In [1]:

```
from IPython.display import IFrame
```

OOP with Python

용어 정리

```
class Person:                       #=> 클래스 정의(선언) : 클래스 객체 생성
    name = '홍길동'                  #=> 멤버 변수(데이터 어트리뷰트)
    def greeting(self):             #=> 멤버 메서드(메서드)
        print(f'{self.name}')
```

```
iu = Person()        # 인스턴스 객체 생성
daniel = Person()    # 인스턴스 객체 생성
iu.name              # 데이터 어트리뷰트 호출
iu.greeting()        # 메서드 호출
```

In [2]:

```
class Person:
    name = '홍길동'
    def greeting(self):
        print(f'{self.name}')

iu = Person()
iu.greeting()
iu.name
```

홍길동

Out[2]:

'홍길동'

- 클래스와 인스턴스간의 관계를 확인해봅시다.

In [3]:

```
isinstance(iu, Person)
```

Out[3]:

True

`self` : 인스턴스 객체 자기자신

- C++ 혹은 자바에서의 this 키워드와 동일함.
- 특별한 상황을 제외하고는 무조건 메서드에서 `self`를 첫번째 인자로 설정한다.
- 메서드는 인스턴스 객체가 함수의 첫번째 인자로 전달되도록 되어있다.

In [5]:

```
# iu를 다시 인사시켜봅시다.
iu.greeting()
```

홍길동

In [7]:

In [7]:

```
# 다르게 인사를 시킬 수도 있습니다.
# 실제로 이렇게 호출이 되는 것과 동일하기에 반드시 self로서 인스턴스 자기자신을 표현해야합니다.
Person.greeting(iu)
```

홍길동

- 클래스 선언부 내부에서도 반드시 self를 통해 데이터 어트리뷰트에 접근 해야 합니다.

In [11]:

```
# 예시를 봅시다.
name = '?'
class Person:
    name = '홍길동'
    def greeting(self):
        print(f'{name}')
```

In [13]:

```
p1 = Person()
print(p1.name)
p1.greeting()
```

홍길동
?

클래스-인스턴스간의 이름공간

- 클래스를 정의하면, 클래스 객체가 생성되고 해당되는 이름 공간이 생성된다.
- 인스턴스를 만들게 되면, 인스턴스 객체가 생성되고 해당되는 이름 공간이 생성된다.
- 인스턴스의 어트리뷰트가 변경되면, 변경된 데이터를 인스턴스 객체 이름 공간에 저장한다.
- 즉, 인스턴스에서 특정한 어트리뷰트에 접근하게 되면 인스턴스 -> 클래스 순으로 탐색을 한다.

In [14]:

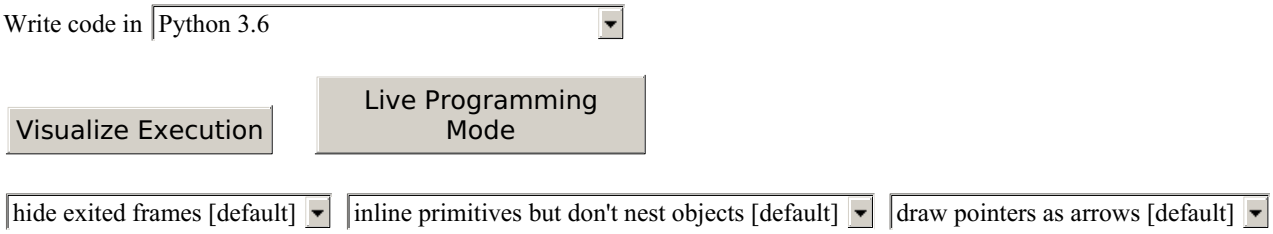
```
iu.name = '아이유'
iu.greeting()
```

아이유

In [15]:

```
# 아래의 Python Tutor를 통해 순차적으로 확인해봅시다.
IFrame('https://goo.gl/ZgNaXB', width='100%', height='500px')
```

Out[15]:



생성자 / 소멸자

- 생성자는 인스턴스 객체가 생성될 때 호출되는 함수이며, 소멸자는 객체가 소멸되는 과정에서 호출되는 함수입니다.

```
    def __init__(self):
        print('생성될 때 자동으로 호출되는 메서드입니다.')

    def __del__(self):
        print('소멸될 때 자동으로 호출되는 메서드입니다.')
```

```
    __someting__
```

위의 형식처럼 양쪽에 언더스코어가 있는 메서드를 스페셜 메서드 혹은 매직 메서드라고 불립니다.

