Assignment3

October 25, 2024

0.0.1 CS2101 - Programming for Science and Finance

Prof. Götz Pfeiffer School of Mathematical and Statistical Sciences University of Galway

---

1 Computer Lab 3

1.1 1. Payments.

Construct a class `Payment` for financial payments with attributes `amount` and `description`. The `amount` should be a floating point number (perhaps rounded to 2 decimal digits). The `description` should be a (short) text describing the purpose of this payment.

Methods to be defined in the class shnould include * a constructor `__init__` to make a `Payment` object and assign vlues to the attributes `amount` and `description`; * a string representation `__repr__`, perhaps printing the `Payment` object in the same was as it was constructed; * an addition function `__add__`, which returns a new `Payment` object whose amount is the sum of two given payements, and whose description is simply `"sum"` or so. * a negation function `__neg__`, which returns a new payment whose amount is the negative of a given payment * a subtraction function `__sub__`, which computes the difference of two given payments as the sum of one and the negative of the other.

[63]:
```python
class Payment:
    def __init__(self, amount : float, description : str):
        self.amount : float = round(amount,2)
        self.description : str = description

    def __repr__(self):
        return f"{self.description} : {self.amount}"

    def __add__(self, other):
        # Adding two payments together
        if isinstance(other, Payment):
            return Payment(self.amount + other.amount, "sum")
        return NotImplemented

    def __neg__(self):
```

```
        return Payment(-self.amount, "negation")

    def __sub__(self, other):
        if isinstance(other, Payment):
            return Payment(self.amount - other.amount, "subtract")
        else:
            return NotImplemented
```

- Test your class and objects with the following examples

[64]:
```
coffee = Payment(2.568, "Coffee")
deposit = Payment(0.50, "Deposit")
print(coffee)
print(deposit)
```

```
Coffee : 2.57
Deposit : 0.5
```

[65]:
```
coffee + deposit
```

[65]: sum : 3.07

[66]:
```
coffee - deposit
```

[66]: subtract : 2.07

1.2 2. Rational Numbers.

- Define a class `Rational` for **rational number objects** as follows.
- A `Rational` object should have two (integer valued) attributes `num` (for the **numerator**) and `den` (for the **denominator**), so that it represents the value `num/den`.
- Care should be taken to keep these attributes in **lowest terms**.
- For this you might need a function `gcd` that computes the **greatest common divisor** of two integers: use the one from class, or `from math import gcd`.

[67]:
```
from math import gcd

gcd(24, 16)
```

[67]: 8

1. Start with a class definition that defines a constructor `__init__` and a string representation method `__repr__` for rational number objects.

[85]:
```
class Rational:
    def __init__(self, num: int, den: int):
        if den == 0:
            raise ValueError("Denominator cannot be zero.")
```

```
        # Reduce the fraction to its lowest terms
        common_divisor = gcd(num, den)
        self.num = num // common_divisor
        self.den = den // common_divisor

        # If the denominator is negative, adjust the sign to keep the␣
↪denominator positive
        if self.den < 0:
            self.num = -self.num
            self.den = -self.den

    def __repr__(self):
        if self.den == 1:
            return str(self.num)
        else:
            return f"{self.num}/{self.den}"
```

2. Test the class on these objects:

[69]:
```
print(Rational(2,3))
print(Rational(-2,3))
print(Rational(2, -3))
```

```
2/3
-2/3
-2/3
```

3. Write a function `neg_rational` that computes and returns the **negative** of a rational number object as a rational number object.

[70]:
```
def neg_rational(rat : Rational) -> Rational:
    return Rational(-rat.num, rat.den)
```

4. Test your function:

[71]:
```
neg_rational(Rational(3, 5))
```

[71]: -3/5

5. Write a function `mul_rationals` that computes and returns the **product** of two rational number objects as a rational number object.

