데이터 구조 활용 : 데이터 구조를 활용하기 위해서는 메서드(method)를 활용

- 메서드는 클래스 내부에 정의한 함수, 사실상 함수 동일
- 데이터 구조.메서드() 형태로 활용! ==> 쉽게 생각하면 주어.동사()
- ex) List.append(10) ==>' List에 10을 더한다 '

순서가 있는 데이터 구조(시퀀스형 데이터 구조)

- 문자열(String) : 문자들의 나열 (변경 불가능한 immutable)
    - s.find(x) = x의 첫 번째 위치를 반환, 없으면 -1 을 반환
    - s.index(x) = x의 첫 번째 위치를 반환, 없으면 오류 발생
    - s.isalpha() = 알파벳 문자 여부
    - s.islower() = 소문자 여부
    - s.isupper() = 대문자 여부
    - s.istitle() = 타이틀 형식 여부
    - s.replace(old, new, count) = 바꿀 대상 글자를 새로운 글자로 바꿔서 반환, count 개수만큼 변경
    - s.strip(chars) = 공백이나 특정 문자를 제거 ==> lstrip() = 왼쪽을 제거, rstrip() = 오른쪽을 제거
    - s.split() = 공백이나 특정 문자를 기준으로 분리
    - ''.join(iterable 요소) = 구분자로 iterable을 합침
    - s.capitalize() = 가장 첫 번째 글자를 대문자로 변경
    - s.title() = 문자열 내 띄어쓰기 기준으로 각 단어의 첫글자는 대문자로, 나머지는 소문자로 변환
    - s.upper() = 모두 대문자로 변경
    - s.lower() = 모두 소문자로 변경
    - s.swapcase() = 대<->소문자 서로 변경

  문자열은 immutable(불변형)인데, 문자열 변경이 되는 이유는 기존의 문자열을 변경하는 것이 아닌,

  변경된 문자열을 새롭게 만들어서 반환 ***즉! 서로 다른 주소 값을 가진 다른 것***

- 리스트(List) : 여러 개의 값을 순서가 있는 구조로 저장하고 싶을 때 사용
    - 생성된 이후 내용 변경이 가능 (가변 자료형, mutable)
    - 순서가 있는 시퀀스로 인덱스를 통해 접근 가능
    - 리스트는 대괄호 혹은 list()를 통해 생성
    - 파이썬에서는 어떠한 자료형도 저장할 수 있으며, 리스트 안에 리스트도 넣을 수 있다.
    - L.append(x) = 리스트 마지막에 항목 x를 추가
    - L.insert(i, x) = 리스트 인덱스 i에 항목 x를 삽입
    - L.remove(x) = 리스트 가장 왼쪽에 있는 항목(첫 번째) x를 제거, 항목이 존재하지 않을 경우, ValueError
    - L.pop() = 리스트 가장 오른쪽에 있는 항목(마지막)을 반환 후 제거
    - L.pop(i) = 리스트의 인덱스 i에 있는 항목을 반환 후 제거
    - L.extend(m) = 순회형 m의 모든 항목들의 리스트 끝에 추가 (+=과 같은 기능)
    - L.index(x, start, end) = 리스트에 있는 항목 중 가장 왼쪽에 있는 항목 x의 인덱스를 반환, 값이 없는 경우 ValueError
    - L.reverse() = 순서를 반대로 뒤집음(정렬 아님X)
    - L.sort() = 원본 리스트를 정렬, None을 반환 <==> sorted(x) 원본 변경X, 정렬된 리스트 반환

- L.count(x) = 리스트에서 항목 x가 몇 개 존재하는지 갯수를 반환

- 튜플 : 여러 개의 값을 순서가 있는 구조로 저장하고 싶을 때 사용
  - 다만 리스트와의 차이점은 생성 후, 담고 있는 값 변경이 불가 (불변 자료형)
  - 튜플은 변경 불가한 불변 자료형이기에 값에 영향을 미치지 않는 메서드만을 지원
  - 리스트 메서드 중 항목을 변경하는 메서드들을 제외하고 대부분 동일