In [16]:

```
# 클래스를 만듭니다.
class Person:
    def __init__(self):
        print('응애')
    def __del__(self):
        print('빠이')
```

In [17]:

```
# 생성시켜봅시다.
p1 = Person()     # Person 클래스의 인스턴스 p1 만들기
```

응애

In [18]:

```
# 소멸시켜봅시다.
del p1
```

빠이

- 생성자 역시 메소드이기 때문에 추가적인 인자를 받을 수 있습니다.

In [19]:

```
# 생성자에서 이름을 추가적으로 받아서 출력해봅시다.
class Person:
    def __init__(self, name):
        print(f'응애, {name}')
    def __del__(self):
        print('빠이')
```

In [20]:

```
# 홍길동이라는 이름을 가진 hong 을 만들어봅시다.
hong = Person('홍길동')
```

응애, 홍길동

- 아래와 같이 모두 사용할 수 있습니다!

```
    def __init__(self, parameter1, parameter2):
        print('생성될 때 자동으로 호출되는 메서드입니다.')
        print(parameter1)

    def __init__(self, *args):
        print('생성될 때 자동으로 호출되는 메서드입니다.')

    def __init__(self, **kwagrs):
        print('생성될 때 자동으로 호출되는 메서드입니다.')
```

- 따라서, 생성자는 값을 초기화하는 과정에서 자주 활용됩니다.
- 아래의 클래스 변수와 인스턴스 변수를 통해 확인해보겠습니다.

클래스 변수 / 인스턴스 변수

```
    class Person:
        population = 0              # 클래스 변수 : 모든 인스턴스가 공유함.

        def __init__(self, name):
            self.name = name        # 인스턴스 변수 : 인스턴스별로 각각 가지는 변수
```

In [23]:

```
# 생성자와 인사하는 메소드를 만들어봅시다.
class Person:
    population = 0

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

    def greeting(self):
        print(f'{self.name} 입니다. 반가워요.')
```

In [24]:

```
# 본인의 이름을 가진 인스턴스를 만들어봅시다.
me = Person('바보')
```

In [25]:

```
# 이름을 출력해봅시다.
me.name
```

Out[25]:

'바보'

In [26]:

```
# 옆자리 친구의 이름을 가진 인스턴스를 만들어봅시다.
friend = Person('멍청이')
```

In [27]:

```
# 이름을 출력해봅시다.
friend.name
```

Out[27]:

'멍청이'

In [28]:

```
# population을 출력해봅시다.
Person.population
```

Out[28]:

```
2
```

In [31]:

```
# 물론, 인스턴스도 접근 가능합니다. 왜일까요?!
# me.population
friend.population
```

Out[31]:

```
2
```

정적 메서드 / 클래스 메서드

- 메서드 호출을 인스턴스가 아닌 클래스가 할 수 있도록 구성할 수 있습니다.
- 이때 활용되는게 정적 메서드 혹은 클래스 메서드입니다.
- 정적 메소드는 객체가 전달되지 않은 형태이며, 클래스 메서드는 인자로 클래스를 넘겨준다.

