

객체지향 프로그래밍

ITA 파이썬 강좌 9강

객체지향 프로그래밍(Object Oriented Programming)이란?

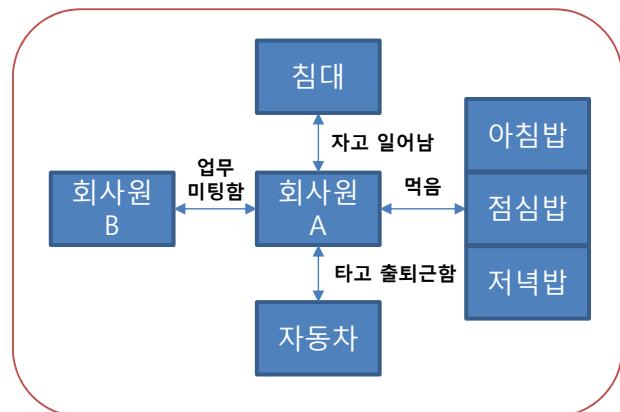
▪ 정의

- 객체 지향 프로그래밍은 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 "객체"들의 모임으로 파악하고자 하는 것이다.

§ 회사원 A의 하루일과

Method

1. 일어난다.
2. 아침 식사를 한다.
3. 출근한다.
4. 회사원 B와 업무 미팅한다.
5. 점심 식사를 한다.
6. 회사원 B와 업무 미팅한다.
7. 저녁 식사를 한다.
8. 퇴근한다.
9. 잠잔다.



클래스(Class)란?

- 정의

- 특정 객체를 생성하기 위해 변수와 메소드를 정의하는 일종의 틀이다.

- 변수 (attribute, state)

- 상태

- 메소드 (method, function)

- 기능

객체(object)

- 속성과 행동을 갖는 그 무엇

- 속성(데이터): 짐작적으로 표현할 수 있는 가치의 데이터들

- 행동(메소드): 객체의 실제 행동, 기능

ex) 학생

- 속성: 나이, 성별, 키, 몸무게, ...

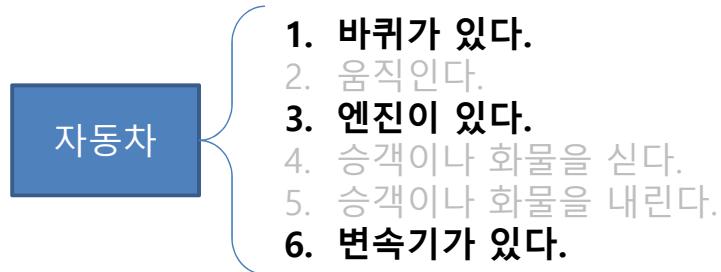
- 행동: 등교, 하교, ...

자동차

1. 바퀴가 있다.
2. 움직인다.
3. 엔진이 있다.
4. 승객이나 화물을 싣다.
5. 승객이나 화물을 내린다.
6. 변속기가 있다.

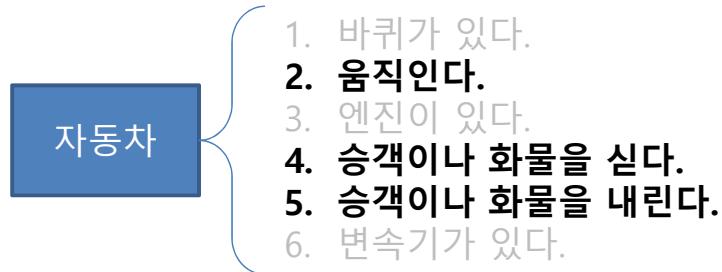
클래스(Class)란?

- 정의
 - 특정 객체를 생성하기 위해 변수와 메소드를 정의하는 일종의 틀이다.
- **변수 (attribute, state)**
 - 상태
- **메소드 (method, function)**
 - 기능



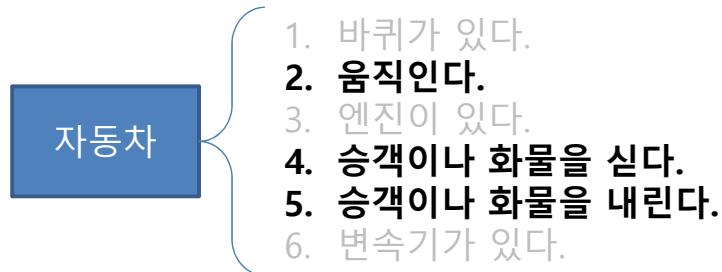
클래스(Class)란?

- 정의
 - 특정 객체를 생성하기 위해 변수와 메소드를 정의하는 일종의 틀이다.
- 변수 (**attribute, state**)
 - 상태
- **메소드 (method, function)**
 - 기능



클래스(Class)란?

- 정의
 - 특정 객체를 생성하기 위해 변수와 메소드를 정의하는 일종의 틀이다.
- 변수 (**attribute, state**)
 - 상태
- 메소드 (**method, function**)
 - 기능



인스턴스(Instance)란?

- 클래스(Class)로부터 만들어지는 객체

Class type Variable



- 클래스(Class)

- 인스턴스(Instance)

클래스 in Python

자동차

1. 바퀴가 있다.
2. 움직인다.
3. 엔진이 있다.
4. 승객이나 화물을 싣다.
5. 승객이나 화물을 내린다.
6. 변속기가 있다.

1. 초기화 (`__init__`)

2. 메소드 구성
3. 인스턴스 생성

class Car:

```
def __init__(self, wheel, engine, transmission):  
    self.wheel = wheel  
    self.engine = engine  
    self.transmission = transmission
```

__init__: 특수한 함
수로 생성자라고 불
리며 처음 instance
가 만들어질 때 호출
된다. 자동 호출

첫 번째에 self 무조건!

멤버 변수 초기화

//this
self → instance 그 자체

self.wheel, self.engine,
self.transmission → 변수
(attribute)

클래스 in Python

자동차

- 1. 바퀴가 있다.
- 2. 움직인다.
- 3. 엔진이 있다.
- 4. 승객이나 화물을 싣다.
- 5. 승객이나 화물을 내린다.
- 6. 변속기가 있다.

1. 초기화 (`__init__`)

2. 메소드 구성

3. 인스턴스 생성

def move(self):

...

def get_in(self):

...

def get_out(self):

...

클래스 in Python

자동차

- 1. 바퀴가 있다.
- 2. 움직인다.
- 3. 엔진이 있다.
- 4. 승객이나 화물을 싣다.
- 5. 승객이나 화물을 내린다.
- 6. 변속기가 있다.

- 1. 초기화 (`__init__`)
- 2. 메소드 구성
- 3. 인스턴스 생성**

```
car1 = Car(2, "gasoline", "auto")
```

instance 생성

```
car2 = Car(4, "diesel", "manual")
```

```
car1.move()
```

'.'을 사용해서 메소드나 변수(**attribute**) 접근 가능

```
print(car2.wheel)
```

상속 (Inheritance)

재사용성, 동일성

- 클래스간의 상하 관계
- A가 B를 포함하는 관계일 때 사용할 수 있다.
- 예를 들어, 동물과 강아지의 관계를 생각해볼 수 있다.

```
class Animal():
```

```
    def __init__(self, is_cute):  
        self.is_cute = is_cute
```

```
class Dog(Animal):
```

```
    def __init__(self, name):
```

```
        Animal.is_cute = True
```

```
        self.name = name
```

다중 상속 문제

Ex)

엄마 IQ 아빠 IQ

```
my_dog = Dog('Nami')
```

아들 IQ : 누구의 IQ?

```
print(my_dog.is_cute)
```

```
print(my_dog.name)
```

상위 클래스의 변수를 사용

할 수 있다.

B is a A : B는 자동차이다

B has a A : B는 푸른 차이다

캡슐화: 속성+메소드

해석의 클래스로
작성

Solve: 멤버 변수를
기본적으로 초기화

오버라이딩

- 메소드가 중복되는 경우, 해당 객체에 가장 우선순위가 높은 메소드가
호출되는 것
- 가장 우선순위가 높은 메소드 외의 다른 메소드가 삭제되는 것은 아님

```
class Animal():
```

```
    def eat(self):
```

```
        print('animal: eat')
```

```
my_dog = Dog('Nami')
```

```
class Dog(Animal):
```

```
    def eat(self):
```

```
        print('dog: eat')
```

```
my_dog.eat()
```

두 eat 함수가 겹치는 것을 확인할 수 있다.

이 경우에 어떤 eat이 불릴까?

my_dog는 Animal보다 Dog에 가까우므로
Dog가 불리게 된다.

super

- 상위 객체의 오버라이딩 당한 메소드를 사용할 때 활용 가능
- 가장 우선순위가 높은 메소드 외의 다른 메소드가 삭제되는 것은 아님
을 확인할 수 있다.

```
class Animal():
```

```
    def eat(self):
```

```
        print('animal: eat')
```

```
my_dog = Dog('Nami')
```

```
my_dog.eat()
```

```
class Dog(Animal):
```

```
    def eat(self):
```

```
        print('dog: eat')
```

```
my_dog.parent_eat()
```

```
    def parent_eat(self):
```

```
        super().eat()
```

my_dog.parent_eat()에서 super()로 인해
상위 객체가 호출되며 상위 객체의 eat() 메소드가 불리게 된다.

추상 클래스

다형성

- 실제로는 불리지 않는 클래스
- 여러 메소드만 가지고 있어 상속받는 클래스에서 **@abstractmethod** 가 붙은 모든 **method**를 구현하여야 한다
- 왜 사용할까? → 상속받는 클래스의 메소드 구현을 강제하기 위함

```
from abc import *          class Dog(Animal):  
  
class Animal(metaclass=ABCMeta):    def eat(self):  
    @abstractmethod  
    def eat(self):                print('dog: eat')  
        pass  
    def fly(self):  
    def fly(self):                print('dog: fly')  
        pass
```

Dog의 eat과 fly를 삭제하면서 확인해보자. 어떤 경우에 Error가 발생하고 왜 발생할까?

Quiz 1.

- **클래스(Class)와 인스턴스(Instance)에 관련된 설명으로 옳은 것을 모두 고르시오.**

1. 아래의 자동차 클래스는 바퀴, 엔진, 변속기라는 변수와 움직임, 상차, 하차라는 메소드로 구성될 수 있다.

2. 이륜자동차 인스턴스를 만들 경우, 바퀴 변수는 2가 된다.

3. 새로운 자동차가 출시되었다면 클래스와 인스턴스를 다시 만들어야 한다.

4. 하나의 클래스로부터 여러 개의 인스턴스가 생성될 수 있다.



- 1. 바퀴가 있다.
- 2. 움직인다.
- 3. 엔진이 있다.
- 4. 승객이나 화물을 싣다.
- 5. 승객이나 화물을 내린다.
- 6. 변속기가 있다.

Quiz 2.

- 아래와 같은 특성을 가진 클래스 Dog을 구현하세요.

- 이름(name), 나이(age), 성별(gender), 소유권자(owner)를 변수(variable)로 갖습니다.
- 초기화 함수를 구현하세요. (이름, 나이, 성별, 소유권자를 모두 초기화해야 합니다.)
- 나이가 1살 증가하는 get_older 메소드를 구현하세요.
- 소유권자가 바뀌는 set_owner 메소드를 구현하세요.

```
Class Dog():
    def __init__(name,
                 age, sex,
                 own):
        self.name=name
        self.age=age
        self.sex=sex
        self.own=own

    def get_older():
        self.age+=1

    def set_owner(own):
        self.own=own
```

Quiz 3.

- 상속, 오버라이딩, **super**, 추상 클래스에 관한 설명으로 옳지 않은 것을 모두 고르세요.

- F 1. 동물-고양이, 사람-남자, 의류-티셔츠, 노트북-핸드폰의 관계는 모두 상 속의 부모, 자식 관계로 표현할 수 있다.
- T 2. A가 부모, B가 자식으로 상속 관계일 때, B의 생성자(`__init__`) 호출 시 A의 생성자(`__init__`)도 호출된다.
- F 3. 추상 클래스의 모든 메소드는 자식 클래스에서 정의되어야 한다.
- F 4. A가 부모, B가 자식으로 상속 관계이며 둘 다 동일한 메소드를 가지고 있을 때, B에서는 A의 메소드를 호출할 수 없다.
- F 5. A가 부모, B가 자식으로 상속 관계이며 A는 `eat(self)`, B는 `eat(self, food)` 메소드를 가지고 있을 때, B의 `eat(self, food)`는 A의 `eat(self)`를 오버라이딩 한다.

2. LinkedList

Array

- 가장 기본적인 자료형 (Data Type)
 - 메모리 공간에 연속적으로 위치
 - 정적 길이 (길이를 바꾸기 쉽지 않다)
 - Ex) 길이가 4인 배열

0	1	2	3						
---	---	---	---	--	--	--	--	--	--

- 중간에 추가하려면, 오른쪽을 전부 옮겨야 한다. Insert의 $T(n) = ?$

배열의 최대 이동

$W(n)$: 첫 번째에 삽입함 때 : $O(n)$

0	4	1	2	3					
---	---	---	---	---	--	--	--	--	--

- Random access가 효율적이다.



0	1	2	3	4	5	6			
---	---	---	---	---	---	---	--	--	--

$O(1)$

Array

- 길이가 매우 큰 배열 중간에 계속 삽입 연산이 일어나면…?
 - 최악의 경우 $O(n)$ 의 삽입이 계속해서 발생
 - 길이가 커질 수록 연속적인 메모리 할당하기도 쉽지 않다.

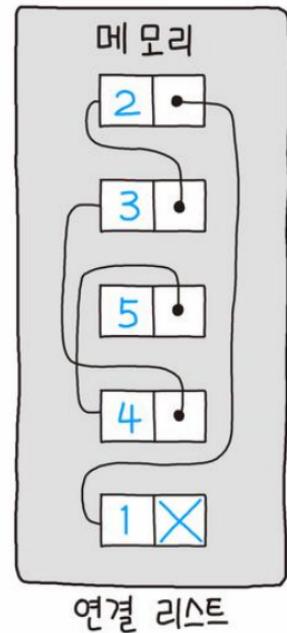
0	1	2	3	4	a	b	c	d	e
---	---	---	---	---	---	---	---	---	---

- 주황색 array에 5를 추가하려면..?
 - 다른 여유 있는 공간 찾아야 한다.
 - 메모리 관리가 효율적이지 않은 OS는 부팅 후 오랜 시간이 지나면 여유 있는 공간 찾기 힘들다.

LinkedList

배열의 단점 보완

- 추상적 자료형(ADT)인 리스트를 구현한 자료구조로, Linked List라는 말 그대로 어떤 데이터 덩어리(이하 노드Node)를 저장할 때 그 다음 순서의 자료가 있는 위치를 데이터에 포함시키는 방식으로 자료를 저장한다. 예를 들어 한 반에 있는 학생들의 자료를 저장한다면, 학생 하나하나의 신상명세 자료를 노드로 만들고, 1번 학생의 신상명세 자료에 2번 학생 신상명세가 어디있는지 표시를 해 놓는 방식이다. 쉽게 생각하면 자료를 비엔나 소시지마냥 줄줄이 엮어놓은 것이다.



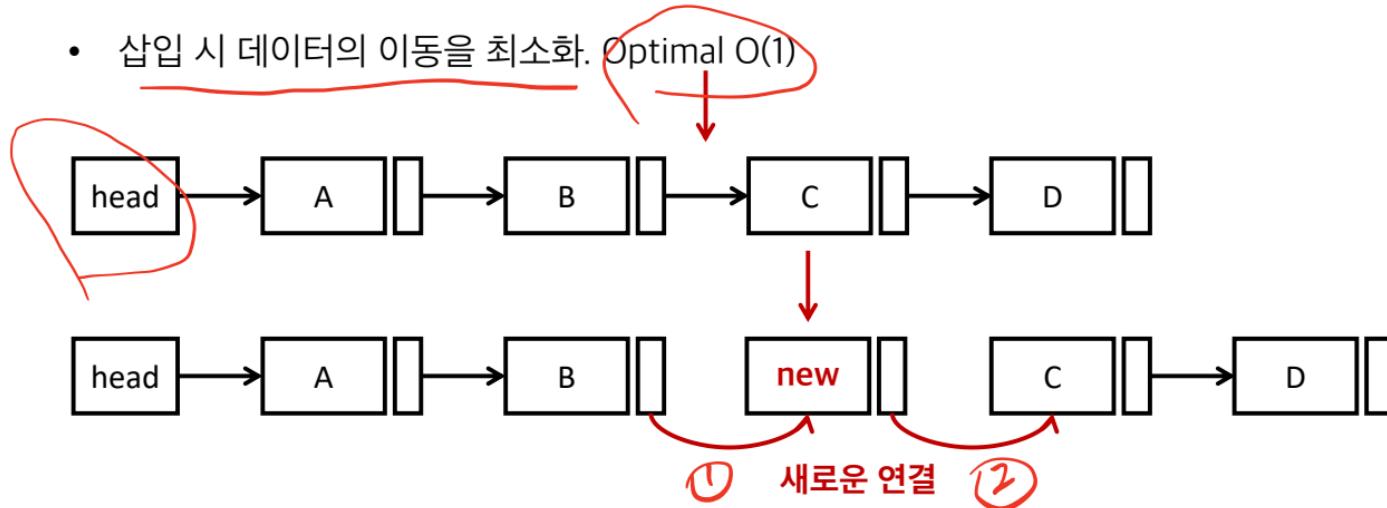
LinkedList

Memory

D		B	C	new		A		head	
---	--	---	---	-----	--	---	--	------	--

- LinkedList의 효율성

- 삽입 시 데이터의 이동을 최소화. Optimal O(1)



LinkedList

Memory

D		B	C	new		A		head	
---	--	---	---	-----	--	---	--	------	--

- LinkedList의 효율성

- 삭제도 간편하다. Optimal O(1)



Python에서는 Garbage Collection을 자동으로 해줌

새로운 연결



B 다음을 D로 바꾸면 합리적.
C 다음은 null이어야 한다.

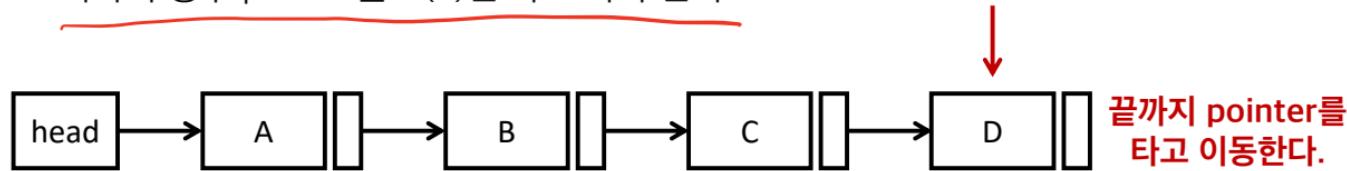
LinkedList

Memory

D		B	C	new		A		head	
---	--	---	---	-----	--	---	--	------	--

- LinkedList의 한계

- 최악의 경우 pointer를 O(n)번 타고 가야 한다.



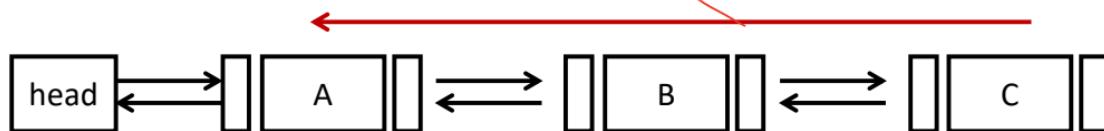
- Head를 잃어버리면…?



Head를 잃어버리면 전부 삭제된다.

Double LinkedList

- 역방향 탐색을 지원한다.
 - Find_node_by_value()로 node를 찾고, 역방향 탐색 가능



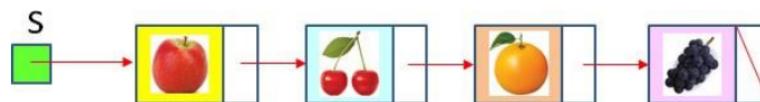
제 2장 연결리스트 (2021년)

리스트

- 일반적인 리스트(List)는 일련의 동일한 타입의 항목(item)들
- 실생활의 예: 학생 명단, 시험 성적, 서점의 신간 서적, 상점의 판매 품목, 실시간 급상승 검색어, 버킷 리스트 등
- 일반적인 리스트의 구현:
 - 1차원 파이썬 리스트(list)
 - 단순연결리스트
 - 이중연결리스트
 - 원형연결리스트

2.1 단순연결리스트

- 단순연결리스트(Singly Linked List)는 동적 메모리 할당을 이용해 리스트를 구현하는 가장 간단한 형태의 자료구조
- 동적 메모리 할당을 받아 노드(node)를 저장하고, 노드는 레퍼런스를 이용하여 다음 노드를 가리키도록 만들어 노드들을 한 줄로 연결시킴



- 연결리스트에서는 삽입이나 삭제 시 항목들의 이동이 필요 없음
- 배열(자바, C, C++언어)의 경우 최초에 배열의 크기를 예측하여 결정해야 하므로 대부분의 경우 배열에 빈 공간을 가지고 있으나, 연결리스트는 빈 공간이 존재하지 않음
- 연결리스트에서는 항목을 탐색하려면 항상 첫 노드부터 원하는 노드를 찾을 때까지 차례로 방문하는 순차탐색 (Sequential Search)을 해야 함

단순연결리스트를 위한 SList 클래스

```
01 class SList:
02     class Node:
03         def __init__(self, item, link):
04             self.item = item
05             self.next = link
06
07     def __init__(self):
08         self.head = None
09         self.size = 0
10
11     def size(self): return self.size
12     def is_empty(self): return self.size == 0
13
14     def insert_front(self, item):
15         if self.is_empty():
16             self.head = self.Node(item, None)
17         else:
18             self.head = self.Node(item, self.head)
19         self.size += 1
20
```

노드 생성자
항목과 다음 노드 레퍼런스

단순연결리스트 생성자
head와 항목 수(size)로 구성

empty인 경우

head가 새 노드 참조

```
21     def insert_after(self, item, p):
22         p.next = SList.Node(item, p.next) ● 새 노드가 p 다음 노드가 됨
23         self.size += 1
24
25     def delete_front(self):
26         if self.is_empty(): ● empty인 경우 에러 처리
27             raise EmptyError('Underflow')
28         else:
29             self.head = self.head.next ● head가 둘째 노드를 참조
30             self.size -= 1
31
32     def delete_after(self, p):
33         if self.is_empty(): ● empty인 경우 에러 처리
34             raise EmptyError('Underflow')
35         t = p.next
36         p.next = t.next ● p 다음 노드를 건너뛰어 연결
37         self.size -= 1
38
```

```
39     def search(self, target):
40         p = self.head
41         for k in range(self.size):
42             if target == p.item: return k
43             p = p.next
44         return None
45
46     def print_list(self):
47         p = self.head
48         while p:
49             if p.next != None:
50                 print(p.item, ' -> ', end=' ')
51             else:
52                 print(p.item)
53             p = p.next
54
55     class EmptyError(Exception):
56         pass
```

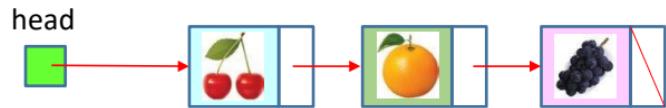
head로부터 순차탐색

탐색 성공

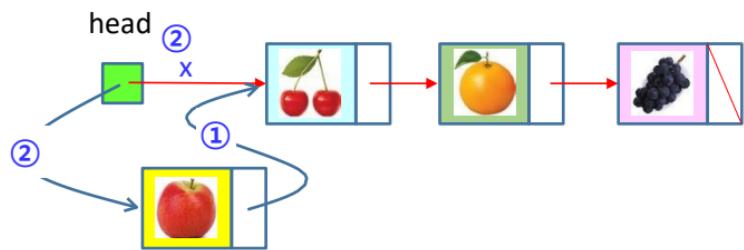
탐색 실패

노드들을 순차탐색

underflow 시 에러 처리



(a) 새 노드 삽입 전



18 self.head = self.Node(item, self.head)

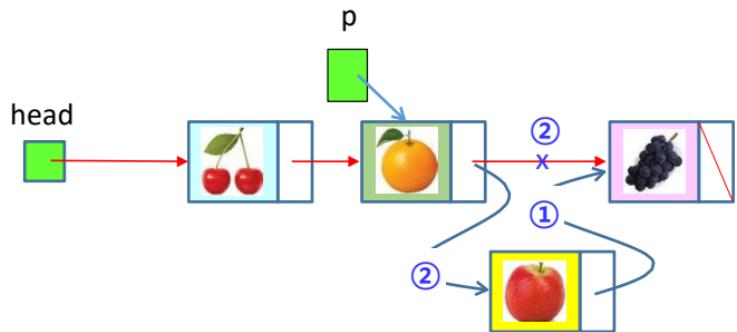
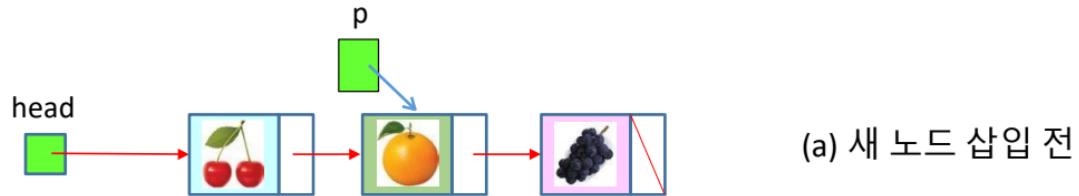
②



