict1, dict2)
print(chain_map)
print(chain_map.maps)
```

```
ChainMap({'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
[{'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'}]
```

.new_child()

cm.new_child(m=None)

ChainMap({}, *cm.maps)

현재의 instance와 동일한 map을 반환

m이 주어지면 map의 앞에 m을 추가한 새로운 map을 반환

Parent 값을 변경하지 않고 사용할 수 있는 map을 만드는 데 사용

.new_child()

```
dict1 = {'color':'red', 'number':'1'}
dict2 = {'size':'small', 'number':'2'}

chain_map = ChainMap(dict1, dict2)
print(f'chain_map : \n{chain_map}')

new_cm1=chain_map.new_child()
print(f'new_cm1 : \n{new_cm1}')
new_cm2=chain_map.new_child(m=1)
print(f'new_cm2 : \n{new_cm2}')
new_cm3=chain_map.new_child(m='test')
print(f'new_cm3 : \n{new_cm3}')
new_cm4=chain_map.new_child('new_child')
print(f'new_cm4 : \n{new_cm4}')
```

```
chain_map :
ChainMap({'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
new_cm1 :
ChainMap({}, {'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
new_cm2 :
ChainMap(1, {'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
new_cm3 :
ChainMap('test', {'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
new_cm4 :
ChainMap('new_child', {'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
```

.parents

cm.parents

ChainMap(*cm.maps)

현재의 instance에서 맨 앞의 것을 제외하고 동일한 map을 반환

맨 앞이 m의 값이 아닌 경우엔 dict하나를 제거하는 것이므로

첫번째 dict가 없는 map이 목적인 경우 유용

.parents()

```
new_cm1=chain_map.new_child()
print(f'new_cm1 : \n{new_cm1}')
new_cm2=chain_map.new_child(m=1)
print(f'new_cm2 : \n{new_cm2}')

new_pm1 = new_cm1.parents
print(f'new_pm1 : \n{new_pm1}')
new_pm2 = new_cm2.parents
print(f'new_pm2 : \n{new_pm2}')
new_pm3 = chain_map.parents
print(f'new_pm3 : \n{new_pm3}')
```

```
new_cm1 :
ChainMap({}, {'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
new_cm2 :
ChainMap(1, {'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
new_pm1 :
ChainMap({'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
new_pm2 :
ChainMap({'color': 'red', 'number': '1'}, {'size': 'small', 'number': '2'})
new_pm3 :
ChainMap({'size': 'small', 'number': '2'})
```

reference

https://docs.python.org/3/library/collections.html#collections.UserDict

https://ash84.io/2018/07/13/chainmap-usage/

Iteration

효율적인 반복에 대한 고찰

여러 가지 iteration 방법

- for문
- List comprehension
- generator
- map
- starmap (itertools)

시간복잡도 비교(append)

n = 100

```
시간복잡도(append)
map: 1.000000000001e-06 seconds
generator: 2.000000000002e-06 seconds
starmap: 4.9999999999980616e-06 seconds
list comprehension: 5.999999999999062e-06 seconds
deque: 1.6000000000002124e-05 seconds
for & append: 2.1000000000000185e-05 seconds
```

n = 10000

```
시간복잡도(append)
generator: 2.000000000002e-06 seconds
map: 2.999999999999531e-06 seconds
starmap: 2.999999999999531e-06 seconds
list comprehension: 0.0005710000000000021 seconds
for & append: 0.0011680000000000024 seconds
deque: 0.00118600000000003 seconds
```

n = 1000000

```
시간복잡도(append)
map: 4.000000000004e-06 seconds
starmap: 6.0000000000060005e-06 seconds
generator: 7.000000000007001e-06 seconds
list comprehension: 0.047458 seconds
deque: 0.09444 seconds
for & append: 0.135905 seconds
```

→ 대체로 map, generator가 가장 빠르고
for문과 deque이 가장 오래 걸림

시간복잡도 비교(sum)

n = 100

```
시간복잡도(sum)
sum: 1.000000000001e-06 seconds
list: 1.1999999999998123e-05 seconds
deque: 1.1999999999998123e-05 seconds
reduce: 1.2999999999999123e-05 seconds
generator: 1.5000000000001124e-05 seconds
```

n = 10000

```
시간복잡도(sum)
sum: 5.6000000000000494e-05 seconds
reduce: 0.00097999999999999948 seconds
list: 0.0010089999999999995 seconds
deque: 0.0012060000000000057 seconds
generator: 0.0016599999999999983 seconds
```

n = 1000000