비시퀀스형 데이터 구조

- 셋(Set) : Set이란 중복되는 요소가 없으며 순서 또한 상관없는 데이터들의 묶음
  - 담고 있는 요소를 삽입, 변경, 삭제 가능한 가변 자료형 (Mutable)
  - 데이터의 중복을 허용하지 않기 때문에 중복되는 원소가 있다면 하나만 저장
  - 순서가 없기 때문에 인덱스를 이용한 접근 불가능
  - 수학에서의 집합을 표현한 컨테이너
  - s.copy() = 셋의 얕은 복사본을 반환

    얕은 복사 => 주소 값을 참조 하고 있다. 즉 A를 복사하여 B를 만들었을 때, B의 값이 바뀌면 A의 값도 같이 바뀜
  - s.add(x) = 항목 x 가 셋 s에 없다면 추가
  - s.pop() = 셋 s에서 랜덤하게 항목을 반환하고, 해당 항목을 제거 set이 비어 있을 경우, KeyError
  - s.remove(s) = 항목 x를 셋 s에서 삭제, 항목이 존재하지 않을 경우, KeyError
  - s.discard(x) = 항목 x 가 셋 s에 있는 경우, 항목 x를 셋 s에서 삭제 ==> ***Error 발생 하지 않음***
  - s.update(t) = 셋 t에 있는 모든 항목 중 셋 s에 없는 항목을 추가
  - s.clear() = 모든 항목을 제거
  - s.isdisjoint(t) = 셋 s가 셋 t의 서로 같은 항목을 하나라도 갖고 있지 않은 경우, True반환(서로소)
  - s.issubset(t) = 셋 s가 셋 t의 하위 셋인 경우, True반환
  - s.issuperset(t) = 셋 s가 셋 t의 상위 셋인 경우, True 반환

- 딕셔너리(Dictionary) : 키-값(key-value) 쌍으로 이뤄진 자료형
  - 딕셔너리의 key는 변경 불가능한 데이터(immutable)만 활용 가능(string, integer, float, boolean, tuple, range)
  - 각 키의 값(value) = 어떠한 형태든 관계없음
  - d.clear() = 모든 항목을 제거
  - d.copy() = 딕셔너리 d의 얕은 복사본을 반환
  - d.keys() = 딕셔너리 d의 모든 키를 담은 뷰를 반환
  - d.values() = 딕셔너리 d 의 모든 값을 담은 뷰를 반환
  - d.items() = 딕셔너리 d의 모든 키-값의 쌍을 담은 뷰를 반환
  - d.get(k) = 키 k의 값을 반환하는데, 키 k가 딕셔너리 d에 없을 경우 None을 반환
  - d.get(k, v) = 키 k의 값을 반환하는데, 키 k가 딕셔너리 d에 없을 경우 v를 반환
  - d.pop(k) = 키 k의 값을 반환하고 키 k인 항목을 딕셔너리 d에서 삭제하는데, 키 k가 딕셔너리 d에 없을 경우 KeyError가 발생
  - d.pop(k, v) =

- 키 k의 값을 반환하고 키 k인 항목을 딕셔너리 d에서 삭제하는데, 키 k가 딕셔너리 d에 없을 경우 v를 반환
- d.update() 딕셔너리 d의 값을 매핑하여 업데이트

할당(Assignment)

- 대입 연산자(=) : 대입 연산자(=)를 통한 복사는 해당 객체에 대한 객체 참조를 복사

객체 지향 프로그래밍 (OOP : Object-Oriented Programming)

- 객체 지향 프로그래밍 (OOP : Object-Oriented Programming)
  - 컴퓨터 프로그래밍의 패러다임 중 하나이다.
  - 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 ***객체*** 들의 모임으로 파악.
  - 프로그램을 여러 개의 독립된 객체들과 그 객체 간의 상호작용으로 파악하는 프로그래밍 방법
  - 각각의 객체는 메세지를 주고받고, 데이터를 처리할 수 있다.
  - 기존의 프로그래밍은 ***절차지향 프로그래밍*** => 순차적인 처리가 중요시 되며 프로그램 전체가 유기적으로 연결되도록 하는 기법 (ex.C언어)
  - 객체 지향의 장/단점
    - 장점 : 클래스 단위로 모듈화시켜 개발할 수 있으므로 많은 인원이 참여하는 대규모 소프트웨어 개발에 적합