①

(b) 새 노드 삽입 후

[그림] insert_front() 함수



22 p.next = SList.Node(item, p.next)

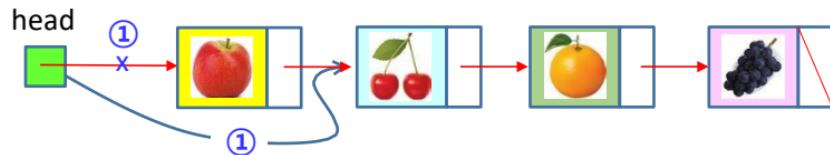


(b) 새 노드 삽입 후

[그림] insert_after() 함수



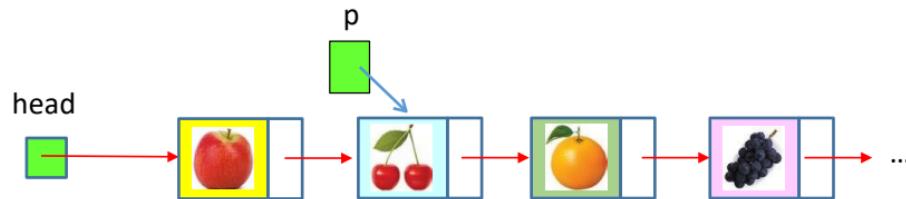
(a) 첫 노드 삭제 전



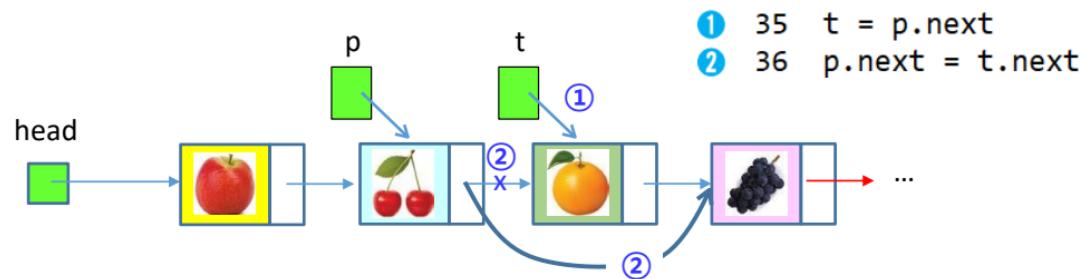
```
29 self.head = self.head.next  
①
```

(b) 첫 노드 삭제 후

[그림] delete_front() 함수



(a) 노드 삭제 전



(b) 노드 삭제 후

[그림] delete_after() 함수

일련의
삽입
삭제
탐색
연산
수행

```
01 from slist import SList
02 if __name__ == '__main__':
03     s = SList()
04     s.insert_front('orange')
05     s.insert_front('apple')
06     s.insert_after('cherry', s.head.next)
07     s.insert_front('pear')
08     s.print_list()
09     print('cherry는 %d번째' % s.search('cherry'))
10    print('kiwi는', s.search('kiwi'))
11    print('배 다음 노드 삭제 후:\t\t', end=' ')
12    s.delete_after(s.head)
13    s.print_list()
14    print('첫 노드 삭제 후:\t\t', end=' ')
15    s.delete_front()
16    s.print_list()
17    print('첫 노드로 망고, 딸기 삽입 후:\t', end=' ')
18    s.insert_front('mango')
19    s.insert_front('strawberry')
20    s.print_list()
21    s.delete_after(s.head.next.next)
22    print('오렌지 다음 노드 삭제 후:\t', end=' ')
23    s.print_list()
```

slist.py에서 SList를 import

이 파일은 파일(모듈)이 메인인가

단순 연결리스트
생성

[자료구조] 실습 2-1(단순 연결리스트)

Console ✘ PyUnit

```
<terminated> main.py [C:\Users\sbbyang\AppData\Local\Programs\Python\Python36-3
pear -> apple -> orange -> cherry
cherry는 3번째
kiwi는 None
배 다음 노드 삭제 후:      pear -> orange -> cherry
첫 노드 삭제 후:          orange -> cherry
첫 노드로 망고, 딸기 삽입 후: strawberry -> mango -> orange -> cherry
오렌지 다음 노드 삭제 후: strawberry -> mango -> orange
```

[자료구조]실습2-1(단순 연결리스트) 수행 결과

2.2 이중연결리스트

- **이중연결리스트(Doubly Linked List)**는 각 노드가 두 개의 레퍼런스를 가지고 각각 이전 노드와 다음 노드를 가리키는 연결리스트
- next
prev
tail
-
- The diagram illustrates a doubly linked list structure. It consists of several nodes, each represented by a yellow vertical rectangle. Between these nodes are light blue horizontal rectangles, which serve as the connection points. Arrows indicate the flow from one node to the next. Specifically, arrows point from the right side of one node's yellow rectangle to the left side of the next node's yellow rectangle, and vice versa. This bidirectional nature is what defines a doubly linked list. Ellipses at both ends of the list indicate that it can have many more nodes.

- 단순연결리스트는 삽입이나 삭제할 때 반드시 이전 노드를 가리키는 레퍼런스를 추가로 알아내야 하고, 역방향으로 노드들을 탐색할 수 없음
- 이중연결리스트는 단순연결리스트의 이러한 단점을 보완하나, 각 노드마다 추가로 한 개의 레퍼런스를 추가로 저장해야 한다는 단점을 가짐

이중연결리스트를 위한 DList 클래스

```
01 class DList:
02     class Node:
03         def __init__(self, item, prev, link):
04             self.item = item
05             self.prev = prev
06             self.next = link
07
08     def __init__(self):
09         self.head = self.Node(None, None, None)
10         self.tail = self.Node(None, self.head, None)
11         self.head.next = self.tail
12         self.size = 0
13
14     def size(self): return self.size
15     def is_empty(self): return self.size == 0
```

노드 생성자
항목과 앞뒤 노드 레퍼런스

이중연결리스트 생성자
head와 tail, 항목 수(size)로 구성

```
17     def insert_before(self, p, item):
18         t = p.prev
19         n = self.Node(item, t, p)●
20         p.prev = n
21         t.next = n
22         self.size += 1
23
24     def insert_after(self, p, item):
25         t = p.next
26         n = self.Node(item, p, t)●
27         t.prev = n
28         p.next = n
29         self.size += 1
30
31     def delete(self, x):
32         -
33
34
35
36
37         return x.item
38
```

The diagram highlights two parts of the code for insertion:

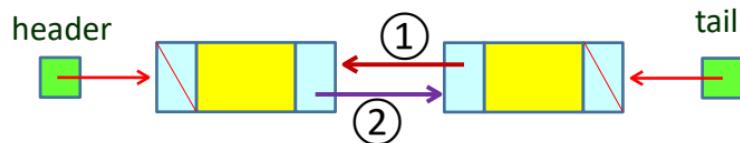
- 새 노드와 앞뒤 노드 연결**: This box covers the assignment statements `p.prev = n`, `t.next = n`, and `self.size += 1`. It indicates that a new node is being connected to its previous and next neighbors, and the size of the list is being updated.
- 새 노드 생성하여 n이 참조**: This box covers the creation of the new node `n` using `n = self.Node(item, t, p)●`. It indicates that a new node is being created and assigned to the variable `n`.

```
39     def print_list(self):
40         if self.is_empty():
41             print('리스트 비어있음')
42         else:
43             p = self.head.next
44             while p != self.tail:
45                 if p.next != self.tail:
46                     print(p.item, ' <=> ', end=' ')
47                 else:
48                     print(p.item)
49             p = p.next
50
51 class EmptyError(Exception):
52     pass
```

노드들을 차례로 방문하기 위해

underflow 시 에러 처리

[자료구조]실습2-2(이중연결리스트)



[그림 2-8] DList 객체 생성

```

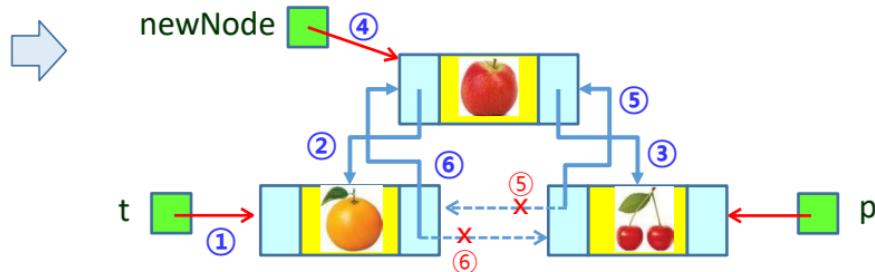
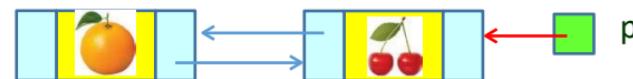
09 self._head = self._Node(None, None, None)
10 self._tail = self._Node(None, self._head, None)
11 self._head._next = self._tail      ①
                                ②

```

```

18     t = p._prev
19     n = self._Node(item, t, p)
20     p._prev = n
21     t._next = n

```

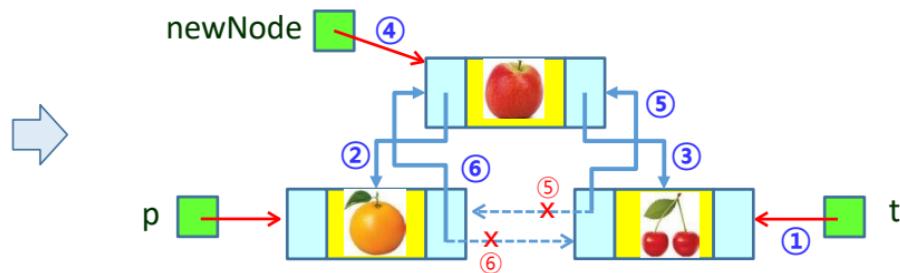


[그림] insert_before()의 삽입 수행

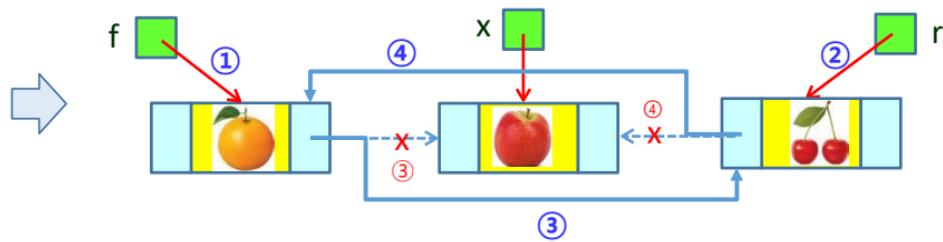
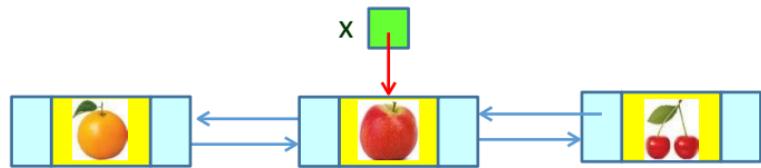
```

25   t = p._next
  1
26   n = self._Node(item, p, t)
  4   2   3
27   t._prev = n
  5
28   p._next = n
  6

```



[그림] insert_after()의 삽입 수행



[그림 2-11] delete()의 삭제 수행

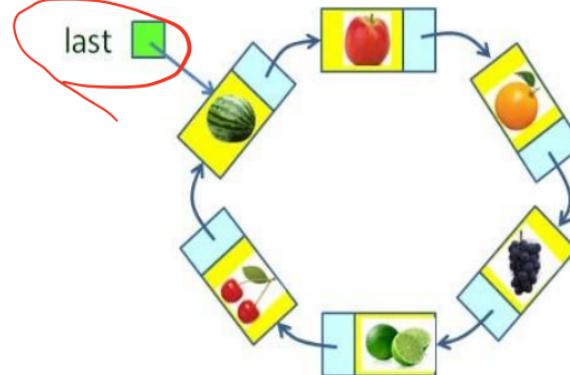
```
이중연결리스트 생성
dlist.py에서 DList를 import
이 파이썬 파일(모듈)이 메인이면
일련의 삽입 삭제 탐색 연산 수행
[자료구조]실습2-2(이중연결리스트)
01 from dlist import DList
02 if __name__ == '__main__':
03     s = DList()
04     s.insert_after(s.head, 'apple')
05     s.insert_before(s.tail, 'orange')
06     s.insert_before(s.tail, 'cherry')
07     s.insert_after(s.head.next, 'pear')
08     s.print_list()
09     print('마지막 노드 삭제 후:\t', end='')
10     s.delete(s.tail.prev)
11     s.print_list()
12     print('맨 끝에 포도 삽입 후:\t', end='')
13     s.insert_before(s.tail, 'grape')
14     s.print_list()
15     print('첫 노드 삭제 후:\t', end='')
16     s.delete(s.head.next)
17     s.print_list()
18     print('첫 노드 삭제 후:\t', end='')
19     s.delete(s.head.next)
20     s.print_list()
21     print('첫 노드 삭제 후:\t', end='')
22     s.delete(s.head.next)
23     s.print_list()
24     print('첫 노드 삭제 후:\t', end='')
25     s.delete(s.head.next)
26     s.print_list()
```

[자료구조]실습2-2(이중연결리스트) 수행 결과

```
Console ✘ PyUnit  
<terminated> main.py [C:\Users\sb.yang\AppData\Local\Programs\Python\Python36-32\python.exe]  
apple <=> pear <=> orange <=> cherry  
마지막 노드 삭제 후: apple <=> pear <=> orange  
맨 끝에 포도 삽입 후: apple <=> pear <=> orange <=> grape  
첫 노드 삭제 후: pear <=> orange <=> grape  
첫 노드 삭제 후: orange <=> grape  
첫 노드 삭제 후: grape  
첫 노드 삭제 후: 리스트 비어있음
```

2-3 원형연결리스트

- 원형연결리스트(Circular Linked List)는 마지막 노드가 첫 노드와 연결된 단순연결리스트
- 원형연결리스트에서는 마지막 노드의 레퍼런스가 저장된 last가 단순연결리스트의 head와 같은 역할



원형연결리스트를 위한 CList 클래스

```
01 class CList:
02     class _Node:
03         def __init__(self, item, link):
04             self.item = item
05             self.next = link
06
07     def __init__(self):
08         self.last = None
09         self.size = 0
10
11     def no_items(self): return self.size
12     def is_empty(self): return self.size == 0
13
14     def insert(self, item):
15         n = self._Node(item, None)
16         if self.is_empty():
17             n.next = n
18             self.last = n
19         else:
20             n.next = self.last.next
21             self.last.next = n
22             self.size += 1
23
```

노드 생성자
항목과 다음 노드 레퍼런스

원형연결리스트 생성자
last와 항목 수(size)로 구성

새 노드 생성하여
n이 참조

새 노드는 자신을 참조하고
last가 새 노드 참조

새 노드는 첫 노드를 참조하고
last가 참조하는 노드와 새 노드 연결

```
24     def first(self):
25         if self.is_empty():
26             raise EmptyError('Underflow')
27         f = self.last.next
28         return f.item
29
30     def delete(self):
31         if self.is_empty():
32             raise EmptyError('Underflow')
33         x = self.last.next
34         if self.size == 1:
35             self.last = None
36         else:
37             self.last.next = x.next
38         self.size -= 1
39         return x.item
40
```

empty 리스트가 됨

last가 참조하는 노드가 두 번째 노드를 연결

```
41     def print_list(self):
42         if self.is_empty():
43             print('리스트 비어있음')
44         else:
45             f = self.last.next
46             p = f
47             while p.next != f:
48                 print(p.item, ' -> ', end=' ')
49                 p = p.next
50             print(p.item)
51
52 class EmptyError(Exception):
53     pass
```

첫 노드가 다시 방문되면
루프 중단

노드들을 차례로 방문하기 위해

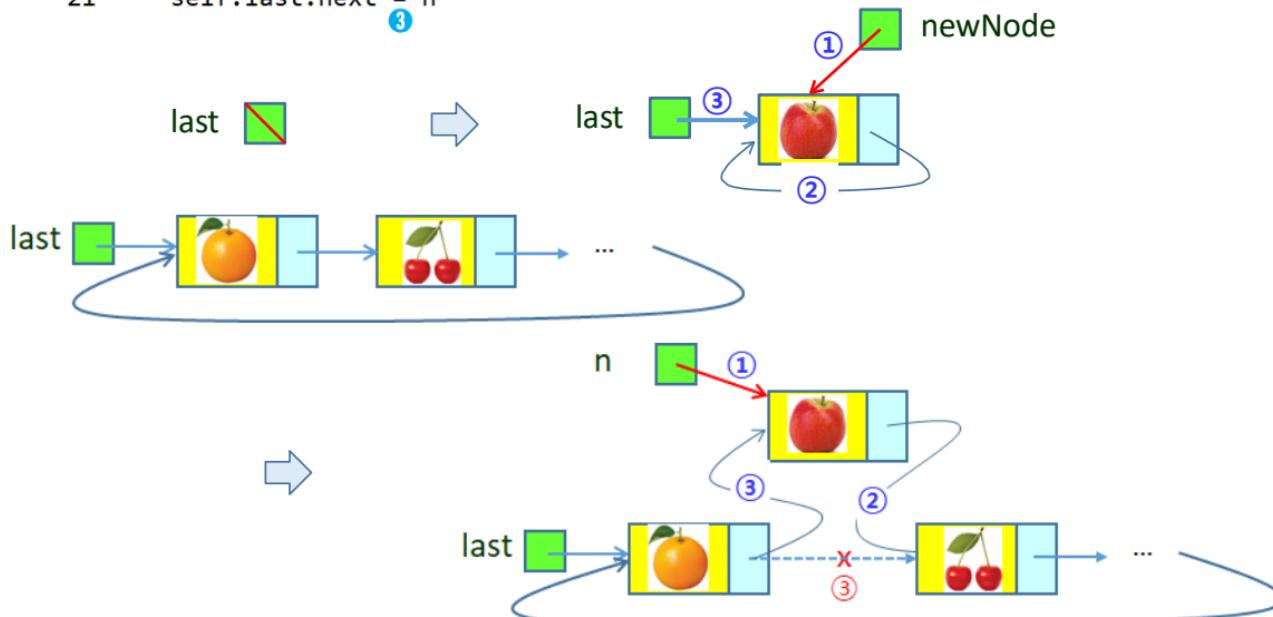
underflow 시 에러 처리

[자료구조]실습2-3(원형 연결리스트)



```
15 n = self.Node(item, None)
16 if self.is_empty():
17     n.next = n
18     self.last = n
19 else:
20     n.next = self.last.next
21     self.last.next = n
```

[그림] insert()의 노드 삽입

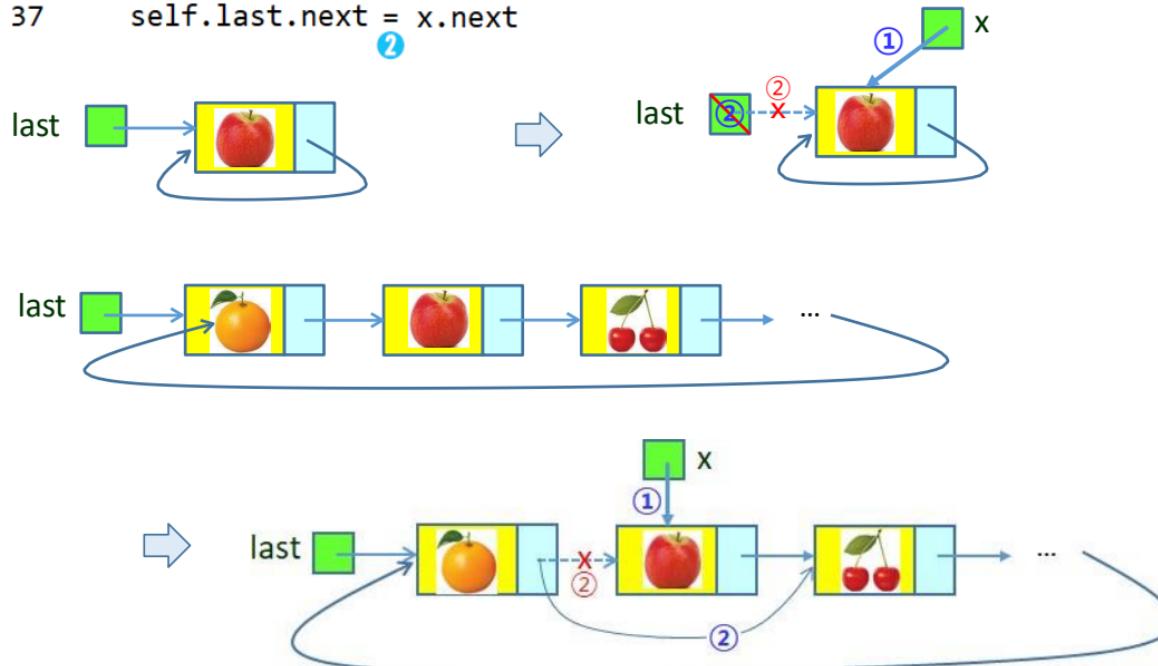


```

33 x = self.last.next
①
34 if self.size == 1:
35     self.last = None
②
36 else:
37     self.last.next = x.next
②

```

[그림 2-15] delete()의 노드 삭제



```
01 from clist import CList
02 if __name__ == '__main__':
03     s = CList()
04     s.insert('pear')
05     s.insert('cherry')
06     s.insert('orange')
07     s.insert('apple')
08     s.print_list()
09     print('s의 길이 = ', s.no_items())
10    print('s의 첫 항목 : ', s.first())
11    s.delete()
12    print('첫 노드 삭제 후: ', end=' ')
13    s.print_list()
14    print('s의 길이 = ', s.no_items())
15    print('s의 첫 항목 : ', s.first())
16    s.delete()
17    print('첫 노드 삭제 후: ', end=' ')
18    s.print_list()
19    s.delete()
20    print('첫 노드 삭제 후: ', end=' ')
21    s.print_list()
22    s.delete()
23    print('첫 노드 삭제 후: ', end=' ')
24    s.print_list()
```

clist.py에서 CList를 import

이 파이썬 파일(모듈)이 메인이라면

원형연결리스트 생성

일련의
삽입
삭제
탐색
연산
수행

[자료구조]실습2-3(원형연결리스트)

[자료구조]실습2-3(원형연결리스트) 수행 결과

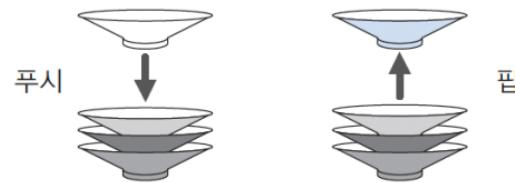
```
Console > PyUnit  
<terminated> main.py [C:\Users\sb.yang\AppData\Local\Programs\Python\Python36-32  
apple -> orange -> cherry -> pear  
s의 길이 = 4  
s의 첫 항목 : apple  
첫 노드 삭제 후: orange -> cherry -> pear  
s의 길이 = 3  
s의 첫 항목 : orange  
첫 노드 삭제 후: cherry -> pear  
첫 노드 삭제 후: pear  
첫 노드 삭제 후: 리스트 비어있음
```

03-1 스택이란?

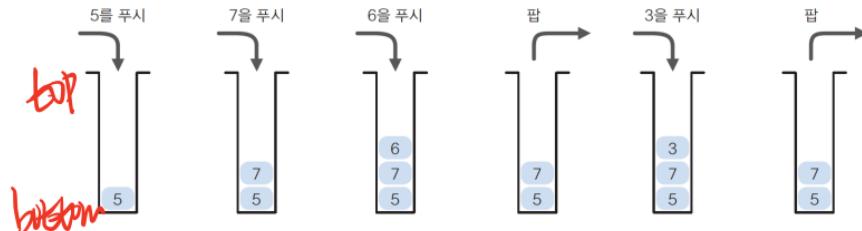
스택 알아보기

스택 stack

- 데이터를 임시 저장할 때 사용하는 자료구조
- 데이터 입력과 출력 순서는 후입선출 LIFO 방식
- 푸시 push: 스택에 데이터를 넣는 작업
- 팝 pop: 스택에서 데이터를 꺼내는 작업
- 꼭대기 top: 푸시하고 팝하는 윗부분
- 바닥 bottom: 푸시하고 팝하는 아랫부분



Lost In
First Out

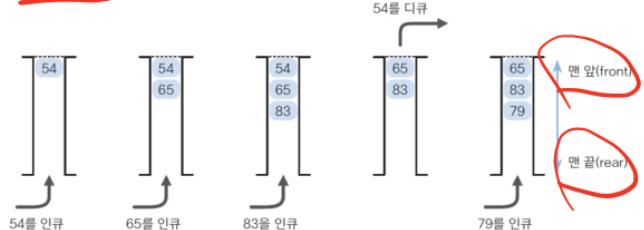


03-2 큐란?

큐 알아보기

큐 queue

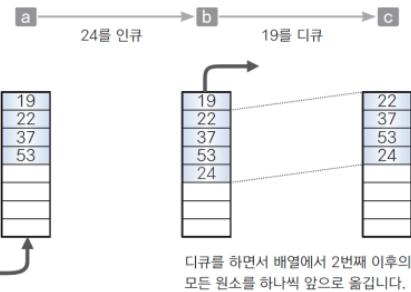
- 스택과 같이 데이터를 임시 저장하는 자료구조
- 가장 먼저 넣은 데이터를 가장 먼저 꺼내는
선입선출(FIFO) 구조
- 인큐 enqueue: 큐에 데이터를 추가하는 작업
- 디큐 dequeue: 큐에서 데이터를 꺼내는 작업
- 프런트 front: 큐에서 데이터를 꺼내는 쪽
- 리어 rear: 큐에 데이터를 넣는 쪽



배열로 큐 구현하기

배열로 큐를 구현한 예

- 24를 인큐하기
 - 맨 끝 데이터 que[3]의 다음 원소인 que[4]에 24를 저장
 - 비교적 적은 비용 cost로 구현 가능
- 19를 디큐하기
 - 19(que[0])를 꺼내고 뒤의 모든 원소를 앞으로 옮겨야 함
 - 데이터를 꺼낼 때마다 이런 처리 작업을 수행해야 한다면 프로그램의 효율성 ↓



03-2 큐란?

