

2025.04.08. SR 파이썬 스터디 3주차

조코딩 05. 클래스, 모듈, 패키지, 예외 처리, 내장함수, 라이브러리

클래스(Class)

함수의 한계

나는 더 편리하게 빵을 사오도록 시키기 위해서, 어떤 빵이든 이름만 알려주면 사 오는 프로그램을 만들었다.

그리 똑똑하진 않아서, 로봇 한 대가 여러 빵을 찾도록 할 생각은 못 했다.

그래서 이전 챕터에서의 로봇을 여러 대 더 사고, 여러 대를 한 번에 보내 각자 빵을 하나씩 집어오게 할 계획을 세웠다.

컴퓨터까지 여러 대 사기엔 너무 비싸서, 이전 챕터에서 쓰던 한 대의 컴퓨터가 이 로봇들을 동시에 원격으로 조종하도록 했다.

기왕 만든 김에, 마트를 다녀오는 코드도 만들었다.

```
def 빵 사오기(빵 이름) :
```

1. 카드를 잡는다.
2. 집 문을 열고 엘리베이터 앞에 도달할 때까지 직진한다.
3. 엘리베이터가 나오면 엘리베이터를 바라본다.
4. 엘리베이터 호출 버튼을 누른다.
5. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
6. 뒤로 돋다.
7. 1층 버튼을 누른다.
8. 엘리베이터 문이 다시 열리면 현관문 앞에 도달할 때까지 직진한다.
9. (현관에서 직진하면 빵집이 있다고 가정) 현관문이 열리면 빵집 앞에 도달할 때까지 직진한다.
10. 빵집의 문을 당긴다.
11. 만약 빵집의 문이 열린다면 입력한 이름을 가진 빵이 나올 때까지 2번 빵집 내부를 탐색한다.
- 11-1. 만약 빵을 찾았다면 그걸 손에 든다. 카운터 쪽으로 가서 카드를 건넨다.
- 점원이 카드를 돌려주면 카드를 다시 잡는다.
- 빵집의 문 앞까지 간다. 문을 당긴다.
- 11-1-2. 만약 2번 탐색할 동안 입력한 이름을 가진 빵을 찾지 못했다면 빵집의 문 앞까지 간다. 문을 당긴다.
- 11-2. 만약 빵집의 문이 열리지 않는다면 뒤로 돋다.
12. 현관이 나올 때까지 직진한다.
13. 현관문이 열리면 엘리베이터 앞에 도달할 때까지 직진한다.
14. 엘리베이터 호출 버튼을 누른다.
15. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
16. 뒤로 돋다.
17. 우리 집 층수와 일치하는 숫자가 써진 버튼을 누른다.
18. 엘리베이터 문이 다시 열리면 엘리베이터 밖으로 나간다.
19. 우리 집 쪽을 바라본다.
20. 집 문 앞에 도달할 때까지 직진한다.
21. 초인종을 누른다.

```
def 마트 다녀오기(물품 이름) :
    <세부 코드 생략>
```

그런데 문제가 있다.

이렇게 코드를 짜면 여러 대의 로봇을 동시에 조종하지 못한다.

그래서 각각의 로봇에 적용할 수 있는 코드를 새로 짬다.

로봇의 이름은 로봇1, 로봇2, 로봇3, ... 으로 했다.

