**5. 리스트**

1) 값의 집합: 리스트

- 리스트: 값의 집합(다른 자료형끼리 묶는 것도 가능, 리스트 안에 리스트 가능)

- 선언 방식: 리스트 이름 = [요소1, 요소2 …] / (비어 있는 리스트) 리스트 이름 = list()

시계, 실내, 검은색, 화면이(가) 표시된 사진

자동 생성된 설명

검은색, 하얀색, 빨간색이(가) 표시된 사진

자동 생성된 설명

- 리스트 내 리스트 접근 가능

- 인덱싱한 값끼리 연산 가능

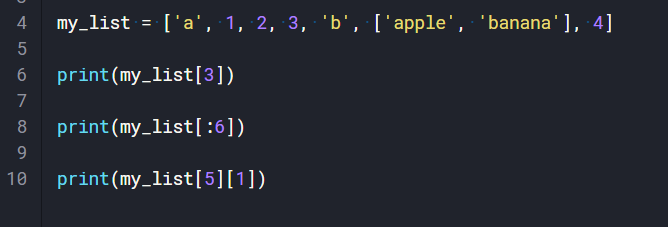
- 문자열 슬라이싱과 마찬가지로 앞쪽 범위 생략 -> 처음부터, 뒤쪽 범위 생략 -> 끝까지

예) lst[:5] , lst[5:]

- 슬라이싱 이용해 리스트 내 리스트 접근 가능

예) lst[0][1:4]

<실습>



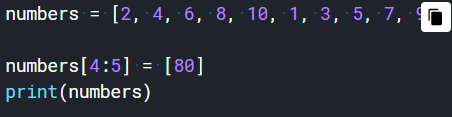
2) 리스트 연산과 수정

- 리스트 연산은 문자열 연산과 동일 기능 수행

- 문자열은 인덱싱 값으로 수정 불가능하나 리스트는 인덱싱 값으로 수정 가능

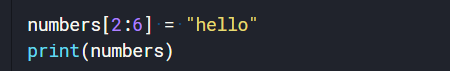
-> 리스트 이름[인덱스값] = 수정값

- 슬라이싱으로 리스트 값 수정할 때 조심해야할 사항

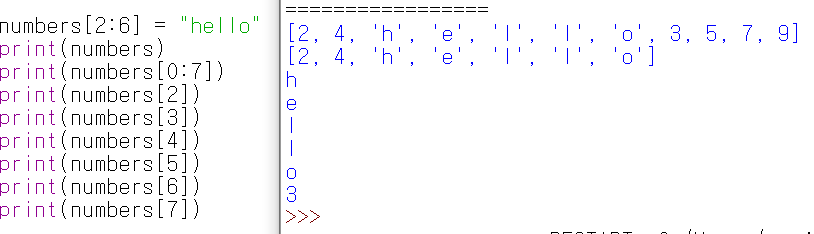




- numbers[4:5]는 인덱스 4만 가져오지만 슬라이싱으로 값을 수정할 때는 단일 값이 아닌 리스트로 대입해서 수정해야한다.

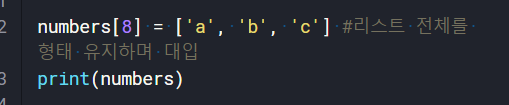






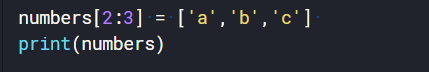
- “hello”는 하나의 문자열이면서 문자의 집합이기에 슬라이싱으로 값을 대입할 때 슬라이싱한 범위에 문자열의 문자를 순차적으로 대입한다.

->처음 numbers에서 3은 인덱스 6였지만 “hello”를 넣어서 **인덱스 7로 밀려났다.**





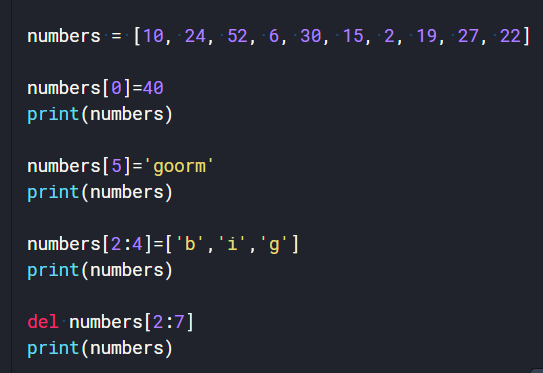
- 슬라이싱한 범위에 리스트 대입하면 리스트의 구성요소들이 numbers에 완전히 포함된다.





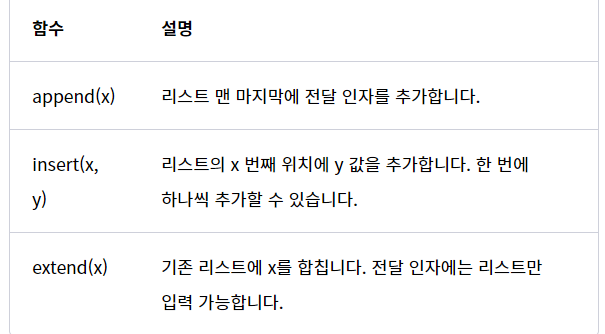
- 인덱스로 리스트를 대입하면 리스트 내 리스트 형태로 포함된다.