      필요한 부분만 수정하기 쉽기 때문에 프로그램의 유지보수가 쉬움

    - 단점 : 설계 시 많은 노력과 시간이 필요함

      실행 속도가 상대적으로 느림

- 객체(컴퓨터 과학)
  - 컴퓨터 과학에서 객체 또는 오브젝트는 클래스에서 정의한 것을 토대로 메모리(실제 저장공간)에 할당된 것으로 프로그램에서 사용되는

    데이터 또는 식별자에 의해 참조되는 공간을 의미하며, 변수, 자료 구조, 함수 또는 메서드가 될 수 있다
  - 객체 = 속성 + 행동
  - ***객체***(이찬혁) = ***속성, 변수***(정보 => 직업 , 생년월일, 국적) + ***행동, 함수, 메서드***(동작 => 랩하기(), 댄스())
  - 클래스(설계도) --> 객체(실제 사례)
  - 클래스로 만든 객체를 인스턴스 라고도 함
    - 객체와 인스턴스의 차이점
      - 이찬혁은 객체다(O)
      - 이찬혁을 가수의 인스턴스다(O)
      - 이찬혁은 인스턴스다(X)
  - 파이썬은 모든 것이 객체 --> 파이썬의 모든 것엔 속성과 행동이 존재
    - ex. [3, 2, 1].sort() -> 리스트. 정렬() --> 객체.행동()

'banana'.upper() -> 문자열.대문자로() --> 객체.행동()

- 객체와 인스턴스
  - [1, 2, 3], [1], [], ['hi'] => 모두 리스트 타입(클래스)의 객체

    '', 'hi', '파이썬' => 모두 문자열 타입(클래스)의 객체

  - 객체 : 객체는 특정 타입의 인스턴스이다.

    123, 900, 5는 모두 int의 인스턴스

    'hello', 'bye'는 모두 string의 인스턴스

    [232, 89, 1], []은 모두 list의 인스턴스

객체와 클래스 문법

- 객체의 설계도인 클래스를 가지고, 객체(인스턴스)를 생성한다 -> ex. Person(클래스) ==> 가수 이지은(인스턴스), 감독 강해피(인스턴스)
- 클래스 : 객체들의 분류 / 설계도
- 인스턴스 : 하나하나의 실체
- ***파이썬은 모든 것이 객체이며 모든 객체는 특정 타입의 인스턴스***

- 클래스 정의 : class MyClass:
- 인스턴스 생성 : my_instance = MyClass()
- 메서드 호출 : my_instance.my_method()
- 속성 : my_instance.my_attribute

- 객체 비교하기
  - == (equal, 동등한)
    - 변수가 참조하는 객체가 동등한(내용이 같은) 경우 True
    - 두 객체가 같아 보이지만 실제로 동일한 대상을 가리키고 있다고 확인해 준 것은 아님 (같은 주소값이 아니라는 의미.)

  - is (identical, 동일한)
    - 두 변수가 동일한 객체를 가리키는 경우 True (같은 주소값을 가짐)

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b, a is b) # True False
print(id(a)) # 2437991554048
print(id(b)) # 2437988582784

# ================================
```

```
a = [1, 2, 3]
b = a

print(a == b, a is b) # True True
print(id(a)) # 2163516693504
print(id(b)) # 2163516693504
```

OOP 속성 (객체 지향 프로그래밍의 속성)

- 특정 데이터 타입/클래스의 객체들이 가지게 될 상태/데이터를 의미
- 클래스 변수 / 인스턴스 변수 가 존재

- 클래스 변수 : 한 클래스의 모든 인스턴스가 공유하는 값을 의미, 같은 클래스의 인스턴스들은 같은 클래스 변수 값을 갖게 됨

  Person1.blood_color 으로 접근 및 할당
- 인스턴스 변수 : 인스턴스가 개인적으로 가지고 있는 속성(attribue), 각 인스턴스들의 고유한 변수

  생성자 메서드(__ init __) 에서 self.name = name 으로 정의

  인스턴스가 생성된 이후 person1.name 으로 접근 및 할당

```
class Person:
    blood_color = 'red' # 클래스 변수
    population = 100    # 클래스 변수

    def __init__(self, name):
        self.name = name # 인스턴스 변수


person1 = Person('싸피')  # person1 이라는 Person의 인스턴스를 생성
print(person1.name) # 싸피
print(Person.blood_color) # red
print(person1.blood_color) # red
# ============================================================