(비효율성을 보여주기 위해 일부러 쓸데없이 길게 코드를 만들었음.)

```
def 로봇1 빵 사오기(빵 이름) :
    1. 명령 대상을 로봇1로 설정한다
    2. 카드를 잡는다.
    3. 집 문을 열고 엘리베이터 앞에 도달할 때까지 직진한다.
    4. 엘리베이터가 나오면 엘리베이터를 바라본다.
    5. 엘리베이터 호출 버튼을 누른다.
    6. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
    7. 뒤로 돋다.
    8. 1층 버튼을 누른다.
    9. 엘리베이터 문이 다시 열리면 현관문 앞에 도달할 때까지 직진한다.
    10. (현관에서 직진하면 빵집이 있다고 가정) 현관문이 열리면 빵집 앞에 도달할 때까지 직진한다.
    11. 빵집의 문을 당긴다.
        11-1. 만약 빵집의 문이 열리다면 입력한 이름을 가진 빵이 나올 때까지 2번 빵집 내부를 탐색한다.
            11-1-1. 만약 빵을 찾았다면 그걸 손에 든다. 카운터 쪽으로 가서 카드를 건넨다.
                점원이 카드를 돌려주면 카드를 다시 잡는다.
                빵집의 문 앞까지 간다. 문을 당긴다.
            11-1-2. 만약 2번 탐색할 동안 입력한 이름을 가진 빵을 찾지 못했다면 빵집의 문 앞까지 간다. 문을 당긴다.
        11-2. 만약 빵집의 문이 열리지 않는다면 뒤로 돋다.
    12. 현관이 나올 때까지 직진한다.
    13. 현관문이 열리면 엘리베이터 앞에 도달할 때까지 직진한다.
    14. 엘리베이터 호출 버튼을 누른다.
    15. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
    16. 뒤로 돋다.
    17. 우리 집 층수와 일치하는 숫자가 써진 버튼을 누른다.
    18. 엘리베이터 문이 다시 열리면 엘리베이터 밖으로 나간다.
    19. 우리 집 쪽을 바라본다.
    20. 집 문 앞에 도달할 때까지 직진한다.
    21. 초인종을 누른다.
```

```
def 로봇1 마트 다녀오기(물품 이름) :
    <세부 코드 생략>
```

```
def 로봇2 빵 사오기(빵 이름) :
```

1. 명령 대상을 로봇2로 설정한다
2. 카드를 잡는다.
3. 집 문을 열고 엘리베이터 앞에 도달할 때까지 직진한다.
4. 엘리베이터가 나오면 엘리베이터를 바라본다.
5. 엘리베이터 호출 버튼을 누른다.

6. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
7. 뒤로 돈다.
8. 1층 버튼을 누른다.
9. 엘리베이터 문이 다시 열리면 현관문 앞에 도달할 때까지 직진한다.
10. (현관에서 직진하면 빵집이 있다고 가정) 현관문이 열리면 빵집 앞에 도달할 때까지 직진한다.
11. 빵집의 문을 당긴다.
- 11-1. 만약 빵집의 문이 열린다면 입력한 이름을 가진 빵이 나올 때까지 2번 빵집 내부를 탐색한다.
- 11-1-1. 만약 빵을 찾았다면 그걸 손에 든다. 카운터 쪽으로 가서 카드를 건넨다.

점원이 카드를 돌려주면 카드를 다시 잡는다.
빵집의 문 앞까지 간다. 문을 당긴다.

- 11-1-2. 만약 2번 탐색할 동안 입력한 이름을 가진 빵을 찾지 못했다면 빵집의 문 앞까지 간다. 문을 당긴다.
- 11-2. 만약 빵집의 문이 열리지 않는다면 뒤로 돈다.
12. 현관이 나올 때까지 직진한다.
13. 현관문이 열리면 엘리베이터 앞에 도달할 때까지 직진한다.
14. 엘리베이터 호출 버튼을 누른다.
15. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
16. 뒤로 돈다.
17. 우리 집 층수와 일치하는 숫자가 써진 버튼을 누른다.
18. 엘리베이터 문이 다시 열리면 엘리베이터 밖으로 나간다.
19. 우리 집 쪽을 바라본다.
20. 집 문 앞에 도달할 때까지 직진한다.
21. 초인종을 누른다.

```
def 로봇2 마트 다녀오기(물품 이름) :
    <세부 코드 생략>
```

- def 로봇3 빵 사오기(빵 이름) :
 1. 명령 대상을 로봇3으로 설정한다
 2. 카드를 잡는다.
 3. 집 문을 열고 엘리베이터 앞에 도달할 때까지 직진한다.
 4. 엘리베이터가 나오면 엘리베이터를 바라본다.
 5. 엘리베이터 호출 버튼을 누른다.
 6. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
 7. 뒤로 돈다.
 8. 1층 버튼을 누른다.
 9. 엘리베이터 문이 다시 열리면 현관문 앞에 도달할 때까지 직진한다.
10. (현관에서 직진하면 빵집이 있다고 가정) 현관문이 열리면 빵집 앞에 도달할 때까지 직진한다.
11. 빵집의 문을 당긴다.
- 11-1. 만약 빵집의 문이 열린다면 입력한 이름을 가진 빵이 나올 때까지 2번 빵집 내부를 탐색한다.
- 11-1-1. 만약 빵을 찾았다면 그걸 손에 든다. 카운터 쪽으로 가서 카드를 건넨다.