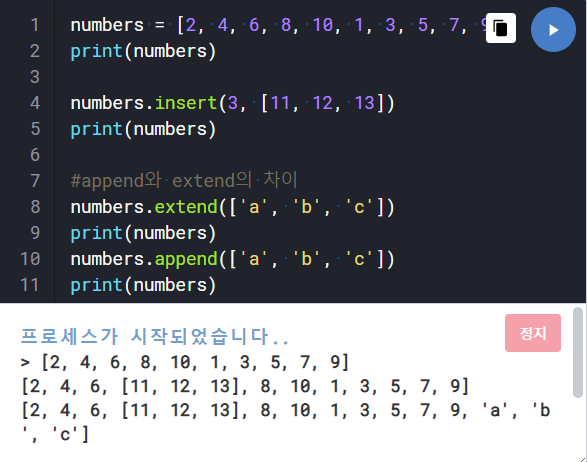
<실습>

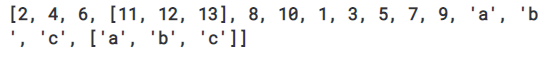


3) 리스트 함수

- 추가하는 함수



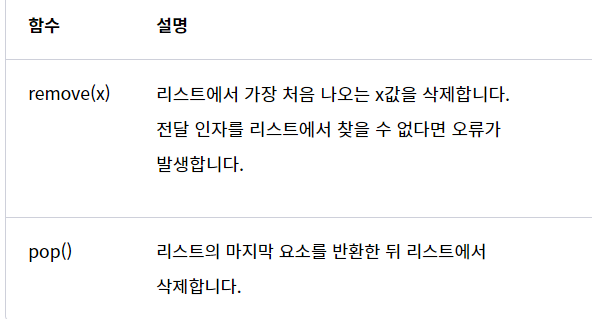


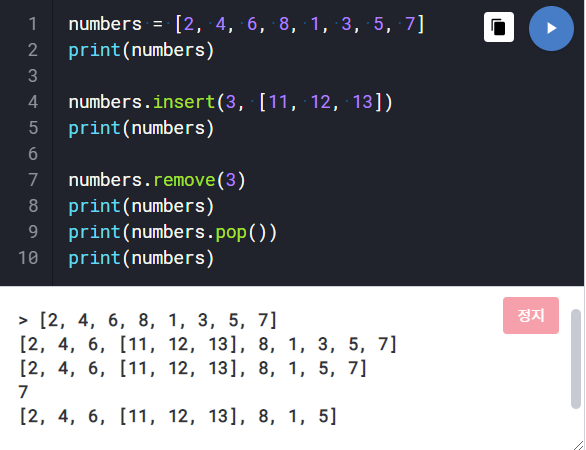


-> append로 리스트를 넣으면 리스트는 하나의 값으로 처리된다,

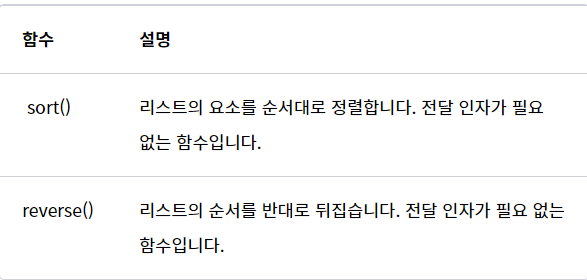
-> extend으로 리스트를 넣으면 리스트 괄호가 풀리고 리스트 요소 하나하나가 값으로 처리된다.

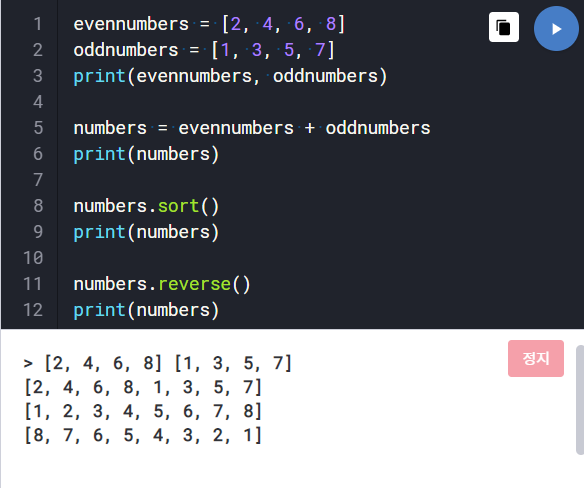
- 삭제하는 함수



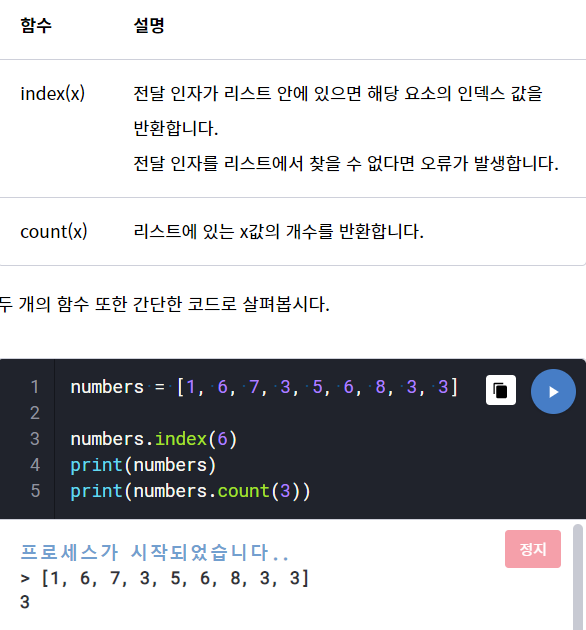


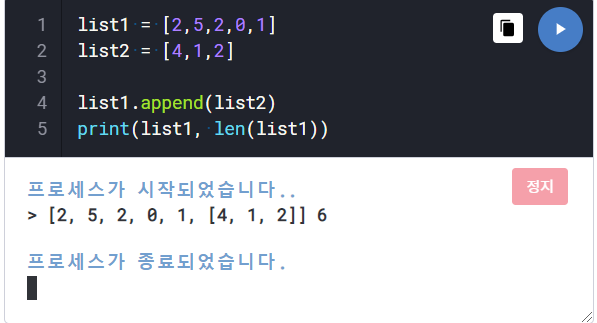
- 정렬하고 순서를 바꾸는 함수



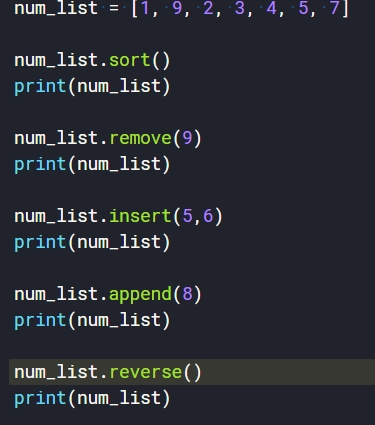


- 리스트의 정보를 알아내는 함수





<실습>



4) 리스트의 심화개념: 값 할당

- 리스트의 두가지 종류

① 빈 리스트 예) a=[], a=list()

② 리스트 생성과 동시에 값이 할당된 리스트

- 빈 리스트를 생성하고 나중에 값을 대입 -> 불가능

예) drawer = []

drawer.append("양말")

drawer[0] = "양말"

- 빈 리스트를 선언했다면 공간을 마련하고 값을 추가해야 한다.

