클래스, 인스턴스, 객체

주요 내용

- 파이썬: 객체 지향 프로그래밍 언어 활용
- 직접 필요한 자료형을 클래스로 정의하고 객체를 생성하여 활용

예제: `Fraction` 클래스 직접 정의

- 클래스, 인스턴, 객체 개념 소개
- `1/2`, `2/7` 처럼 기약분수들의 자료형 역할 수행
- 분수들의 덧셈, 크기 비교 등 지원

클래스 선언과 생성자

- `__init__()` 메서드
    - 클래스의 인스턴스를 생성하는 **생성자**
    - 인스턴스의 속성으로 저장될 정보와 관련된 값을 인자로 받음
- `Fraction` 클래스의 생성자: 분자와 분모 저장

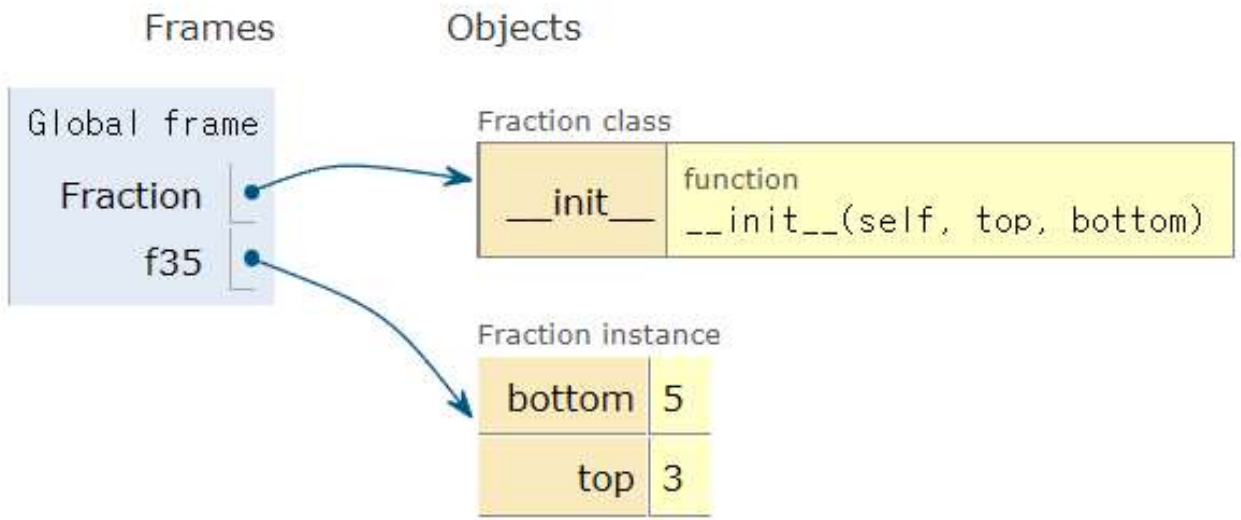
In [1]:
```
class Fraction:
    def __init__(self, top, bottom):
        """생성자 메서드
        top: 분자
        bottom: 분모
        """

        self.top = top
        self.bottom = bottom
```

인스턴스(객체) 생성

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

- `Fraction` 클래스의 인스턴스 생성
- `__init__()` 메서드: 분자는 3, 분모는 5로 지정
- 최종적으로 `3/5`에 해당하는 객체 생성

메모리 상태 변화


Frames
Objects
Global frame
Fraction
f35
Fraction class
__init__
function
__init__(self, top, bottom)
Fraction instance
bottom
5
top
3

`self`의 역할

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

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

- 즉, `self` 매개변수는 현재 생성되는 객체를 인자로 사용함

인스턴스 변수와 속성

- 인스턴스 변수
    - 클래스 내부에서 `self`와 함께 선언된 변수
    - `Fraction` 클래스의 인스턴스 변수: `top` 과 `bottom`
    - 클래스의 영역scope에서만 의미를 가짐.