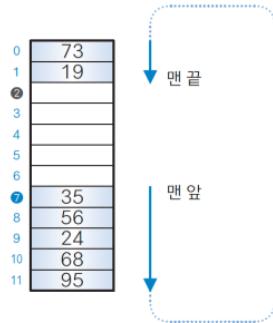
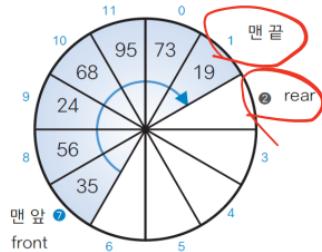
링 버퍼로 큐 구현하기 – (1)

▪ 링 버퍼 ring buffer

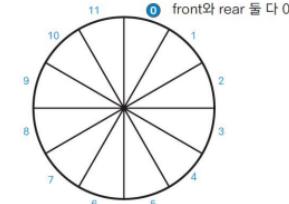
- 배열 맨 끝의 원소 뒤에 맨 앞의 원소가 연결되는 자료구조
- 링 버퍼로 큐를 구현하면 원소를 옮길 필요 없이
front와 rear의 값을 업데이트 하는 것만으로 인큐와 디큐를 수행할 수 있음

● 프런트(front): 맨 앞 원소의 인덱스

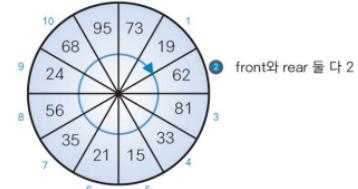
● 리어(rear): 맨 끝 원소 바로 뒤의 인덱스(다음 인큐되는 데이터가 저장되는 위치)



a 비어 있는 큐(no = 0)



b 가득 차 있는 큐(no = 12)

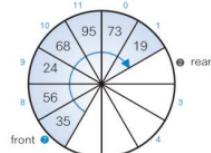


03-2 큐란?

링 버퍼로 큐 구현하기 – (2)

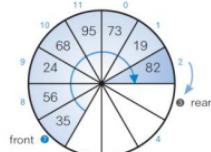
▪ 링 버퍼 ring buffer

a 큐인 상태



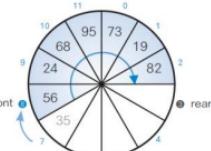
0	73
1	19
2	●
3	●
4	●
5	●
6	●
7	35
8	56
9	24
10	68
11	95

b 82를 인큐



0	73
1	19
2	●
3	●
4	●
5	●
6	●
7	35
8	56
9	24
10	68
11	95

c 35를 디큐



0	73
1	19
2	●
3	●
4	●
5	●
6	●
7	●
8	35
9	56
10	24
11	68

- a : 7개의 데이터 35, 56, 24, 68, 95, 73, 19가 늘어선 순서대로 que[7], que[8], ..., que[11], que[0], que[1]에 저장됨. front 값은 7, rear 값은 2

- b : a에서 82를 인큐한 다음의 상태
맨 끝의 다음에 위치한 que[rear], 즉 que[2]에 82를 저장하고 rear값을 1 증가시켜 3으로 만듦

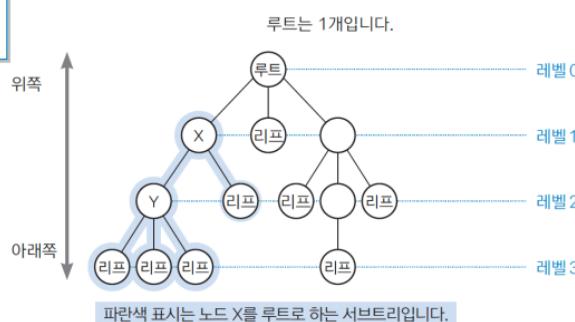
- c : b에서 35를 디큐한 다음의 상태
맨 앞 원소인 que[front], 즉 que[7]의 값인 35를 꺼내고 front값을 1 증가시켜 8로 만듦

04-1 트리 구조

트리의 구조와 관련 용어 – (1)

트리 tree

- 트리는 노드 node 와 가지 edge 로 구성됨
- 각 노드는 가지를 통해 다른 노드와 연결됨



루트 root

- 트리의 가장 위쪽에 있는 노드(1개만 존재)

리프 leaf

- 가장 아래쪽에 있는 노드
- 단말 노드 terminal node, 외부 노드 external node라고도 함
- 물리적으로 가장 밑에 위치하는 것이 아닌 가지가 더 이상 뻗어나갈 수 없는 마지막에 노드가 있다는 의미

비단말 노드 non-terminal node

- 리프를 제외한 노드(루트 포함)
- 내부 노드 internal node라고도 함

자식 child

- 어떤 노드와 가지가 연결되었을 때 아래쪽 노드
- 노드는 자식을 여러 개 가질 수 있음(리프는 자식 없음)

04-1 트리 구조

트리의 구조와 관련 용어 – (2)

■ 트리 tree

- 부모 parent
 - 어떤 노드와 가지가 연결되었을 때 가장 위쪽에 있는 노드
 - 노드의 부모는 하나 (루트는 부모 없음)
- 형제 sibling
 - 부모가 같은 노드
- 조상 ancestor
 - 어떤 노드에서 위쪽으로 가지를 따라가면 만나는 모든 노드
- 자손 descendant
 - 어떤 노드에서 아래쪽으로 가지를 따라가면 만나는 모든 노드

▪ 레벨 level

- 루트에서 얼마나 멀리 떨어져 있는지를 나타내는 것
- 루트의 레벨은 0이고, 가지가 하나씩 아래로 뻗어 내려갈 때마다 레벨이 1씩 증가

▪ 차수 degree

- 각 노드가 갖는 자식의 수
- 모든 노드의 차수가 n 이하인 트리를 n 진 트리라고 함

▪ 높이 height

- 루트에서 가장 멀리 있는 리프까지의 거리(레벨 최댓값)

▪ 서브트리 subtree

- 어떤 노드를 루트로 하고 그 자손으로 구성된 트리

▪ 빈 트리 None tree

- 노드와 가지가 전혀 없는 트리(널 트리 null tree라고도 함)

//depth

04-1 트리 구조

순서 트리와 무순서 트리

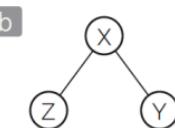
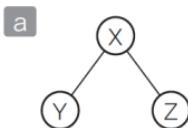
순서 트리 ordered tree

- 형제 노드의 순서 관계가 있는 트리

BST

무순서 트리 unordered tree

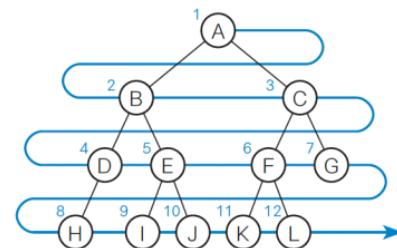
- 형제 노드의 순서 관계를 구별하지 않는 트리



순서 트리의 검색 – (1)

너비 우선 검색 breadth-first search

- 폭 우선 검색, 가로 검색, 수평 검색
- 낮은 레벨부터 왼쪽에서 오른쪽으로 검색
- 한 레벨에서 검색을 마치면 다음 레벨로 내려가는 방법



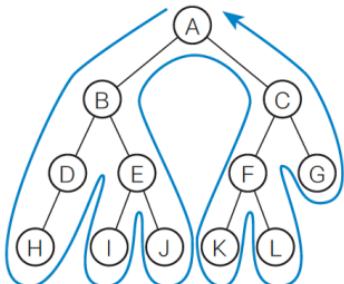
A → B → C → D → E → F → G → H → I → J → K → L

04-1 트리 구조

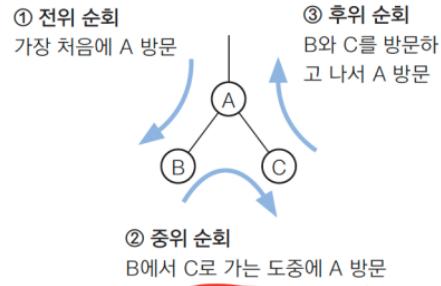
순서 트리의 검색 – (2)

깊이 우선 검색(depth-first search)

- 세로 검색, 수직 검색
- 리프에 도달할 때까지 아래쪽으로 내려가면서 검색하는 것을 우선으로 하는 방법
- 리프에 도달해서 더 이상 검색할 곳이 없으면 일단 부모 노드로 돌아가고 그 다음 자식 노드로 내려감



깊이 우선 검색과 스캔 방법



- A에서 B로 내려가기 직전에 A를 1번 지나갑니다.
- B에서 C로 지나가는 도중에 A를 1번 지나갑니다.
- C에서 A로 돌아올 때 A를 1번 지나갑니다.

- 각 노드를 최대 3번 지나감

04-1 트리 구조

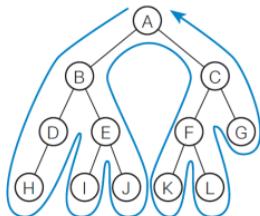
순서 트리의 검색 – (3)

▪ 깊이 우선 검색(depth-first search)

- 어느 시점에 실제로 노드를 방문하는지에 따라 세 종류의 스캔 방법으로 구분

1. 전위 순회 preorder

노드 방문 → 왼쪽 자식 → 오른쪽 자식



- A → B → D → H → I → J → C → F → K → L → G

2. 중앙 순회 inorder

왼쪽 자식 → 노드 방문 → 오른쪽 자식

- H → D → B → I → E → J → A → K → F → L → C → G

3. 후위 순회 postorder

왼쪽 자식 → 오른쪽 자식 → 노드 방문

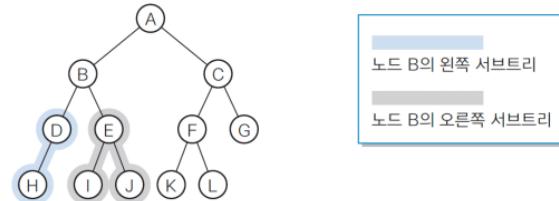
- H → D → I → J → E → B → K → L → F → G → C → A

04-2 이진 트리와 이진 검색 트리

이진 트리 알아보기

이진 트리 binary tree

- 노드가 왼쪽 자식 left child과 오른쪽 자식 right child만을 갖는 트리 (두 자식 가운데 하나 또는 둘 다 존재하지 않는 노드가 있어도 상관 없음)

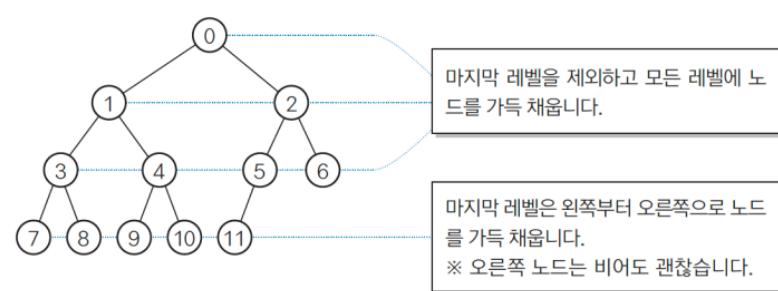


- 왼쪽 자식과 오른쪽 자식을 구분하는 특징
- 왼쪽 서브트리 left subtree
 - 왼쪽 자식을 루트로 하는 서브트리
- 오른쪽 서브트리 right subtree
 - 오른쪽 자식을 루트로 하는 서브트리

완전 이진 트리 알아보기

완전 이진 트리 complete binary tree

- 루트부터 아래쪽 레벨로 노드가 가득 차 있고, 같은 레벨 안에서 왼쪽부터 오른쪽으로 노드가 채워져 있는 이진 트리



04-2 이진 트리와 이진 검색 트리

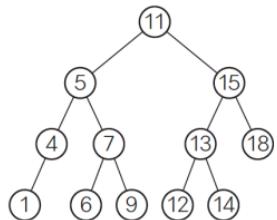
이진 검색 트리 알아보기

■ 이진 검색 트리 binary search tree

- 이진 검색 트리 노드 조건

- 원쪽 서브트리 노드의 키값은 자신의 노드 키값보다 작아야 함
- 오른쪽 서브트리 노드의 키값은 자신의 노드 키값보다 커야 함

- 키값이 같은 노드는 복수로 존재하지 않음



- 이진 검색 트리가 알고리즘에서 폭넓게 사용되는 이유

- 구조가 단순함
- 중위 순회의 깊이 우선 검색을 통하여 노드값을 오름차순으로 얻을 수 있음
- 이진 검색과 비슷한 방식으로 아주 빠르게 검색 가능
- 노드를 삽입하기 쉬움

Left < parent < Right

- 중위 순회의 깊이 우선 검색 시 키값 오름차순 나열

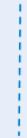
```
1 → 4 → 5 → 6 → 7 → 9 → 11 → 12 → 13 → 14 → 15 → 18
```



감사합니다

실패하더라도 무언가 배우는 것이
아무것도 시도하지 않는 것보다 낫다.

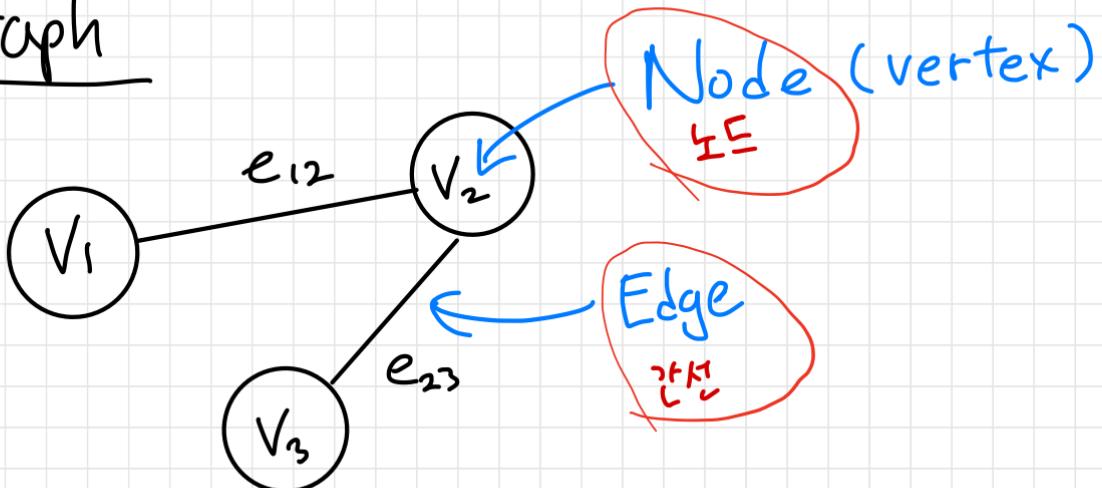
You are better off trying something and having it not work
and learning from that than not doing anything at all.



마크 저커버그

Mark Zuckerberg, 페이스북 설립자

Graph

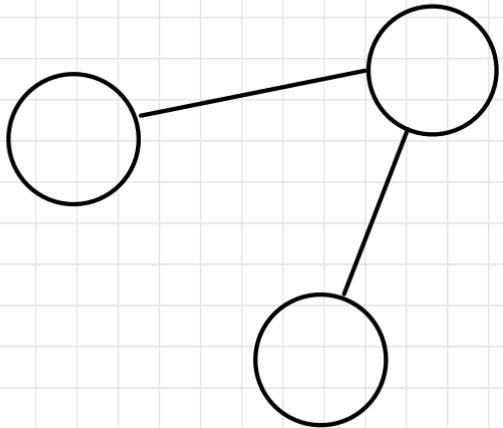


$$G = (V, E)$$

\downarrow

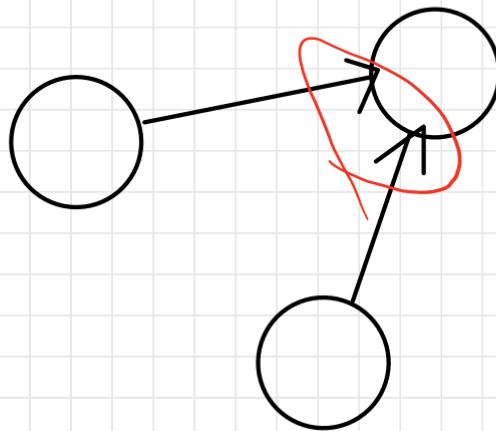
$$\{e_{12}, e_{23}\}$$
$$\{v_1, v_2, v_3\}$$

vertex, Edge



무방향 그래프

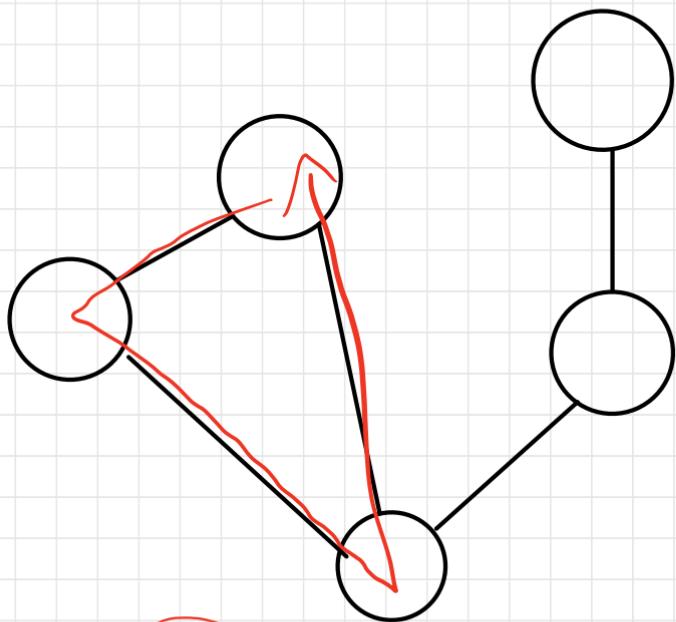
(페이스북 친구, 상하수도망)



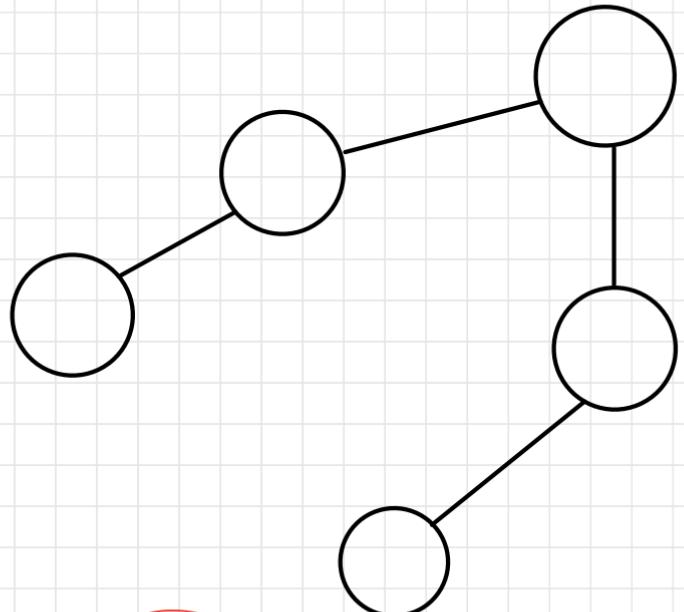
방향 그래프

(ex: 일방 통행, 팔로우)

순환 그래프



순환 그래프 (cyclic)
Cycle 존재

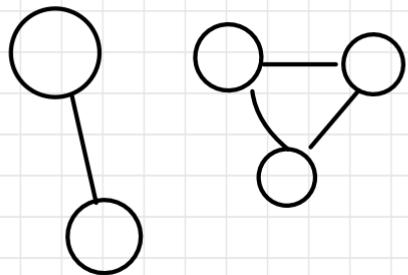


비순환 그래프 (acyclic)

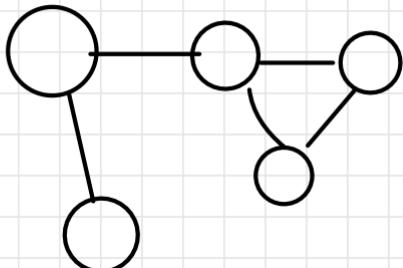
트리 (Tree)

Graph

1) 연결 그래프

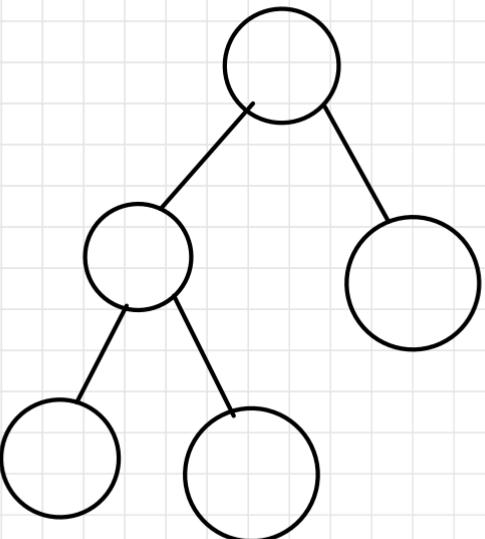


연결 그래프 X



연결 그래프

2) 비순환 그래프
(acyclic)

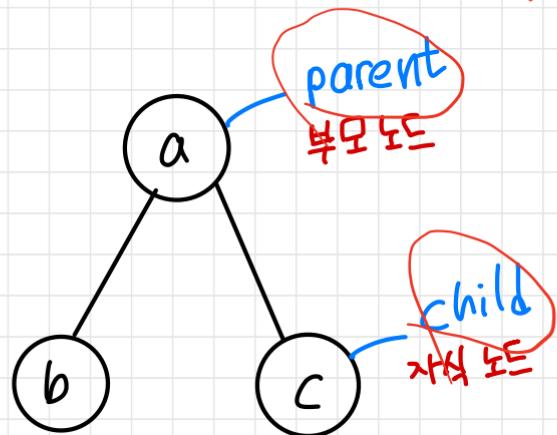


EPI (Tree)

