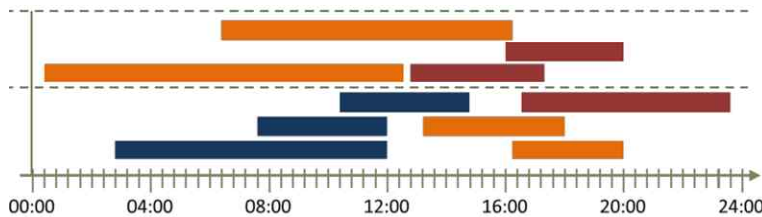
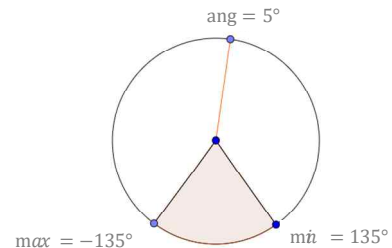


이 예습 자료에서는 **range(범위)**를 처리하는 기능과 이를 구현하는 방법에 대해 알아보겠습니다. 특히 수업 자료 Shade.py 파일에 포함된 AngleRange 클래스의 코드를 이해하는 것이 목표입니다.

여기서 말하는 range는 **연속된 값의 범위**를 말합니다. 예를 들어  $3 \leq x \leq 5$ 는 3에서 5까지 모든 실수의 range를 나타낸다고 할 수 있습니다. 또 다른 예로  $\{1, 2, 3, 4, 5\}$ 는 1에서 5까지의 모든 정수의 range라고 할 수 있습니다. Range는 아래 그림과 같이 시간, 각도를 포함한 다양한 연속된 값의 범위를 나타내기 위해 사용됩니다.



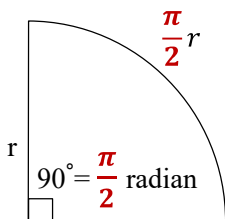
(a) 시간 구간(interval)을 range로 나타낸 예



(b) 각도 범위를 range로 나타낸 예

이번 시간 문제 풀이에는 특히 **각도의 범위**를 사용하게 되므로 이를 저장하는 클래스가 어떻게 구현되는지 보겠습니다. 특히 AngleRange 클래스는 **이러한 범위의 목록**을 저장합니다. 즉 여러 범위를 함께 저장하는 클래스입니다. 이를 이해하기 위해서 먼저 (1) 각도를 나타내는 방법을 이해하고, 이러한 (2) 각도의 범위 하나를 저장하는 방법을 살펴본 후, (3) 이러한 범위 여럿으로 이루어진 목록을 저장하고 처리하는 방법을 보도록 하겠습니다.

### <각도를 나타내는 방법>



- **각도의 단위와 직각(right angle):** 각도를 나타내기 위해 프로그래밍 언어에서는 도(°) 단위보다 (국제 표준인) 라디안 단위를 더 자주 사용합니다. 라디안 단위는 원 주위를 따라 회전한 각도를 나타내는데, 특히 원 주위를 따라 이동한 거리와 반지름 간의 비율을 나타냅니다. 예를 들어 반지름  $r$ 인 원 주위를 따라 한 번 회전(turn)하였다면  $2\pi r$ 의 거리를 이동한 것이므로  $\frac{2\pi r}{r} = 2\pi$  라디안의 각도를 회전한 것으로 봅니다. 즉  $360^\circ$ 가  $2\pi$  라디안에 해당합니다. 마찬가지로  $\frac{1}{4}$ 회전에 해당하는 직각은( $90^\circ$ )  $\frac{1}{4} \times 2\pi = \frac{\pi}{2}$  라디안입니다. 결국 도(°) 단위의 값에  $\frac{2\pi}{360} = \frac{\pi}{180}$ 을 곱하면 라디안 단위의 값이 되고, 반대로 라디안 단위의 값에  $\frac{180}{\pi}$ 를 곱하면 도(°) 단위의 값을 얻을 수 있습니다.