점원이 카드를 돌려주면 카드를 다시 잡는다.
빵집의 문 앞까지 간다. 문을 당긴다.

- 11-1-2. 만약 2번 탐색할 동안 입력한 이름을 가진 빵을 찾지 못했다면 빵집의 문 앞까지 간다. 문을 당긴다.
- 11-2. 만약 빵집의 문이 열리지 않는다면 뒤로 돈다.
12. 현관이 나올 때까지 직진한다.
13. 현관문이 열리면 엘리베이터 앞에 도달할 때까지 직진한다.

14. 엘리베이터 호출 버튼을 누른다.
15. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
16. 뒤로 돈다.
17. 우리 집 층수와 일치하는 숫자가 써진 버튼을 누른다.
18. 엘리베이터 문이 다시 열리면 엘리베이터 밖으로 나간다.
19. 우리 집 쪽을 바라본다.
20. 집 문 앞에 도달할 때까지 직진한다.
21. 초인종을 누른다.

```
def 로봇3 마트 다녀오기(물품 이름) :
    <세부 코드 생략>
    ...
```

이렇게 코드를 짜면 각각의 로봇을 컴퓨터 한 대로 조종할 수 있다.

그러나 다들 느꼈겠지만 비효율적이다. 일단 보기 가 안 좋다.
 그리고 저장공간도 많이 차지하는 데다가, 로봇을 더 들여올 때마다 들여온 만큼 저 코드를 또다시 써넣어야 한다.
 무엇보다 저 코드를 수정할 일이 생기면 같은 수정 작업을 여러 번 반복해야 한다.
 만약 로봇이 100대 있다면? 상상하기도 싫다...

어차피 똑같은 코드를 계속 복제하는 거라면, 그냥 원본 코드만 따로 저장해 놓고 쓸 일이 생길 때마다 불러올 순 없을까?

그렇게 하기 위해 생긴 개념이 클래스(Class)이다.

클래스(Class)

- 같은 함수를 여러 곳에서 쓰기 편하도록 묶어 놓은 것.
- 더 정확한 정의: 변수와 메소드(클래스 안에서 정의된 함수)의 묶음.

```
class 로봇제어 :
    def 빵 사오기(self, 빵 이름) :
        1. 명령 대상을 로봇1로 설정한다
        2. 카드를 잡는다.
        3. 집 문을 열고 엘리베이터 앞에 도달할 때까지 직진한다.
        4. 엘리베이터가 나오면 엘리베이터를 바라본다.
        5. 엘리베이터 호출 버튼을 누른다.
        6. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
        7. 뒤로 돈다.
        8. 1층 버튼을 누른다.
        9. 엘리베이터 문이 다시 열리면 현관문 앞에 도달할 때까지 직진한다.
        10. (현관에서 직진하면 빵집이 있다고 가정) 현관문이 열리면 빵집 앞에 도달할 때까지 직진한다.
        11. 빵집의 문을 당긴다.
            11-1. 만약 빵집의 문이 열린다면 입력한 이름을 가진 빵이 나올 때까지 2번 빵집 내부를 탐색한다.
                11-1-1. 만약 빵을 찾았다면 그걸 손에 든다. 카운터 쪽으로 가서 카드를 건넨다.
                    점원이 카드를 돌려주면 카드를 다시 잡는다.
                    빵집의 문 앞까지 간다. 문을 당긴다.
                11-1-2. 만약 2번 탐색할 동안 입력한 이름을 가진 빵을 찾
```

지 못했다면 빵집의 문 앞까지 간다. 문을 당긴다.

- 11-2. 만약 빵집의 문이 열리지 않는다면 뒤로 돈다.
12. 현관이 나올 때까지 직진한다.
13. 현관문이 열리면 엘리베이터 앞에 도달할 때까지 직진한다.
14. 엘리베이터 호출 버튼을 누른다.
15. 문이 열린 것이 식별되면 엘리베이터 안으로 들어간다.
16. 뒤로 돈다.
17. 우리 집 층수와 일치하는 숫자가 써진 버튼을 누른다.
18. 엘리베이터 문이 다시 열리면 엘리베이터 밖으로 나간다.
19. 우리 집 쪽을 바라본다.
20. 집 문 앞에 도달할 때까지 직진한다.
21. 초인종을 누른다.

```
def 마트 다녀오기(self, 물품 이름) :
    <세부 코드 생략>
```

로봇1제어 = 로봇제어
로봇1제어.마트 다녀오기(새우깡)

로봇2제어 = 로봇제어
로봇2제어.빵 사오기(우유식빵)

이런 식으로 클래스는 함수를 묶을 수 있다.
다시 정리하면,

1. 함수는 명령어들을 묶어놓은 것이고,
2. 클래스는 함수들을 묶어놓은 것이다. 클래스로 묶인 함수는 특별히 '메소드'라고 부른다.

