

1. 클래스, 인스턴스, 객체

주요 내용

- 클래스, 인스턴스, 객체 개념 확인
- 예제: 기약분수들의 자료형인 `Fraction` 클래스 선언
- `Fraction` 클래스의 인스턴스: `1/2`, `2/7` 등과 같은 기약분수
- 분수들의 덧셈, 동등성 비교 등을 지원하는 메서드 활용

1.1. 클래스 선언과 생성자

1.1.1. 클래스 선언

- 파이썬 클래스의 선언은 다음 형식을 따름.

```
class 클래스명:  
    ...  
    # 속성 지정 및 메서드 선언
```

1.1.2. 생성자

- `Fraction` 클래스의 생성자: 분자와 분모를 입력받아 저장
- `__init__()` 메서드
 - 클래스의 인스턴스를 생성하는 **생성자**
 - 인스턴스의 속성으로 저장될 정보와 관련된 값 저장

```
In [1]: class Fraction:  
    def __init__(self, top, bottom):  
        """생성자 메서드  
        top: 분자  
        bottom: 분모  
        """  
  
        self.top = top  
        self.bottom = bottom
```

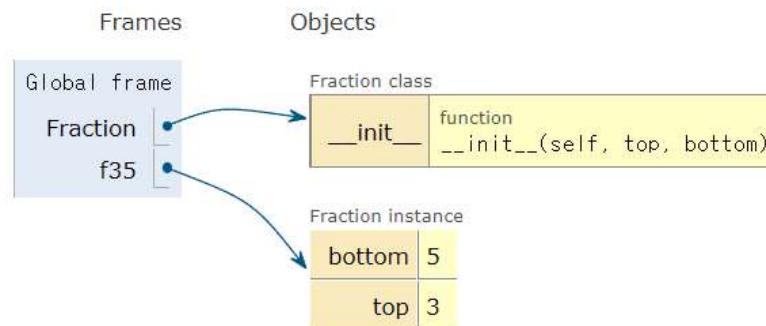
1.2. 인스턴스와 객체

객체는 특정 클래스의 인스턴스로 생성된다.

1.2.1. 인스턴스 생성

```
In [2]: f35 = Fraction(3, 5)
```

- `__init__()` 메서드: 분자는 3, 분모는 5로 지정
- 최종적으로 $\frac{3}{5}$ 에 해당하는 객체 생성



self의 기능

- `f35 = Fraction(3, 5)` 방식으로 변수 할당이 실행될 때 내부에서 다음이 실행됨

```
__init__(f35, 3, 5)
```

- 즉, `self` 매개변수는 현재 생성되는 객체를 인자로 사용함

1.2.2. 인스턴스 변수와 속성

- 인스턴스 변수: 클래스 내부에서 `self` 와 함께 선언된 변수.
- `Fraction` 클래스의 인스턴스 변수: `top` 과 `bottom`
- 클래스 내부에서만, 따라서 생성되는 객체 고유의 속성을 가리키는 역할 수행
- 생성된 객체의 속성을 가리킨다는 의미에서 인스턴스 속성으로도 불림.

1.3. 매직 메서드

인스턴스 메서드

- 메서드: 클래스 내부에서 선언된 함수
- 인스턴스 메서드
 - 첫째 인자로 `self` 를 사용하는 메서드
 - 클래스의 인스턴스가 생성되어야만 활용될 수 있음.

매직 메서드

- 클래스에 기본적으로 포함되는 인스턴스 메서드
- 클래스가 기본적으로 갖춰야 하는 기능 제공
- 메서드 이름이 밑줄 두 개로 감싸짐.

dir() 함수

- 객체(클래스의 인스턴서)에 포함된 인스턴스 변수와 매직 메서드로 구성된 리스트 반환.

```
In [3]: print(dir(Fraction))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

1.3.1. 객체 출력

`__str__()` 매직 메서드

- `f35` 는 '3/5'에 해당하는 분수를 가리켜야 함.
- 그런데 `print()` 함수를 이용하여 이 사실을 확인하려면 이해할 수 없는 결과가 출력됨.