-> 값을 하나 추가: append함수 사용, 여러 값을 한번에 추가: extend함수 사용

예) drawer = []

drawer.append("양말")

print(drawer, drawer[0])

- del 기능: 값을 삭제하는 기능

예) del drawer[3]

print(drawer)

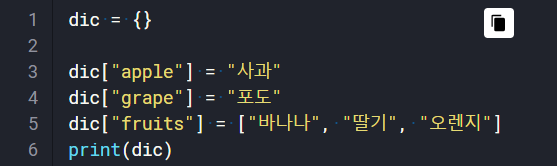
**6. 딕셔너리**

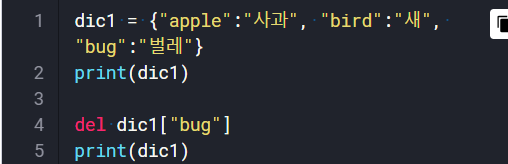
- 값들의 집합이나 순서가 존재하지 않는다.

- key : value 형태로 값을 저장하고 딕셔너리 이름 = **{** key1: value1, key2 : value2 … **}**의 형식

- 딕셔너리는 리스트와 달리 빈 딕셔너리 선언해도 함수 사용없이 대입해서 값을 추가할 수 있다

-> 변수 이름 **[** key **]** = value

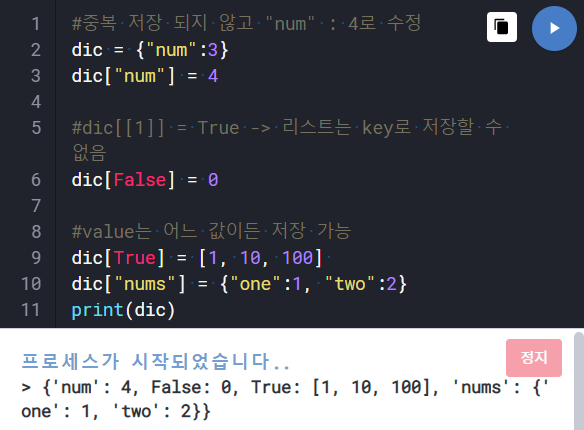




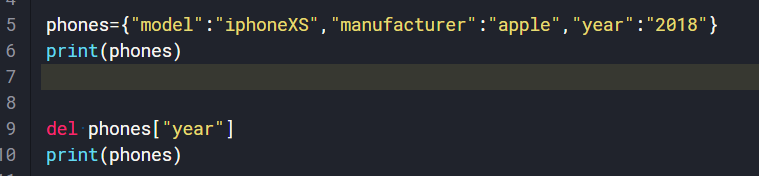
- key에 리스트 사용 불가능

- key는 중복 불가능

- 헷갈렸던 예제

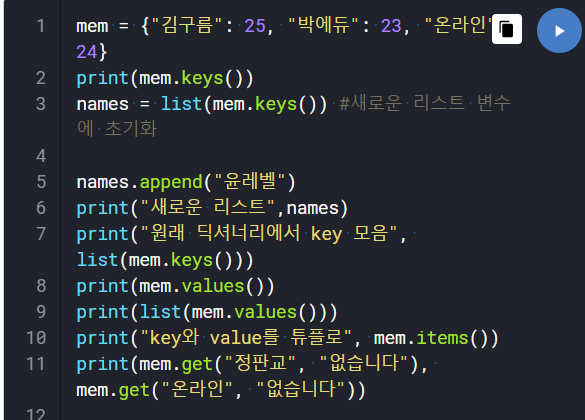


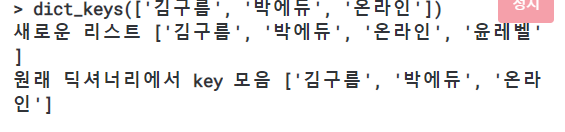
<실습>

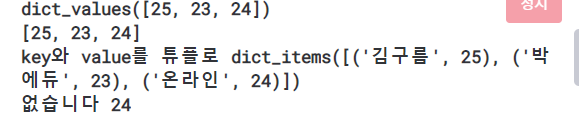


2) 딕셔너리 함수







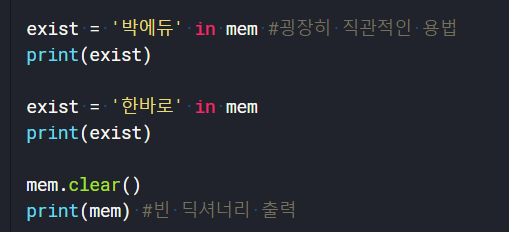


- x.keys()가 실행되면 dict\_keys라는 객체 반환한다. -> 리스트 형태로 필요하다면 list(x.keys( ))

- key가 존재하지 않으면 none 반환

- none이 아니라 다른 값을 출력하고 싶다면 x.get(key, 출력하고 싶은 값)으로 작성

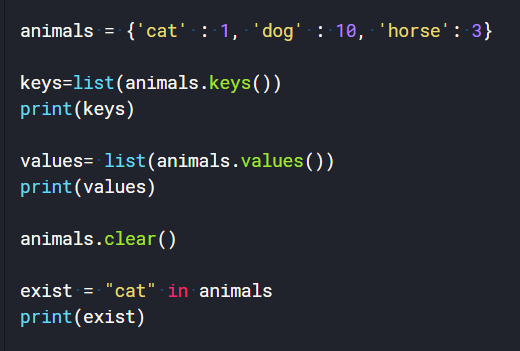
- value 값이 존재한다면 x.get(key, 출력하고 싶은 값) 이런 형태로 작성했어도 출력하고 싶은 값이 아닌 value 값이 나온다.





- key 값이 존재하면 true 아니면 false 반환한다.

<실습>



**7. 튜플과 집합**

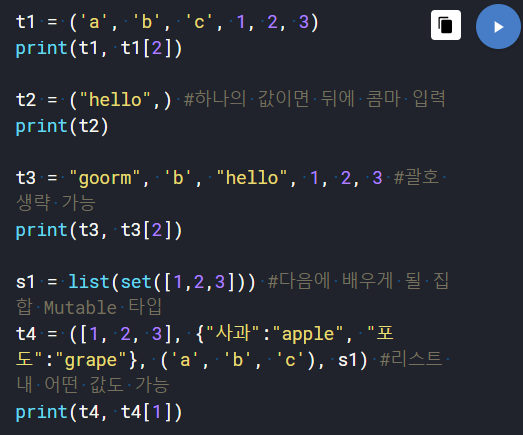
1) 변하지 않는 튜플

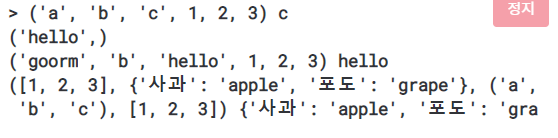
- immutable 타입: 튜플, 문자열

-> 인덱스로 값 수정 불가능

- mutable: 리스트, 딕셔너리

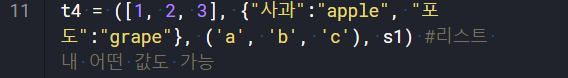
- **( )**을 사용해 선언 예) t1=(‘a’, ‘b’, ‘d’)