클래스는 다른 곳에 무한정 복제할 수 있다.
클래스의 복제는 변수를 지정하듯이 하면 된다.

위의 코드에서 유념할 점이 두 가지 있다.

첫 번째는 def문에 self라는 변수는 갑자기 왜 튀어나온 것인지이고,
두 번째는 그럼에도 불구하고 하나의 코드로 로봇을 여러 대 조종할 수 없다는 문제는 해결되지 않았다는 것이다.



self는 어디서 튀어나온 것인가?

클래스로 묶인 함수, 다시말해 클래스 안에서 정의된 함수는 따로 '메소드'라 불린다고 했다. 그 이유는 그 성질이 특별해서이다.

클래스 안에 있는 메소드들은 그 클래스가 복제된 변수에서만 사용된다.
예를 들어

로봇1제어 = 로봇제어

라는 명령어로 '로봇1제어'라는 변수에서 '로봇제어'라는 클래스를 쓸 것이라고 선언했다고 하자.

근데 이 상황에서

로봇5.빵 사오기(우유식빵)

a.빵 사오기(우유식빵)

고양이.빵 사오기(우유식빵)

이런 식으로 클래스와 관련 없는 변수에다가 클래스에 있는 기능(메소드)을 적용할 순 없다는 것이다.

로봇1.빵 사오기

이렇게 클래스가 정의되어 있는 로봇1이라는 변수에만 메소드를 적용할 수 있다.

그리고 그 변수를 클래스 안에서는 self라고 부른다.

클래스 안에서 정의된 모든 메소드는 기본적으로 self 인자를 괄호 안에 넣어줘야 한다.

하나의 코드로 로봇을 여러 대 조종할 수 없는 문제는 아직도 해결되지 않았다.

```
class 로봇제어 :
    def 빵 사오기(self, 빵 이름) :
        1. 명령 대상을 로봇1로 설정한다
```

로봇제어 클래스의 이 부분에 집중해 보자.

지금도 여전히 명령 대상을 로봇1로만 설정할 수 있다는 걸 알 수 있다..

따라서 애초에 클래스를 선언할 때부터 이게 어떤 로봇을 제어하는 클래스인지 지정할 필요가 있다.

로봇1제어 = 로봇제어(로봇1)

로봇2제어 = 로봇제어(로봇2)

이런 식으로!!

그걸 구현하기 위해 나온 것이 '생성자', `_init_` 이다.

✓ `_init_()`

- 클래스 선언 시에 사용할 수 있는 특수 기능 중 하나임.
- 클래스를 선언할 때 `init` 밑에 딸린 코드는 같이 실행됨.
- `init`이 없어도 클래스 생성이 가능함. 클래스를 불러올 때 초기값을 설정해줘야 할 때 `init`을 사용함.
- `_init_()`의 괄호 안에 변수를 써 주면 클래스를 선언할 때도 `init`이 요구하는 변수를 써 줘야 함.
ex)

```
class A :
    def __init__(self, a, b) :
        self.a = a
        self.b = b
k = A(1, 2)
```

이 기능을 사용하면

```
class 로봇제어 :
    def __init__(self, 로봇이름) :
        self.로봇이름 = 로봇이름
    def 빵 사오기(빵 이름) :
        1. 명령 대상을 self.로봇이름으로 설정한다
```

이렇게 로봇 이름을 따로 지정해 줄 수 있다.

그럼

```
로봇1제어 = 로봇제어(로봇1)
로봇1제어.빵 사오기
```

```
로봇2제어 = 로봇제어(로봇2)
로봇2제어.빵 사오기
```

이제 어떤 로봇이든 자유롭게 명령할 수 있게 됐다.

지금까진 설명하기 편하게 코드에 한글을 섞어 사용하다 보니 너무 추상적인 설명을 했다.