```
In [4]: print(f35)
```

```
<__main__.Fraction object at 0x7f4bb384d7f0>
```

`__str__()` 매직 메서드 재정의

- `__str__()` 메서드를 필요한 방식으로 재정의해야 함.

```
In [5]: class Fraction:  
    def __init__(self, top, bottom):  
        self.top = top  
        self.bottom = bottom  
  
    def __str__(self):  
        return f"{self.top}/{self.bottom}" # 3/5, 1/2 형식으로 출력
```

```
In [6]: f35 = Fraction(3, 5)  
print(f35)
```

3/5

`__repr__()` 매직 메서드

- 반면에 `f35`를 확인하면 제대로 출력되지 않음.

```
In [7]: f35
```

```
Out[7]: <__main__.Fraction at 0x7f4bb3882d80>
```

`__repr__()` 매직 메서드 재정의

- `__repr__()` 메서드를 필요한 방식으로 재정의해야 함.

```
In [8]: class Fraction:  
    def __init__(self, top, bottom):  
        self.top = top  
        self.bottom = bottom  
  
    def __str__(self):  
        return f"{self.top}/{self.bottom}"  
  
    def __repr__(self):  
        return f"{self.top}/{self.bottom}"
```

```
In [9]: f35 = Fraction(3, 5)  
f35
```

```
Out[9]: 3/5
```

`__repr__()` 대 `__str__()`

```
In [10]: class Fraction:  
    def __init__(self, top, bottom):  
        self.top = top  
        self.bottom = bottom  
  
    def __str__(self):  
        return f"{self.top}/{self.bottom}" # 3/5, 1/2 형식으로 출력  
  
    def __repr__(self):  
        return f"{self.bottom}분의 {self.top}" # 5분의 3 형식으로 출력
```

```
In [11]: f35 = Fraction(3, 5)  
f35
```

Out[11]: 5분의 3

```
In [12]: print(f35)
```

3/5

`__repr__()` 만 사용 가능

```
In [13]: class Fraction:  
    def __init__(self, top, bottom):  
        self.top = top  
        self.bottom = bottom  
  
    def __repr__(self):  
        return f"{self.bottom}/{self.top}"
```

```
In [14]: f35 = Fraction(3, 5)  
f35
```

```
Out[14]: 5/3
```

```
In [15]: print(f35)
```

```
5/3
```

1.3.2. 인스턴스 연산

- 덧셈 연산이 지원되지 않음.

```
In [16]: f14 = Fraction(1, 4)
          f12 = Fraction(1, 2)
```

```
In [17]: f14 + f12
```

```
-----
TypeError: unsupported operand type(s) for +: 'Fraction' and 'Fraction'
Cell In[17], line 1
----> 1 f14 + f12
```

```
Traceback (most recent call last)
```

```
TypeError: unsupported operand type(s) for +: 'Fraction' and 'Fraction'
```

`__add__()` 매직 메서드

- 아래 성질을 이용하는 `__add__()` 매직 메서드 선언 필요

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + cb}{bd}$$

```
In [18]: class Fraction:  
    def __init__(self, top, bottom):  
        self.top = top  
        self.bottom = bottom  
  
    def __repr__(self):  
        return f"{self.top}/{self.bottom}"  
  
    def __add__(self, other):  
        new_top = self.top * other.bottom + self.bottom * other.top  
        new_bottom = self.bottom * other.bottom  
  
        return Fraction(new_top, new_bottom)
```

```
In [19]: f14 = Fraction(1, 4)  
f12 = Fraction(1, 2)  
  
f14 + f12
```

```
Out[19]: 6/8
```

기약분수 처리: gcd() 함수

- $1/4 + 1/2$ 의 계산이 $3/4$ 가 아닌 $6/8$ 로 계산됨.
- 즉, 기약분수로의 변환이 진행되지 않음.
- 분모와 분자를 두 수의 최대공약수로 나눠야 함.