Person.blood_color = 'orange' # 클래스 변수 blood_color를 변경
print(Person.blood_color) # orange
print(person1.blood_color) # orange

# ============================================================

person1.blood_color = 'green'   # 인스턴스 변수 blood_color를 변경
print(Person.blood_color) # green
print(person1.blood_color) # orange
```

- 만약 인스턴스가 생성 될 때마다 클래스 변수가 늘어나도록 설정하려고 한다면

```
class Person:
    count = 0

    def __init__(self, name):
        self.name = name
        Person.count += 1

person1 = Person('부울경 1반 왕자님')
person2 = Person('부울경 1반 공주님')
print(Person.count)  # 2
print(person1.count)  # 2
print(person2.count)  # 2
```

OOP 기초 (객체 지향 프로그래밍의 기초)

- 메서드 : 특정 데이터 타입/ 클래스의 객체에 공통적으로 적용 가능한 행위(함수)

```
class Person:

    def talk(self):
        print('안녕')

    def eat(self, food):
        print(f'오늘은 {food} 먹어야지')

person1 = Person()
person1.talk() #  안녕
person1.eat('삐쓰까또레부르쥬미쳴라햄페스츄리치즈나쵸스트링스파게티') # 오늘은 삐쓰까또레부르쥬미쳴라햄페스츄리치즈나쵸스트링스파게티 먹어야지
```

메서드의 종류

- 인스턴스 메서드
  - 인스턴스 변수를 사용하거나, 인스턴스 변수에 값을 설정하는 메서드
  - 클래스 내부에 정의되는 메서드의 기본
  - 호출 시, 첫번째 인자로 인스턴스 자기자신(self)이 전달됨

  - self : 인스턴스 자기자신

    파이썬에서 인스턴스 메서드는 호출 시 첫번째 인자로 인스턴스 자신이 전달되게 설계, 매개변수 이름으로 self를 첫 번째 인자로 정의

  - 생성자(constructor) 메서드 : 인스턴스 객체가 생성될 때 자동으로 호출되는 메서드, 인스턴스 변수들의 초기값을 설정

    인스턴스 생성, __ init __ 메서드 자동 호출

- 매직 메서드 : Double underscore(__) 가 있는 메서드는 특수한 동작을 위해 만들어진 메서드로, 특정 상황에 자동으로 불리는 메서드

  객체의 특수 조작 행위를 지정(함수, 연산자 등)

  - __ str __ : 해당 객체의 출력 형태를 지정, 프린트 함수를 호출할 때, 자동으로 호출
  - __ del __ : 인스턴스 객체가 소멸되기 직전에 호출되는 메서드

```
class Person:

    def __init__(self):
        pass

    def __str__(self):
        return '저 졸려요...'

    def __del__(self):
        print("잠을 자러 갔습니다")

person1 = Person()
print(person1) # 저 졸려요...
# 인스턴스 객체가 소멸되기 직전 출력문 ==> 잠을 자러 갔습니다
```

- 클래스 메서드
  - 클래스가 사용할 메서드
  - @classmethod 데코레이터를 사용하여 정의
  - 호출 시, 첫번째 인자로 클래스(cls)가 전달됨

```
class Person:
    count = 0 # 클래스 변수

    def __init__(self, name): # 인스턴스 변수 설정
        self.name = name
        Person.count += 1

    @classmethod  # 데코레이터 사용
    def number_of_population(cls):   # 첫번째 인자(파라미터)로 cls(class)를 전달
        print(f'인수수는 {cls.count} 입니다.') # cls.count는 해당 클래스의 클래스 변수 count 를 적용