실제 코드 형식으로 init문이 포함된 클래스를 구현해 보면 다음과 같다.

```
In [29]: class ControlRobot : # 로봇제어

    def __init__(self, robot_name) : # init문. 클래스 선언과 동시에 실행된다. 클래스
        self.robot_name = robot_name

    def get_bread(self, bread_name) :
        print(f'1. 명령 대상을 {self.robot_name}으로 지정한다.')
        print('2. 빵집에 간다.')
        print(f'3. {bread_name} 찾는다. 있으면 산다.')
        print('4. 집으로 돌아온다.')
        print('---끝---\n')

robot1 = ControlRobot('robot1') # 드디어 로봇1을 조종할 것이라고 명시할 수 있게 되었다.
robot1.get_bread('우유식빵')

robot10000 = ControlRobot('robot10000') # 이제 로봇을 100대를 들여오는 10000대를 조종할 수 있게 되었다.
robot10000.get_bread('에그타르트')
```

1. 명령 대상을 robot1으로 지정한다.
 2. 빵집에 간다.
 3. 우유식빵 찾는다. 있으면 산다.
 4. 집으로 돌아온다.
- 끝---

1. 명령 대상을 robot10000으로 지정한다.
 2. 빵집에 간다.
 3. 에그타르트 찾는다. 있으면 산다.
 4. 집으로 돌아온다.
- 끝---

상속(Inheritance)

- 클래스는 다른 클래스의 기능을 물려받을 수 있음.
- 상속의 효과는 한 클래스 안에 있는 모든 메서드를 다른 클래스에 복붙하는 것과 같음.
- 물려준 클래스를 부모, 물려받은 클래스를 자식이라고 함.
- 자식 클래스에 있는 메서드와 부모 클래스에 있는 메서드의 이름이 똑같으면, 자식 클래스에 있는 걸 사용함.

```
In [38]: class Control(ControlRobot) :
    pass # 아무 것도 하지 않고 넘어감.

robot = Control('robot')
robot.get_bread('우유식빵')
```

1. 명령 대상을 robot으로 지정한다.
 2. 빵집에 간다.
 3. 우유식빵 찾는다. 있으면 산다.
 4. 집으로 돌아온다.
- 끝---

모듈(Module)

- 명령어, 함수, 클래스를 모두 묶은 것.
 - 간단하게 말하면 파이썬 파일 그 자체. 클래스든 함수든 그게 써져 있는 파일 전체임.
1. 함수: 변수, 명령어를 묶은 것.
 2. 클래스: 변수, 함수(메소드)를 묶은 것.
 3. 모듈: 변수, 함수, 클래스를 묶은 것.
- 한 파이썬 파일에서 'import'라는 명령어를 사용해 같은 폴더에 있는 다른 파이썬 파일을 불러올 수 있음.
 - 그렇게 불러와진 파일 또는 그렇게 불러오게끔 설계된 파일을 모듈이라고 함.

ex)

```
files
└─apple.py
└─banana.py
```

이런 폴더 구조가 있다고 하자.

files 폴더 안에 apple.py와 banana.py 파일이 들어 있는 구조다.

이때 banana.py 파일 안에서

```
import apple
```

명령어를 실행한다면, apple.py 파일을 banana.py에 불러온(복붙한) 것이다.

이려면 apple.py에서 쓴 모든 것들을 banana.py에서도 같이 쓸 수 있다.

그러나 banana.py 파일 안에서

```
import melon
```

이걸 실행하는 건 안 된다.

melon.py파일이 컴퓨터 어딘가에 실제로 존재한다 하더라도 banana.py 파일과 같은 폴더 안에 들어있지 않으므로 import 할 수 없다.

if __name__ == '__main__' 조건문

(그냥 참고만 해도 됨)

- 해당하는 파이썬 파일이 import되지 않고 직접 실행되면, 파이썬 자체적으로 가지고 있는 __name__이라는 변수에 '__main__'이라는 문자열이 할당됨.
- 만약 파이썬 파일이 그 파일 원본이 아닌 다른 파일에서 import되어 실행되는 거라면, __name__이라는 변수에 '__main__'이 아니라 원본 파일의 이름이 할당됨.
- 따라서 __name__ == '__main__' 이라면, 내가 실행시킨 파일이 다른 파일을 통해 실행된 게 아니라 직접 실행되었단 것을 뜻함.
- 이걸 이용해서, 파일을 직접 실행했을 때만 특정 코드가 실행되도록 만들 수 있음.