- 인스턴스 속성
    - 클래스의 인스턴스가 생성되면서 인스턴스 변수가 가리키는 값
    - 클래스의 인스턴스가 생성되어야만 의미를 가짐

`__dict__` 속성

- 인스턴스 변수와 인스턴스 속성 정보를 사전으로 저장

In [3]:
```
f35.__dict__
```

Out[3]:
```
{'top': 3, 'bottom': 5}
```

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
- 밑줄 두 개로 감싸진 이름을 가진 메서드
- 클래스가 기본적으로 갖춰야 하는 기능 제공

`dir()` 함수

- 객체(클래스의 인스턴서)에 포함된 인스턴스 속성, 매직 메서드들의 리스트 확인

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

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__ini
t__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__
sizeof__', '__str__', '__subclasshook__', '__weakref__', 'bottom', 'to
p']
```

`__str__()` 메서드

In [5]:
```
print(f35)
```

```
<__main__.Fraction object at 0x000002ABAACBAC50>
```

In [6]:
```
class Fraction:
    """Fraction 클래스"""

    def __init__(self, top, bottom):
        """생성자 메서드
        top: 분자
        bottom: 분모
        """
        self.top = top
        self.bottom = bottom

    def __str__(self):
        return f"{self.top}/{self.bottom}"  # 3/5, 1/2 형식으로 출력
```

In [7]:
```
f35 = Fraction(3, 5)
```

In [8]:
```
print(f35)
```

```
3/5
```

`__repr__()` 메서드

In [9]:
```
f35
```

Out[9]:
```
<__main__.Fraction at 0x2abaabd0c40>
```

In [10]:
```python
class Fraction:
    """Fraction 클래스"""

    def __init__(self, top, bottom):
        """생성자 메서드
        top: 분자
        bottom: 분모
        """
        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.top}/{self.bottom}"  # 3/5, 1/2 형식으로 출력
```

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

In [12]:
```python
f35
```

Out[12]:
```
3/5
```

In [13]:
```
class Fraction:
    """Fraction 클래스"""

    def __init__(self, top, bottom):
        """생성자 메서드
        top: 분자
        bottom: 분모
        """
        self.top = top
        self.bottom = bottom

#     def __str__(self):
#         return f"{self.top}/{self.bottom}"  # 3/5, 1/2 형식으로 출력

    # __str__() 메서드 대신 활용 가능
    def __repr__(self):
        return f"{self.bottom}분의 {self.top}"  # 5분의 3 형식으로 출력
```

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

In [15]:
```
f35
```

Out[15]: 5분의 3

In [16]:
```
print(f35)
```

5분의 3

`__add__()` 메서드

In [17]:
```
class Fraction:
    """Fraction 클래스"""

    def __init__(self, top, bottom):
        """생성자 메서드
        top: 분자
        bottom: 분모
        """
        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 [18]:
```
f14 = Fraction(1, 4)
f12 = Fraction(1, 2)

f14 + f12
```

Out[18]: 6/8

기약분수 처리: `gcd()` 함수 활용

In [19]:
```
def gcd(m, n):
    while m % n != 0:
        m, n = n, m % n
    return n
```

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

```
2
4
```

In [21]:
```
class Fraction:
    """Fraction 클래스"""

    def __init__(self, top, bottom):
        """생성자 메서드
        top: 분자
        bottom: 분모
        """
        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 [22]:
```
f14 = Fraction(1, 4)
f12 = Fraction(1, 2)