Python서는 math 모듈의 radians(degree) 함수를 사용하면 도(°) 단위의 값 degree를 라디안 단위로 변환할 수 있습니다. 아래 코드는  $90^\circ$ 를 라디안 단위로 변환한 예입니다. 90에  $\frac{\pi}{180}$ 을 곱한 값도 구해서 math.radians(90)과 비교해 보면 같은 결과를 얻을 수 있습니다. math.pi는  $\pi$ 를 나타내는 상수입니다.

```
>>> import math
>>> math.radians(90)
1.5707963267948966
```

```
>>> 90 * math.pi / 180
1.5707963267948966
```

Python에서 각도를 입력으로 받는 함수 대부분은 앞에서 본 것처럼 라디안 단위를 입력으로 받습니다. 예를 들어 삼각함수 중  $\sin(90^\circ)$ 을 구하고자 한다면  $\sin$  함수를 의미하는 `math.sin()`을 호출하며 아래와 같이 라디안 단위로 변환한 값을 입력으로 주면 됩니다.

```
>>> math.sin(math.radians(90))
1.0
```

만약 라디안 단위의 값을 도( $^\circ$ ) 단위로 변환하고자 한다면 `math.degrees(radian)` 함수를 사용하면 됩니다. 아래 코드는  $\pi$  라디안을 도( $^\circ$ ) 단위로 변환한 예입니다.  $\pi$ 에  $\frac{180}{\pi}$ 를 곱한 값도 구해서 `math.degrees(math.pi)`와 비교해 보면 같은 결과를 얻을 수 있습니다.

```
>>> math.degrees(math.pi)
180.0
>>> math.pi * 180 / math.pi
180.0
```

**[Q]** Python에서 삼각 함수  $\cos(180^\circ)$ 을 구하고자 한다. 보기 중 올바른 코드는?

```
math.cos(180)
math.cos(math.degrees(180))
math.cos(math.radians(180))
```

※ 이 자료에서 **[Q]**로 제시된 문제는 학습 후 풀이하는 온라인 퀴즈에 그대로 나오니 학습하면서 그때그때 문제를 풀어 두세요. 온라인 퀴즈에서 보기의 순서는 바뀔 수 있으니 유의하세요. (예: 보기 1이 보기 2가 되고, 보기 2가 보기 1이 될 수 있음)

#### <단일 range를 표현하는 방법>

연속된 각도의 range는 **처음 값**과 **마지막 값** 2개로 나타낼 수 있습니다. 이를 `AngleRange` 클래스에서는 2-tuple (from, to)를 사용해 나타냅니다. 이때 from과 to는 라디안 단위의 값으로  $-\pi(-180^\circ) < \text{from} \leq \text{to} \leq \pi(180^\circ)$ 를 만족합니다. 예를 들어  $(-\frac{\pi}{2}, \frac{\pi}{2})$ 는  $-90^\circ \sim 90^\circ$ 까지의 범위를 나타냅니다.

2-tuple (from, to)가 변수 range에 저장되어 있다면, `range[0]`가 첫 원소인 from을, `range[1]`이 두 번째 원소인 to를 나타냅니다. 아래 코드를 참조하세요.

```
>>> range = (0.1, 0.2)
>>> range[0]
0.1
```

```
>>> range[1]
0.2
```