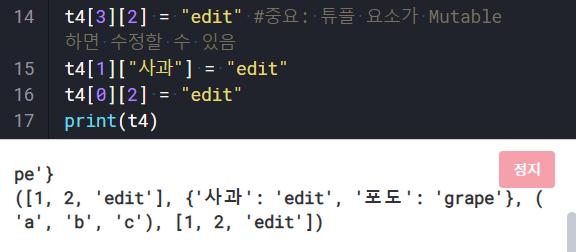
- 



- 괄호 생략 가능 -> 괄호 생략하면 자동으로 튜플로 초기화된다 예) t1=‘a’, ‘b’, ‘d

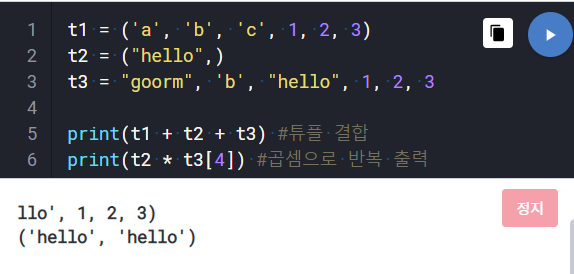
- 한 개의 요소만 초기화할 시 요소 뒤에 콤마를 꼭 입력해야 한다.





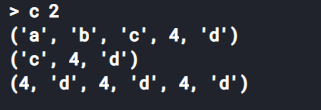
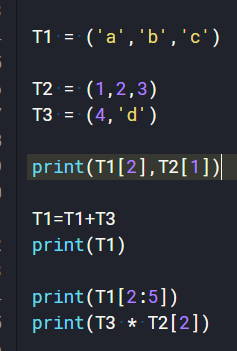
- 튜플 안에 있는 mutable한 값은 수정 가능하다

- 값 수정을 제외하고는 리스트의 기능을 모두 지원한다 -> 연산 가능



- 리스트 함수 중 함수 값을 변경하지 않는 함수는 그대로 사용 가능 예) Index(), count()

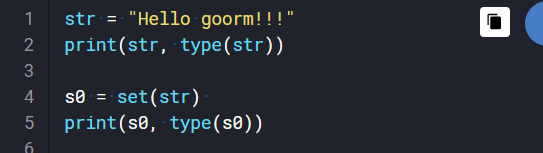
<실습>

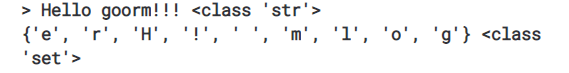


2) 중복과 순서가 없는 집합

- **{}** 이용하여 선언 및 초기화 가능

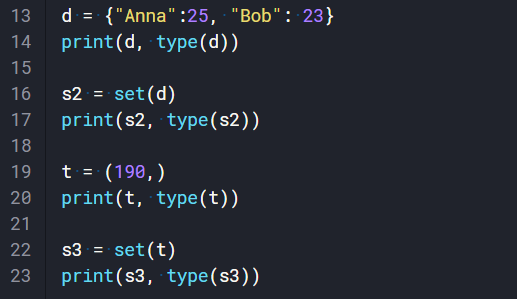
- 다른 묶음 자료형을 집합으로 변형할 수 있다

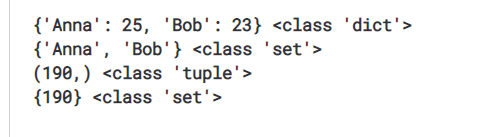




- 요소의 순서가 없다. -> 인덱싱 사용 불가능

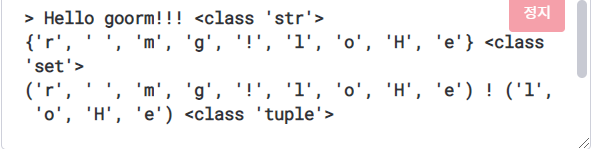
- 중복되는 값은 하나만 저장한다.





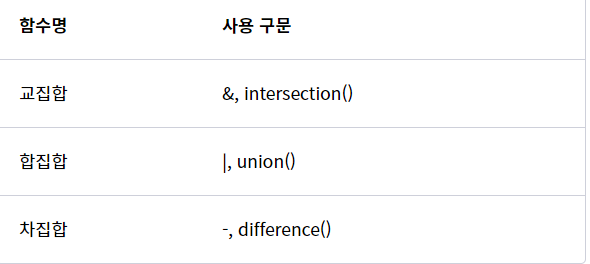
- 딕셔너리는 key만 저장한다.