```
In [20]: def gcd(m, n):  
    while m % n != 0:  
        m, n = n, m % n  
    return n
```

```
In [21]: print(gcd(6, 14))  
print(gcd(8, 20))
```

2

4

```
In [22]: class Fraction:
    def __init__(self, top, bottom):
        self.top = top
        self.bottom = bottom

    def __repr__(self):
        return f"{self.top}/{self.bottom}"

    def __add__(self, other):
        new_top = self.top * other.bottom + self.bottom * other.top
        new_bottom = self.bottom * other.bottom
        common = gcd(new_top, new_bottom)

        return Fraction(new_top // common, new_bottom // common)
```

```
In [23]: f14 = Fraction(1, 4)
f12 = Fraction(1, 2)

f14 + f12
```

```
Out[23]: 3/4
```

연산 실행: self 와 other

- `f14 + f12` 표현식이 실행되면 실제로는 `f14` 객체의 `__add__()` 메서드가 다음과 같이 호출됨.
- 즉, `f14` 를 기준(self)으로 다른 값(other) `f12` 와의 덧셈이 실행됨.

```
In [24]: f14.__add__(f12)
```

```
Out[24]: 3/4
```

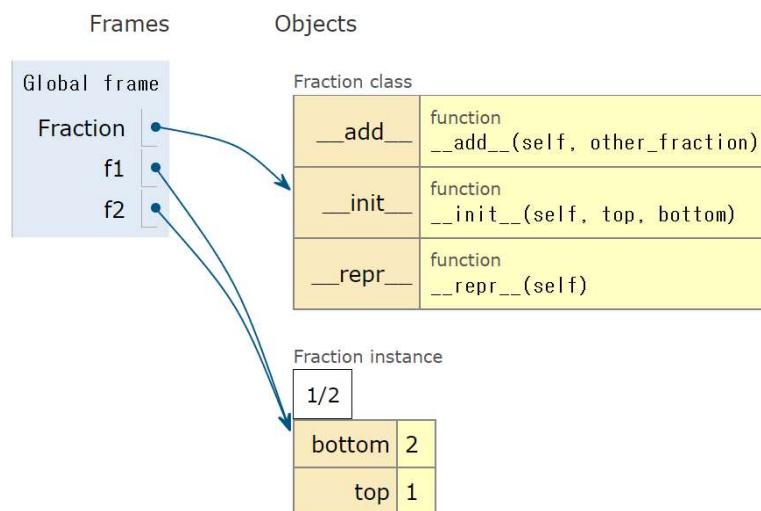
1.3.3. 인스턴스 동일성/동등성

- 두 객체의 동일성_{identity}: 두 객체가 동일한 메모리 주소에 저장되었는가에 따라 결정됨.
- 두 객체의 동등성_{equality}: 두 객체가 가리키는 값들의 동일성으로 결정

is 연산자: 동일성 판단

```
In [25]: f1 = Fraction(1, 2)
f2 = f1
print("동일성:", f1 is f2, ", ", "동등성:", f1 == f2)
```

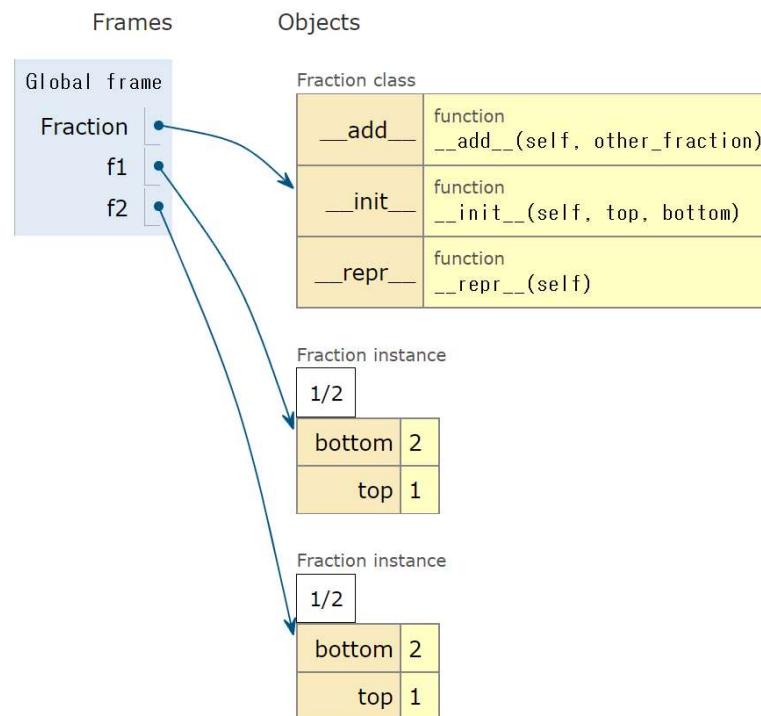
동일성: True , 동등성: True



== 연산자: 동등성 판단