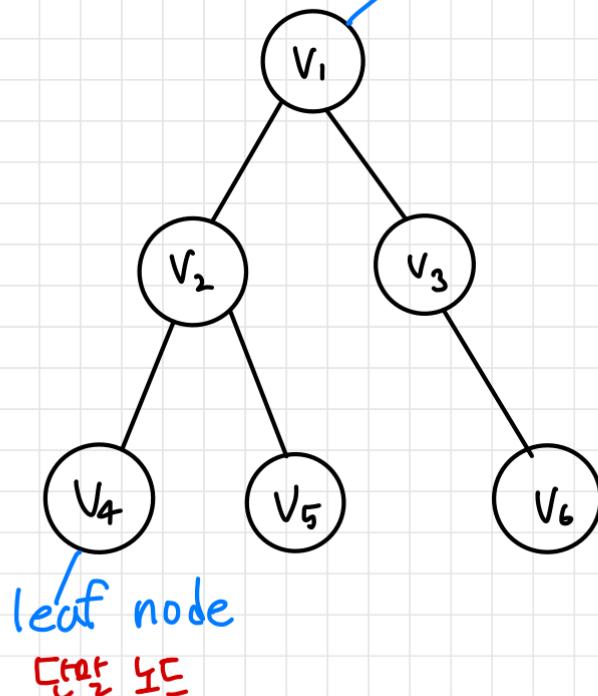
연결 + 비순환

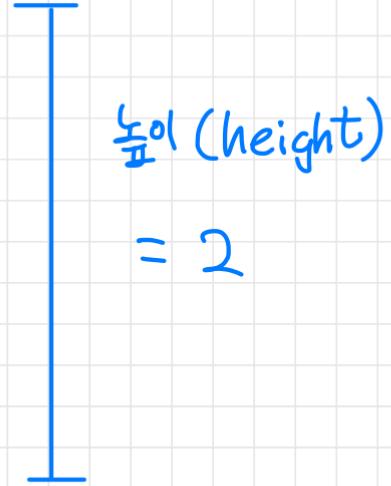
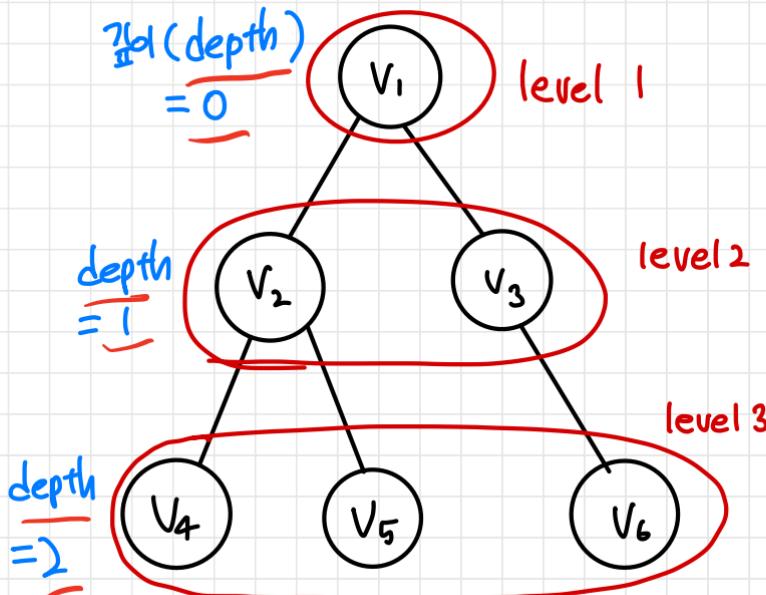
루트가 있는 모든 선관X

root node
루트 노드



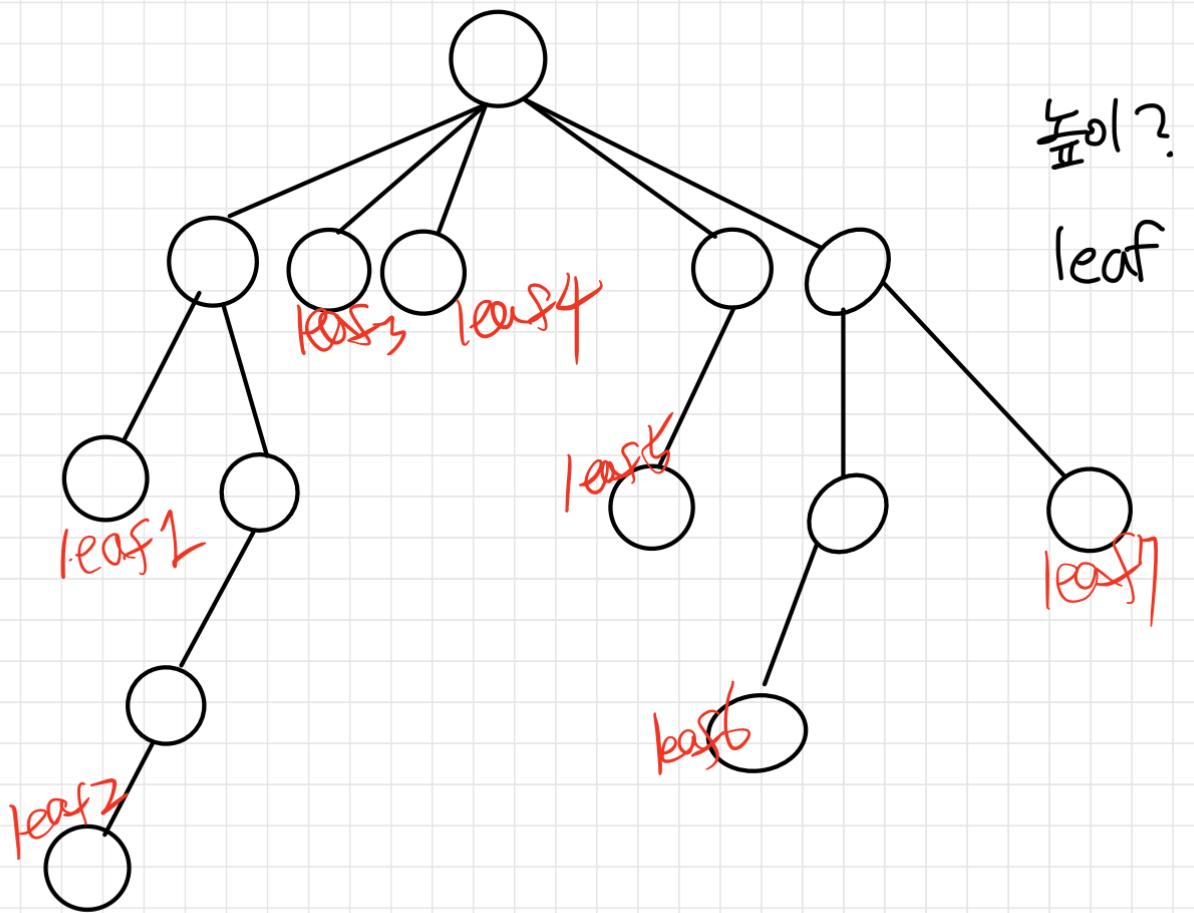
a, b, c : node의 key
(value)





깊이 (Depth) : root에서부터 특정 node 까지에
 존재하는 edge의 개수.

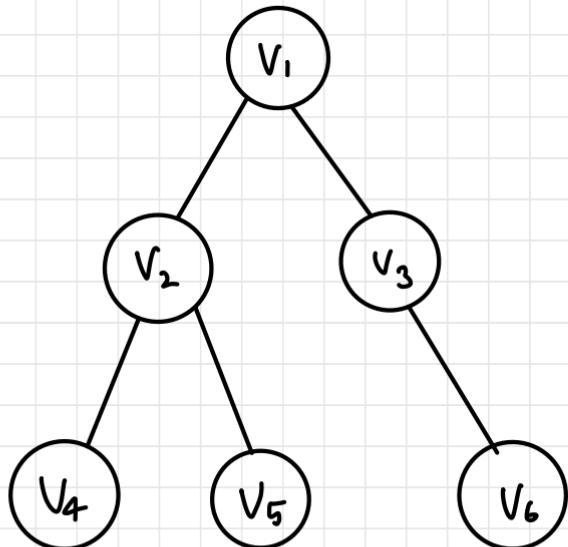
높이 (height) : root에서 가장 먼 node의 깊이.



노드? 4

leaf node? 7

이진 트리 (Binary Tree)

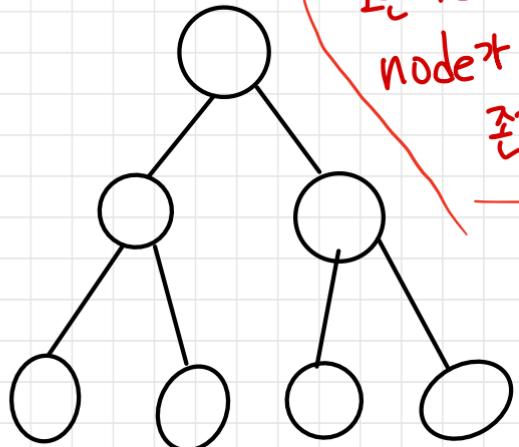


최대 2개의 자식 노드를 가지는 트리.

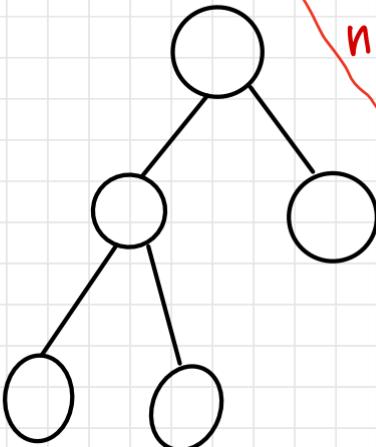
왼쪽 자식과 오른쪽 자식을 구분한다.



이진 트리의 종류



모든 level에
node가 꽉 차워져
존재.



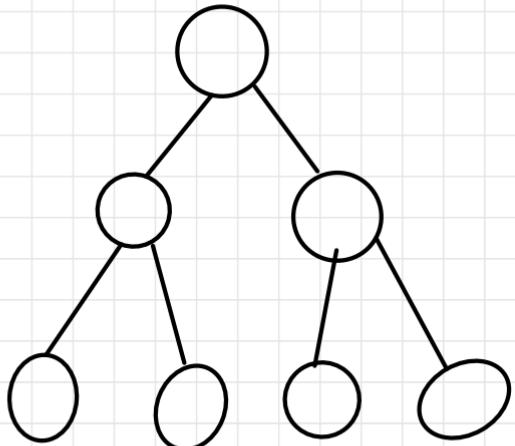
마지막 level 전까지
node가 꽉 차워짐.

포화이진 트리 (full binary tree)

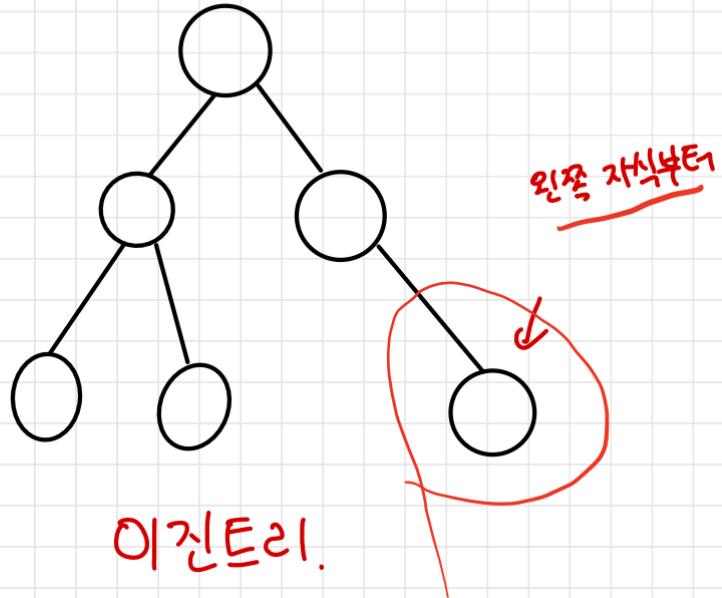
각 level node 수: 2^{depth}



완전 이진트리 (complete ")

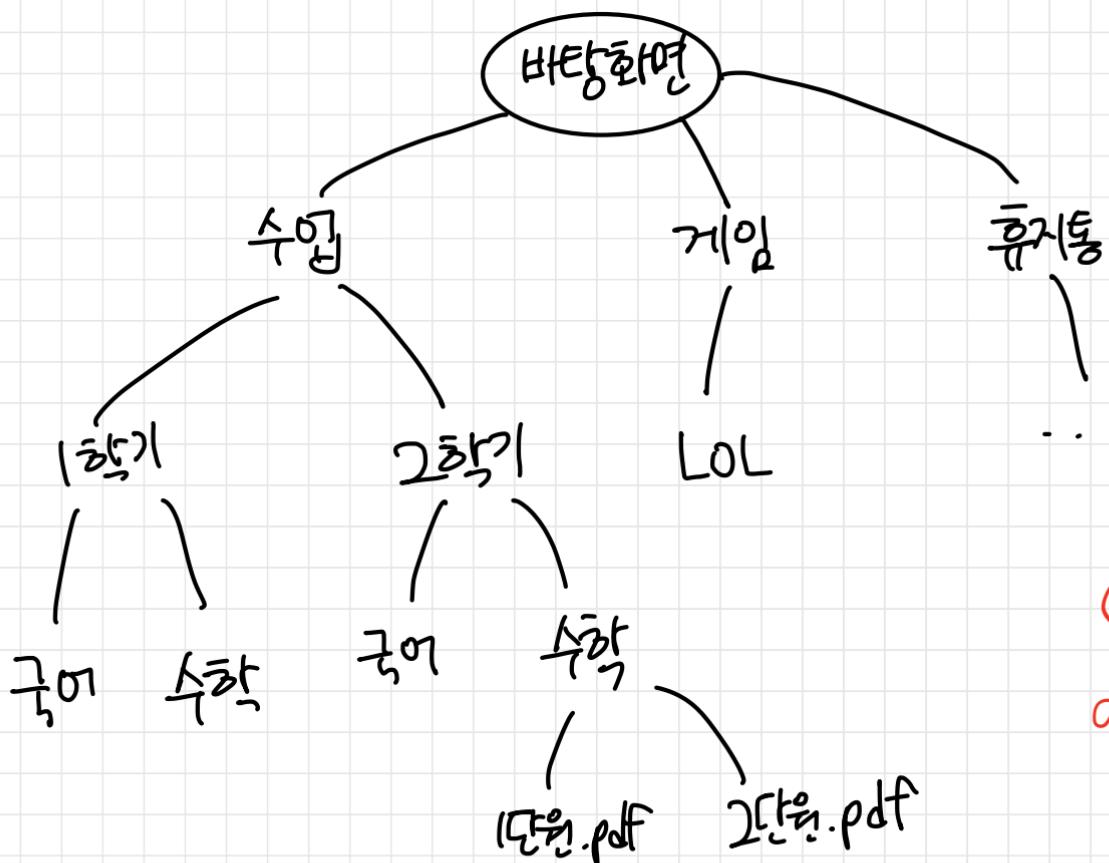


포함 & 완전 이진트리.

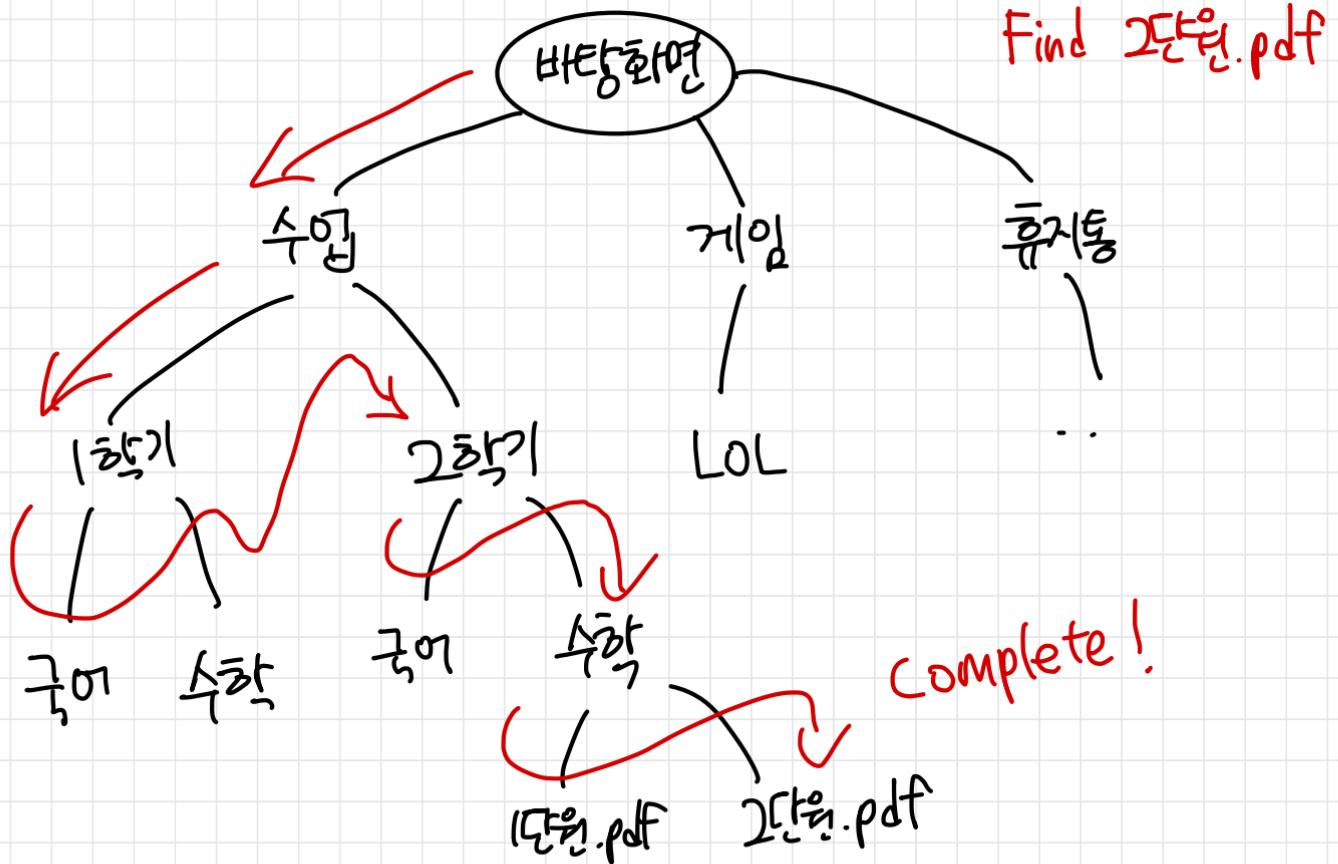


이진트리.

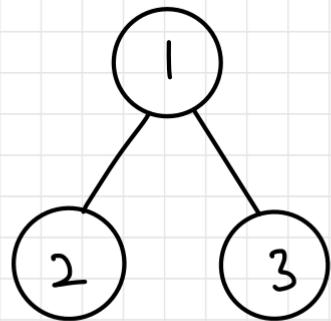
완전 이진트리 X



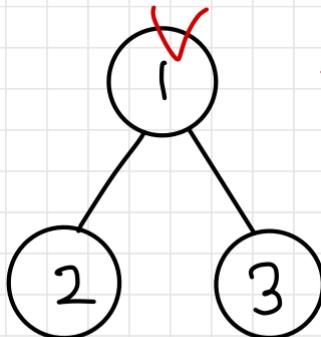
Cycle 종지 X
예외) 바로가기



이진트리 순회하기



이진트리

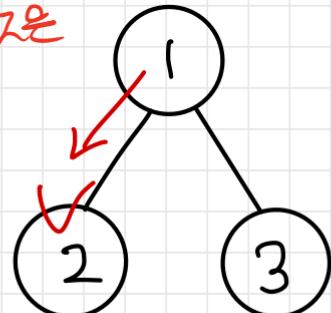


1번 node 방문

2번을 가려면 1번을 방문해야한다.

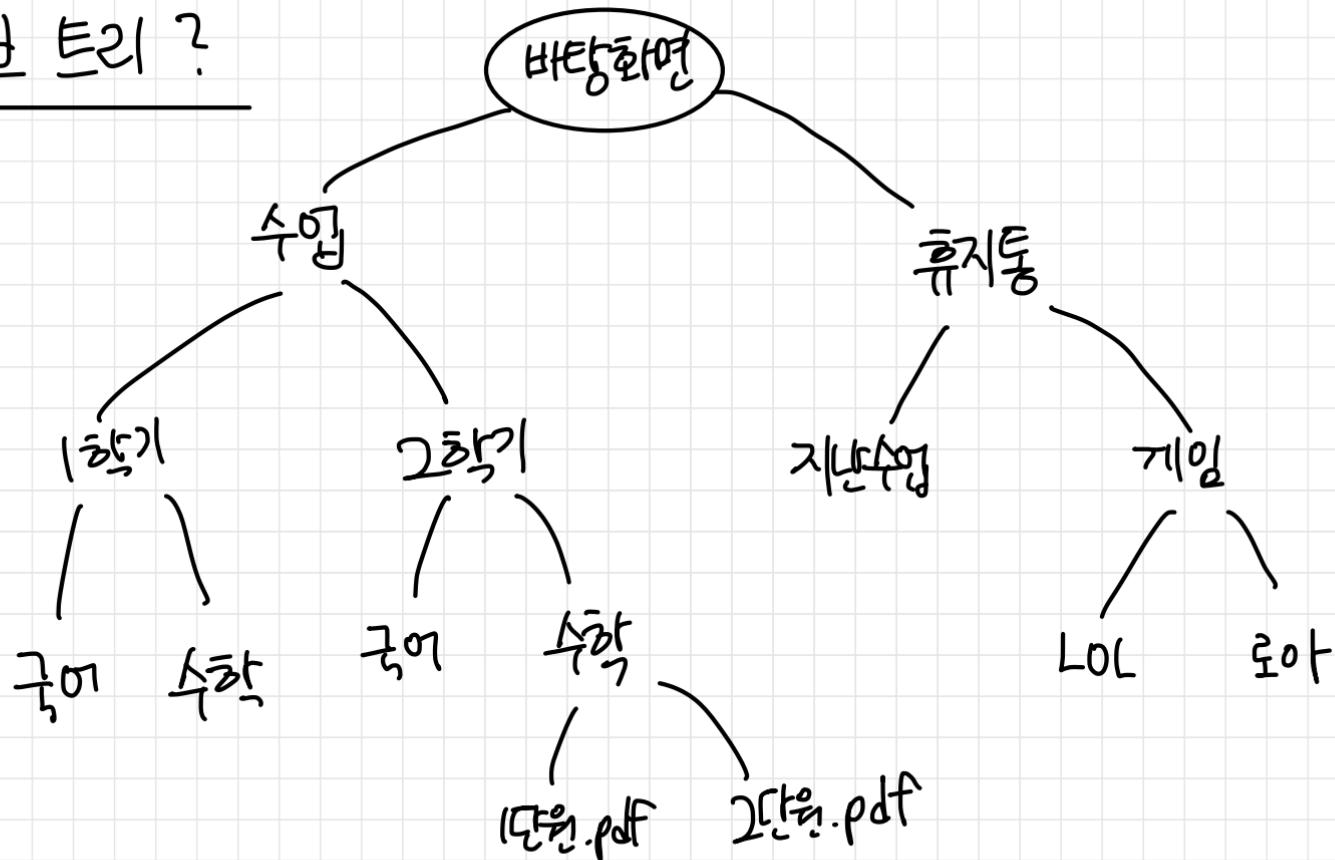
T/F ? F

지나가는 것은
방문 X

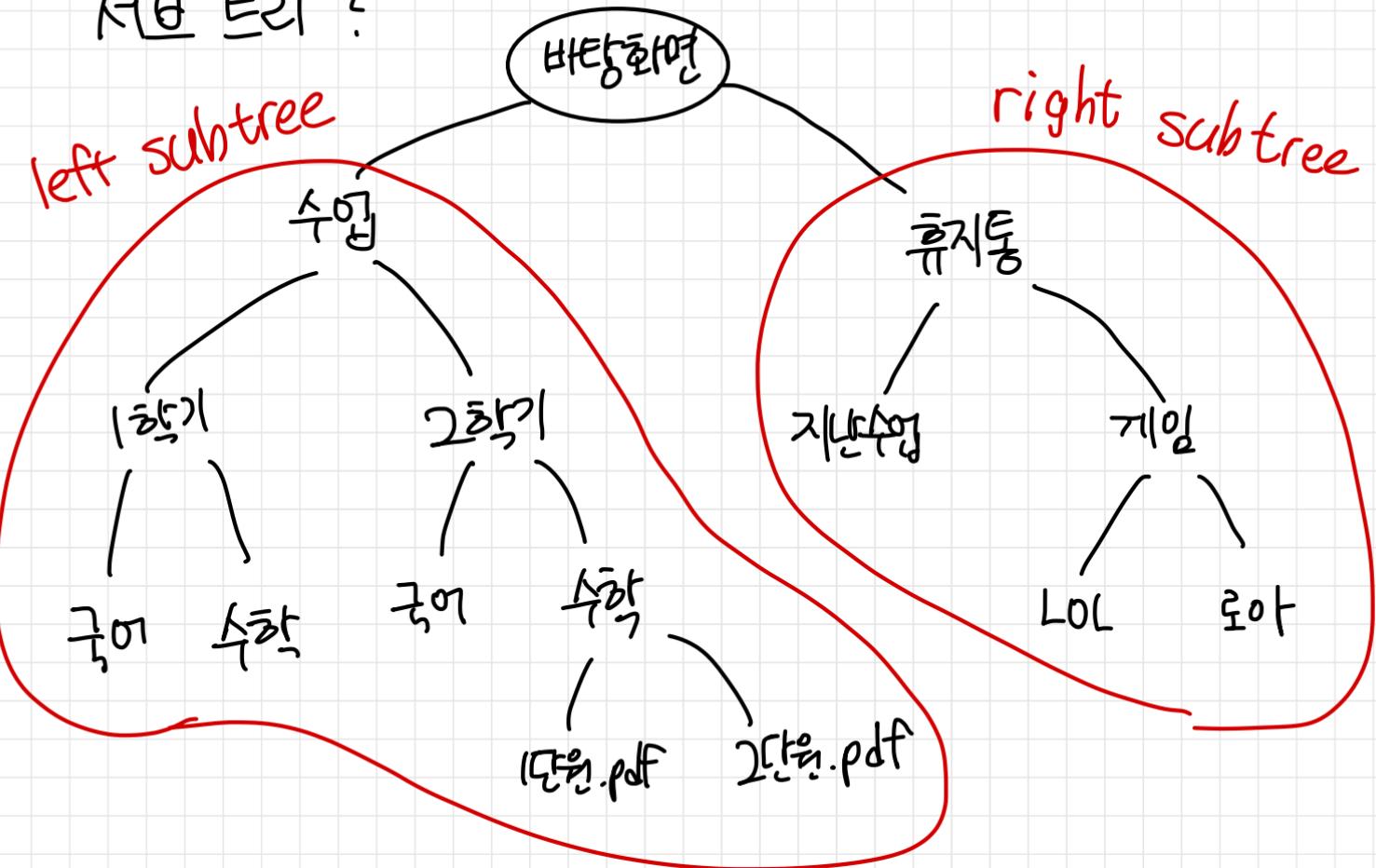


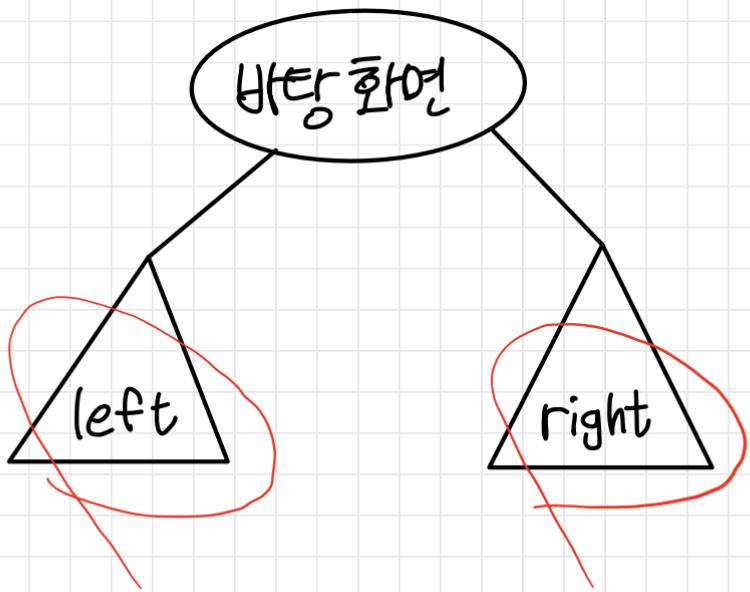
2번 node 부터 방문

서브 트리 ?