In [32]:

```
# Person 클래스가 인사할 수 있는지 확인해보겠습니다.
Person.greeting()     # 클래스로는 인스턴스를 호출할 수 없습니다.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-32-6d9af1c11039> in <module>
      1 # Person 클래스가 인사할 수 있는지 확인해보겠습니다.
----> 2 Person.greeting()

TypeError: greeting() missing 1 required positional argument: 'self'
```

In [33]:

```
# 이번에는 Dog class를 만들어보겠습니다.
# 클래스 변수 num_of_dogs 통해 개가 생성될 때마다 증가시키도록 하겠습니다.
# 개들은 각자의 이름과 나이를 가지고 있습니다.
# 그리고 bark() 메서드를 통해 짖을 수 있습니다.
class Dog:
    num_of_dog = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Dog.num_of_dog += 1
    def bark(self):
        print(f'멍멍, {self.name}')
```

In [34]:

```
# 각각 이름과 나이가 다른 인스턴스를 3개 만들어봅시다.
babo = Dog('바보', 3)
mc = Dog('멍청이', 2)
dd = Dog('똥개', 1)
```

In [36]:

```
# babo.bark()
mc.bark()
```

```
멍멍, 멍청이
```

- staticmethod는 다음과 같이 정의됩니다.

```
@staticmethod
def methodname():
    codeblock
```

In [37]:

```
# 단순한 static method를 만들어보겠습니다.
class Dog:
    num_of_dog = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Dog.num_of_dog += 1
    def bark(self):
        print(f'멍멍, {self.name}')

    @staticmethod
    def info():
        print('개입니다.')
```

In [38]:

```
# 3마리를 만들어보고,
babo = Dog('바보', 3)
mc = Dog('멍청이', 2)
dd = Dog('똥개', 1)
```

In [39]:

```
# info 메소드를 호출해 봅니다.
Dog.info()
```

개입니다.

- classmethod는 다음과 같이 정의됩니다.

```
@classmethod
def methodname(cls):
    codeblock
```

In [40]:

```
# 개의 숫자를 출력하는 classmethod를 만들어보겠습니다.
class Dog:
    num_of_dog = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Dog.num_of_dog += 1
    def bark(self):
        print(f'멍멍, {self.name}')

    @classmethod
    def count(cls):
        print(cls)
        print(f'{cls.num_of_dog}마리')
```

In [41]:

```
# 3마리를 만들어보고,
babo = Dog('바보', 3)
mc = Dog('멍청이', 2)
dd = Dog('똥개', 1)
```

In [42]:

```
# count 메소드를 호출해 봅니다.
Dog.count()
```

```
<class '__main__.Dog'>
3마리
```

실습 - 정적 메소드

> 계산기 class인 `Calculator`를 만들어봅시다.

- 정적 메소드 : 두 수를 받아서 각각의 연산을 한 결과를 반환(return)

1. `add()` : 덧셈
2. `sub()` : 뺄셈
3. `mul()` : 곱셈
4. `div()` : 나눗셈

In [43]:

```
# 아래에 코드를 작성해주세요.
class Calculator:
    @staticmethod
    def add(a, b):
        return a + b
    def sub(a, b):
        return a - b
    def mul(a, b):
        return a * b
    def div(a, b):
        return a / b
```

In [44]:

```
Calculator.add(1, 3)
```

Out[44]:

```
4
```

실습 - 스택

> `Stack` 클래스를 간략하게 구현해봅시다.
>
> Stack : 스택은 LIFO(Last in First Out)으로 구조화된 자료구조를 뜻합니다.

1. `empty()` : 스택이 비었다면 참을 주고,그렇지 않다면 거짓이 된다.
2. `top()` : 스택의 가장 마지막 데이터를 넘겨준다. 스택이 비었다면 None을 리턴해주세요.
3. `pop()` : 스택의 가장 마지막 데이터의 값을 넘겨주고 해당 데이터를 삭제한다. 스택이 비었다면 None을 리턴해주세요.
4. `push()` : 스택의 가장 마지막 데이터 뒤에 값을 추가한다. 리턴값 없음

다 완료하신 분들은 repr을 통해 스택의 아이템들을 예쁘게 출력까지 해봅시다.