f14 + f12
```

Out[22]: 3/4

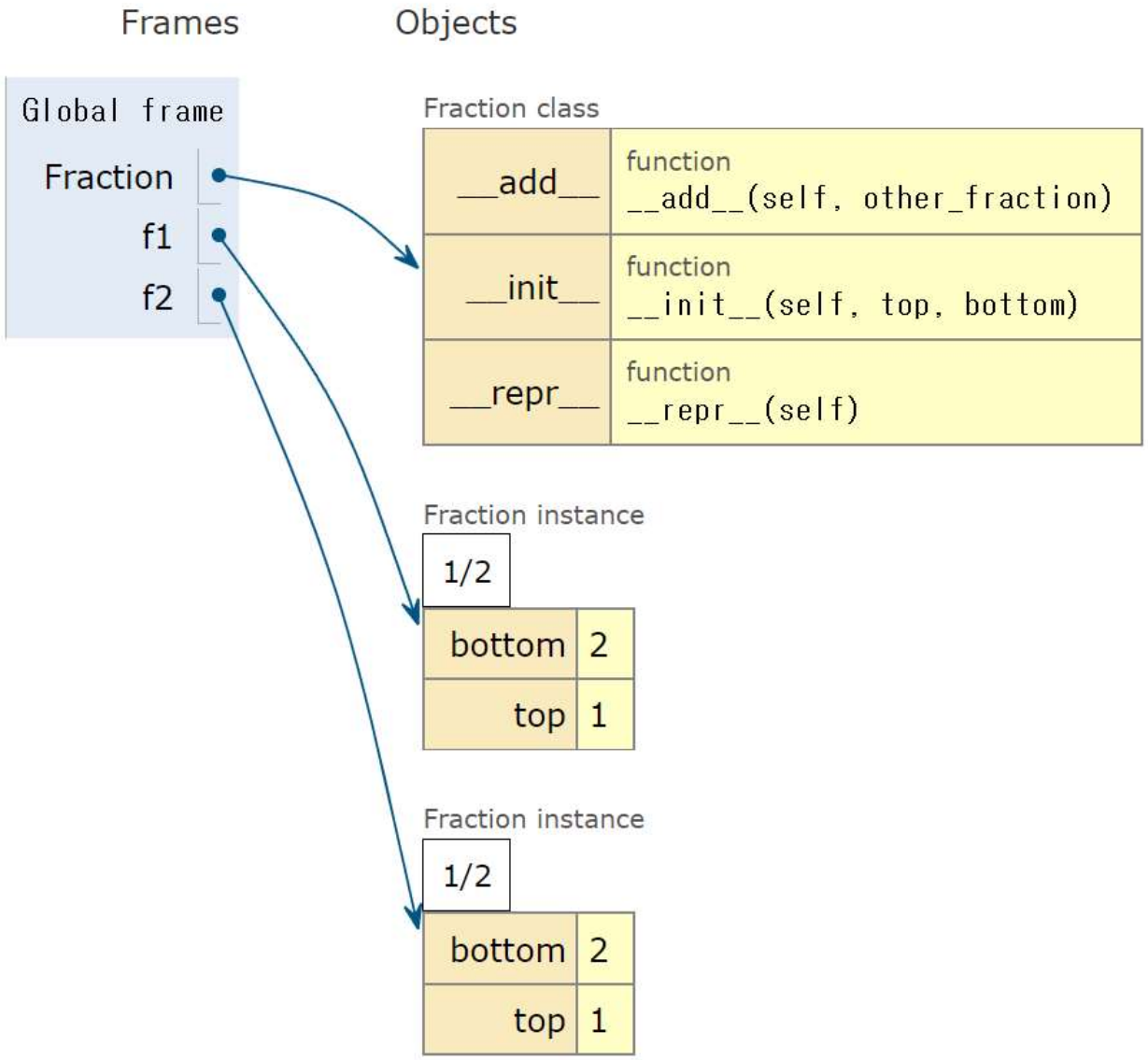
`__eq__()` 메서드

- 두 객체의 동일성identity: 두 객체가 동일한 메모리 주소에 저장되었는가에 따라 결정됨
- 두 객체의 동등성equality: 메모리의 주소가 아니라 객체가 표현하는 값의 동일성 여부 판정

- 아래 `f1`과 `f2`는 동일하지도, 동등하지도 않음.

In [23]:
```
f1 = Fraction(1, 2)
f2 = Fraction(1, 2)