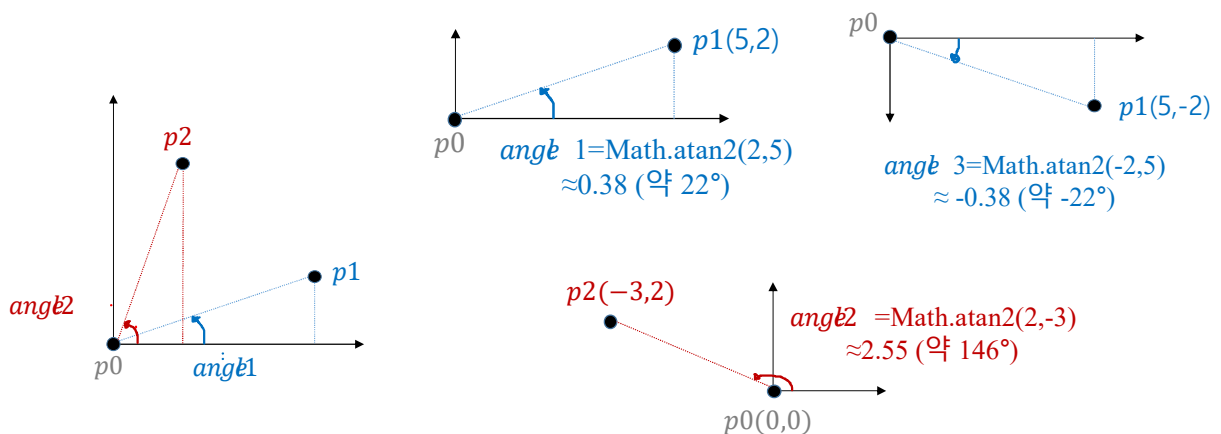
#### <여러 range를 표현하고 생성하는 방법>

AngleRange 클래스는 하나 이상의 각도의 range를 표현하는 클래스로, 멤버 변수 self.ranges에 이러한 range로 이루어진 리스트를 저장합니다. 예를 들어 self.ranges = [(0.1, 0.2), (0.3, 0.4)]라면 두 개의 range (0.1, 0.2)와 (0.3, 0.4)를 저장함을 의미합니다.

AngleRange 클래스 객체는 처음에는 0 ~ 2개의 range를 저장한 상태로 초기화되며, 이후에 다른 객체와 병합하면서 더 많은 range를 저장할 수 있습니다. AngleRange 객체를 생성하고 초기화하는 생성자 함수는 아래와 같습니다.

```
00 class AngleRange:
01     def __init__(self, x0, y0, x1, y1, x2, y2):
02         def minMax(v1, v2):
03             if v1 <= v2: return v1, v2
04             else: return v2, v1
05
06         self.ranges = []
07         if x0 != None: # if x0 == None, create an empty instance
08             angleMin, angleMax = minMax(math.atan2(y1-y0, x1-x0), math.atan2(y2-y0, x2-x0))
09             if angleMax - angleMin <= math.pi: self.ranges.append((angleMin, angleMax))
10         else:
11             self.ranges.append((-math.pi, angleMin))
12             self.ranges.append((angleMax, math.pi))
```

Python에서 클래스의 생성자 함수는 항상 같은 이름인 “\_\_init\_\_”을 가집니다. AngleRange 클래스의 생성자는 라인 01~12에 정의되어 있는데, 입력으로 직교 좌표계에 속한 3개 점의 좌표 (x0, y0), (x1, y1), (x2, y2)를 받습니다. 이번 시간 문제에서 이러한 3개 점이 만드는 각도의 range가 필요하기 때문입니다. 이러한 range를 만드는 예인 아래 그림을 보며 방법을 이해해 보겠습니다.



(a)  $p_0, p_1, p_2$ 의 예

(b)  $\text{atan2}()$  method 반환 값의 예

(라인 01) 각도를 구할 3개 점  $p_0(x_0, y_0)$ ,  $p_1(x_1, y_1)$ ,  $p_2(x_2, y_2)$ 의 좌표를 인자로 받는데,  $p_0$ 를 기준으로  $p_1 \sim p_2$ 까지의 각도의 범위를 반환합니다. 그림 (a)의 예를 보면  $p_0$ 와  $p_1$ 이 이루는 각도가  $angle1$ ,  $p_0$ 와  $p_2$ 가 이루는 각도가  $angle2$  입니다. 이럴 때  $[angle1, angle2]$ 를 반환합니다. 두 각도의 차이는  $\pi$  ( $180^\circ$ ) 보다 작다고 가정합니다.