In [74]:

```
# 여기에 코드를 작성해주세요.
class Stack:
    def __init__(self):
        self.items = []

    def empty(self):
        return not self.items

    def top(self):
```

```
        if self.items:
            return self.items[-1]

    def pop(self):
        if self.items:
            return self.items.pop()

    def push(self, elements):
        self.items.append(elements)
```

In [76]:

```
# 인스턴스를 하나 만들고 메소드 조작을 해봅시다.
s1 = Stack()
```

In [78]:

```
s1.push('hi')
```

In [79]:

```
print(s1)
```

```
<__main__.Stack object at 0x000002561FC4AE10>
```

In [80]:

```
s1.empty()
```

Out[80]:

```
False
```

In [82]:

```
s1.items
```

Out[82]:

```
['hi', 'hi']
```

In [84]:

```
s1.pop()
```

Out[84]:

```
'hi'
```

In [87]:

```
s1.empty()
```

Out[87]:

```
True
```

In [89]:

```
s1.items
```

Out[89]:

```
[]
```

연산자 오버라이딩(중복 정의)

- 파이썬에 기본적으로 정의된 연산자를 직접적으로 정의하여 활용할 수 있습니다.
- 몇가지만 소개하고 활용해봅시다.

```
+  __add__
-  __sub__
*  __mul__
<  __lt__
<= __le__
== __eq__
!= __ne__
>= __ge__
>  __gt__
```

In [91]:

```
# 사람과 사람을 더하면, 나이의 합을 반환하도록 만들어봅시다.
class Person:
    population = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.population += 1

    def greeting(self):
        print(f'{self.name}입니다. 반가워요.')

    def __add__(self, other):
        return f'나이의 합은 {self.age + other.age}'

    def __sub__(self, other):
        return f'나이의 차는 {self.age - other.age}'
```

In [92]:

```
# 연산자를 호출해봅시다.
p1 = Person('바보', 30)
p2 = Person('멍청이', 28)
```

In [94]:

```
# 원하는 연산자로 사람과 사람을 비교해보세요.
p1 + p2
```

Out[94]:

```
'나이의 합은 58'
```

In [95]:

```
# 원하는 연산자로 사람과 사람을 비교해보세요.
p1 - p2
```

Out[95]:

```
'나이의 차는 2'
```

In [ ]:

```
# 파이썬 내부를 살펴봅시다.
print(1 + 3)
print('1' + '3')
# 이렇게 + 연산자가 서로 다르게 활용될 수 있는 이유는 파이썬 내부적으로 각각 다른 클래스마다 다른 정의가 있기 때문입니다.
```

상속

기초

- 클래스에서 가장 큰 특징은 '상속' 기능을 가지고 있다는 것이다.
- 부모 클래스의 모든 속성이 자식 클래스에게 상속 되므로 코드재사용성이 높아집니다.

```
class DerivedClassName(BaseClassName):
    code block
```

In [96]:

```
# 인사만 할 수 있는 간단한 사람 클래스를 만들어봅시다.
class Person:
    def __init__(self, name):
        self.name = name

    def greeting(self):
        print(f'안녕! 난 {self.name}야.')
```

In [97]:

```
# 사람 클래스를 상속받아 학생 클래스를 만들어봅시다.
class Student(Person):
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id
```

In [98]:

```
# 학생을 만들어봅시다.
s1 = Student('바보', '20190101')
```

In [99]:

```
# 부모 클래스에 정의를 했음에도 메소드를 호출 할 수 있습니다.
s1.greeting()
```

```
안녕! 난 바보야.
```

- 이처럼 상속은 공통된 속성이나 메소드를 부모 클래스에 정의하고, 이를 상속받아 다양한 형태의 사람들을 만들 수 있습니다.

In [100]:

```
# 진짜 상속관계인지 확인해봅시다.
issubclass(Person, Student)
```

Out[100]:

```
False
```

In [102]:

```
issubclass(Student, Person)
```

Out[102]:

```
True
```