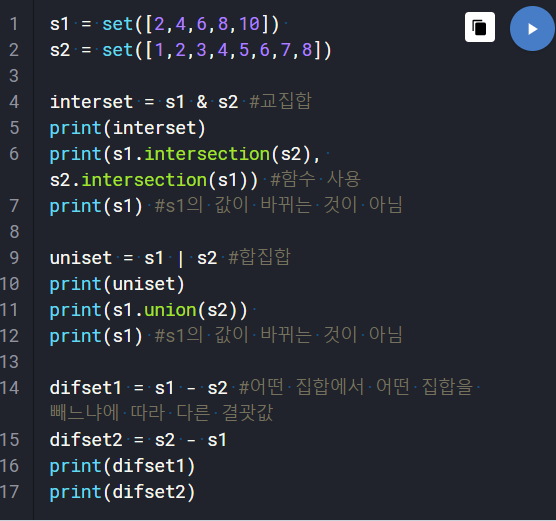


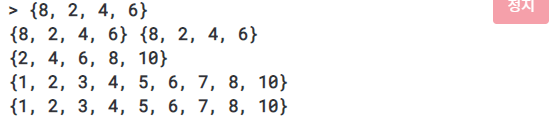


- set -> tuple : tuple(), set -> list : list() 사용

- 집합 함수

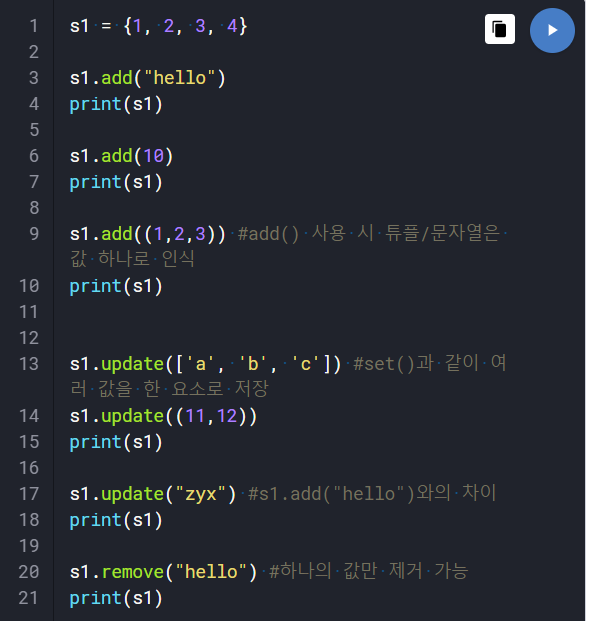


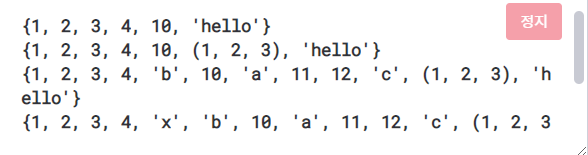


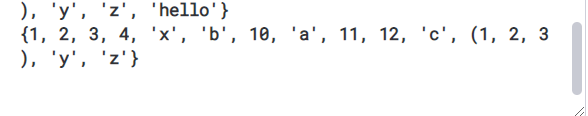












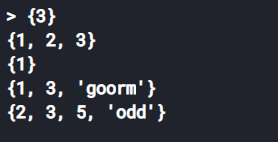
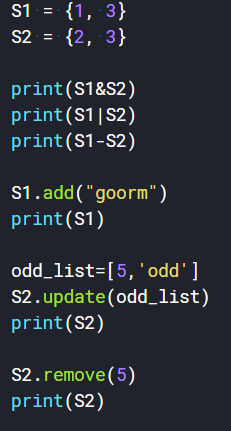
- 튜플은 한 개의 원소로 사용 가능하나 리스트와 집합 자체는 집합의 원소로 사용할 수 없다.

-> 튜플은 s1. add((1,2,3))로 (1,2,3)을 하나의 요소로 저장할 수 있음 / s1. Update((1,2,3))로 1, 2, 3 따로 저장도 가능함

-> 리스트는 add함수 이용해 [‘a’, ‘b’, ‘c’]를 하나의 요소로 저장할 수 없음 / s1.update([‘a’, ‘b’, ‘c’])로 ‘a’, ‘b’, ‘c’ 따로 저장은 가능함

- s1. Update(“ab”) 하면 ‘a’, ‘b’ 따로 저장된다.

<실습>



**10. 함수**

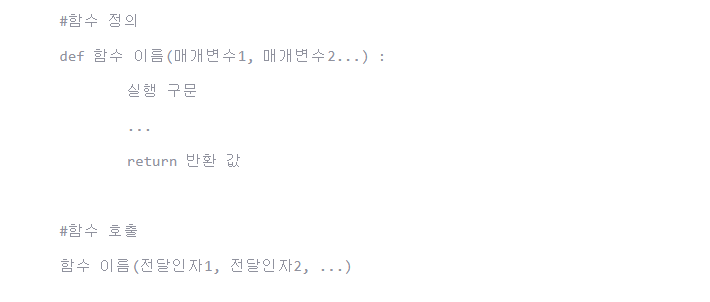
1) 함수의 역할

- 함수의 역할

① 지정 기능을 실행하는 단위

- 사용자가 만든 함수에 인자를 전달하면 연산 결과가 반환된다.

- 함수 선언 방식



- def: 함수를 정의하는 키워드

- 매개변수: 함수를 호출할 때 전달받은 인자를 함수 안에서 사용할 수 있도록 연결하는 역할

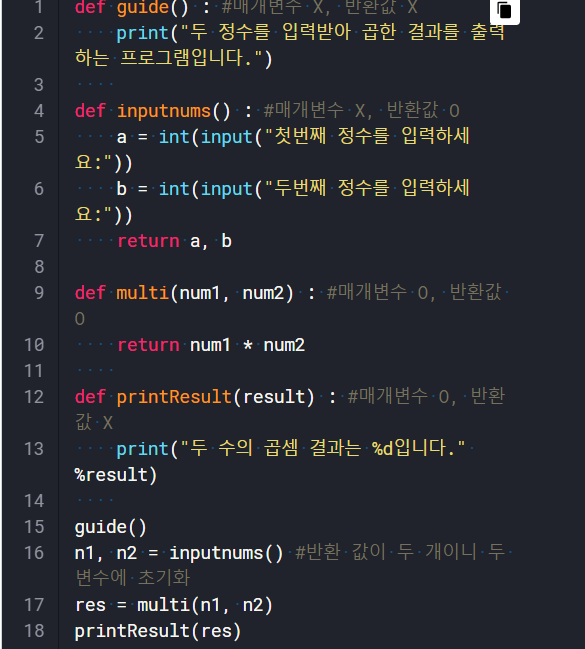
- return: 반환할 값

② 코드의 가독성과 프로그램의 효율성 증대

- 함수는 특정 기능을 위해 만든 여러 문장을 묶어서 실행하는 코드 블록 단위로써, 코드의 가독성과 프로그램의 효율성 증대를 위해 사용된다.

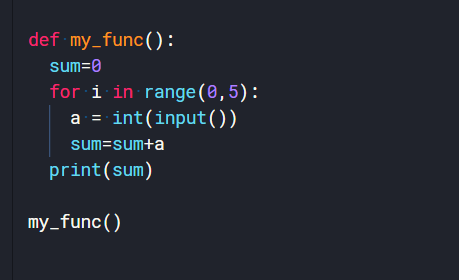
- 함수의 이름은 이 함수가 무슨 기능을 하는지 알아보기 쉽게 설정한다.

2) 함수의 사용 형태

2

- 매개 변수와 반환 값은 있을 수도 있고 없을 수도 있다.

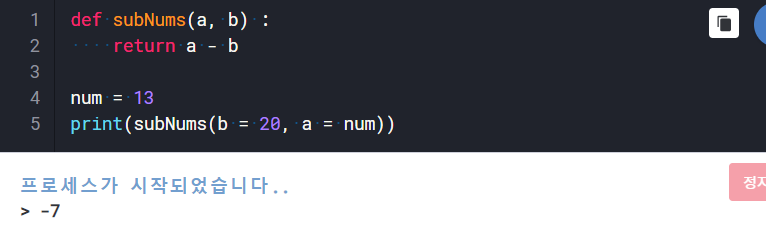
<실습>



-> sum이라는 변수의 위치 중요! 반드시 함수 안에 넣을 것!