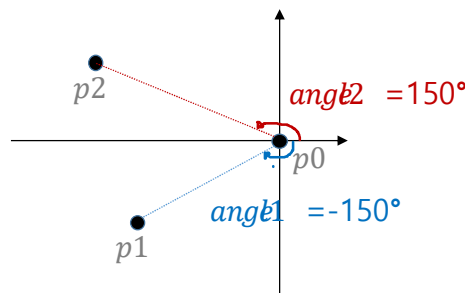
(라인 02 ~ 04) 생성자 함수가 내부적으로 사용하는 함수  $minMax(v1, v2)$ 를 정의합니다. 입력  $v1$ 과  $v2$ 를 받아 두 값을 반환하는 함수인데,  $v1$ 과  $v2$  중 더 작은 값(min)을 첫 번째 값으로, 더 큰 값(max)을 두 번째 값으로 반환합니다.

(라인 06) 멤버 변수 `self.ranges`를 비어 있는 리스트 `[]`로 초기화합니다. 여기에 앞으로 찾게 되는 각도의 범위를 추가합니다.

(라인 07) 생성자 함수의 입력  $x_0$ 가 `None`이 아닐 때만 이어지는 라인 08 ~ 12를 실행해 3개 점이 만드는 각도의 range를 계산합니다.  $x_0$ 가 `None`이라면 이어지는 라인이 실행되지 않으므로 `self.ranges`가 비어 있는 상태로 객체 생성이 끝납니다.

(라인 08)  $angle1$ 과  $angle2$ 를 구하기 위해  $\tan$ 의 역함수를 사용합니다. 예를 들어  $angle1$ 에 대해서는  $\tan(angle1) = \frac{y1-y0}{x1-x0}$  이며, 따라서  $angle1 = \tan^{-1}(\frac{y1-y0}{x1-x0})$  입니다. 라인 08의 '`math.atan2(y1-y0, x1-x0)`'가 이 값을 구하는 부분입니다. `math.atan2(y, x)` 함수는  $\tan$ 의 역함수 값을 구하며 라디안 단위로  $-\pi(-180^\circ) \sim \pi(180^\circ)$  사이의 값을 반환합니다. 그림 (b)에서 `math.atan2(y, x)` 함수의 반환 값 예를 보세요. 이어지는 `math.atan2(y2-y0, x2-x0)`는 같은 방식으로  $angle2$ 를 구합니다. 이렇게 구한 두 각도 중 작은 쪽을 `angleMin`에, 큰 쪽을 `angleMax`에 저장합니다.

(라인 09 ~ 12) 앞에서 구한 각도의 범위를 `self.ranges`에 추가하는 부분입니다. 많은 경우 하나의 range인  $[angle1, angle2]$ 를 추가하면 되지만 예외적으로 이를 두 range로 쪼개어 반환해야 하는 경우가 있어 if 조건문을 사용하게 되었습니다. 아래 그림과 같이 계산한 두 각도  $angle1$ ,  $angle2$ 의 차이가  $180^\circ$ 를 초과할 때는 하나의 범위인  $-150^\circ \sim 150^\circ$ 가 아닌 두 개의 범위  $-180^\circ \sim -150^\circ$ 와  $150^\circ \sim 180^\circ$ 를 추가합니다.  $p_1$ 과  $p_2$ 가 이루는 각도가  $\pi(180^\circ)$  보다 작다고 가정했기 때문입니다. 이 조건을 나타내는 부분이 라인 10 ~ 12입니다. 이에 해당하지 않는 때는 라인 9와 같이 한 개의 범위를 추가합니다.

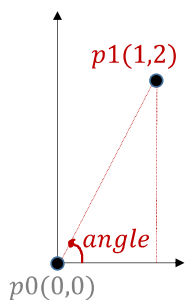


아래 코드는 지금까지 본 `AngleRange`의 생성자를 사용해 객체를 생성하고 출력한 예입니다. `ag1`은 생성자의 입력으로 `None`을 주었으므로 range 없이 비어 있는 객체가 됩니다. `ag2`는 한 개의 range만 포함한 객체가 되며 (라인 09), `ag3`은 두 개의 range를 포함한 객체가 됩니다 (라인 10 ~ 11).

```
>>> ag1 = AngleRange(None, None, None, None, None, None)
>>> print(ag1)

>>> ag2 = AngleRange(0, 0, 5, -2, 5, 2)
>>> print(ag2)
(-0.3805063771123649, 0.3805063771123649)
>>> ag3 = AngleRange(0, 0, -5, 2, -5, -2)
>>> print(ag3)
(-3.141592653589793, -2.761086276477428) (2.761086276477428, 3.141592653589793)
```