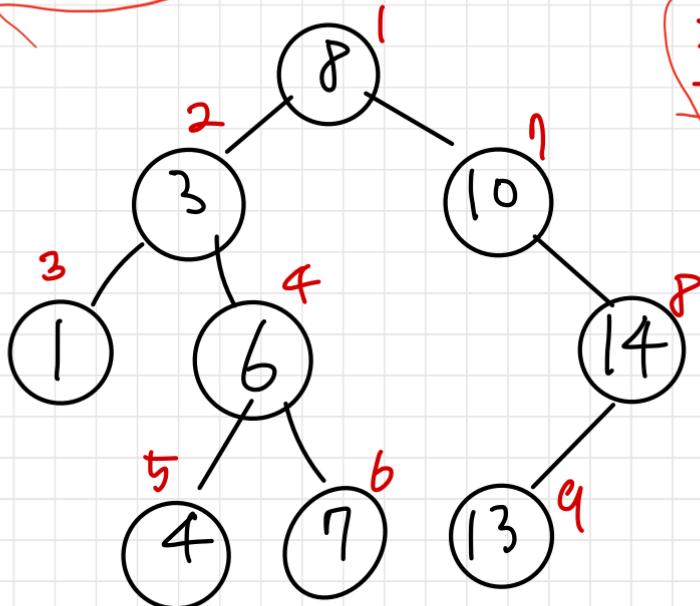


서브 트리 ?





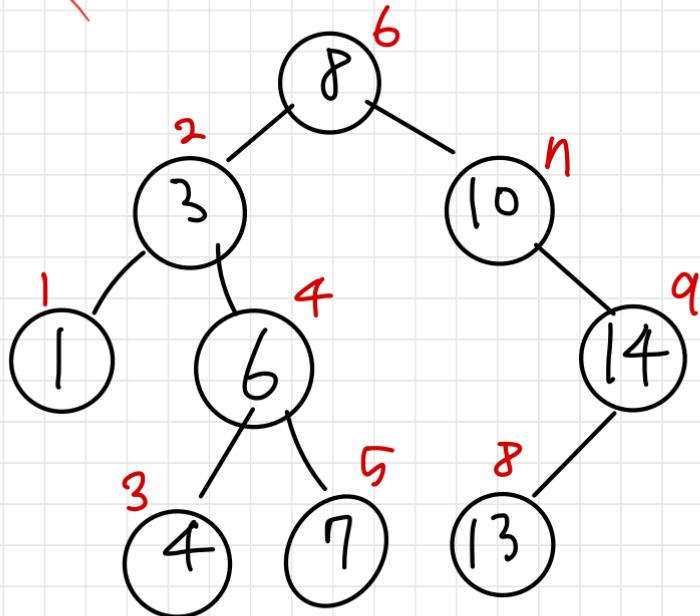
전위 순회 (Preorder traversal) parent 현재 순회



자기 자신 → 왼쪽 자식 → 오른쪽 자식

8 → 3 → 1 → 6 → 4 → 7 → 10 → 14 → 13

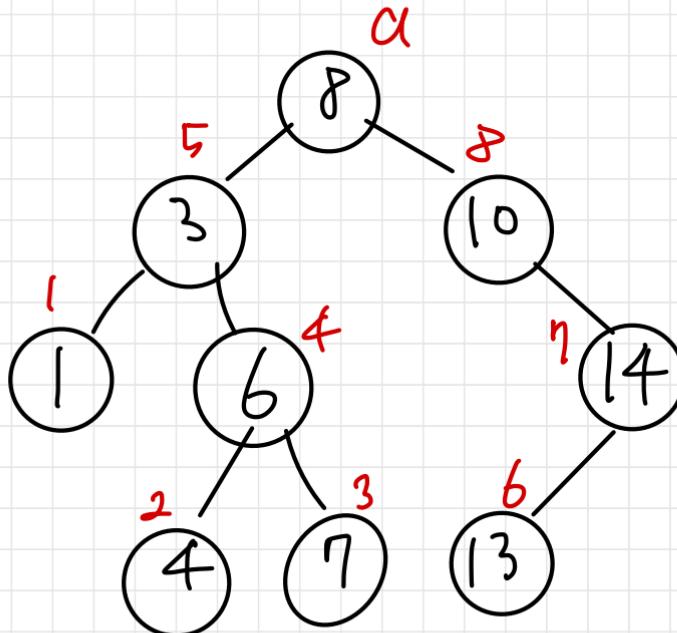
중위 순회 (Inorder traversal)



왼쪽 자식 → 자기 자신 → 오른쪽 자식

1 → 3 → 4 → 6 → 7 → 8 → 10 → 13 → 14

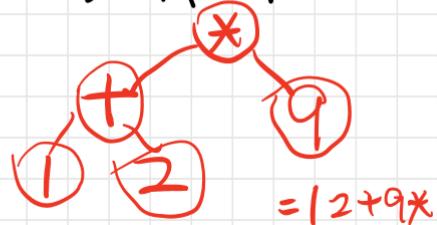
후위 순회 (Postorder traversal)



왼쪽 자식 → 오른쪽 자식 → 자기 자신

1 → 4 → 7 → 6 → 3 → 13 → 14 → 10 → 8

$$\begin{aligned} & 1 \ 2 + \\ & = 1 + 2 \end{aligned}$$

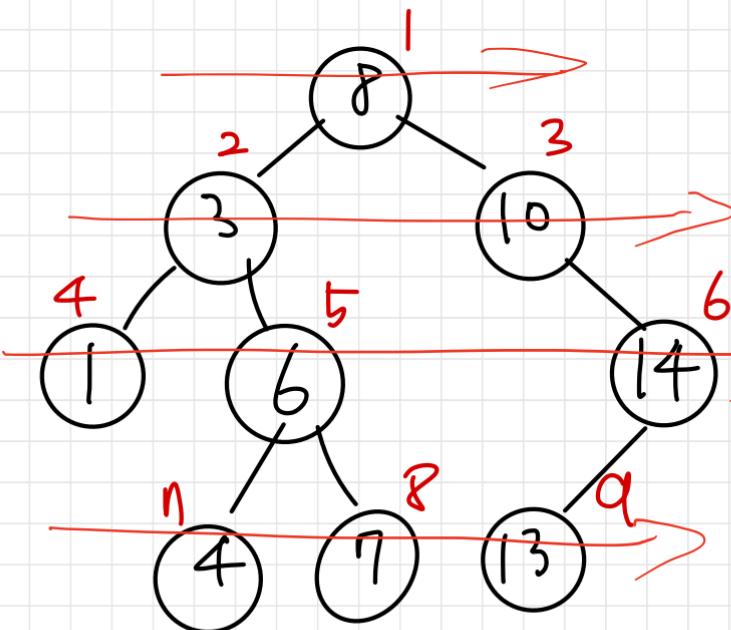


사칙연산을 Postorder 순으로 하면
컴퓨터가 이해할 수 있는 형태로 바뀐다

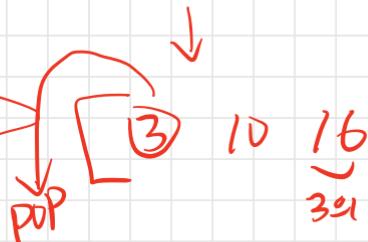
정가운데 순회 순회 (level-order traversal)

Breadth - First - Search

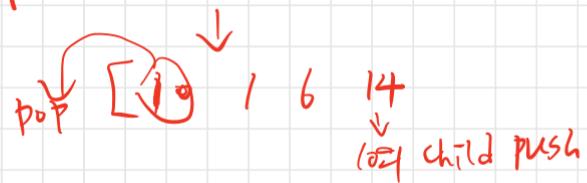
8 → 3 → 10 → 1 → 6 → 14 → 4 → 7 → 13



8's child push

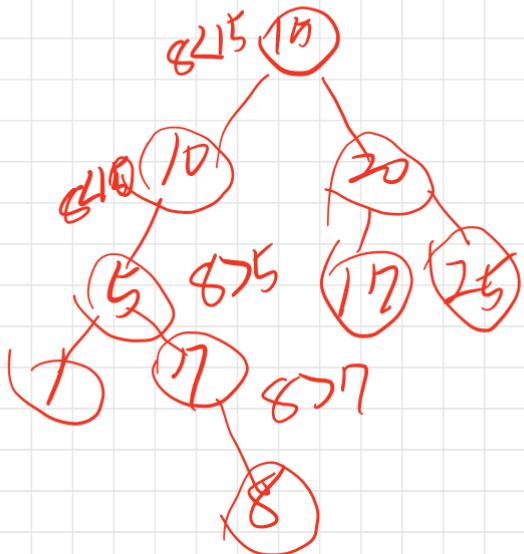


3's child push



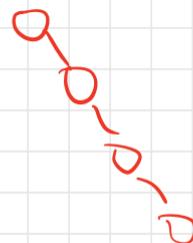
Binary Search Tree (BST)

target: 8



$$T(n) = O(\log n)$$

$$W(n) = O(n) \Rightarrow$$



* Red-black tree
python 자료구조도 이 알고리즘을
사용합니다.

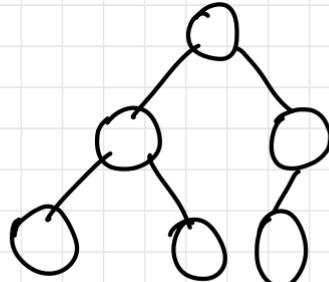
Min heap

Tree-based의 자료구조.

내 자식은 항상 나보다 크다.

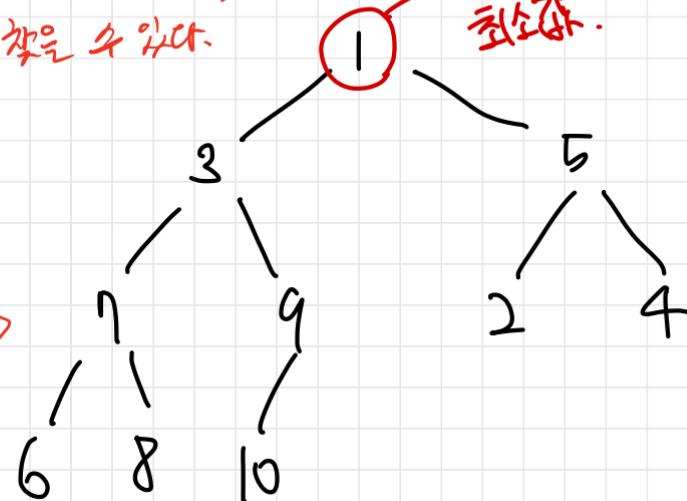
⇒ root는 tree의 최소값.

Complete binary tree.



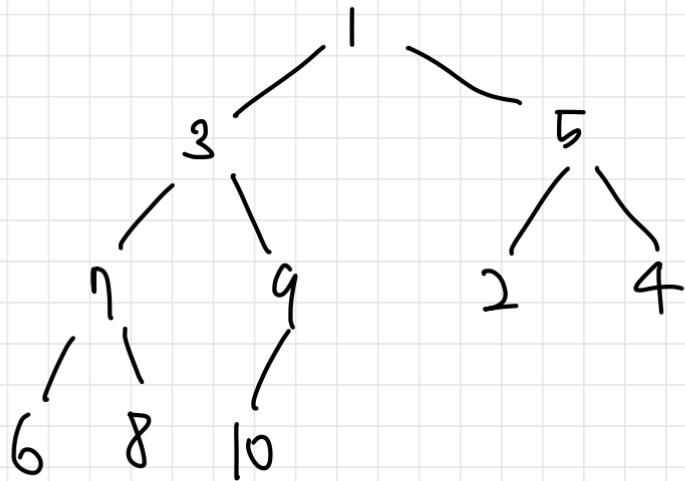
O(1)에 min, max를 찾을 수 있다.

트리의 최소값.



array로 구현

Heap insert

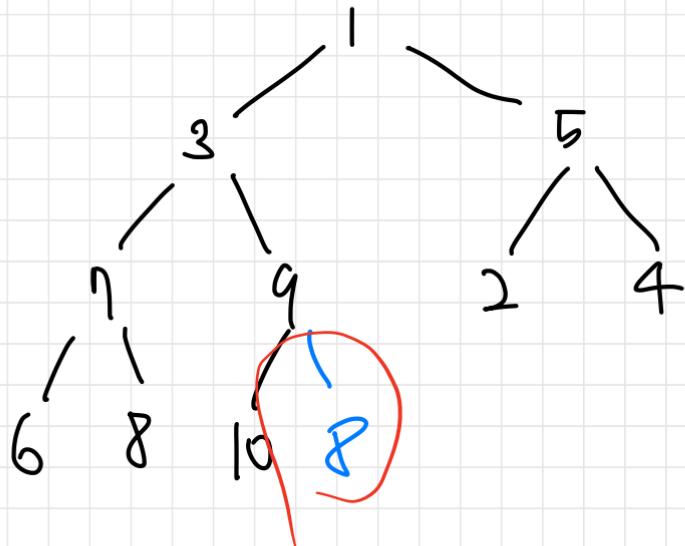


8을 삽입?

heap order : complete binary tree
root가 min

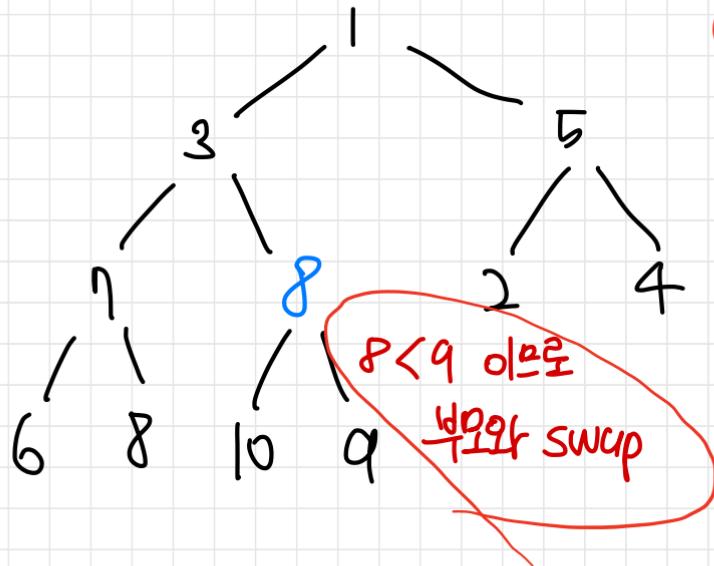
heap order가 유지되도록 하자.

Heap insert



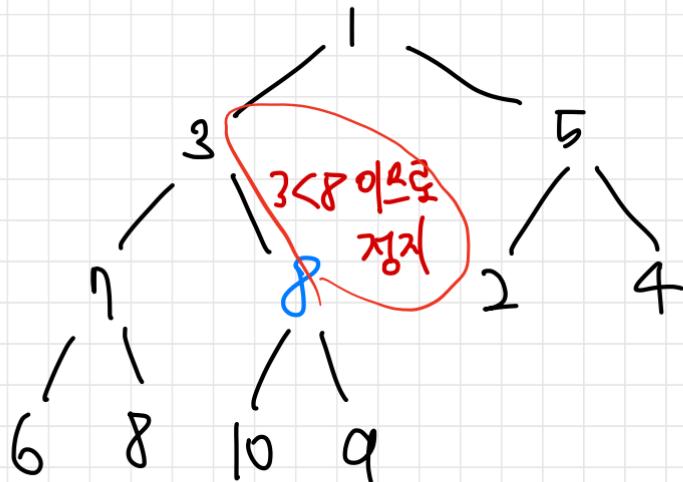
① 8을 일단 마지막에 삽입

Heap insert



② 8보다 작은 부모와 만날 때까지
위로 Swap하면서 올라간다.

Heap insert



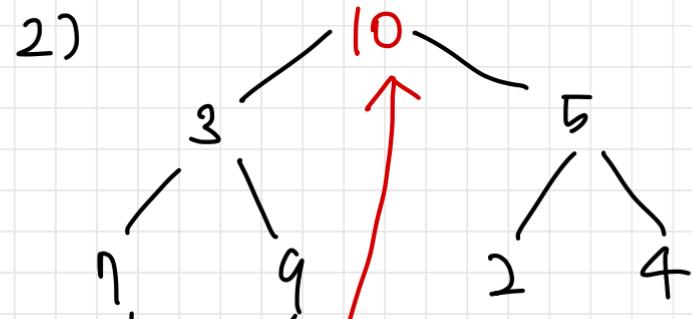
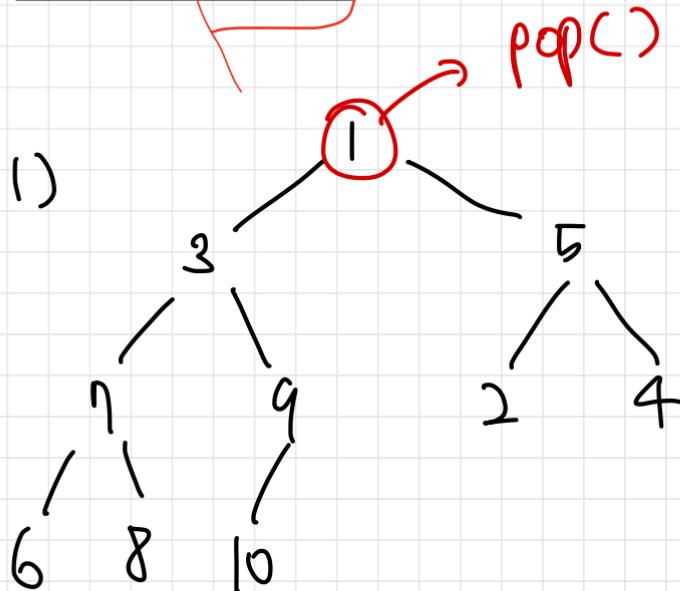
8보다 작은 부모와 만날 때까지

위로 Swap하면서 올라간다.

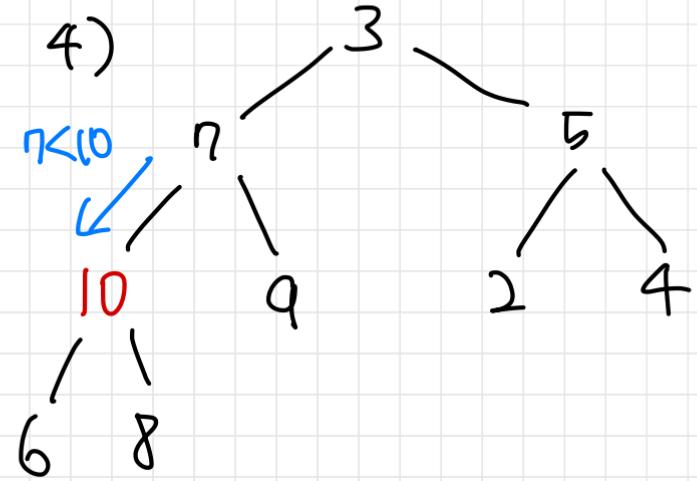
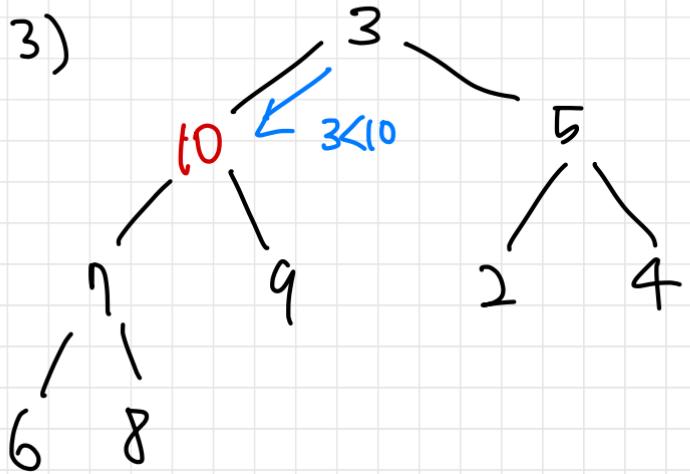
3<8 이므로 등반 중단.

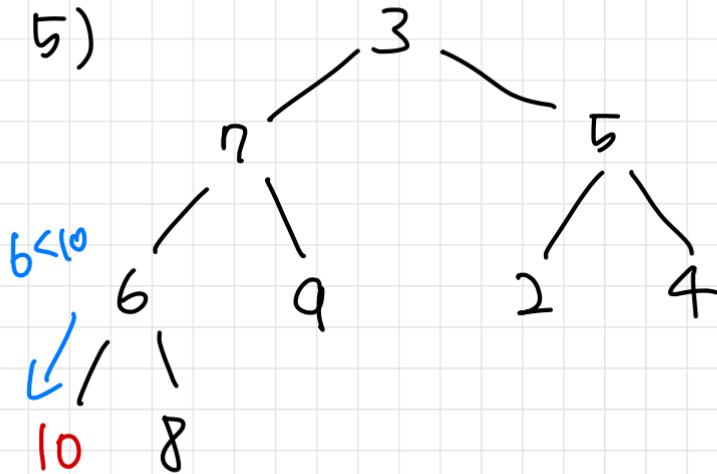
$O(\log n)$

Heap pop



가장 마지막
node가 top으로
올라온다.

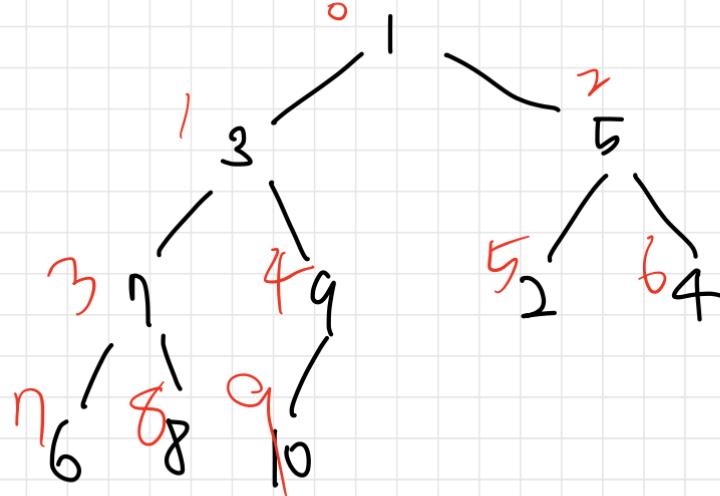




가장 마지막 node가 top으로 올라온 뒤,

자기보다 큰 node를 만날 때까지 자식과 비교하여 내려간다.

초소값 확인은 $O(1)$



idx 0 1 2 3 4 5 6 7 8 9
 key 1 3 5 7 9 2 4 6 8 10
 ↗ j ← idx
 ↗ 나누기 후 부터
 parent(j) = (j-1)//2
 parent_idx

$$\text{ex)} \text{parent}(7 \rightarrow 3) = (3-1)//2 = 1$$

$$\text{parent}(8 \rightarrow 8) = (8-1)//2 = 3$$

$$\text{left}(j) = 2 \times j + 1$$

$$\text{right}(j) = 2 \times j + 2$$

idx 2의 left idx

$$\text{ex)} \text{left}(2) = 2 \times 2 + 1 = 5$$

Numpy Library

ITA 파이썬 강좌 7강

KAIST

Information Technology Academy

1. Numpy 소개



Why NumPy?

- 정의
 - 대규모의 다차원 배열과 행렬 연산에 필요한 다양한 효율적인 함수를 제공하는 파이썬 패키지
 - Numerical Python의 약자
- 사용

```
In [1]: import numpy as np
```

```
In [2]: np.__version__
```

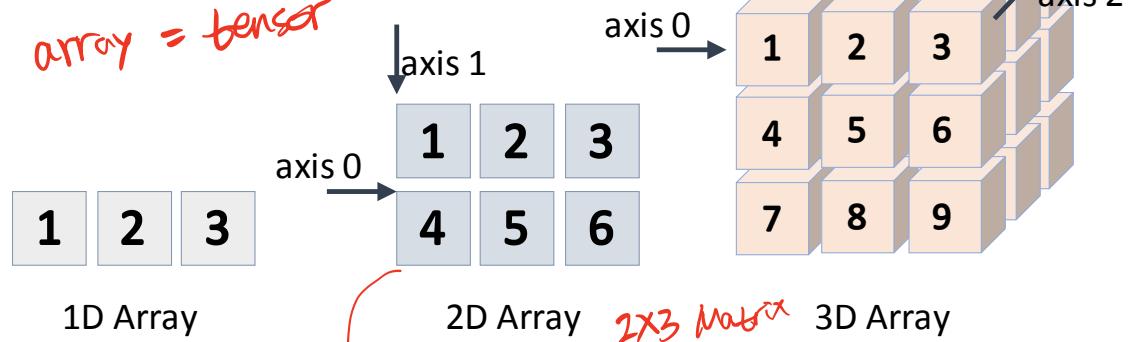
```
Out[2]: '1.16.0'
```



NumPy Array

- NumPy는 아래와 같이 다차원 배열을 지원한다.

3D 이상의 array = tensor



```
In [4]: arr = np.array([[1, 2, 3], [4, 5, 6]])
```

```
type:<class 'numpy.ndarray'>
shape: (2, 3), dimension: 2, dtype:int64
Array's Data:
[[1 2 3]
 [4 5 6]]
```

type을 항상 체크

$m \times n$ 형태 $\in \mathbb{R}^{m \times n}$

NumPy Array v.s. Python list

- NumPy Array는 대규모의 배열 및 행렬 연산에 유리하다.
 - NumPy는 적은 메모리를 쓰고, 빠르다
 - C 코드로 짜여진 라이브러리이며, 데이터를 벡터화하여 계산한다.
 - 배열 및 행렬간의 직관적인 직접 계산이 가능하다.