패키지(Package)

- 모듈을 묶은 폴더.
- 여러 파이썬 파일 및 폴더를 묶은 것.

import

어떤 사람이 수치해석에 유용한 여러 모듈을 만들고, 이를 묶어서 'numpy'라는 폴더를 만들었다고 하자.

이 폴더가 패키지다. 이것도 import를 통해 불러올 수 있다.

```
import numpy
```

패키지의 이름이 쓰기 불편하면, as를 활용해서 더 편한 이름을 지정할 수 있다.

```
import numpy as np
```

이럼 이 import문이 있는 파이썬 파일에 한해서는 numpy라는 패키지를 np라고 부를 수 있다.

numpy 패키지의 구조가 다음과 같다고 하자.

```
numpy
└random
    └choice.py
```

numpy 폴더 안에 random 폴더가 있고, 그 안에 choice.py 파일이 있는 구조다.
여기서 choice.py를 불러오려면

```
import numpy.random.choice
```

를 해 주면 된다. 폴더의 계층을 .(점)으로 구분해서 지정할 수 있다.

매번 저 긴 걸 다 입력하긴 귀찮을 수 있다.

그럼 from문을 사용해서 choice만 따로 빼서 사용할 수도 있다.

```
from numpy.random import choice
```

영어 뜻 그대로 numpy 안의 random 폴더에서 choice를 가져오는 것이다.



라이브러리(Library)

- 패키지를 모아 놓은 것.
- 여기에 저장되어 있는 수많은 패키지들을 개개인이 가져올 수 있음.

참고

'패키지'와 '라이브러리'는 엄밀하게 말하면 다른 개념이지만, 사실상 엄격한 구분 없이 혼용되고 있음.



표준 라이브러리

- 파이썬 자체적으로 지원하는 패키지들을 모아 놓은 것.
- 파이썬을 깔면 파이썬 프로그램 안에 같이 설치됨.
- 따로 설치할 필요 없이 바로 import해올 수 있음.



외부 라이브러리

- 파이썬 외부에 존재하는 라이브러리.

- 실력 있는 프로그래머나 프로그래밍 회사가 만들어 놓은 패키지가 아주 많이 있음.
 - 이걸 올려놓은 데이터베이스 서버가 있는데, 그 서버를 통해 설치(install)할 수 있음.
 - PyPI(Python Package Index)
 - 대표적 외부 라이브러리 서버
 - 오픈소스로 패키지를 올리고 다운받을 수 있음.
 - 파이썬 콘솔에서 다음 명령어를 통해 패키지 다운 가능:
`pip install 패키지이름`
 - pip: PyPI에서 패키지를 가져오는 프로그램. 위의 명령어는 pip를 호출하고 install 명령을 내린 것.
-

예외 처리

- 원래는 코드 실행 중 오류가 발생하면 코드 실행을 종료하고 에러 메시지를 띄움.
- 오류가 발생해도 코드 실행 종료 없이 정해진 명령을 수행하게 할 수 있는데, 이걸 예외 처리라고 함.

```
try:
    오류 발생할 수 있는 구문
except :
    오류 발생했을 때 실행되는 구문
else :
    오류 발생하지 않았을 때 실행되는 구문
finally :
    오류가 발생하건 발생하지 않건 마지막에 실행되는 구문
```

: 마치 if 조건문처럼 동작함.

except문의 여러 형태

1. except :

기본적 형태. 오류 발생 시 except : 다음에 나오는 명령어를 수행함.

2. except 오류이름 :

특정 오류 발생시에만 명령어 수행하게 만들 수 있음.

3. except - pass

`pass`를 사용해서 오류 발생했을 때 아무 것도 안 하게, 즉 오류를 무시하게 만들 수 있음.

4. except 오류이름 as e :

특정 오류 발생시에만 명령어를 수행하게 만드는데, 이때 오류 이름을 e라는 변수에

할당해서 편하게 불러올 수 있음.

✓ try - except문의 유용성

- 코드 실행의 장애물이 되는 에러를, 코드의 실행조건 중 하나로 이용할 수 있게 함.
- while True 무한반복문의 탈출조건으로 쓸 수 있음.
- 백준 등에서 풀 수 있는 코딩테스트에서, 입출력 개수가 정해지지 않았을 때 활용할 수 있음.