person1 = Person('우영우')
person2 = Person('이준혁')
person3 = Person('정명석')
print(Person.count) # 3
```

  - 데코레이터 :
    - 함수를 어떤 함수로 꾸며서 새로운 기능을 부여
    - @데코레이터(함수명) 형태로 함수 위에 작성
    - 순서대로 적용 되기 때문에 작성 순서가 중요

- 클래스 메서드와 인스턴스 메서드
  - 클래스 메서드 -> 클래스 변수 사용
  - 인스턴스 메서드 -> 인스턴스 변수 사용
  - 그렇다면 인스턴스 변수, 클래스 변수 모두 사용하고 싶다면?
    - 클래스는 인스턴스 변수 사용이 불가
    - 인스턴스 메서드는 클래스 변수, 인스턴스 변수 둘 다 사용이 가능

- 정적(static) 메서드
  - 정적(static) 메서드는 인스턴스 변수, 클래스 변수를 전혀 다루지 않는 메서드
  - 속성을 다루지 않고 단지 기능(행동)만을 하는 메서드를 정의할 때, 사용, 즉 객체 상태나 클래스 상태를 수정할 수 없음
  - @staticmethod 데코레이터를 사용하여 정의
  - 일반 함수처럼 동작하지만, 클래스의 이름공간에 귀속됨 --> 그래서 주로 해당 클래스 한정하는 용도로 사용

```
class Person:

    def __init__(self, name): # 인스턴스 변수 설정
        self.name = name


    @staticmethod
    def check_rich(money):   # static은 self, cls 사용 X
        return money > 10000

person1 = Person('우영우')

print(Person.check_rich(10)) # False   # static은 클래스로 접근 가능
print(person1.check_rich(20000)) # True   # static은 인스턴스로도 접근 가능
```

- 인스턴스와 클래스 간의 이름 공간(namespace)
  - 클래스를 정의하면, 클래스와 해당하는 이름 공간 생성
  - 인스턴스를 만들면, 인스턴스 객체가 생성되고 이름 공간 생성
  - 인스턴스에서 특정 속성에 접근하면, 인스턴스-클래스 순으로 탐색

```
class Person:
    name = 'unknown'

    def talk(self):
        print(self.name)

p1 = Person()
p1.talk() # unknown --> p1은 인스턴스 변수가 정의되어 있지 않아 클래스 변수 name의 값인 unknown이 출력됨
```

```
#
==========================================================================
==========================================

# p2 인스턴스 변수 설정 전/후
p2 = Person()
p2.talk() # unknown
p2.name = '강동원'
p2.talk() # 강동원 --> p2는 인스턴스 변수가 정의되어 인스턴스 변수 강동원이 출력
됨
#
==========================================================================
==========================================

print(Person.name) # unknown --> Person 클래스의 name 값이 강동원으로 변경된
것이 아닌 p2의 인스턴스의 name이 강동원으로 저장
print(p1.name) # unknown
print(p2.name) # 강동원
```

- 다시 한번 메서드 정리
  - 인스턴스 메서드 : 호출한 인스턴스를 의미하는 self 매개 변수를 통해 인스턴스 조작
  - 클래스 메서드 : 클래스를 의미하는 cls 매개 변수를 통해 클래스를 조작, 인스턴스 메서드는 호출할 수 없다.
  - 스태틱 메서드 : 클래스 변수나 인스턴스 변수를 사용하지 않는 경우에 사용, 객체 or 클래스 상태를 수정할 수 없다.

```
class MyClass:

    def method(self):
        return 'instance method', self

    @classmethod
    def classmethod(cls):
        return 'classmethod', cls #

    @staticmethod
    def staticmethod():
        return 'staticmethod'

# 인스턴스 메서드를 호출한 결과
obj = MyClass()
print(obj.method()) # ('instance method', <__main__.MyClass object at
0x0000021AC5CBCDC0>)
print(MyClass.method(obj)) # ('instance method', <__main__.MyClass object at
0x0000021AC5CBCDC0>)#
#
==========================================================================
====================

# 클래스 자체에서 각 메서드를 호출하는 경우
print(MyClass.classmethod()) # ('classmethod', <class '__main__.MyClass'>)
```

```
print(MyClass.staticmethod()) # staticmethod
print(MyClass.method()) # TypeError: method() missing 1 required positional
argument: 'self'
# 클래스 자체에서 인스턴스 메서드를 호출하려 해서 타입 에러
#
==========================================================================
====================