Python list

```
In [5]: a = [1, 2, 3]
         b = [4, 5, 6]

         c = a + b
         print(c)

[1, 2, 3, 4, 5, 6]
```

NumPy array

```
In [6]: a = np.array([1, 2, 3])
         b = np.array([4, 5, 6])

         c = a + b
         print(c)

[5 7 9]
```



NumPy Basic

Type

- List와는 달리, 한 array 안의 data type은 모두 같다.

```
In [10]: arr = np.array([1, 2, 3], dtype = float)
print(arr)
print(type(arr))
print(arr.dtype)
```

[1. 2. 3.]
<class 'numpy.ndarray'>
float64

type 지정

* help(np.array)

↓
doc string

Rank

- 배열의 차원(dimension)을 의미하며, 차원의 크기는 shape로 표시한다.

```
In [11]: arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr.shape)
```

(2, 3)

한정 type, shape을 확인해야 한다.



NumPy 배열 상태 검사

- 배열의 상태는 아래와 같이 검사할 수 있다.

```
In [39]: a = np.array([[1, 2, 3, 4, 5], [8, 9, 10, 11, 12],  
                     [[4, 5, 6, 7, 8], [10, 11, 12, 13, 14]]])  
  
# 배열 shape  
print('shape: ', a.shape)  
# 배열 길이  
print('length: ', len(a))  
# 배열 차원  
print('dimension: ', a.ndim)  
# 배열 요소 수  
print('size: ', a.size)  
# 배열 타입  
print('type: ', a.dtype)  
# 배열 타입 이름  
print('type name: ', a.dtype.name)  
# 배열 타입 변환  
print('to float: \n', a.astype(np.float))  
  
shape: (2, 2, 5)  
length: 2  
dimension: 3  
size: 20  
type: int64  
type name: int64  
to float:  
[[[ 1.  2.  3.  4.  5.]  
 [ 8.  9. 10. 11. 12.]]  
  
[[ 4.  5.  6.  7.  8.]  
 [10. 11. 12. 13. 14.]]]
```

2. Numpy 배열 생성



NumPy 기본 배열 생성 및 초기화

zeros()

default float

원하는 shape의 numpy array를 생성하며, 초기값은 모두 0이다.

```
In [16]: a = np.zeros((2, 3))  
print(a)
```

```
[[0. 0. 0.]  
 [0. 0. 0.]]
```

ones()

원하는 shape의 numpy array를 생성하며, 초기값은 모두 1이다.

```
In [17]: a = np.ones((2, 3))  
print(a)
```

```
[[1. 1. 1.]  
 [1. 1. 1.]]
```

NumPy 기본 배열 생성 및 초기화

full()

원하는 shape의 numpy array를 생성하며, 초기값을 설정할 수 있다.

```
In [18]: a = np.full((2, 3), 7)  
print(a)
```

```
[[7 7 7]  
 [7 7 7]]
```

2x3 matrix 7로 초기화

eye()

원하는 크기의 identity matrix를 생성한다.

```
In [19]: a = np.eye(3)  
print(a)
```

```
[[1. 0. 0.]  
 [0. 1. 0.]  
 [0. 0. 1.]]
```



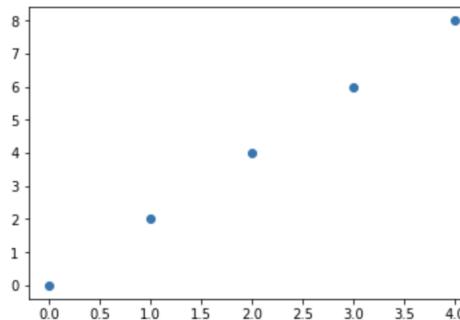
NumPy 데이터 생성 함수

Range 25 뷔주

- `arange([start,] stop[, step])`
 - [start, stop) 범위에서 step 간격으로 데이터를 생성하여 배열을 만드는 함수. 데이터의 간격 기준으로 배열 생성.

```
In [42]: a = np.arange(0, 10, 2)
print(a)
[0 2 4 6 8]    => numpy.array
```

```
In [43]: import matplotlib.pyplot as plt
plt.plot(a, 'o')
plt.show()
```



X축 그리기 때 자주 사용

NumPy 데이터 생성 함수

| linear space

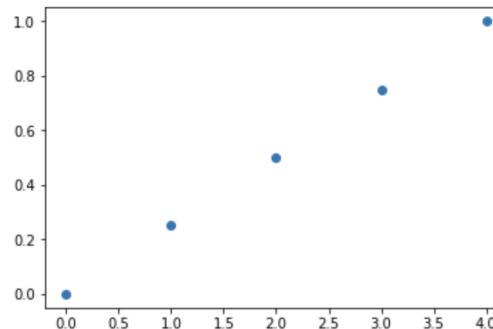
. linspace(start, stop, num=50, endpoint=True, retstep=False)

[start, stop) 범위에서 num개의 수를 균일한 간격으로 생성하여 배열을 만드는 함수. 요소 개수를 기준으로 균등 간격의 배열 생성.

```
In [40]: a = np.linspace(0, 1, 5)  
print(a)
```

```
[0.  0.25 0.5  0.75 1. ]
```

```
In [41]: import matplotlib.pyplot as plt  
plt.plot(a, 'o')  
plt.show()
```



→ False vs endpoint 미포함
True가 default



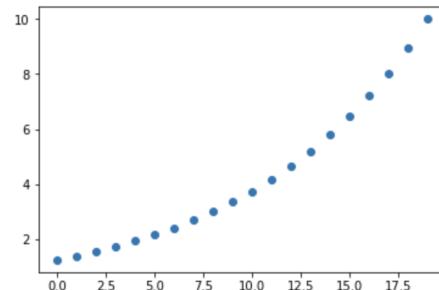
NumPy 데이터 생성 함수

. logspace(start, stop, num=50, endpoint=True, base=10.0)

base에 해당하는 로그 스케일로 [start, stop] 범위에서 num 개수만큼
균등 간격으로 데이터를 생성한 후 배열을 만드는 함수.

```
In [49]: a = np.logspace(0.1, 1, 20, endpoint=True)
print(a)
[ 1.25892541  1.40400425  1.565802   1.74624535  1.94748304  2.1719114
 2.42220294  2.70133812  3.0126409   3.35981829  3.74700446  4.17881006
 4.66037703  5.19743987  5.79639395  6.46437163  7.2093272   8.04013161
 8.9666781   10. ]
```

```
In [50]: import matplotlib.pyplot as plt
plt.plot(a, 'o')
plt.show()
```



Start
base ~ base Stop
num 개

NumPy 난수 기반 배열 생성

- NumPy는 난수 발생 및 배열 생성을 가능하게 하는 `numpy.random` 모듈을 제공한다.
- 난수 발생시 시작점을 설정함으로써 난수 발생을 재연할 수 있다.

```
In [12]: # 특정 seed 값을 이용하면 이후 난수가 재연 가능하도록 할 수 있다.  
np.random.seed(100)
```

Pseudo-random ; 유사随即

NumPy 난수 기반 배열 생성

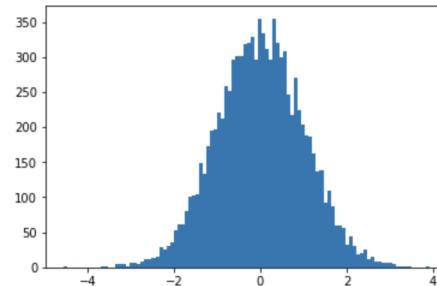
자료분석

random.normal(loc=0.0, scale=1.0, size=None)

평균과 표준편차를 이용해 정규분포 확률 밀도에서 표본을 추출하여 원하는 shape의 배열을 생성한다.

```
In [2]: mean = 0  
std = 1  
a = np.random.normal(mean, std, (2, 3))  
print(a)  
[[ 0.19698362  1.28631988 -2.32856336]  
 [ 0.30050865  0.50803789  0.16482721]]
```

```
In [3]: data = np.random.normal(0, 1, 10000)  
import matplotlib.pyplot as plt  
plt.hist(data, bins=100)  
plt.show()
```



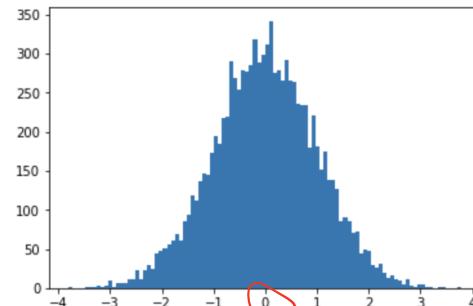
NumPy 난수 기반 배열 생성

- random.randn(d0, d1, ..., dn)
 - (d_0, d_1, \dots, d_n) shape의 배열을 생성 후 표준 정규 분포(standard normal distribution)에서 표본을 추출하여 난수를 생성한다.

```
In [6]: a = np.random.randn(2, 4)
print(a)
[[ 0.08602882 -0.38172755  0.47040927 -1.44387654]
 [-0.44912463 -0.59200318 -0.27760492 -0.52657155]]
```

standard normal dist $\mu=0$
 $\sigma=1$

```
In [7]: data = np.random.randn(10000)
import matplotlib.pyplot as plt
plt.hist(data, bins=100)
plt.show()
```



NumPy 난수 기반 배열 생성

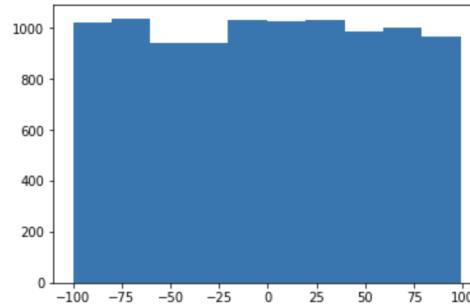
- random.randint(low, high=None, size=None, dtype='l')
- [low, high] 범위에서 균등분포(Uniform Distribution)을 이용해 정수 표본을 추출하여 지정된 shape의 배열을 생성한다.

```
In [8]: a = np.random.randint(5, 10, size=(2, 4))
print(a)
[[8 5 9 9]
 [5 9 7 6]]
```

low high

```
In [9]: data = np.random.randint(-100, 100, 10000)
import matplotlib.pyplot as plt
plt.hist(data, bins=10)
plt.show()
```

10000개 데이터



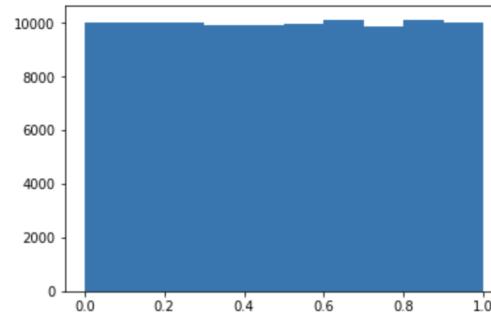
NumPy 난수 기반 배열 생성

- random.random(size=None)
- [0.0, 1.0) 범위에서 균등분포(Uniform Distribution)을 이용해 표본을 추출하여 지정된 shape의 배열을 생성한다.

```
In [10]: a = np.random.random((2, 4))
print(a)

[[0.25928536 0.53626885 0.93058457 0.4155561 ]
 [0.78407289 0.55597148 0.04916261 0.60873568]]
```

```
In [11]: data = np.random.random(100000)
import matplotlib.pyplot as plt
plt.hist(data, bins=10)
plt.show()
```



NumPy 배열 참조

Indexing

- comma를 이용하여 배열 값에 접근할 수 있다.
- 단, -1은 배열의 마지막 요소의 인덱스를 의미한다.

```
In [26]: a = np.array([1, 2, 3])
print(a[2])
print(a[-1])
```

```
3
3
```

```
In [27]: a = np.array([[1, 2, 3], [4, 5, 6]])
print(a[0, 1])
print(a[0][1])
```

```
2
2
```

```
In [28]: a = np.array([[[1, 2, 3, 4, 5], [8, 9, 10, 11, 12]],
                     [[4, 5, 6, 7, 8], [10, 11, 12, 13, 14]]])
print(a[1, 0, 4])
print(a[0, 0, 4])
```

8
5

↑ ↑ axis 0
 ↑ axis 1
 ↑ axis 2

NumPy 배열 참조

Slicing

여러 배열 요소를 참조할 때, axis별로 범위를 지정하여 참조할 수 있다.

```
In [35]: a = np.array([1, 2, 3, 4])
print(a[1:3])
print(a[2:])
```

[2 3]
[3 4]

```
In [36]: a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(a[:, 1:3])
print(a[:, 1:-1])
```

[[2 3]
 [6 7]]
 [[2 3]
 [6 7]]]

```
In [37]: a = np.array([[[1, 2, 3, 4, 5], [8, 9, 10, 11, 12]],
                   [[4, 5, 6, 7, 8], [10, 11, 12, 13, 14]]])
print(a[:, 1:, 3:5])
```

[[[11 12]
 [13 14]]]

3. Numpy 배열 변환



NumPy 배열 전치

transpose()

전치행렬

```
In [141]: a = np.array([[1, 2, 3],  
                      [4, 5, 6],  
                      [7, 8, 9]])  
print(a.T)  
print(a.transpose())
```

a.T : property
a.transpose() : Method

```
[[1 4 7]  
 [2 5 8]  
 [3 6 9]]  
[[1 4 7]  
 [2 5 8]  
 [3 6 9]]
```

NumPy 배열 크기 변환

- **flatten()** *Deep Copy*
n차원 배열을 1차원 배열로 바꾸는 함수.

```
In [54]: a = np.array([[1,2],[3,4]])
b = a.flatten()
print(b)
```

```
[1 2 3 4]
```

- **ravel()** *Shallow Copy*
n차원 배열의 1차원 배열 view를 제공하는 함수.
변환된 배열은 기존 배열을 참조하므로 원본 배열의 값이 변경될 경우
변환된 배열의 값도 함께 변경된다.

```
In [56]: a = np.array([[1,2],[3,4]])
b = a.ravel()
print(b)
```

```
[1 2 3 4]
```

NumPy 배열 크기 변환

reshape()

배열의 shape를 변경함

$$4 \times 5 \rightarrow 5 \times 4$$

$$2 \times 8 \rightarrow 4 \times 4$$

수정 전후의 전체 요소 개수가 일치하지 않는다면 ValueError.

```
In [60]: a = np.array([[[1, 2, 3, 4, 5], [8, 9, 10, 11, 12]],  
                   [[4, 5, 6, 7, 8], [10, 11, 12, 13, 14]]])  
  
print('original shape: ', a.shape) Shape은 2x2x5  
b = a.reshape((4, 5))  
print('transformed shape: ', b.shape)  
print(b)
```

```
original shape: (2, 2, 5)  
transformed shape: (4, 5)  
[[ 1  2  3  4  5]  
 [ 8  9 10 11 12]  
 [ 4  5  6  7  8]  
 [10 11 12 13 14]]
```

NumPy 배열 크기 변환

- resize()** *reshape() 이 데이터 보존에 더 안전하다.*
배열의 shape를 변경함.

1. 수정 전후의 전체 요소 개수가 일치하는 경우

```
In [68]: a = np.random.randint(1, 10, (2, 6))
print(a)

print('\nreshape to (6, 2)')
a.resize((6, 2))
print(a)
```

```
[[9 2 1 8 7 3]
 [1 9 3 6 2 9]]
```

```
reshape to (6, 2)
[[9 2]
 [1 8]
 [7 3]
 [1 9]
 [3 6]
 [2 9]]
```

NumPy 배열 크기 변환

. resize()

배열의 shape를 변경함.

2. 수정 전후의 전체 요소 개수가 일치하지 않고 줄어드는 경우 삭제

```
In [69]: a = np.random.randint(1, 10, (2, 6))
print(a)
```

느려진 O 이 붙는다.

```
print('\nreshape to (2, 10)')
a.resize((2, 10))
print(a)
```

```
[[2 6 5 3 9 4]
 [6 1 4 7 4 5]]
```

```
reshape to (2, 10)
[[2 6 5 3 9 4 6 1 4 7]
 [4 5 0 0 0 0 0 0 0 0]]
```

NumPy 배열 크기 변환

- `resize()`
- 배열의 `shape`를 변경함.
 3. 수정 전후의 전체 요소 개수가 일치하지 않고 늘어나는 경우: 0으로 초기화

```
In [70]: a = np.random.randint(1, 10, (2, 6))
print(a)
```

```
print('\nreshape to (3, 3)')
a.resize((3, 3))
print(a)
```

```
[[8 7 4 1 5 5]
 [6 8 7 7 3 5]]
```

```
reshape to (3, 3)
[[8 7 4]
 [1 5 5]
 [6 8 7]]
```

NumPy 배열 추가, 삭제

- append(arr, values, axis=None)

- 배열의 끝에 값을 추가

- 1. axis 지정 없이 추가: 1차원 배열로 변형되어 추가됨.

```
In [71]: a = np.arange(1, 10).reshape(3, 3)
b = np.arange(10, 19).reshape(3, 3)
result = np.append(a, b)
print(a)
print(b)
print(result)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
 [[10 11 12]
 [13 14 15]
 [16 17 18]]
 [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18]
```

NumPy 배열 추가, 삭제

- append(arr, values, axis=None)

- 배열의 끝에 값을 추가

- 2. axis = 0 *행에 추가*

```
In [72]: a = np.arange(1, 10).reshape(3, 3)
b = np.arange(10, 19).reshape(3, 3)
result = np.append(a, b, axis=0)
print(a)
print(b)
print(result)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
 [[10 11 12]
 [13 14 15]
 [16 17 18]]
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]
 [16 17 18]]]
```



NumPy 배열 추가, 삭제

- append(arr, values, axis=None)

- 배열의 끝에 값을 추가

- 3. axis =1 *열에 추가*

```
In [73]: a = np.arange(1, 10).reshape(3, 3)
b = np.arange(10, 19).reshape(3, 3)
result = np.append(a, b, axis=1)
print(a)
print(b)
print(result)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
 [[10 11 12]
 [13 14 15]
 [16 17 18]]
 (  [[ 1   2   3  10  11  12]
 [ 4   5   6  13  14  15]
 [ 7   8   9  16  17  18]])
```

NumPy 배열 추가, 삭제

insert(arr, obj, values, axis=None)

중간위로 삽입

배열의 axis 방향 obj번째 index에 요소 추가 (axis=None이면 1차원 변환)

```
In [80]: a = np.arange(1, 10).reshape(3, 3)
print(a)
print('no axis')
b = np.insert(a, 1, 999)
print(b)
print('axis 0') idx
c = np.insert(a, 1, 999, axis=0)
print(c)
print('axis 1')
d = np.insert(a, 1, 999, axis=1) idx
print(d)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
no axis:
[ 1 999  2  3  4  5  6  7  8  9]
axis 0
[[ 1  2  3]
 [999 999 999]
 [ 4  5  6]
 [ 7  8  9]]
axis 1
[[ 1 999  2  3]
 [ 4 999  5  6]
 [ 7 999  8  9]]
```

idx 1 행에 추가

idx 1 열에 추가

차원 맞추면
 가능

Ex: np.insert(a, 1, [999, 999, 999],
axis=0)

NumPy 배열 추가, 삭제

- ~~delete(arr, obj, axis=None)~~ *중단으로 삭제*
- 배열의 axis 방향 obj번째 index의 요소 제거 (axis=None이면 1차원 변환)

```
In [81]: a = np.arange(1, 10).reshape(3, 3)
print(a)
print('no axis')
b = np.delete(a, 1)
print(b)
print('axis 0')
c = np.delete(a, 1, axis=0)
print(c)
print('axis 1')
d = np.delete(a, 1, axis=1)
print(d)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
no axis
[1 3 4 5 6 7 8 9]
axis 0
[[1 2 3]
 [7 8 9]]
axis 1
[[1 3]
 [4 6]
 [7 9]]
```

NumPy 배열 결합

- concatenate(a1, a2, ...), axis=0
 - axis 방향으로 a1, a2, ... 배열을 결합

```
In [84]: a = np.arange(1, 7).reshape((2, 3))
b = np.arange(7, 13).reshape((2, 3))
result = np.concatenate((a, b), axis=0)
print(a)
print(b)
print(result)
```

```
[[1 2 3]
 [4 5 6]]
 [[ 7  8  9]
 [10 11 12]]
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

transpose가 필요할 때
transpose를 먼저 reshape 후 concat

NumPy 배열 결합

- concatenate((a1, a2, ...), axis=0)
 - axis 방향으로 a1, a2, ... 배열을 결합

```
In [85]: a = np.arange(1, 7).reshape((2, 3))
b = np.arange(7, 13).reshape((2, 3))
result = np.concatenate((a, b), axis=1)
print(a)
print(b)
print(result)
```

```
[[1 2 3]
 [4 5 6]]
 [[ 7  8  9]
 [10 11 12]]
 [[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

NumPy 배열 결합

append 및 Code 가독성이 좋아질 수 있음.

vstack(tup)

Vertical

튜플로 설정된 여러 배열을 수직 방향(axis=0)으로 연결

```
In [86]: a = np.arange(1, 7).reshape((2, 3))
b = np.arange(7, 13).reshape((2, 3))
result = np.vstack((a, b))
print(a)
print(b)
print(result)
```

```
[[1 2 3]
 [4 5 6]]
 [[ 7  8  9]
 [10 11 12]]
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]]
```

NumPy 배열 결합

hstack(tup)

horizontal

튜플로 설정된 여러 배열을 수평 방향(axis=1)으로 연결

```
In [87]: a = np.arange(1, 7).reshape((2, 3))
b = np.arange(7, 13).reshape((2, 3))
result = np.hstack((a, b))
print(a)
print(b)
print(result)
```

```
[[1 2 3]
 [4 5 6]]
 [[ 7   8   9]
 [10 11 12]]
 [[ 1   2   3   7   8   9]
 [ 4   5   6 10 11 12]]
```

NumPy 배열 분리

horizontal

· `hsplit(arr, indices_or_sections)`

배열을 수평 방향(axis=1)으로 분할하는 함수

```
In [89]: a = np.arange(1, 25).reshape((4, 6))
result1, result2, result3 = np.hsplit(a, 3)
print(a)
print(result1)
print(result2)
print(result3)
```

3개로 분할

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
 [[ 1  2]
 [ 7  8]
 [13 14]
 [19 20]]
 [[ 3  4]
 [ 9 10]
 [15 16]
 [21 22]]
 [[ 5  6]
 [11 12]
 [17 18]
 [23 24]]
```

NumPy 배열 분리

- . hsplit(arr, indices_or_sections)
 - . 배열을 수평 방향(axis=1)으로 분할하는 함수

```
In [93]: a = np.arange(1, 25).reshape((4, 6))
result1, result2, result3, result4 = np.hsplit(a, [1, 3, 5])
print(a)
print(result1)
print(result2)
print(result3)
print(result4)
```

0:1, 1:3, 3:5, 5:

```
[ 1  2  3  4  5  6]
[ 7  8  9 10 11 12]
[13 14 15 16 17 18]
[19 20 21 22 23 24]
[[ 1]
 [ 7]
 [13]
 [19]]
[[ 2  3]
 [ 8  9]
 [14 15]
 [20 21]]
[[ 4  5]
 [10 11]
 [16 17]
 [22 23]]
[[ 6]
 [12]
 [18]
 [24]]
```

NumPy 배열 분리

Vertical

- `vsplit(arr, indices_or_sections)`
- 배열을 수직 방향(axis=0)으로 분할하는 함수

```
In [91]: a = np.arange(1, 25).reshape((4, 6))
result1, result2 = np.vsplit(a, 2)
print(a)
print(result1)
print(result2)
```

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
[[13 14 15 16 17 18]
 [19 20 21 22 23 24]]
```

NumPy 배열 분리

- . vsplit(arr, indices_or_sections)
 - . 배열을 수직 방향(axis=0)으로 분할하는 함수

```
In [94]: a = np.arange(1, 25).reshape((4, 6))
result1, result2, result3 = np.vsplit(a, [1, 3])
print(a)
print(result1)
print(result2)
print(result3)
```

0:1, 1:3, 3:

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
 [[1 2 3 4 5 6]]
 [[ 7  8  9 10 11 12]
 [13 14 15 16 17 18]]
 [[19 20 21 22 23 24]]
```

4. Numpy 배열 연산



NumPy 배열 산술 연산

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])  
b = np.array([[8, 7, 6, 5], [4, 3, 2, 1]])
```

덧셈

```
In [100]: print(a + b)  
          print(np.add(a, b))
```

```
[[9 9 9 9]  
 [9 9 9 9]]  
 [[9 9 9 9]  
 [9 9 9 9]]
```

broadcast



뺄셈

```
In [101]: print(a - b)  
          print(np.subtract(a, b))
```

```
[[ -7 -5 -3 -1]  
 [ 1  3  5  7]]  
 [[ -7 -5 -3 -1]  
 [ 1  3  5  7]]
```

NumPy 배열 산술 연산

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])  
b = np.array([[8, 7, 6, 5], [4, 3, 2, 1]])
```

곱셈

```
In [102]: print(a * b)  
print(np.multiply(a, b))
```

```
[[ 8 14 18 20]  
 [20 18 14  8]]  
[[ 8 14 18 20]  
 [20 18 14  8]]
```

정답 : np.matmul(a, b)

나눗셈