```
In [26]: f1 = Fraction(1, 2)
f2 = Fraction(1, 2)
print("동일성:", f1 is f2, ", ", "동등성:", f1 == f2)
```

동일성: False , 동등성: False



`__eq__()` 매직 메서드

- `__eq__` 매직 메서드: 두 객체의 동등성을 정의하여 동일하진 않지만 동등하도록 판정되도록 하게 만듦.
- 두 분수의 동등성은 아래와 같이 판단됨:

$$\frac{a}{b} = \frac{c}{d} \iff ad = bc$$

```
In [27]: class Fraction:
    def __init__(self, top, bottom):
        self.top = top
        self.bottom = bottom

    def __repr__(self):
        return f"{self.top}/{self.bottom}"

    def __add__(self, other):
        new_top = self.top * other.bottom + self.bottom * other.top
        new_bottom = self.bottom * other.bottom
        common = gcd(new_top, new_bottom)

        return Fraction(new_top // common, new_bottom // common)

    def __eq__(self, other): # 분수 객체의 동등성
        first_top = self.top * other.bottom
        second_top = other.top * self.bottom

        return first_top == second_top
```

```
In [28]: f1 = Fraction(1, 2)
f2 = Fraction(1, 2)
print("동일성:", f1 is f2, ", ", "동등성:", f1 == f2)
```

동일성: False , 동등성: True

1.4. 인스턴스 메서드

- 매직 메서드 이외에 클래스에 다른 기능을 제공하는 인스턴스 메서드 선언 가능

예제: 분모와 분자 추출

```
In [29]: class Fraction:
    def __init__(self, top, bottom):
        self.top = top
        self.bottom = bottom
    def __repr__(self):
        return f'{self.top}/{self.bottom}'
    def __add__(self, other):
        new_top = self.top * other.bottom + self.bottom * other.top
        new_bottom = self.bottom * other.bottom
        common = gcd(new_top, new_bottom)
        return Fraction(new_top // common, new_bottom // common)
    def __eq__(self, other):
        first_top = self.top * other.bottom
        second_top = other.top * self.bottom
        return first_top == second_top

    def numerator(self):
        return self.top      # 분자 반환
    def denominator(self):
        return self.bottom  # 분모 반환
```

```
In [30]: f3 = Fraction(2, 3)
print("분자:", f3.numerator())
print("분모:", f3.denominator())
```

분자: 2
분모: 3

예제: 부동소수점으로의 변환

- 분수를 부동소수점으로 변환

```
In [31]: class Fraction:
    def __init__(self, top, bottom):
        self.top = top
        self.bottom = bottom
    def __repr__(self):
        return f'{self.top}/{self.bottom}'
    def __add__(self, other):
        new_top = self.top * other.bottom + self.bottom * other.top
        new_bottom = self.bottom * other.bottom
        common = gcd(new_top, new_bottom)
        return Fraction(new_top // common, new_bottom // common)
    def __eq__(self, other):
        first_top = self.top * other.bottom
        second_top = other.top * self.bottom
        return first_top == second_top
    def numerator(self):
        return self.top
    def denominator(self):
        return self.bottom

    def to_float(self):
        return self.numerator() / self.denominator()
```

```
In [32]: f3 = Fraction(2, 3)
print(f3.to_float())
```

0.6666666666666666