3) 매개변수

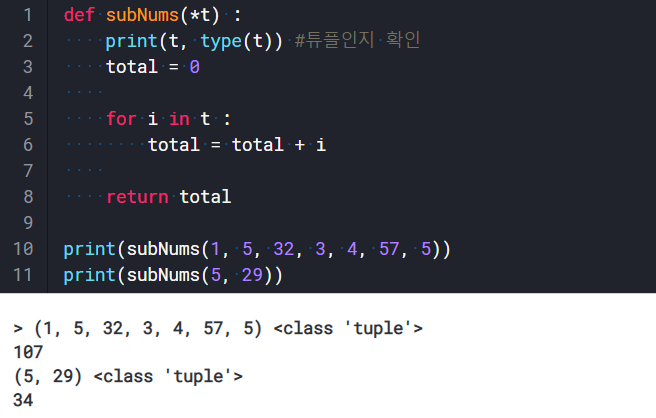
- 파이썬에서는 함수를 호출할 때 매개변수를 지정하여 인자를 전달할 수 있다(자주 사용 x)



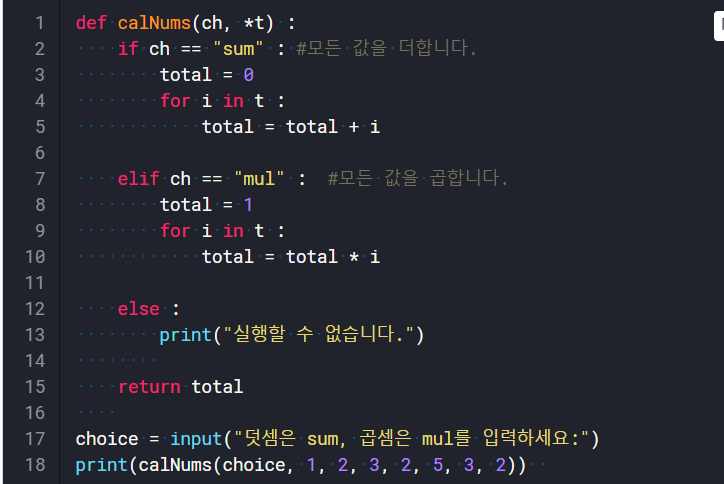
- 가변 인자 함수: 동일한 기능을 수행하지만 매개변수만 다른 함수

- 가변 인자 함수는 매개변수 앞에 \*입력해서 생성

->def 함수 이름(\*매개변수):



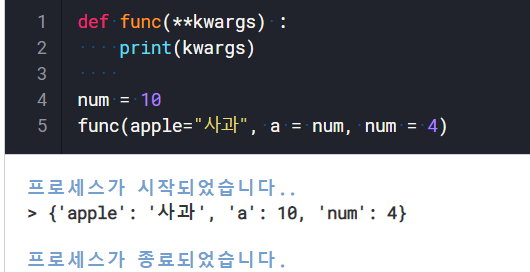
-가변 인자 함수의 매개변수는 일반 매개변수와 같이 사용 가능하나 일반 매개변수가 가변 인자 앞에 나와야 한다.

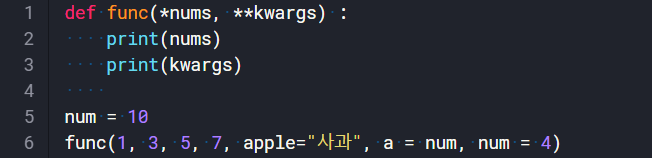


- 키워드 매개변수: 매개변수 앞에 \*\* 입력해 전달인자를 딕셔너리 형태로 받는 것

- key는 따옴표로 감싸지 않는다. -> key에는 문자열을 입력할 수 없다 (+숫자)

- 키워드 매개변수는 가변인자와 같이 사용 가능하나 키워드 형태의 인수는 맨 뒤에 온다.

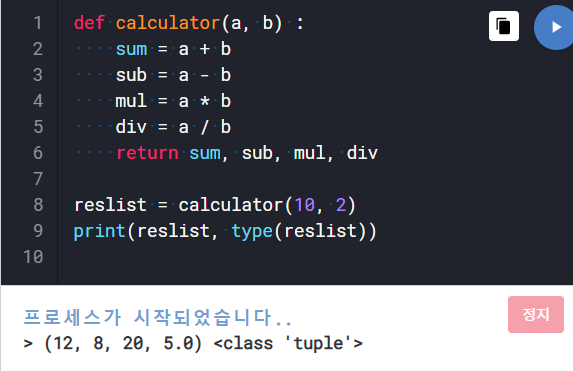




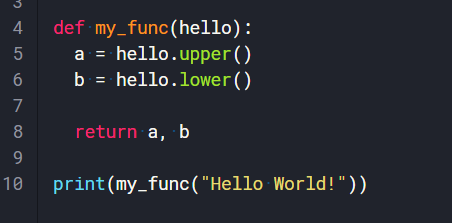
4) 반환값

- 반환값: 함수 내 연산 결과값이자 함수를 종료하는 키워드

- 반환값이 여러 개일 때 자동으로 튜플로 반환한다. 단 튜플로 할당할 수 있는 변수가 필요하다



<실습>



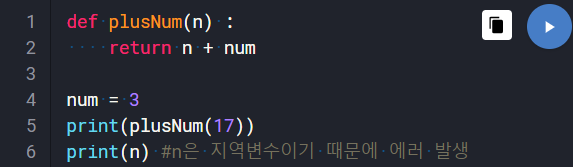
-> 매개변수.함수() (0) / 함수.함수() (x) 예) my\_func().upper()

