特殊方法（魔术方法）

官方定义好的，以两个下划线开头且以两个下划线结尾来命名的方法。

在特定情况下，它会被自动调用，不需要我们主动调用该方法。

__init__(self [, ...])

- 初始化方法，在实例化过程中调用

```
class Ex:

    def __init__(self, arg1, arg2):
        print(f"__init__被调用, arg1:{arg1}, arg2:{arg2}")


Ex("a", "b")  # 实例化
```

__call__(self [, ...])

- 当实例对象像函数那样被“调用”时，会调用该方法

```
class Ex:

    def __call__(self, arg1, arg2):
        print(f"__call__被调用, arg1:{arg1}, arg2:{arg2}")


e = Ex()
e("a", "b")
```

__getitem__(self, key)

- 当执行 self[key] 操作时，会调用该方法

```
class Ex:

    def __getitem__(self, key):
        print(f"__getitem__被调用, key: {key}")
        print(["a", "b", "c"][key])
        print({0: "零", 1: "壹", 2: "贰"}[key])


e = Ex()
e[2]
```

__len__(self)

- 对实例对象求长度时，会调用该方法，要求必需返回整数类型

```
class Ex:

    def __len__(self):
        return 1234


e = Ex()
print(len(e))
```

__repr__(self) / __str__(self)

- 实例对象转字符串时，会调用该方法，要求必需返回字符串类型

```python
class Ex:

    def __repr__(self):
        return "__repr__被调用"

    # def __str__(self):
    #     return "__str__被调用"


e = Ex()
print(str(e))
print(f"{e}")
print(e)  # print会转成字符串再输出
```

__add__(self, other)

- 实例对象进行加法操作时会调用该方法，要求只要加法左边有当前类的实例对象即可

```python
class Number:

    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return self.num + other


n = Number(6)
print(n + 7)  # 实例对象在左边
```

__radd__(self, other)

- 实例对象进行加法操作时会调用该方法，要求加法右边有当前类的实例对象且左边没有

```
class Number:

    def __init__(self, num):
        self.num = num

    def __radd__(self, other):
        return other + self.num


n = Number(6)
print(7 + n)  # 实例对象在右边
```

__sub__(self, other)

- 实例对象进行减法操作时会调用该方法，要求只要减法左边有当前类的实例对象即可

```
class Number:

    def __init__(self, num):
        self.num = num

    def __sub__(self, other):
        return self.num - other


n = Number(6)
print(n - 4)  # 实例对象在左边
```

__rsub__(self, other)

- 实例对象进行减法操作时会调用该方法，要求减法右边有当前类的实例对象且左边没有

```
class Number:

    def __init__(self, num):
        self.num = num

    def __rsub__(self, other):
        return other - self.num


n = Number(6)
print(4 - n)  # 实例对象在右边
```

__mul__(self, other)

- 实例对象进行乘法操作时会调用该方法，要求只要乘法左边有当前类的实例对象即可

```
class Number:

    def __init__(self, num):
        self.num = num

    def __mul__(self, other):
        return self.num * other


n = Number(6)
print(n * 4)  # 实例对象在左边
```

__rmul__(self, other)

- 实例对象进行乘法操作时会调用该方法，要求乘法右边有当前类的实例对象且左边没有

```
class Number:

    def __init__(self, num):
        self.num = num

    def __rmul__(self, other):
        return other * self.num


n = Number(6)
print(4 * n)  # 实例对象在右边
```

__truediv__(self, other)

- 实例对象进行除法操作时会调用该方法，要求只要除法左边有当前类的实例对象即可

```
class Number:

    def __init__(self, num):
        self.num = num

    def __truediv__(self, other):
        return self.num / other


n = Number(6)
print(n / 3)  # 实例对象在左边
```

__rtruediv__(self, other)

- 实例对象进行除法操作时会调用该方法，要求除法右边有当前类的实例对象且左边没有

```
class Number:

    def __init__(self, num):
        self.num = num

    def __rtruediv__(self, other):
        return other / self.num


n = Number(6)
print(3 / n)  # 实例对象在右边
```

__neg__(self)

- 实例对象进行相反数操作时会调用该方法

```
class Ex:

    def __neg__(self):
        return 1234


e = Ex()
print(-e)
```

案例：实现分数运算

```
"""
根据约分的规则，定义一个求最大公约数的函数：
- 分母为负数时，返回最大公约数的相反数，可以确保分子分母都为负数时，负号消掉
  还可以确保负号永远在分子
- 分母为0时，返回的最大公约数为0，这样可以保证在除零时正常报错
"""
def get_gcd(num1, num2):
```

```
    for i in range(min(abs(num1), abs(num2)), 0, -1):
        if not (num1 % i or num2 % i):
            return -i if num2 < 0 else i
    return num2


def other_to_frac(obj):
    if type(obj) == Fraction:
        return obj
    if isinstance(obj, int):
        return Fraction(obj, 1)
    if type(obj) == float:
        p = 10 ** len(str(obj).split('.')[1])
        return Fraction(int(obj*p), p)
    raise TypeError(f"unsupported operand type(s): <class
'Fraction'> and {type(obj)}")


class Fraction:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        gcd = get_gcd(self.a, self.b)
        x, y = self.a // gcd, self.b // gcd
        if y == 1:
            return f'{x}'
        return f'{x} / {y}'

    def __add__(self, other):
        other = other_to_frac(other)
        return Fraction(self.a * other.b + other.a * self.b, self.b
* other.b)

    def __radd__(self, other):
        return self + other

    def __sub__(self, other):
        other = other_to_frac(other)
```

```
        return Fraction(self.a * other.b - other.a * self.b, self.b
* other.b)

    def __rsub__(self, other):
        other = other_to_frac(other)
        return other - self

    def __mul__(self, other):
        other = other_to_frac(other)
        return Fraction(self.a * other.a, self.b * other.b)

    def __rmul__(self, other):
        return self * other

    def __truediv__(self, other):
        other = other_to_frac(other)
        return Fraction(self.a * other.b, self.b * other.a)

    def __rtruediv__(self, other):
        other = other_to_frac(other)
        return other / self


f1 = Fraction(1, 2)
f2 = Fraction(6, 3)
f3 = Fraction(1, -4)
f4 = Fraction(-1, 4)
print(f1 + f2
print(f1 - f2)
print(f1 * f2)
print(f1 / f2)
print(f1 + 3)
print(f1 - 3.2)
print(f1 * False)
print(f1 / True)
print(3 + f1)
print(3.2 - f1)
print(True * f1)
print(False / f1)
print(0.5 + 1 / f2 * True - 4 * f3 + f4)
```