**[Q]** 아래 그림에서 붉은색 'angle'로 표기된 각도를 계산하기 위해서는 `math.atan2` 함수의 인자로 무엇을 전달해야 하는가?



```
math.atan2(1,2)
math.atan2(2,1)
```

**[Q]** 아래와 같이 `AngleRange` 객체를 생성하면 몇 개의 `range`를 포함한 객체가 생성되는가?

```
ag = AngleRange(0, 0, -2, 2, -2, -2)
```

- 입력으로 `None`이 주어졌으므로 0개의 `range`를 포함한다.
- 3개 점이 만드는 각도가 180도 이내이므로 `angleMin ~ angleMax`까지 1개의 `range`를 포함한다.
- 3개 점이 만드는 각도가 180도를 넘어가므로 `-180도 ~ angleMin`까지와 `angleMax ~ 180도`까지 2개의 `range`를 포함한다.




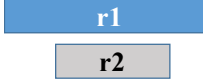


### <두 `range`의 교집합 구하기>

이제부터는 각도의 `range`에 대해 자주 수행하는 기능(연산) 중 이번 시간에 사용할 기능에 대해 차례대로 알아보겠습니다.

두 `range` `r1`과 `r2`의 교집합을 구하려면 먼저 이들이 겹치는지(교집합이 존재하는지) 확인해야 합니다. 다음 조건 (a)를 만족할 때 `r1`과 `r2`가 겹친다고 할 수 있습니다.

$$\max(r1.from, r2.from) < \min(r1.to, r2.to) \quad (a)$$







아래 표의 예제를 보며 두 range가 겹치는 경우 조건 (a)를 만족함을 확인하세요. 경우 ①, ②와 같이 두 range가 겹치는 경우 위 조건을 만족합니다. 하지만 경우 ③과 같이 겹치지 않는 경우 조건을 만족하지 않습니다. 특히 두 range가 겹칠 때 이들의 교집합은  $\max(r1.from, r2.from)$ 에서  $\min(r1.to, r2.to)$ 까지의 range가 됨도 확인하세요.

경우	예제		from, to 값의 관계
①일부 겹침			$\max(r1.from, r2.from) < \min(r1.to, r2.to)$
②포함			$\max(r1.from, r2.from) < \min(r1.to, r2.to)$
③겹치지 않음			$\max(r1.from, r2.from) > \min(r1.to, r2.to)$

AngleRange 클래스에는 아래와 같은 정적 함수가 있으며, 이는 지금까지 본 조건에 따라 두 range가 겹치는지 아닌지를 판단합니다.

```
20 def intersectAngles(range1, range2):
21     return max(range1[0], range2[0]) < min(range1[1], range2[1])
```

지금까지는 두 개의 단일 range 간에 겹치는지를 판단하였습니다. 이를 활용하면 여러 range로 구성된 AngleRange 클래스 객체 간에 겹치는 부분이 있는지를 판단할 수 있습니다. 예를 들어 아래 표의 경우 ①에서는 푸른색 리스트와 회색 리스트 간에 겹치는 부분이 있습니다. 하지만 경우 ②에서는 두 리스트 간에 겹치는 부분이 없습니다. 이처럼 두 리스트를 비교하려면 이들을 구성하는 range를 적절히 짝지어 비교하면 됩니다. 특히 이어지는 멤버 함수 intersectWith(self, ag)는 이러한 방법으로 두 AngleRange 객체에 속한 range끼리 겹치는지 비교합니다.