6) 지역변수와 전역변수

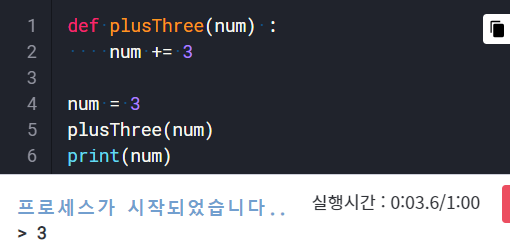
- 지역 변수: 정해진 범위에서만 사용 가능한 변수

- 전역 변수: 코드 전체에서 사용할 수 있는 변수

- 헷갈리기 쉬운 예제



-> n은 지역변수이기 때문에 에러 발생

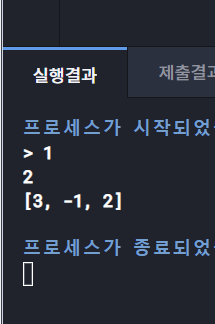
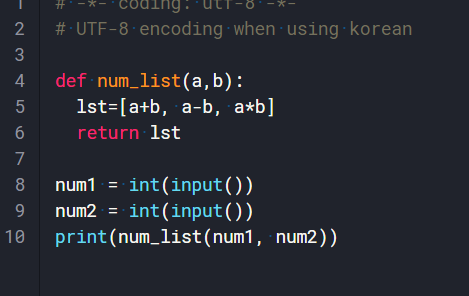


-> 함수를 호출하고 지역변수 num에 3을 전달해 여기에 3을 더한다. 지역변수 num은 6으로 변경된다. 전역변수 num을 출력한다. 고로 3이 출력된다.

- 전역변수는 어디서든 참조할 수 있으나 함수 안에서 전역변수에 새로운 값을 대입할 수는 없다. (Global 이라는 키워드 사용하면 가능하지만 자주 쓰이지 않는다.)

- 전역변수의 단점 -> 메모리 비효율적으로 사용, 코드의 흐름 파악하는 데 방해

<실습>



**12. 클래스**

1) 클래스의 개념과 필요성

- 붕어빵을 예로 든다면) 붕어빵 틀=클래스 / 붕어빵=객체

-> 붕어빵 틀에 어떤 재료를 넣고 얼마나 구웠느냐에 따라 다른 붕어빵이 만들어진다.

- 특정객체를 지정해서 말하면->객체 / 그 객체를 생성한 클래스와의 관계를 얘기하면->인스턴스

2) 객체와 클래스 변수

- class **Triangle** : // triangle 이라는 클래스 선언

pass

**tri1** = Triangle() // 만들어 놓은 클래스로 객체를 생성

**tri2** = Triangle()

**tri3** = Triangle()

**tri4** = Triangle()

-class Triangle :

**height = 10** //클래스 변수

**bottom = 4**

tri1 = Triangle()

print(tri1.**height**)

print(tri1.**bottom**)

tri2 = Triangle()

print(tri2.**height**)

print(tri2.**bottom**)

- 클래스 변수: 클래스 안에 선언된 변수

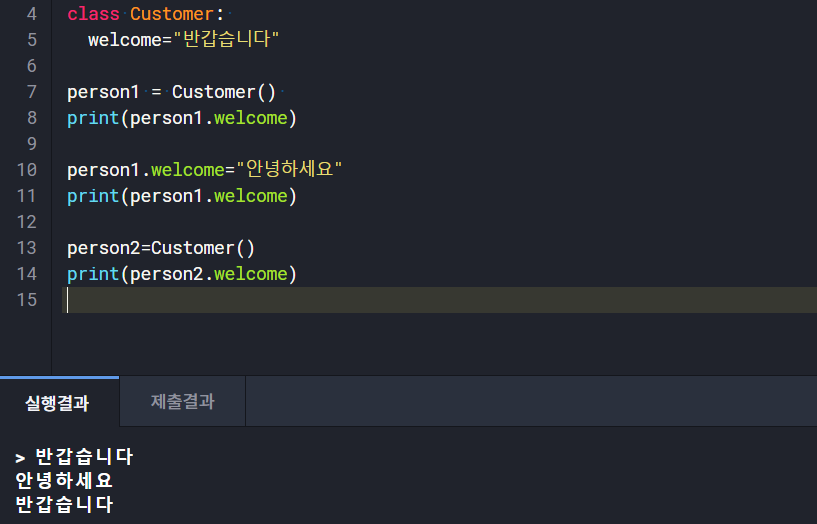
- 클래스 변수는 같은 클래스 끼리 만들어진 인스턴스끼리 공유하고 접근 가능한 변수

- 선언된 객체로 클래스 변수를 접근하려면 객체이름. 변수이름 형식으로 작성

- 객체에서 클래스 변수 값 변경 -> 클래스에 영향 X

- 클래스에서 클래스 변수 변경 -> 객체에서 불러오는 클래스 변수에 영향 O

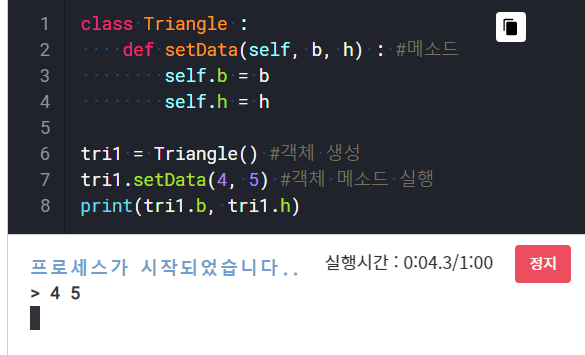
<실습>



3) 인스턴스 변수와 메소드

- 메소드: 클래스 안에서 어떠한 기능을 수행하는 함수 -> 객체 정보를 지정할 수 있음

- 메소드 함수는 사용자가 전달 인자를 입력하지 않는 매개변수 self를 사용한다



- setData 메소드의 매개변수 self는 객체 tril을 전달받아 b, h에 4, 5를 할당하는 역할을 수행

- self.b=b = tril.b=b 일맥상통

- self.변수이름 형식으로 객체필드에 저장한 변수는 tril.b와 같은 형식으로 값을 가져올 수 있다.