# 인스턴스는 클래스 메서드와 스태틱 메서드 모두 접근할 수 있다.
print(obj.classmethod()) # ('classmethod', <class '__main__.MyClass'>)
print(MyClass.classmethod()) # ('classmethod', <class '__main__.MyClass'>)
print(obj.staticmethod()) # staticmethod
```

객체지향의 핵심 4가지

- 추상화(Abstraction) : 현실 세계를 프로그램 설계에 반영하여 복잡한 것은 숨기고, 필요한 것만 들어내기
    - ex. Person => Professor , Student ==> Professor과 Student의 공통적인 요소들만 따로 모아 Person이라는 클래스에 모아 추상화 시킴

- 상속(Inheritance) : 두 클래스 사이 부모(상) - 자식(하) 관계를 정립하는 것, 모든 파이썬 클래스는 object를 상속 받음
    - 하위 클래스는 상위 클래스에 정의된 속성, 행동, 관계 및 제약 조건을 모두 상속 받음
    - 부모 클래스의 속성, 메서드가 자식 클래스에 상속되므로, 코드 재사용성이 높아짐
    - 하위 클래스에서 공통적으로 사용하는 메서드들을 상위 클래스로부터 물려받아 메서드의 재사용을 높일 수 있다.
    - super() : 자식(하위) 클래스에서 부모(상위)클래스를 사용하고 싶은 경우 사용
    - 메서드 오버라이딩을 통해 자식 클래스에서 재정의 가능 ==> 메서드 오버라이딩이란 메서드를 새로 재정의 한다는 의미
    - 상속관계에서의 이름 공간은 인스턴스, 자식 클래스, 부모 클래스 순으로 탐색
    - 다중 상속
        - 두 개 이상의 클래스를 상속 받는 경우
        - 상속받은 모든 클래스의 요소를 활용 가능함
        - 중복된 속성이나 메서드가 있는 경우 상속 순서에 의해 결정됨

          ex. class FirstChild(Dad, Mom) 의 경우 같은 메서드 호출 시 앞에 있는 Dad를 우선 상속
    - mro(Method Resolution Order) : 해당 인스턴스의 클래스가 어떤 부모 클래스를 가지는지 확인하는 메서드

      --> 해당 인스턴스로 부터의 모든 조상 클래스를 확인 하는 메서드

- 다형성(Polymorphism) : 동일한 메서드가 클래스에 따라 다르게 행동할 수 있음을 의미,

  즉 서로 다른 클래스에 속해있는 객체들이 동일한 메세지에 대해 다른 방식으로 응답할 수 있음

- 메서드 오버라이딩 : 상속받은 메서드를 재정의
  - 클래스 상속 시, 부모 클래스에서 정의한 메서드를 자식 클래스에서 변경
  - 부모 클래스의 메서드 이름과 기본 기능은 그대로 사용하지만, 특정 기능을 바꾸고 싶을 때 사용

```
class Person:

        def __init__(self, name) -> None:
                self.name = name

        def talk(self):
            print(f'반갑습니다. {self.name}입니다.')

class Professor(Person):

        def talk(self):
            print(f'{self.name} 일세.')

class Student(Person):

        def talk(self):
            super().talk()
            print(f'저는 학생입니다.')


p1 = Professor('김교수')
p1.talk() # 김교수 일세.

s1 = Student('이학생')
s1.talk() # (super().talk()로 인한 우선 출력)반갑습니다. 이학생입니다. --> 저는 학생입니다.
```

- 캡슐화(Encapsulation) : 객체의 일부 구현 내용에 대해 외부로부터의 직접적인 액세스를 차단 --> ex. 주민등록번호

  파이썬에서 암묵적으로 존재하지만, 언어적으로는 존재하지 않음

  - 접근제어자 종류
    - Public Access Modifier : 언더바 없이 시작하는 메서드나 속성, 어디서나 호출이 가능,

      하위 클래스에서 오버라이딩 가능, 일반적으로 작성되는 메서드와 속성의 대다수를 차지

```
class Person:

    def __init__(self, name, age) -> None:
            self.name = name
            self.age = 30

# Person 클래스의 인스턴스인 p1은 이름(name)과 나이(age) 모두 접근 가능.
p1 = Person('김싸피', 30)
print(p1.name)
print(p1.age)
```

- Protected Access Modifier : ***언더바 1개***로 시작하는 메서드나 속성, 암묵적 규칙에 의해 부모 클래스 내부와 자식 클래스만 호출 가능