```
In [103]: print(a / b)  
print(np.divide(a, b))
```

```
[[ 0.125      0.28571429  0.5       0.8      ]]  
[ 1.25       2.          3.5       8.       ]]  
[[ 0.125      0.28571429  0.5       0.8      ]]  
[ 1.25       2.          3.5       8.       ]]
```

NumPy 배열 산술 연산

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])  
b = np.array([[8, 7, 6, 5], [4, 3, 2, 1]])
```

element
wise

지수

exponentiation

```
In [104]: print(np.exp(b))
```

```
[ [2.98095799e+03 1.09663316e+03 4.03428793e+02 1.48413159e+02]  
[ 5.45981500e+01 2.00855369e+01 7.38905610e+00 2.71828183e+00] ]
```

제곱근

Square root

```
In [105]: print(np.sqrt(a))
```

```
[ [1. 1.41421356 1.73205081 2. ]  
[ 2.23606798 2.44948974 2.64575131 2.82842712] ]
```

로그

base e

10^{-4} ; e는 10

```
In [106]: print(np.log(a))
```

```
[ [0. 0.69314718 1.09861229 1.38629436]  
[ 1.60943791 1.79175947 1.94591015 2.07944154] ]
```

NumPy 배열 산술 연산

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])  
b = np.array([[8, 7, 6, 5], [4, 3, 2, 1]])
```

element
wise

sin

```
In [107]: print(np.sin(a))
```

```
[[ 0.84147098  0.90929743  0.14112001 -0.7568025 ]  
 [-0.95892427 -0.2794155   0.6569866   0.98935825]]
```

COS

```
In [108]: print(np.cos(a))
```

```
[[ 0.54030231 -0.41614684 -0.9899925  -0.65364362]  
 [ 0.28366219  0.96017029  0.75390225 -0.14550003]]
```

tan

```
In [109]: print(np.tan(a))
```

```
[[ 1.55740772 -2.18503986 -0.14254654  1.15782128]  
 [-3.38051501 -0.29100619  0.87144798 -6.79971146]]
```

NumPy 배열 산술 연산

수학적 해석
column row

$$2 \times 4, 4 \times 7 \Rightarrow 2 \times 7$$

Dot product (내적)

```
In [111]: a = np.arange(1, 10).reshape(3, 3)
          b = np.arange(9, 0, -1).reshape(3, 3)
          print(a)
          print(b)
          print(np.dot(a, b))
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
 [[9 8 7]
 [6 5 4]
 [3 2 1]]
 [[ 30  24  18]
 [ 84  69  54]
 [138 114  90]]
```

NumPy 배열 비교 연산

```
a = np.arange(1, 10).reshape(3, 3)
b = np.arange(9, 0, -1).reshape(3, 3)
```

요소별 비교 (Element-wise)

```
In [122]: a == b
```

```
Out[122]: array([[False, False, False],
                  [False, True, False],
                  [False, False, False]])
```

```
In [123]: a > b
```

```
Out[123]: array([[False, False, False],
                  [False, False, True],
                  [True, True, True]])
```

배열 비교 (Array-wise)

```
In [124]: np.array_equal(a, b)
```

```
Out[124]: False
```



NumPy 배열 집계 함수

집계 함수(Aggregate Functions)들은 axis를 기준으로 계산되며,
axis를 지정하지 않을 경우 axis=None이다.

1	2	3
4	5	6
7	8	9

Axis = None

1	2	3
4	5	6
7	8	9

Axis = 0

행을 ~~입~~에는 ~~쪽~~으로
계산

⇒ 제일 큰 끝에다가
다놓는다.

1	2	3
4	5	6
7	8	9

Axis = 1

제일 큰 끝에다가

하나 작은 끝에

NumPy 배열 집계 함수

```
a = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

합계

```
In [127]: print(a.sum(), np.sum(a))
          print(a.sum(axis=0), np.sum(a, axis=0))
          print(a.sum(axis=1), np.sum(a, axis=1))
```

45 45
[12 15 18] [12 15 18]
[6 15 24] [6 15 24]

*각 행에서
동행으로의 sum*

a.shape
→ (3, 3)

ex) (2,3,4)
axis 0 ⇒ (3,4)
axis 1 ⇒ (2,4)
axis 2 ⇒ (1,3)

누적 합계

```
In [130]: print(np.cumsum(a))
          print(np.cumsum(a, axis=0))
          print(a.cumsum(axis=1))
```

[1 3 6 10 15 21 28 36 45]
[[1 2 3]
 [5 7 9]
 [12 15 18]]
[[1 3 6]
 [4 9 15]
 [7 15 24]]

Cumulative는
앞축 끝사킨다.

NumPy 배열 집계 함수

```
a = np.array([[1, 2, 3],  
             [4, 5, 6],  
             [7, 8, 9]])
```

최대값

```
In [131]: print(a.max(), np.max(a))  
print(a.max(axis=0), np.max(a, axis=0))  
print(a.max(axis=1), np.max(a, axis=1))
```

```
9 9  
[7 8 9] [7 8 9]  
[3 6 9] [3 6 9]
```

최소값

```
In [132]: print(a.min(), np.min(a))  
print(a.min(axis=0), np.min(a, axis=0))  
print(a.min(axis=1), np.min(a, axis=1))
```

```
1 1  
[1 2 3] [1 2 3]  
[1 4 7] [1 4 7]
```

NumPy 배열 집계 함수

```
a = np.array([[1, 2, 3],  
             [4, 5, 6],  
             [7, 8, 9]])
```

평균

```
In [133]: print(a.mean(), np.mean(a))  
print(a.mean(axis=0), np.mean(a, axis=0))  
print(a.mean(axis=1), np.mean(a, axis=1))
```

```
5.0 5.0  
[4. 5. 6.] [4. 5. 6.]  
[2. 5. 8.] [2. 5. 8.]
```

표준편차

```
In [134]: print(a.std(), np.std(a))  
print(a.std(axis=0), np.std(a, axis=0))  
print(a.std(axis=1), np.std(a, axis=1))
```

```
2.581988897471611 2.581988897471611  
[2.44948974 2.44948974 2.44948974] [2.44948974 2.44948974 2.44948974]  
[0.81649658 0.81649658 0.81649658] [0.81649658 0.81649658 0.81649658]
```

NumPy 배열 집계 함수

```
a = np.array([[1, 2, 3],  
             [4, 5, 6],  
             [7, 8, 9]])
```

- 중앙값

```
In [136]: print(np.median(a))  
          print(np.median(a, axis=0))  
          print(np.median(a, axis=1))
```

```
5.0  
[4. 5. 6.]  
[2. 5. 8.]
```

- 상관계수

기본에 따라 Python critical로 처리가 됨.

```
In [138]: print(np.corrcoef(a))
```

```
[[1. 1. 1.]  
 [1. 1. 1.]  
 [1. 1. 1.]]
```

NumPy 배열 Broadcasting

- Shape이 다른 배열의 연산 또한 가능하다.

1	1	1
2	2	2
3	3	3

0	1	2
0	1	2
0	1	2

1	1	1
2	2	2
3	3	3

0	1	2
0	1	2
0	1	2

1	1	1
2	2	2
3	3	3

0	1	2

1	1	1
2	2	2
3	3	3

0	1	2
0	1	2
0	1	2

1
2
3

0	1	2

1	1	1
2	2	2
3	3	3

0	1	2
0	1	2
0	1	2

0	2	3
2	3	4
3	4	5

NumPy 배열 Broadcasting

- Shape이 다른 배열의 연산 또한 가능하다.

```
In [143]: a = np.arange(1, 25).reshape(4, 6)
          print(a)
          print(a + 100)
```

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
 [[101 102 103 104 105 106]
 [107 108 109 110 111 112]
 [113 114 115 116 117 118]
 [119 120 121 122 123 124]]
```

NumPy 배열 Broadcasting

- . Shape이 다른 배열의 연산 또한 가능하다.

```
In [144]: a = np.arange(5).reshape((1, 5))
b = np.arange(5).reshape((5, 1))
print(a)
print(b)
print(a + b)
```

```
[[0 1 2 3 4]]
[[0]
 [1]
 [2]
 [3]
 [4]]
[[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]
 [4 5 6 7 8]]
```



Quiz 1.

1. 다음 배열들의 shape는 어떻게 될 것인가?

In []: `a = np.array([1, 2, 3])` (3,)

In []: `b = np.array([[1, 2], [3, 4], [5, 6]])` (3,2)

In []: `c = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])` (2,2,3)
o△□

2. [0.0, 1.0) 사이의 랜덤한 값으로 이루어진 shape (2, 3, 4)인 배열을 정의하고, indexing을 사용하여 가장 마지막 요소에 접근하여라.

`ans = np.random.rand(2,3,4)`

`ans[-1][-1][-1]`, `ans[-1,-1,-1]`, `ans[1,2,3]`

Quiz 2.

- [10, 100) 사이의 랜덤한 값으로 이루어진 shape (4, 4)인 배열을 정의하고

$a = np.random.randint(10, 100, size=(4, 4), endpoint=False)$

- 이 배열의 전치행렬을 구한 후 $a.T$

- shape를 (2, 8) 형태로 바꾸고 $a.T.reshape(2, 8)$

- 수평 방향으로 [:2], [2:5], [5:7], [7:] 형태가 되도록 나누어라.

$np.hsplit(a.T.reshape(2, 8), [2, 5, 7])$

Quiz 3.

1. [10, 100) 범위 내에서 shape (4, 6)인 두 배열을 정의하고 두 배열의 내적을 구하여라.
$$a = np.random.randint(10, 100, size=(4, 6), endpoint=False)$$

$$b = np.random.randint(10, 100, size=(4, 6), endpoint=False)$$
2. 내적한 배열의 axis=0 방향의 평균과 표준편차를 구하여라.
$$m = np.mean(data1, axis=0)$$

$$std = np.std(data1, axis=0)$$
3. 구한 평균과 표준편차를 이용하여 해당 배열을 정규화하여라.

$$\frac{(data - m)}{std}$$

Pandas Library

ITA 파이썬 강좌 8강

KAIST

Information Technology Academy

1. Pandas 소개



Pandas Library

- 소개

- 파이썬에서 사용하는 데이터분석 라이브러리
- 행과 열로 이루어진 데이터 객체를 만들어 다룰 수 있게 됨
- 보다 안정적으로 대용량의 데이터를 처리할 수 있음

- 사용

```
In [1]: import numpy as np  
        import pandas as pd
```

Pandas v. s. NumPy

- Pandas library는 다양한 형태의 데이터를 다루기에 용이하다.
 - NumPy library가 numerical value를 다루는 데에 최적화되어있었다면, Pandas library는 보다 다양한 형태의 여러 데이터를 함께 다루기 용이함
 - Panel data: 여러 개체들을 복수의 시간에 걸쳐 추적하여 얻는 데이터
 - Big data analysis에 사용될 수 있다.
 - Flexible indexing: non-integer index를 사용할 수 있다.

2. Pandas 자료구조



Pandas 자료구조

- Pandas에서 사용되는 자료구조로는 Series와 DataFrame이 있다.
 - Series: 1차원 배열
 - DataFrame: 2차원 배열

Index 0	
Index 1	
Index 2	
Index 3	
Index 4	
Index 5	
Index 6	

Series

	Column 0	Column 1	Column 2
Index 0			
Index 1			
Index 2			
Index 3			
Index 4			
Index 5			
Index 6			

DataFrame

Series 생성

- pd.Series(data=None, index=None)

```
In [2]: grades = pd.Series(range(70, 100, 2))
print(grades)
```

index

0	70
1	72
2	74
3	76
4	78
5	80
6	82
7	84
8	86
9	88
10	90
11	92
12	94
13	96
14	98

dtype: int64

Series 생성

- Series data의 descriptive statistics 확인

```
In [3]: print(grades.describe())
print(len(grades))
```

```
count    15.000000
mean     84.000000
std      8.944272
min      70.000000
25%     77.000000
50%     84.000000
75%     91.000000
max     98.000000
dtype: float64
15
```

Series 생성

- Index를 지정한 Series 생성

```
In [4]: height = pd.Series([160, 170, 180], index=['Amy', 'Tom', 'Michael'])
print(height)
print(height.Amy, height['Amy'], height[0])
```

```
Amy      160
Tom      170
Michael  180
dtype: int64
160 160 160
```

- Dictionary를 사용한 Series 생성

```
In [5]: nations = pd.Series({'Korea':82, 'USA':1, 'China':'cn'})
print(nations)
```

```
Korea    82
USA      1
China    cn
dtype: object
```

dictionary

| data type 상관없어

— Python의 string

DataFrame 생성

- Dictionary를 사용하여 DataFrame을 생성할 수 있다.
 - Dictionary의 key는 column이 된다.
 - Dictionary의 item은 모두 같은 길이의 리스트여야 한다.

```
In [14]: scores = {'Amy':[92, 80, 70],  
                 'Tom':[56, 87, 100],  
                 'Michael':[96, 78, 88]}  
scores_df = pd.DataFrame(scores)  
  
scores_df
```

Out[14]:

	Amy	Tom	Michael
0	92	56	96
1	80	87	78
2	70	100	88

DataFrame 생성

- Index를 지정하여 DataFrame을 생성하거나 생성한 DataFrame의 index를 바꿀 수 있다.

```
In [17]: scores_df.index = ['Math', 'Science', 'Art']  
scores_df
```

Out[17]:

	Amy	Tom	Michael
Math	92	56	96
Science	80	87	78
Art	70	100	88

```
In [18]: scores_df_with_index = pd.DataFrame(scores,  
                                         index = ['Math', 'Science', 'Art'])  
scores_df_with_index
```

Out[18]:

	Amy	Tom	Michael
Math	92	56	96
Science	80	87	78
Art	70	100	88

Index를 지정해서 DF 생성

DataFrame 생성

- Column을 지정하여 DataFrame을 생성할 수 있다.
 - Index와 달리 Dictionary에 없는 column도 포함하여 생성할 수 있다.
 - Dictionary에 없는 column은 Nan으로 초기화된다.

```
In [33]: df = pd.DataFrame(scores,
                           index = ['Math', 'Science', 'Art'],
                           columns = ['Amy', 'Tom', 'Michael', 'Clara'])  
df
```

```
Out[33]:
```

	Amy	Tom	Michael	Clara
Math	92	56	96	NaN
Science	80	87	78	NaN
Art	70	100	88	NaN

* column name 소문자로 바꾸기

$df.columns = [x.lower() \text{ for } x \text{ in } df.columns]$

```
In [37]: df.columns = ['P1', 'P2', 'P3', 'P4']  
df
```

기존 column name 유지

```
Out[37]:
```

	P1	P2	P3	P4
Math	92	56	96	NaN
Science	80	87	78	NaN
Art	70	100	88	NaN

DataFrame 살펴보기

- 생성한 DataFrame의 index와 column, 값을 확인할 수 있다.

```
In [20]: scores_df
```

```
Out[20]:
```

	Amy	Tom	Michael
Math	92	56	96
Science	80	87	78
Art	70	100	88

df.shape

df.shape
df.info()

```
In [22]: scores_df.index
```

```
Out[22]: Index(['Math', 'Science', 'Art'], dtype='object')
```

```
In [23]: scores_df.columns
```

```
Out[23]: Index(['Amy', 'Tom', 'Michael'], dtype='object')
```

```
In [24]: scores_df.values
```

```
Out[24]: array([[ 92,   56,   96],
       [ 80,   87,   78],
       [ 70,  100,   88]])
```

DataFrame 살펴보기

- 각 index와 column의 이름, DataFrame의 이름을 정의할 수 있다.

```
In [28]: scores_df.index.name = 'Subjects'  
scores_df.columns.name = 'Name'  
scores_df.name = 'Scores'  
scores_df
```

Out[28]:

	Name	Amy	Tom	Michael
Subjects				
Math	92	56	96	
Science	80	87	78	
Art	70	100	88	

DataFrame 살펴보기

- describe() 함수를 이용하여 DataFrame의 계산 가능한 값들을 확인할 수 있다.

* index_col 설정

In [38]: scores_df.describe()

Out [38]:

	Name	Amy	Tom	Michael
count	3.000000	3.000000	3.000000	
mean	80.666667	81.000000	87.333333	
std	11.015141	22.605309	9.018500	
min	70.000000	56.000000	78.000000	
25%	75.000000	71.500000	83.000000	
50%	80.000000	87.000000	88.000000	
75%	86.000000	93.500000	92.000000	
max	92.000000	100.000000	96.000000	

* DataFrame
인덱스를 통해
데이터를
접근할 수 있다

Pd.to_excel excel

Pd.to_csv CSV

Pd.to_csv, delimiter='W' tsv

Pd.read_excel excel

Pd.read_csv CSV

Pd.read_csv, delimiter='t' tsv

* DataFrame 생성 방식

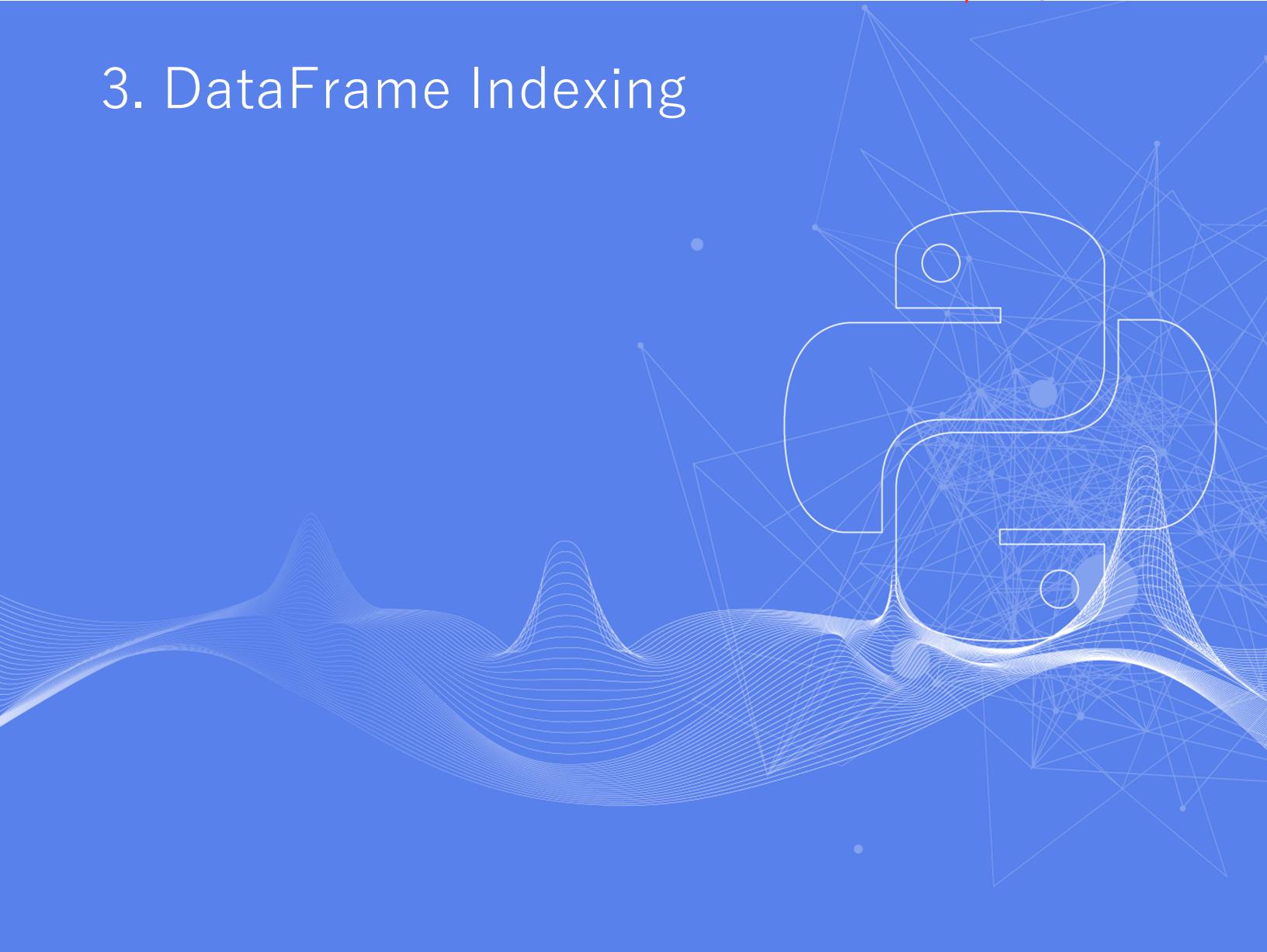
① Scores = {'Amy': [1, 2, 3],
'Tom': [4, 5, 6],
'Michael': [7, 8, 9]}

Pd.DataFrame(Scores)

② Scores2 = [{"Amy": 1, "Tom": 4, "Michael": 7},
 {"Amy": 2, "Tom": 5, "Michael": 8},
 {"Amy": 3, "Tom": 6, "Michael": 9}]

Pd.DataFrame(Scores2,
 index=['math', 'S', 'A'])

3. DataFrame Indexing



DataFrame Indexing

- DataFrame을 만들 때 index, columns를 설정하지 않으면 기본값으로 0부터 시작하는 정수형 숫자로 index와 column이 초기화된다.

```
In [41]: df = pd.DataFrame(np.random.randn(6, 4))
df.columns = ['A', 'B', 'C', 'D']
df.index = pd.date_range('20201129', periods=6)
df
```

Out[41]:

	A	B	C	D
2020-11-29	-0.292296	-0.406187	0.494777	-0.589141
2020-11-30	1.497354	2.337008	-0.006118	1.710427
2020-12-01	0.727822	0.063547	0.066877	-0.143830
2020-12-02	-1.383203	-0.242607	-2.262568	0.879890
2020-12-03	-0.944349	-0.070250	-1.037501	0.844512
2020-12-04	1.712479	-0.609560	-0.735014	-0.643639

cell 아래 /list로 들어갈 수
있어
문자열 수식을 실행
할 때 이미 쓰는 것 지양!
pickle library 사용하면
파일로 저장 후 불러오면 깊은 타입
그대로 불러올 수 있다.

Column을 선택하고 조작하기

- Column 이름을 사용하여 DataFrame의 column에 접근한다.

```
In [67]: df['A']
```

```
Out[67]: 2020-11-29    -0.096245
          2020-11-30    -0.347351
          2020-12-01     0.607475
          2020-12-02    -0.717678
          2020-12-03     0.844395
          2020-12-04    -1.644254
Name: A, dtype: float64
```

```
In [68]: df.A
```

☞ 새로운 컬럼은 만들 때는 약된다.

```
Out[68]: 2020-11-29    -0.096245
          2020-11-30    -0.347351
          2020-12-01     0.607475
          2020-12-02    -0.717678
          2020-12-03     0.844395
          2020-12-04    -1.644254
Name: A, dtype: float64
```

Column을 선택하고 조작하기

- 여러 column을 함께 선택할 수 있다.

```
In [69]: df[['A', 'D']]
```

* Column 이름을 list로 뒤에서 불러오면 된다.

```
Out[69]:
```

	A	D
2020-11-29	-0.096245	-0.648267
2020-11-30	-0.347351	-0.701743
2020-12-01	0.607475	0.324370
2020-12-02	-0.717678	-0.343311
2020-12-03	0.844395	0.536590
2020-12-04	-1.644254	0.515602

Column을 선택하고 조작하기

- Column의 값을 수정할 수 있다.

```
In [70]: df['D'] = 0.5 # broadcasting  
df['C'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6] # 개수 맞추어서 넣어야 한다.  
df
```

Out[70]:

	A	B	C	D
2020-11-29	-0.096245	0.081155	0.1	0.5
2020-11-30	-0.347351	0.848011	0.2	0.5
2020-12-01	0.607475	-0.633725	0.3	0.5
2020-12-02	-0.717678	-1.170083	0.4	0.5
2020-12-03	0.844395	1.534665	0.5	0.5
2020-12-04	-1.644254	-0.864390	0.6	0.5

Column을 선택하고 조작하기

- 새로운 column을 추가할 수 있다.

```
In [71]: df['E'] = np.arange(6)  
df
```

Out[71]:

	A	B	C	D	E
2020-11-29	-0.096245	0.081155	0.1	0.5	0
2020-11-30	-0.347351	0.848011	0.2	0.5	1
2020-12-01	0.607475	-0.633725	0.3	0.5	2
2020-12-02	-0.717678	-1.170083	0.4	0.5	3
2020-12-03	0.844395	1.534665	0.5	0.5	4
2020-12-04	-1.644254	-0.864390	0.6	0.5	5

Column을 선택하고 조작하기

- Series를 새로운 column으로 추가할 수 있다.

```
In [72]: data = pd.Series([-1.2, -1.5, -1.7],  
                         index=['2020-11-30', '2020-12-01', '2020-12-02'])  
df['F'] = data  
df
```

period=3)) 으로 써야함

Out[72]:

	A	B	C	D	E	F
2020-11-29	-0.096245	0.081155	0.1	0.5	0	NaN
2020-11-30	-0.347351	0.848011	0.2	0.5	1	-1.2
2020-12-01	0.607475	-0.633725	0.3	0.5	2	-1.5
2020-12-02	-0.717678	-1.170083	0.4	0.5	3	-1.7
2020-12-03	0.844395	1.534665	0.5	0.5	4	NaN
2020-12-04	-1.644254	-0.864390	0.6	0.5	5	NaN

Column을 선택하고 조작하기

- 산술 연산을 이용하여 새로운 column을 추가할 수 있다.