[72]:
```
def mul_rationals(lft, rgt):
    return Rational(lft.num * rgt.num, lft.den * rgt.den)
```

6. Test your function:

[73]:
```
mul_rationals(Rational(2, 3), Rational(3,4))
```

[73]: 1/2

7. Write a function `add_rationals` that computes and returns the **sum** of two rational number objects as a rational number object.

[74]:
```
def add_rationals(lft, rgt):
    return Rational(lft.num*rgt.den + rgt.num*lft.den, lft.den * rgt.den)
```

8. Test your function:

[75]:
```
add_rationals(Rational(1,2), Rational(2,3))
```

[75]:
```
7/6
```

9. Write a function `sub_rationals` that computes and returns the **difference** of two rational number objects as a rational number object, perhaps as the sum of one and the negative of the other.

[76]:
```
def sub_rationals(lft, rgt):
    return Rational(lft.num*rgt.den - rgt.num*lft.den, lft.den * rgt.den)
```

10. Test your function:

[77]:
```
sub_rationals(Rational(2, 3), Rational(1,2))
```

[77]:
```
1/6
```

11. Write a function `eq_rationals` that tests two rational number objects for equality, i.e., it returns `True` if the two represent the same value, and it returns `False` if not.

[78]:
```
def eq_rationals(lft, rgt):
    return lft.num * rgt.den == rgt.num * lft.den
```

12. Test your function:

[79]:
```
print(eq_rationals(Rational(1,2), Rational(2,4)))
print(eq_rationals(Rational(1,2), Rational(3,4)))
```

```
True
False
```

13. Now rewrite your class definition for `Rational` objects with **special methods**

- `__neg__` similar to `neg_rational`
- `__mul__` similar to `mul_rationals`
- `__add__` similar to `add_rationals`
- `__sub__` similar to `sub_rationals`
- `__eq__` similar to `eq_rationals`

[80]:
```
from math import gcd

class Rational:
    def __init__(self, num: int, den: int):
        if den == 0:
```

```
                raise ValueError("Denominator cannot be zero.")

        # Reduce the fraction to its lowest terms
        common_divisor = gcd(num, den)
        self.num = num // common_divisor
        self.den = den // common_divisor

        # If the denominator is negative, adjust the sign to keep the␣
↪denominator positive
        if self.den < 0:
                self.num = -self.num
                self.den = -self.den

    def __repr__(self):
        if self.den == 1:
            return str(self.num)
        else:
            return f"{self.num}/{self.den}"

    def __neg__(self):
        return Rational(-self.num, self.den)

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

    def __add__(self, other):
        return Rational(self.num*other.den + self.num*other.den, self.den *␣
↪other.den)

    def __sub__(self, other):
        return Rational(self.num*other.den - self.num*other.den, self.den *␣
↪other.den)

    def __eq__(self, other):
        return self.num * other.den == other.num * self.den

    def __gt__(self, other):
        return self.num * other.den > other.num * self.den

    def __ge__(self, other):
        return self.num * other.den >= other.num * self.den
```

14. Test your class:

[81]:
```
print(Rational(2, 3))
print(-Rational(3, 5))
print(Rational(2, 3) * Rational(3, 4))
```

```python
print(Rational(1, 2) + Rational(2, 3))
print(Rational(2, 3) - Rational(1, 2))
print(Rational(1, 2) == Rational(2, 4))
print(Rational(1, 2) == Rational(3, 4))
```

```
2/3
-3/5
1/2
1
0
True
False
```

15. What needs to be done so that comparisons like `python Rational(1, 2) < Rational(2, 3)` `Rational(1, 2) <= Rational(2, 3)` are evaluated in a meaningful way?

[82]:
```python
# def __gt__(self, other):
#         return self.num * other.den > other.num * self.den

# def __ge__(self, other):
#     return self.num * other.den >= other.num * self.den
```

[83]:
```python
Rational(1,2) < Rational(2,3)
```

[83]: True

[84]:
```python
Rational(1,2) <= Rational(2,3)
```

[84]: True