```
class Person:

    def __init__(self, name, age):
            self.name = name
            self._age = age

    def get_age(self):
        return self._age

# 인스턴스를 만들고 get_age 메서드를 활용하여 호출할 수 있습니다.
p1 = Person('김싸피', 30)
print(p1.get_age()) # 30

# _age에 직접 접근하여도 확인이 가능, 파이썬에서는 암묵적으로 활용될 뿐.
print(p1._age) # 30
```

- Private Access Modifier : ***언더바 2개***로 시작하는 메서드나 속성, 본 클래스 내부에서만 사용이 가능, 하위 클래스 상속 및 호출 불가능

```
class Person:

    def __init__(self, name, age):
            self.name = name
            self.__age = age

    def get_age(self):
        return self.__age

# 인스턴스를 만들고 get_age 메서드를 활용하여 호출할 수 있습니다.
p1 = Person('김싸피', 30)
print(p1.get_age()) # 30

# __age에 직접 접근이 불가.
```

```
print(p1.__age) # AttributeError: 'Person' object has no attribute
'__age'
```

- getter 메서드와 setter 메서드 : 변수에 접근할 수 있는 메서드를 별도로 생성
  - getter 메서드 : 변수의 값을 읽는 메서드
    - @property 데코레이터 사용
  - setter 메서드 : 변수의 값을 설정하는 성격의 메서드
    - @변수. setter 사용

에러/예외 처리(Error/Exception Handling)

- 버그 : 프로그래밍 언어 코볼의 발명자 그레이스 호퍼가 발견, 소프트웨어에서 발생하는 문제를 버그라고 부름
- 디버깅 : 잘못된 프로그램을 수정하는 것을 의미. --> de(없앤다) + bugging(버그)
  - 에러 메시지가 발생하는 경우(해당 하는 위치를 찾아 메시지를 해결)
  - 디버깅의 도구: print 함수 활용, 개발 환경(text editor, IDE) 등에서 제공하는 기능 활용, python tutor 활용, 뇌컴파일, 눈디버깅

- 문법 에러(Syntax Error) : SyntaxError가 발생하면, 파이썬 프로그램은 실행이 되지 않음
  줄에서 에러가 감지된 가장 앞의 위치를 가리키는 캐럭(caret)기호 (^)를 표시

  - Invalid syntax : 문법오류
  - assign to literal : 잘못된 할당
  - EOL(End of Line) : ex. print('hello <---- 이렇게 실행했을 경우
  - EOF (End of File) : ex. print(<----- 이렇게 실행했을 경우

- 예외(Exception) : 실행 도중 예상치 못한 상황을 맞이하면, 프로그램 실행을 멈춤
  - 문장이나 표현식이 문법적으로 올바르더라도 발생하는 에러
  - 실행 중에 감지되는 에러들을 예외라고 부름
  - 사용자 정의 예외를 만들어 관리할 수 있음
  - ZeroDivisionError : 0으로 나누고자 할 때 발생
  - NameError : namespace상에 이름이 없는 경우
  - TypeError : 타입 불일치
  - ValueError : 타입은 올바르나 값이 적절하지 않거나 없는 경우
  - IndexError : 인덱스가 존재하지 않거나 범위를 벗어나는 경우
  - KeyError : 해당 키가 존재하지 않는 경우
  - ModuleNotFoundError : 해당 모듈이 존재하지 않는 경우
  - ImportError : Module은 있으나 존재하지 않는 클래스/함수를 가져오는 경우
  - KeyboardInterrupt : 임의로 프로그램을 종료하였을 때
  - IndentationError : Indentation이 적절하지 않는 경우

- 예외 처리 : try 문(statement) / except 절(clause)을 이용하여 예외 처리를 할 수 있음

- try : 오류가 발생할 가능성이 있는 코드를 실행

  예외가 발생되지 않으면, except 없이 실행 종료
- except : try 문에서 예외가 발생하면, except 절이 실행

  예외 상황을 처리하는 코드를 받아서 적절한 조치를 취함
- else : try 문에서 예외가 발생하지 않으면 실행함
- finally : 예외 발생 여부와 관계없이 항상 실행함

```
try:                    # try 문은 반드시 한 개 이상의 except문이 필요
    try 명령문
except 예외그룹_1 as 변수_1:
    예외처리 명령문 1
except 예외그룹_2 as 변수_2:
    예외처리 명령문 2
finally:
    finally 명령문
```