경우	예제			비고
①일부 겹침				self.ranges[2]와 ag.ranges[1]이 겹침
②겹치지 않음				어떤 range도 겹치지 않음

```
30 def intersectWith(self, ag):
31     indexSelf, indexAg = 0, 0
32     while indexSelf < len(self.ranges) and indexAg < len(ag.ranges):
33         if AngleRange.intersectAngles(self.ranges[indexSelf], ag.ranges[indexAg]): return True
34         if AngleRange.lessThanAngles(self.ranges[indexSelf], ag.ranges[indexAg]): indexSelf += 1
35         else: indexAg += 1
36     return False
```

intersectWith(self, ag) 함수는 이 함수를 호출한 객체 self의 ranges 리스트와 함수 인자로 받은 객체 ag의 ranges 리스트를 비교해 서로 겹치는 range가 있다면 True를 반환하고, 그러한 range가 없다면 False를 반환합니다. 이를 위해 ranges에 저장한 range가 from 순으로 (from이 같다면 to 순으로) 정렬되어 있다고 가정합니다. 비교 순서는 다음과 같습니다. 먼저 시작 index를 0으로 설정해서 (라인 31) 두 리스트의 첫 range끼리 겹치는지 비교합니다. 이후에는 방금 비교한 두 range 중 더 작은 쪽의 index를 1 증가시키는 것을 반복합니다 (라인 34 ~ 35). 이러한 순서로 비교하다가 겹치는 range가 나오면 True를 반환하고 (라인 33), 겹치는 range 없이 비교를 마치면 False를 반환합니다 (라인 36). 예를 들어 위 표의 경우 ①이라면 다음 순으로 비교합니다.

```
self.ranges[0]와 ag.ranges[0] 비교 → 겹치지 않음 → 더 작은 self쪽 index 1 증가
self.ranges[1]와 ag.ranges[0] 비교 → 겹치지 않음 → 더 작은 ag쪽 index 1 증가
self.ranges[1]와 ag.ranges[1] 비교 → 겹치지 않음 → 더 작은 self쪽 index 1 증가
self.ranges[2]와 ag.ranges[1] 비교 → 겹침 → True 반환
```

위 표의 경우 ②라면 다음 순으로 비교합니다.

```
self.ranges[0]와 ag.ranges[0] 비교 → 겹치지 않음 → 더 작은 self쪽 index 1 증가
self.ranges[1]와 ag.ranges[0] 비교 → 겹치지 않음 → 더 작은 self쪽 index 1 증가
self.ranges[2]와 ag.ranges[0] 비교 → 겹치지 않음 → 더 작은 ag쪽 index 1 증가
ag쪽에 더는 비교할 range가 남아있지 않으므로 while loop 종료 → False 반환
```

이러한 방식으로 비교하면 두 리스트의 길이의 합  $\text{len}(\text{self.ranges}) + \text{len}(\text{ag.ranges})$ 에 비례하는 시간 이내에 비교를 완료할 수 있습니다. 만약 2개의 중첩 loop을 사용해 모든 가능한 range의 쌍을 비교했다면  $\text{len}(\text{self.ranges}) * \text{len}(\text{ag.ranges})$ 에 비례한 시간이 필요했을 것입니다.

아래 코드는 지금까지 본 intersectWith 함수를 사용해 세 AngleRange 객체 ag1, ag2, ag3 간에 겹치는 range가 있는지 검사한 예입니다. ag1와 ag2 간에는 겹치는 range의 쌍이 하나 이상 있으므로 True를 반환합니다. 하지만 ag3는 이들과 겹치는 range가 없으므로 False를 반환합니다.

```
>>> ag1 = AngleRange(None, None, None, None, None, None)
>>> ag1.ranges.append((0, 0.3))
>>> ag1.ranges.append((0.4, 0.6))
>>> ag1.ranges.append((0.8, 1.0))
>>> ag2 = AngleRange(None, None, None, None, None, None)
>>> ag2.ranges.append((0.2, 0.5))
>>> ag2.ranges.append((0.7, 0.9))
>>> ag3 = AngleRange(None, None, None, None, None, None)
>>> ag3.ranges.append((1.1, 1.2))
>>> print(ag1.intersectWith(ag2))
True
>>> print(ag1.intersectWith(ag3))
False
>>> print(ag2.intersectWith(ag3))
False
```

[Q] 두 range (1, 4)와 (2, 5)의 교집합은 무엇인가?