print(f1 is f2) # 동일성
print(f1 == f2) # 동등성
```

```
False
False
```

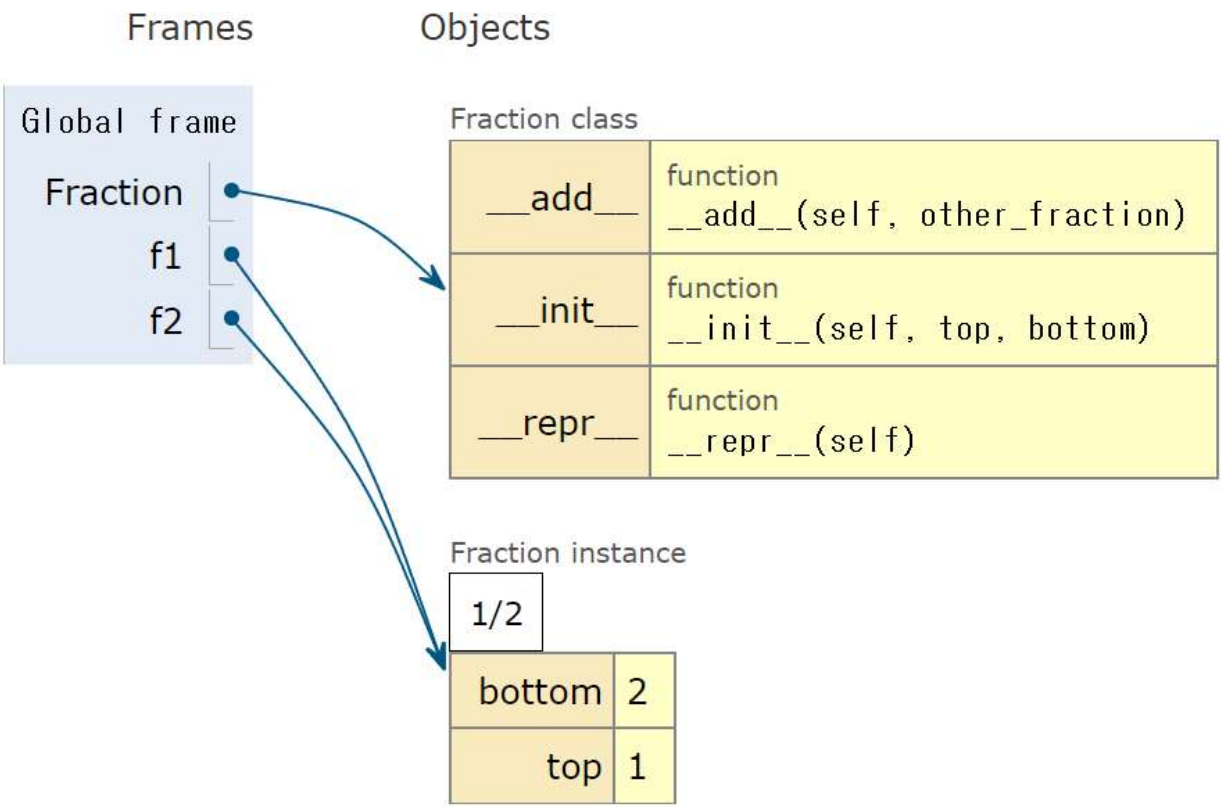

Frames
Objects
Global frame
Fraction
f1
f2
Fraction class
__add__
function
__add__(self, other_fraction)
__init__
function
__init__(self, top, bottom)
__repr__
function
__repr__(self)
Fraction instance
1/2
bottom
2
top
1
Fraction instance
1/2
bottom
2
top
1

동일성

- 동일한 객체를 가리키면 동일하면서 동시에 동등함.

In [24]:
```
f1 = Fraction(1, 2)
f2 = f1

print(f1 is f2) # 동일성
print(f1 == f2) # 동등성
```

```
True
True
```


Frames
Objects
Global frame
Fraction
f1
f2
Fraction class
__add__
function
__add__(self, other_fraction)
__init__
function
__init__(self, top, bottom)
__repr__
function
__repr__(self)
Fraction instance
1/2
bottom
2
top
1

동등성

- 두 분수의 동등성:

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

- `__eq__` 매직 메서드: 두 객체의 동등성 정의

In [25]:

```
class Fraction:
    """Fraction 클래스"""

    def __init__(self, top, bottom):
        """생성자 메서드
        top: 분자
        bottom: 분모
        """
        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
```

In [26]:
```
f1 = Fraction(1, 2)
f2 = Fraction(1, 2)

print(f1 is f2) # 동일성
print(f1 == f2) # 동등성
```

```
False
True
```

`self`와 `other`

- 주로 이항 연산자를 정의할 때 활용됨
    - `self`: 객체 자신. 연산의 중심 역할 수행
    - `other`: 다른 객체. 연산에 필요한 역할 수행

- `f1 == f2` 는 `__eq__()` 메서드를 다음과 같이 호출함

```
f1.__eq__(f2)
```

- 내부적으로는 다음과 같이 실행됨

```
__eq__(f1, f2)
```

인스턴스 메서드: 분모와 분자 추출

- `numerator()` 메서드: 분자 반환
- `denominator()` 메서드: 분모 반환

```
class Fraction:
    """Fraction 클래스"""

    def __init__(self, top, bottom):
        """생성자 메서드
        top: 분자
        bottom: 분모
        """
        self.top = top
        self.bottom = bottom

    ...

    def numerator(self):
        return self.top

    def denominator(self):
        return self.bottom
```

```
>>> f3 = Fraction(2, 3)

>>> f3.numerator()
2

>>> f3.denominator()
3
```

인스턴스 메서드: 부동소수점으로의 변환

- `to_float()` 메서드: 분수를 부동소수점으로 변환

```
class Fraction:
    """Fraction 클래스"""

    def __init__(self, top, bottom):
        """생성자 메서드
        top: 분자
        bottom: 분모
        """
        self.top = top
        self.bottom = bottom

        ...

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

```
>>> f3 = Fraction(2, 3)

>>> f3.to_float()
0.6666666666666666
```