- self.변수이름 = 인스턴스 변수

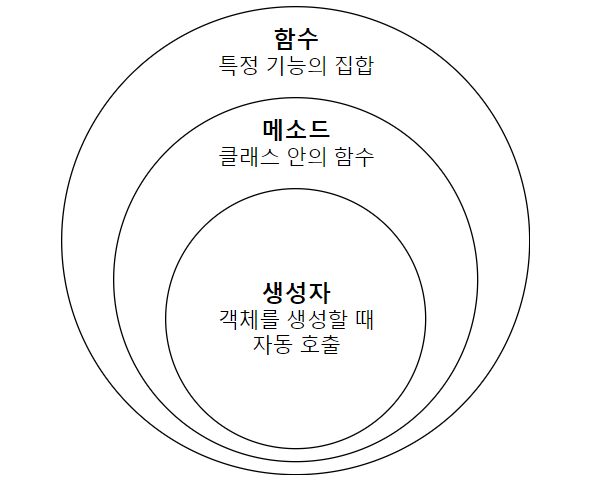
- 클래스 변수와 인스턴스 변수의 예제



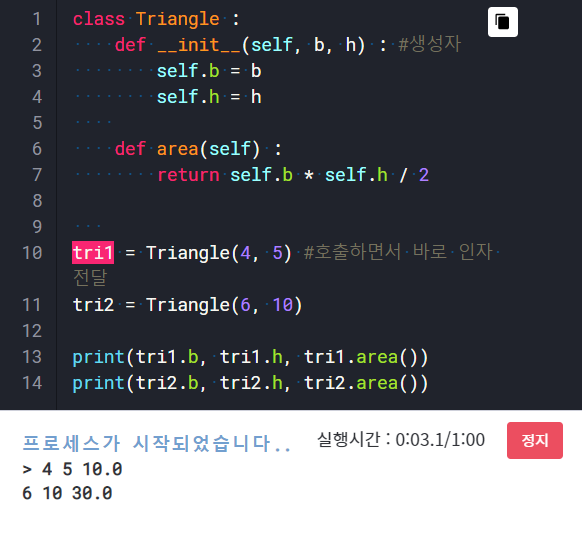
4) 생성자와 메소드

- 생성자: 객체 생성과 동시에 초기값을 설정하는 메소드

- 기본 생성자는 클래스를 선언할 때 자동으로 만들어지며, 객체를 만들 때 자동으로 실행된다.



- 생성자는 이름을 마음대로 설정할 수 있는 메소드와 달리 \_\_init\_\_라는 이름 사용



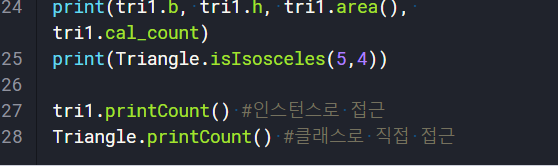
- 인스턴스 메소드: self로 인스턴스의 영역에 접근하는 메소드

- 정적 메소드: self 매개변수를 갖고 있지 않는 메소드

- 정적 메소드는 @staticmethod라고 표시해야 한다.

- 클래스 메소드: 클래스 변수에 접근할 때 사용하며, cls로 클래스 변수를 전달받는다.

-> 인스턴스 필드에 접근하지 않을 때 정적 메소드나 클래스 메소드를 사용, 특히 클래스 변수에 접근해야 할 때 클래스 메소드로 접근하는 것이 용이함

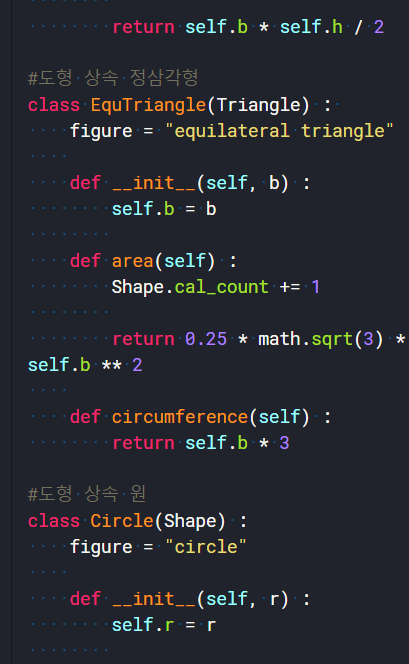
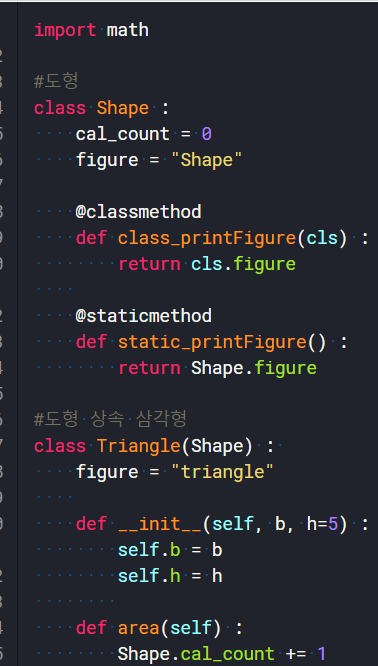


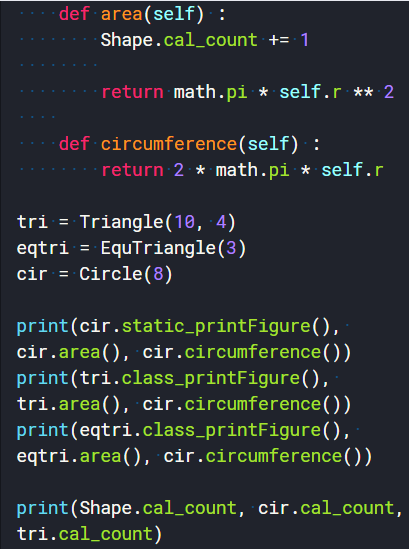
5) 클래스 상속

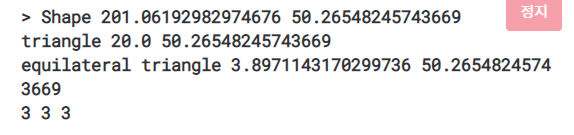
- 도형을 예로 든다면) 삼각형은 도형의 고유 특징을 물려받아 사용하고, 정삼각형은 삼각형의 고유 특징을 물려받아 사용한다

-> 정삼각형 클래스는 삼각형 클래스를 상속받았다

-> 정삼각형 클래스는 삼각형 클래스를 부모 클래스로 부르고 반대로는 자식 클래스라고 부른다







- 부모 클래스에서 선언된 클래스 메소드는 자식 클래스의 인스턴스의 클래스 변수를 참조

- 클래스 메소드는 부모 클래스에서 클래스 메소드를 실행 or 자식클래스 변수에 접근할 때 사용

- 자식 클래스로 생성한 인스턴스는 부모 클래스의 메소드에 접근 가능

- 부모 클래스로 생성한 메소드는 자식 클래스에 접근 불가능

- 메소드: 자식 클래스에서 부모 클래스의 메소드를 재정의하는 것

6) 클래스 심화

- 객체 지향 프로그래밍에서 클래스는 다형성의 특징을 가진다.

- 같은 정보를 공유하는 인스턴스는 다형성을 가진다.

- 부모 클래스를 여러 개 상속 가능

- 오버 로딩: 한 클래스 안에서 같은 이름의 메소드를 여러 번 선언하는 것