```
시간복잡도(sum)
sum: 0.005473999999999979 seconds
reduce: 0.06593100000000007 seconds
deque: 0.08004800000000001 seconds
list: 0.08939999999999998 seconds
generator: 0.12432300000000002 seconds
```

→ 대체로 sum이 가장 빠르고
generator가 가장 오래 걸림

시간복잡도 비교(pop)

n = 100

```
시간복잡도(pop)
deque: 1.1000000000000593e-05 seconds
list: 1.4000000000000123e-05 seconds
generator: 1.4000000000000123e-05 seconds
```

n = 10000

```
시간복잡도(pop)
deque: 0.001017000000000004 seconds
list: 0.0010589999999999974 seconds
generator: 0.0015529999999999988 seconds
```

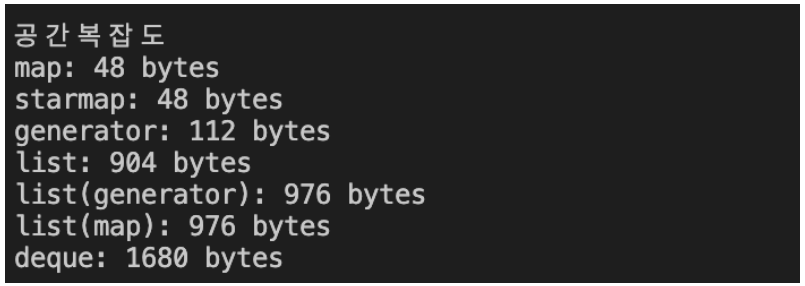
n = 1000000

```
시간복잡도(pop)
list: 0.08920600000000001 seconds
deque: 0.09049299999999993 seconds
generator: 0.11444100000000001 seconds
```

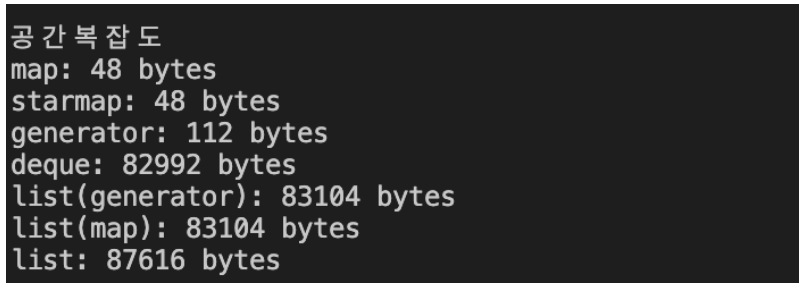
→ 대체로 deque이 가장 빠르고
generator가 가장 오래 걸림

공간복잡도 비교

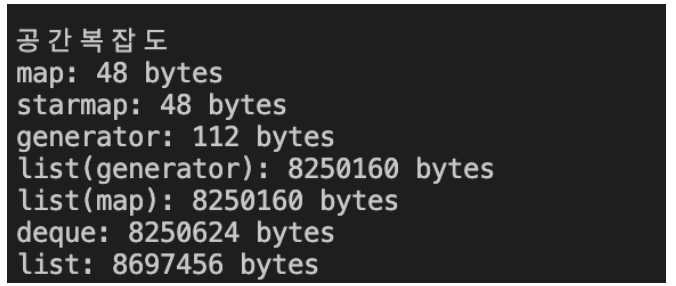
n = 100



n = 10000



n = 1000000



→ 대체로 map, starmap이 메모리를 가장 적게 차지하고 list가 가장 많이 차지함

→ list(generator)와 list(map)의 크기가 같고 일반 list보다는 작음

결과 분석

- List comprehension이 for loop보다 빠른 이유: append 함수 실행시간
- map과 generator가 다른 방식에 비해 메모리를 훨씬 적게 차지하는 이유: 리스트를 메모리에 저장하지 않고 호출될 때 결과값을 반환하기 때문
- 추가할 내용:
  - n 값이 더 커질 경우
  - 시간복잡도가 $O(N^2)$, $O(N^3)$ 인 경우
  - 단순히 i가 아니라 복잡한 함수를 넣는 경우
  - 저장된 값을 변환하는 경우
  - 그래프 그려서 비교

Reference

- https://leadsift.com/loop-map-list-comprehension/ (loop, map, list comprehension 속도 비교)
- https://docs.python.org/ko/3/library/itertools.html (itertools)
- https://www.linkedin.com/pulse/list-comprehension-python-always-faster-than-alex-falkovskiy (list vs list comprehension)
- https://stackoverflow.com/questions/30245397/why-is-a-list-comprehension-so-much-faster-than-appending-to-a-list (list vs list comprehension)