```
In [73]: df['G'] = df['C'] - df['D'] # Broad Casting이라고 한다는 암시.  
df['H'] = df['B'] > 0 True / False  
df
```

Out[73]:

	A	B	C	D	E	F	G	H
2020-11-29	-0.096245	0.081155	0.1	0.5	0	NaN	-0.4	True
2020-11-30	-0.347351	0.848011	0.2	0.5	1	-1.2	-0.3	True
2020-12-01	0.607475	-0.633725	0.3	0.5	2	-1.5	-0.2	False
2020-12-02	-0.717678	-1.170083	0.4	0.5	3	-1.7	-0.1	False
2020-12-03	0.844395	1.534665	0.5	0.5	4	NaN	0.0	True
2020-12-04	-1.644254	-0.864390	0.6	0.5	5	NaN	0.1	False

Column을 선택하고 조작하기

- column을 삭제할 수 있다.

```
In [74]: del df['E']      #제거하지는 않는다.  
df
```

Out[74]:

	A	B	C	D	F	G	H
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	-0.4	True
2020-11-30	-0.347351	0.848011	0.2	0.5	-1.2	-0.3	True
2020-12-01	0.607475	-0.633725	0.3	0.5	-1.5	-0.2	False
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False
2020-12-03	0.844395	1.534665	0.5	0.5	NaN	0.0	True
2020-12-04	-1.644254	-0.864390	0.6	0.5	NaN	0.1	False

Row를 선택하고 조작하기

- 기본적으로 이전의 배열과 같은 방식의 indexing도 가능하다.

```
In [75]: df[0:3]
```

Out[75]:

	A	B	C	D	F	G	H
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	-0.4	True
2020-11-30	-0.347351	0.848011	0.2	0.5	-1.2	-0.3	True
2020-12-01	0.607475	-0.633725	0.3	0.5	-1.5	-0.2	False

```
In [76]: df[:-1]
```

Out[76]:

	A	B	C	D	F	G	H
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	-0.4	True
2020-11-30	-0.347351	0.848011	0.2	0.5	-1.2	-0.3	True
2020-12-01	0.607475	-0.633725	0.3	0.5	-1.5	-0.2	False
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False
2020-12-03	0.844395	1.534665	0.5	0.5	NaN	0.0	True

Row를 선택하고 조작하기

- loc을 이용하면 index를 사용하여 행에 접근할 수 있다.

```
In [77]: df.loc['2020-11-30']
```

```
Out[77]: A    -0.347351  
          B     0.848011  
          C      0.2  
          D      0.5  
          F     -1.2  
          G     -0.3  
          H      True  
Name: 2020-11-30, dtype: object
```

Series로 불러온다.

loc은 endPoint = True 옥.

```
In [78]: df.loc['2020-12-01':'2020-12-03']
```

```
Out[78]:
```

	A	B	C	D	F	G	H
2020-12-01	0.607475	-0.633725	0.3	0.5	-1.5	-0.2	False
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False
2020-12-03	0.844395	1.534665	0.5	0.5	NaN	0.0	True

Row를 선택하고 조작하기

- loc 을 이용하면 index를 사용하여 행의 특정 열에 접근할 수 있다.

```
In [80]: df.loc[:, 'F'] # == df['F']
```

```
Out[80]: 2020-11-29    NaN  
2020-11-30    -1.2  
2020-12-01    -1.5  
2020-12-02    -1.7  
2020-12-03    NaN  
2020-12-04    NaN  
Name: F, dtype: float64
```

```
In [81]: df.loc[:, ['F', 'A']]
```

```
Out[81]:
```

	F	A
2020-11-29	NaN	-0.096245
2020-11-30	-1.2	-0.347351
2020-12-01	-1.5	0.607475
2020-12-02	-1.7	-0.717678
2020-12-03	NaN	0.844395
2020-12-04	NaN	-1.644254

Row를 선택하고 조작하기

- loc 을 이용하면 새로운 행을 추가할 수 있다.

```
In [82]: df.loc['2020-12-05', :] = [1, 2, 3, 4, -2, -0.7, True]  
df
```

Out[82]:

	A	B	C	D	F	G	H
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	-0.4	True
2020-11-30	-0.347351	0.848011	0.2	0.5	-1.2	-0.3	True
2020-12-01	0.607475	-0.633725	0.3	0.5	-1.5	-0.2	False
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False
2020-12-03	0.844395	1.534665	0.5	0.5	NaN	0.0	True
2020-12-04	-1.644254	-0.864390	0.6	0.5	NaN	0.1	False
2020-12-05	1.000000	2.000000	3.0	4.0	-2.0	-0.7	True

Row를 선택하고 조작하기

- iloc은 index의 번호를 사용한다. *문자*

```
In [83]: df.iloc[3]
```

```
Out[83]: A    -0.717678  
          B    -1.17008  
          C      0.4  
          D      0.5  
          F     -1.7  
          G     -0.1  
          H    False  
Name: 2020-12-02, dtype: object
```

```
In [84]: df.iloc[3:5, 0:2]
```

```
Out[84]:
```

	A	B
2020-12-02	-0.717678	-1.170083
2020-12-03	0.844395	1.534665

Row를 선택하고 조작하기

- iloc 은 index의 번호를 사용한다.

```
In [85]: df.iloc[[0, 1, 3], [1, 2]]
```

```
Out[85]:
```

	B	C
2020-11-29	0.081155	0.1
2020-11-30	0.848011	0.2
2020-12-02	-1.170083	0.4

```
In [86]: df.iloc[:, 1:4]
```

```
Out[86]:
```

	B	C	D
2020-11-29	0.081155	0.1	0.5
2020-11-30	0.848011	0.2	0.5
2020-12-01	-0.633725	0.3	0.5
2020-12-02	-1.170083	0.4	0.5
2020-12-03	1.534665	0.5	0.5
2020-12-04	-0.864390	0.6	0.5
2020-12-05	2.000000	3.0	4.0



Boolean Indexing

- Column 'A'의 값이 0보다 큰 Boolean data

```
In [87]: df['A'] > 0
Out[87]: 2020-11-29    False
          2020-11-30    False
          2020-12-01     True
          2020-12-02    False
          2020-12-03     True
          2020-12-04    False
          2020-12-05     True
Name: A, dtype: bool
```

- Column 'A'의 값이 0보다 큰 모든 행의 data

```
In [88]: df.loc[df['A'] > 0, :]
Out[88]:
          A      B      C      D      F      G      H
2020-12-01  0.607475 -0.633725  0.3   0.5  -1.5  -0.2  False
2020-12-03  0.844395  1.534665  0.5   0.5   NaN   0.0   True
2020-12-05  1.000000  2.000000  3.0   4.0  -2.0  -0.7   True
```

Boolean Indexing

- 특정 값과 일치하는 행의 특정 column 데이터

```
In [89]: df.loc[df['H']==True, ['A', 'C']]
```

```
Out[89]:
```

	A	C
2020-11-29	-0.096245	0.1
2020-11-30	-0.347351	0.2
2020-12-03	0.844395	0.5
2020-12-05	1.000000	3.0

- 논리 연산 응용

```
In [90]: df.loc[(df['A']>0) & (df['B'] < 0), :]
```

```
Out[90]:
```

	A	B	C	D	F	G	H
2020-12-01	0.607475	-0.633725	0.3	0.5	-1.5	-0.2	False

Boolean Indexing

- 새로운 값 대입

```
In [91]: df.loc[df['A'] > 0, 'B'] = 0  
df
```

Out[91]:

	A	B	C	D	F	G	H
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	-0.4	True
2020-11-30	-0.347351	0.848011	0.2	0.5	-1.2	-0.3	True
2020-12-01	0.607475	0.000000	0.3	0.5	-1.5	-0.2	False
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False
2020-12-03	0.844395	0.000000	0.5	0.5	NaN	0.0	True
2020-12-04	-1.644254	-0.864390	0.6	0.5	NaN	0.1	False
2020-12-05	1.000000	0.000000	3.0	4.0	-2.0	-0.7	True

4. Data Drop



Missing Data 처리하기

Not a Number

- Missing value는 Nan으로 나타나며, np.nan으로 지정할 수 있다.

```
In [93]: df['I'] = [1.0, np.nan, 3.5, 4.1, 8.0, np.nan, 2.9]  
df
```

Out [93]:

	A	B	C	D	F	G	H	I
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	-0.4	True	1.0
2020-11-30	-0.347351	0.848011	0.2	0.5	-1.2	-0.3	True	NaN
2020-12-01	0.607475	0.000000	0.3	0.5	-1.5	-0.2	False	3.5
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False	4.1
2020-12-03	0.844395	0.000000	0.5	0.5	NaN	0.0	True	8.0
2020-12-04	-1.644254	-0.864390	0.6	0.5	NaN	0.1	False	NaN
2020-12-05	1.000000	0.000000	3.0	4.0	-2.0	-0.7	True	2.9

Missing Data 처리하기

- dropna()

- Nan값을 없애는 데에 쓰일 수 있는 함수

1. how='any' : 행의 값 중 하나라도 nan일 경우 그 행을 없앤다.

```
In [94]: dropped_df = df.dropna(how='any')  
dropped_df
```

Out [94]:

	A	B	C	D	F	G	H	I
2020-12-01	0.607475	0.000000	0.3	0.5	-1.5	-0.2	False	3.5
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False	4.1
2020-12-05	1.000000	0.000000	3.0	4.0	-2.0	-0.7	True	2.9

2. how='all' : 행의 값이 모두 nan일 경우 그 행을 없앤다.

Missing Data 처리하기

- fillna()

- Nan값에 다른 값을 넣는 함수

```
In [96]: filled_df = df.fillna(value='9.9')
filled_df
```

Out[96]:

	A	B	C	D	F	G	H	I
2020-11-29	-0.096245	0.081155	0.1	0.5	9.9	-0.4	True	1
2020-11-30	-0.347351	0.848011	0.2	0.5	-1.2	-0.3	True	9.9
2020-12-01	0.607475	0.000000	0.3	0.5	-1.5	-0.2	False	3.5
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False	4.1
2020-12-03	0.844395	0.000000	0.5	0.5	9.9	0.0	True	8
2020-12-04	-1.644254	-0.864390	0.6	0.5	9.9	0.1	False	9.9
2020-12-05	1.000000	0.000000	3.0	4.0	-2	-0.7	True	2.9

Missing Data 처리하기

- isnull()

notnull()

- Nan인지 확인하는 함수

Return DataFrame

```
In [97]: check_df = df.isnull()  
check_df
```

Out[97]:

	A	B	C	D	F	G	H	I
2020-11-29	False	False	False	False	True	False	False	False
2020-11-30	False	True						
2020-12-01	False							
2020-12-02	False							
2020-12-03	False	False	False	False	True	False	False	False
2020-12-04	False	False	False	False	True	False	False	True
2020-12-05	False							

Missing Data 처리하기

- isnull()
 - Nan인지 확인하는 함수
 - 이를 이용하여 특정 열에서 nan값을 가지는 행만을 추출하기

NaN은 numpy에 있는 값이자.

np.nan != None

a=np.nan
b=np.nan
a==b? F

```
In [98]: df.loc[df.isnull()['F'], :]
```

Out[98]:

	A	B	C	D	F	G	H	I
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	-0.4	True	1.0
2020-12-03	0.844395	0.000000	0.5	0.5	NaN	0.0	True	8.0
2020-12-04	-1.644254	-0.864390	0.6	0.5	NaN	0.1	False	NaN

행 또는 열 drop

- 특정 행 drop

```
In [99]: dropped_df = df.drop('2020-11-30')
```

// df.drop(index='2020-11-30')

= df.drop(index=datetime(2020, 11, 30))

```
Out[99]:
```

	A	B	C	D	F	G	H	I
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	-0.4	True	1.0
2020-12-01	0.607475	0.000000	0.3	0.5	-1.5	-0.2	False	3.5
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False	4.1
2020-12-03	0.844395	0.000000	0.5	0.5	NaN	0.0	True	8.0
2020-12-04	-1.644254	-0.864390	0.6	0.5	NaN	0.1	False	NaN
2020-12-05	1.000000	0.000000	3.0	4.0	-2.0	-0.7	True	2.9

행 또는 열 drop

- 특정 행 drop

list로 몇개를 대입해 drop

```
In [100]: dropped_df = df.drop(['2020-11-30', '2020-12-05'])  
dropped_df
```

Out[100]:

	A	B	C	D	F	G	H	I
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	-0.4	True	1.0
2020-12-01	0.607475	0.000000	0.3	0.5	-1.5	-0.2	False	3.5
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	-0.1	False	4.1
2020-12-03	0.844395	0.000000	0.5	0.5	NaN	0.0	True	8.0
2020-12-04	-1.644254	-0.864390	0.6	0.5	NaN	0.1	False	NaN

행 또는 열 drop

- 특정 열 drop

// df.drop(columns='G')

```
In [101]: df.drop('G', axis=1)
```

Out [101]:

	A	B	C	D	F	H	I
2020-11-29	-0.096245	0.081155	0.1	0.5	NaN	True	1.0
2020-11-30	-0.347351	0.848011	0.2	0.5	-1.2	True	NaN
2020-12-01	0.607475	0.000000	0.3	0.5	-1.5	False	3.5
2020-12-02	-0.717678	-1.170083	0.4	0.5	-1.7	False	4.1
2020-12-03	0.844395	0.000000	0.5	0.5	NaN	True	8.0
2020-12-04	-1.644254	-0.864390	0.6	0.5	NaN	False	NaN
2020-12-05	1.000000	0.000000	3.0	4.0	-2.0	True	2.9

행 또는 열 drop

- 특정 열 drop

```
In [102]: df.drop(['G', 'B'], axis=1)
```

```
Out[102]:
```

	A	C	D	F	H	I
2020-11-29	-0.096245	0.1	0.5	NaN	True	1.0
2020-11-30	-0.347351	0.2	0.5	-1.2	True	NaN
2020-12-01	0.607475	0.3	0.5	-1.5	False	3.5
2020-12-02	-0.717678	0.4	0.5	-1.7	False	4.1
2020-12-03	0.844395	0.5	0.5	NaN	True	8.0
2020-12-04	-1.644254	0.6	0.5	NaN	False	NaN
2020-12-05	1.000000	3.0	4.0	-2.0	True	2.9

5. Data 분석용 함수



외부 데이터 읽고 쓰기

- Pandas는 csv 파일, text 파일, excel 파일 등 다양한 외부 리소스의 데이터를 읽고 쓸 수 있는 기능을 제공한다.

```
In [4]: df = pd.read_csv('./test.csv')
```

```
df
```

Out[4]:

	Name	Math	Science	English
0	Amy	80	100	70
1	Tom	70	80	100
2	Michael	100	90	80
3	Clara	30	50	70

```
In [5]: df.to_csv('./test.csv')
```

통계적 함수들

- Pandas DataFrame에 적용되는 함수들은 다음과 같다.

count	전체 성분 중 nan이 아닌 값의 갯수
min, max	전체 성분의 최소, 최댓값을 계산
argmin, argmax	전체 성분의 최솟값, 최댓값이 위치한 (정수)인덱스를 반환
idxmin, idxmax	전체 인덱스 중 최솟값, 최댓값
quantile	전체 성분의 특정 사분위수에 해당하는 값을 반환 (0~1 사이)
sum	전체 성분의 합
mean	전체 성분의 평균
median	전체 성분의 중간값
mad	전체 성분의 평균값으로부터의 절대편차의 평균을 계산
std, var	전체 성분의 표준편차, 분산
cumsum	맨 첫 번째 성분부터 각 성분까지의 누적합 (0부터 더해짐)
cumprod	맨 첫 번째 성분부터 각 성분까지의 누적곱 (1부터 곱해짐)
corr	두 열의 상관계수
cov	두 열의 공분산

통계적 함수들

- sum 함수 예시

```
In [103]: data = [[1.4, np.nan], [7.1, -4.5],
                 [np.nan, np.nan], [0.7, -1.3]]
df = pd.DataFrame(data, columns=["o", "x"], index=["a", "b", "c", "d"])
df
```

Out[103]:

	o	x
a	1.4	NaN
b	7.1	-4.5
c	NaN	NaN
d	0.7	-1.3

통계적 함수들

- sum 함수 예시
 - 각 열에 대해서 계산

```
In [104]: df.sum(axis=0)
```

```
Out[104]: o    9.2
           x   -5.8
           dtype: float64
```

- 각 행에 대해서 계산

```
In [105]: df.sum(axis=1)
```

```
Out[105]: a    1.4
           b    2.6
           c    0.0
           d   -0.6
           dtype: float64
```

통계적 함수들

- sum 함수 예시
 - Nan을 포함한 계산

```
In [106]: df.sum(axis=1, skipna=False)
```

```
Out[106]: a      NaN  
          b      2.6  
          c      NaN  
          d     -0.6  
          dtype: float64
```

- 특정 행 또는 열에서의 계산

```
In [108]: df['o'].sum()
```

```
Out[108]: 9.2
```

```
In [109]: df.loc['b'].sum()
```

```
Out[109]: 2.5999999999999996
```

통계적 함수들

- 상관계수와 공분산

```
In [110]: df = pd.DataFrame(np.random.randn(6, 4),  
                           columns=["A", "B", "C", "D"],  
                           index=pd.date_range("20201129", periods=6))  
df
```

Out[110]:

	A	B	C	D
2020-11-29	-1.829777	-0.264528	-0.411297	1.715259
2020-11-30	0.998870	0.467620	-0.741384	-0.550029
2020-12-01	-0.155624	0.089798	-0.180783	-0.314453
2020-12-02	0.548072	-1.361619	0.076763	-1.125462
2020-12-03	0.497136	-1.435453	1.869985	-1.212721
2020-12-04	0.985031	-0.867669	-0.461819	0.168926

통계적 함수들

- 상관계수와 공분산
 - A열과 B열의 상관계수

```
In [111]: df['A'].corr(df['B'])
```

```
Out[111]: -0.1889820257079857
```

- A열과 B열의 공분산

```
In [112]: df['A'].cov(df['B'])
```

```
Out[112]: -0.15817569637686102
```

Sorting

- DataFrame의 행과 열을 특정 기준에 따라 오름차순 또는 내림차순으로 정렬할 수 있다.

```
In [113]: dates = df.index  
random_dates = np.random.permutation(dates)  
df = df.reindex(index=random_dates, columns=['D', 'B', 'C', 'A'])  
df
```

Out[113]:

	D	B	C	A
2020-12-01	-0.314453	0.089798	-0.180783	-0.155624
2020-12-04	0.168926	-0.867669	-0.461819	0.985031
2020-12-02	-1.125462	-1.361619	0.076763	0.548072
2020-11-29	1.715259	-0.264528	-0.411297	-1.829777
2020-11-30	-0.550029	0.467620	-0.741384	0.998870
2020-12-03	-1.212721	-1.435453	1.869985	0.497136

index, columns 정렬

* df.reset_index()
df.reset_index(drop=True)

Sorting

- sort_index()
 - 행 또는 열 기준의 정렬
 - Index가 오름차순이 되도록 정렬

```
In [117]: sorted_df = df.sort_index(axis=0)  
sorted_df
```

Out[117]:

	D	B	C	A
2020-11-29	1.715259	-0.264528	-0.411297	-1.829777
2020-11-30	-0.550029	0.467620	-0.741384	0.998870
2020-12-01	-0.314453	0.089798	-0.180783	-0.155624
2020-12-02	-1.125462	-1.361619	0.076763	0.548072
2020-12-03	-1.212721	-1.435453	1.869985	0.497136
2020-12-04	0.168926	-0.867669	-0.461819	0.985031

Sorting

sort_index()

- 행 또는 열 기준의 정렬
- Column이 오름차순이 되도록 정렬

```
In [116]: sorted_df = df.sort_index(axis=1)      ↳ 행 기준 default
```

Out[116]:

	A	B	C	D
2020-12-01	-0.155624	0.089798	-0.180783	-0.314453
2020-12-04	0.985031	-0.867669	-0.461819	0.168926
2020-12-02	0.548072	-1.361619	0.076763	-1.125462
2020-11-29	-1.829777	-0.264528	-0.411297	1.715259
2020-11-30	0.998870	0.467620	-0.741384	-0.550029
2020-12-03	0.497136	-1.435453	1.869985	-1.212721

Sorting

- sort_index()
 - 행 또는 열 기준의 정렬
 - Column이 내림차순이 되도록 정렬

내림차순

```
In [118]: sorted_df = df.sort_index(axis=1, ascending=False)  
sorted_df
```

Out[118]:

	D	C	B	A
2020-12-01	-0.314453	-0.180783	0.089798	-0.155624
2020-12-04	0.168926	-0.461819	-0.867669	0.985031
2020-12-02	-1.125462	0.076763	-1.361619	0.548072
2020-11-29	1.715259	-0.411297	-0.264528	-1.829777
2020-11-30	-0.550029	-0.741384	0.467620	0.998870
2020-12-03	-1.212721	1.869985	-1.435453	0.497136

Sorting

- sort_value()
 - 값 기준의 정렬
 - A열의 값이 오름차순이 되도록 정렬

```
In [120]: sorted_df = df.sort_values(by='A')
sorted_df
```

```
Out[120]:
```

	D	B	C	A
2020-11-29	1.715259	-0.264528	-0.411297	-1.829777
2020-12-01	-0.314453	0.089798	-0.180783	-0.155624
2020-12-03	-1.212721	-1.435453	1.869985	0.497136
2020-12-02	-1.125462	-1.361619	0.076763	0.548072
2020-12-04	0.168926	-0.867669	-0.461819	0.985031
2020-11-30	-0.550029	0.467620	-0.741384	0.998870

Sorting

- sort_value()
 - 값 기준의 정렬
 - B열의 값이 내림차순이 되도록 정렬

```
In [121]: sorted_df = df.sort_values(by='B', ascending=False)
sorted_df
```

Out[121]:

	D	B	C	A
2020-11-30	-0.550029	0.467620	-0.741384	0.998870
2020-12-01	-0.314453	0.089798	-0.180783	-0.155624
2020-11-29	1.715259	-0.264528	-0.411297	-1.829777
2020-12-04	0.168926	-0.867669	-0.461819	0.985031
2020-12-02	-1.125462	-1.361619	0.076763	0.548072
2020-12-03	-1.212721	-1.435453	1.869985	0.497136

기타 함수

- unique()

행 또는 열의 유니크한 값만 얻기

```
In [125]: df['E'] = ['jazz', 'pop', 'classical', 'pop', 'jazz', 'edm']
df
```

Out[125]:

	D	B	C	A	E
2020-12-01	-0.314453	0.089798	-0.180783	-0.155624	jazz
2020-12-04	0.168926	-0.867669	-0.461819	0.985031	pop
2020-12-02	-1.125462	-1.361619	0.076763	0.548072	classical
2020-11-29	1.715259	-0.264528	-0.411297	-1.829777	pop
2020-11-30	-0.550029	0.467620	-0.741384	0.998870	jazz
2020-12-03	-1.212721	-1.435453	1.869985	0.497136	edm

```
In [126]: df['E'].unique()
```

```
Out[126]: array(['jazz', 'pop', 'classical', 'edm'], dtype=object)
```

기타 함수

- value_counts()
 - 행 또는 열에서 값에 따른 개수 얻기

```
In [128]: df['E'].value_counts()
```

```
Out[128]: pop      2  
          jazz     2  
          edm      1  
          classical 1  
          Name: E, dtype: int64
```

기타 함수

- isin()

- 지정한 행 또는 열에서 입력 값이 있는지 확인

```
In [129]: df['E'].isin(['jazz', 'edm'])
```

```
Out[129]: 2020-12-01    True
2020-12-04    False
2020-12-02    False
2020-11-29    False
2020-11-30    True
2020-12-03    True
Name: E, dtype: bool
```

- E열의 값이 jazz나 edm인 모든 행 구하기

```
In [130]: df.loc[df['E'].isin(['jazz', 'edm']), :]
```

```
Out[130]:
```

	D	B	C	A	E
2020-12-01	-0.314453	0.089798	-0.180783	-0.155624	jazz
2020-11-30	-0.550029	0.467620	-0.741384	0.998870	jazz
2020-12-03	-1.212721	-1.435453	1.869985	0.497136	edm

기타 함수

- apply()
 - 직접 만든 함수 적용하기

```
In [132]: del df['E']
```

```
func = lambda x: x.max() - x.min()
df.apply(func, axis=0)
```

```
Out[132]: D    2.927979
          B    1.903073
          C    2.611369
          A    2.828647
          dtype: float64
```

Quiz 1.

- 다음 데이터 프레임에서 지정하는 데이터를 추출하거나 처리해라.

```
data = {'Math':[80, 70, 100, 30],  
       'Science':[100, 80, 90, 50],  
       'English':[70, 100, 80, 70]}  
columns = ['Math', 'Science', 'English']  
index = ['Amy', 'Tom', 'Michael', 'Clara']  
df = pd.DataFrame(data, index=index, columns=columns)
```

- 모든 학생의 수학 점수를 Series로 나타내라. $\text{df}[\text{loc}[:, \text{'Math'}]]$
- 모든 학생의 과학과 영어 점수를 DataFrame으로 나타내라. $\text{df}[\text{loc}[:, [\text{'Science'}, \text{'English'}]]]$
- 모든 학생의 각 과목 평균 점수를 새로운 열로 추가해라. $\text{df}[\text{'mean'}] = \text{df}[\text{mean}](\text{axis}=1)$
- Tom의 영어 점수를 90점으로 수정하고 평균 점수를 다시 계산해라 $\text{df}[\text{loc}[\text{'Tom'}, \text{'English'}]] = 90$
- 각 학생에 대하여 수학 점수가 80점 이상인 학생은 Pass, 그렇지 못한 학생은 Fail로 표시하는 Result column을 추가해라.
- 평균 점수가 높은 순서대로 DataFrame을 정렬하여라.