- (1, 4)
- (1, 5)
- (2, 4)
- (2, 5)
- 교집합이 존재하지 않음


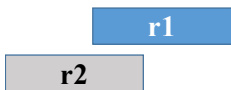
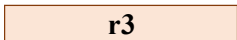
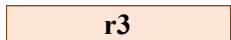
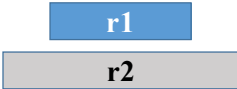
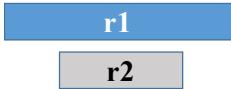
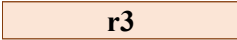
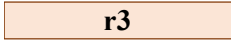
#### <두 range의 합집합 구하기(union, merge)>

같은 리스트에 속한 두 range가 겹칠 때는 한 range로 결합하는 것이 편리한 때도 있습니다. 예를 들어 자동차 전방의 한 장애물이 1 ~ 3시 방향의 시야를 가리며 다른 장애물이 2 ~ 4시 방향의 시야를 가린다면 결과적으로 두 범위를 결합한 1 ~ 4시 방향의 시야가 가린다고 말할 수 있습니다.

두 range r1과 r2가 겹치는 경우(교집합이 존재하는 경우) 이들의 합집합 r3은 다음 조건 (b)와 같이 from의 최솟값에서 to의 최댓값까지의 range입니다.

$$\begin{aligned} r3.from &= \min(r1.from, r2.from); & (b) \\ r3.to &= \max(r1.to, r2.to) \end{aligned}$$

아래 표의 예제 각각을 보며 두 range가 겹치는 경우 합집합은 조건 (b)와 같이 결정됨을 확인하세요.

경우	예제	
① 두 range가 일부 겹치는 경우		
①의 합집합		
② 한 range가 다른 range에 완전히 포함되는 경우		
②의 합집합		

AngleRange 클래스에는 아래와 같은 정적 함수가 있으며, 이 함수는 지금까지 본 조건에 따라 두 range의 합집합을 생성하여 반환합니다.

```
40 def mergeAngles(range1, range2):
41     return (min(range1[0], range2[0]), max(range1[1], range2[1]))
```

지금까지는 두 개의 단일 range 간의 합집합을 구했습니다. 이를 활용하면 여러 range로 구성된 AngleRange 클래스 객체 간의 합집합을 구할 수 있습니다. 이번에도 두 리스트를 구성하는 range를 적절히 짝지어 결합해 가면 됩니다. 특히 이어지는 멤버 함수 mergeWith(self, ag)는 이러한 방법으로 두 AngleRange 객체에 속한 range 간의 합집합을 구합니다.

```
50 def mergeWith(self, ag):
51     result = []
```



```

52  indexSelf, indexAg = 0, 0
53  while indexSelf < len(self.ranges) and indexAg < len(ag.ranges):
54      if AngleRange.intersectAngles(self.ranges[indexSelf], ag.ranges[indexAg]):
55          if self.ranges[indexSelf][1] >= ag.ranges[indexAg][1]:
56              self.ranges[indexSelf] = AngleRange.mergeAngles(self.ranges[indexSelf], ag.ranges[indexAg])
57              indexAg += 1
58          else:
59              ag.ranges[indexAg] = AngleRange.mergeAngles(self.ranges[indexSelf], ag.ranges[indexAg])
60              indexSelf += 1
61      else:
62          if AngleRange.lessThanAngles(self.ranges[indexSelf], ag.ranges[indexAg]):
63              result.append(self.ranges[indexSelf])
64              indexSelf += 1
65          else:
66              result.append(ag.ranges[indexAg])
67              indexAg += 1
68
69      while indexSelf < len(self.ranges):
70          result.append(self.ranges[indexSelf])
71          indexSelf += 1
72      while indexAg < len(ag.ranges):
73          result.append(ag.ranges[indexAg])
74          indexAg += 1
75
76      self.ranges = result

```

mergeWith(self, ag) 함수는 이 함수를 호출한 객체 self의 ranges 리스트와 함수 인자로 받은 객체 ag의 ranges 리스트를 병합하여 그 결과를 self.ranges에 저장합니다. 이를 위해 ranges에 저장한 range가 from 순으로 (from이 같다면 to 순으로) 정렬되어 있다고 가정합니다. 병합 순서는 intersectWith() 함수와 유사합니다. 먼저 시작 index를 0으로 설정해서 (라인 52) 두 리스트의 첫 range끼리 겹치는지 비교합니다. 이후에는 방금 비교한 두 range 중 한쪽의 index를 1 증가시키는 것을 반복하다가 더는 비교할 range가 없어지면 while loop을 중단합니다 (라인 53 ~ 67). 이러한 순서로 비교하다가 겹치는 range가 나오면 두 range를 병합하며 (라인 54 ~ 60), 겹치지 않는다면 더 작은 range를 결과 리스트 result에 추가해 갑니다 (라인 62 ~ 67). self와 ag 중 한쪽 리스트의 range 비교가 다 끝났다면 while loop을 벗어난 후, 아직 비교하지 않고 남은 range를 모두 result에 추가합니다 (라인 69 ~ 74). 마지막으로 결과 리스트를 self.ranges에 저장합니다 (라인 76).



예를 들어 위 그림의 경우라면 다음 순으로 병합합니다.

self.ranges[0]와 ag.ranges[0] 비교 → 겹치지 않음 → 더 작은 self.ranges[0]를 result에 추가하고 self

쪽 index 1 증가

self.ranges[1]와 ag.ranges[0] 비교 → 겹치지 않음 → 더 작은 ag.ranges[0]를 result에 추가하고 ag쪽 index 1 증가

self.ranges[1]와 ag.ranges[1] 비교 → 겹치지 않음 → 더 작은 self.ranges[1]을 result에 추가하고 self 쪽 index 1 증가

self.ranges[2]와 ag.ranges[1] 비교 → 겹침 → 두 range를 병합해 더 큰 self.ranges[2]에 저장하고 더 작은 ag쪽 index 1 증가

ag쪽에 더는 비교할 range가 남아있지 않으므로 while loop 종료 → self 쪽에 남은 range인 self.ranges[2]를 result에 추가하고 종료

결과적으로 result에는 다음과 같은 4개의 range가 담깁니다. 병합 이후에도 from 순으로 정렬된 상태를 계속 유지함을 확인하세요. 이러한 방식으로 병합하면 두 리스트의 길이의 합  $\text{len}(\text{self.ranges}) + \text{len}(\text{ag.ranges})$ 에 비례하는 시간 이내에 병합을 완료할 수 있습니다.

[self.ranges[0], ag.ranges[0], self.ranges[1], ag.ranges[1]과 self.ranges[2]의 합집합]

아래 코드는 지금까지 본 mergeWith 함수를 사용해 세 AngleRange 객체 ag1, ag2, ag3의 합집합을 구해 ag1에 저장하는 예입니다. ag1와 ag2 간에는 겹치는 range 들이 여럿 있으므로 이들이 결과 리스트에서 병합되었음을 확인하세요. 하지만 ag3는 이들과 겹치는 range가 없으므로 (1.1, 1.2)가 그대로 결과 리스트에 포함됩니다.

```
>>> ag1 = AngleRange(None, None, None, None, None, None)
>>> ag1.ranges.append((0, 0.3))
>>> ag1.ranges.append((0.4, 0.6))
>>> ag1.ranges.append((0.8, 1.0))
>>> ag2 = AngleRange(None, None, None, None, None, None)
>>> ag2.ranges.append((0.2, 0.5))
>>> ag2.ranges.append((0.7, 0.9))
>>> ag3 = AngleRange(None, None, None, None, None, None)
>>> ag3.ranges.append((1.1, 1.2))
>>> ag1.mergeWith(ag2)
>>> ag1.mergeWith(ag3)
>>> print(ag1)
(0, 0.6) (0.7, 1.0) (1.1, 1.2)
```

**[Q]** 두 range (1, 4)와 (2, 5)의 합집합은 무엇인가?

(1, 4)

(1, 5)

(2, 4)

(2, 5)

합집합이 존재하지 않음