Special methods

• Rational

Rational numbers are ratios of two integers $p/q$, where $p$ is called the numerator and $q$ is called the denominator. The denominator $q$ is non-zero.

Operations on rationals include addition, multiplication, and comparisons: $=$, $<>$, $<$, $>$, $<=$, $==$
Attributes for Rational

Special Attributes in python (magic methods)

```python
== -__eq__

> __gt__

< __lt__

print(object) __str__
```

Created automatically with empty body

You can implement your corresponding code
Protecting against mistakes

Bad inputs can cause programs to crash

For Rational Class:

• What if num and denom are not integers?

• What if denom is 0?
Data Encapsulation

Data encapsulation (aka Data hiding) is the implementation details of a class are kept hidden from the user.

- The user should only perform a restricted set of operations on the “hidden” members of the class, through special methods.
- This is where getter and setter methods come in (will see these in a bit).
Getters, setters, and properties

• Basic idea: make accesses (read, write) to attributes go through special getter and setter methods
Example Property

```python
class Point:
    def __init__(self, x: Union[int, float], y: Union[int, float]) -> None:
        self.x, self.y = x, y
```

Change of requirement:

X will only have values between 0 and 1000
Example Property

class Point:

    def __int__(self, x: Union[int|float],
               y: Union[int|float]) -> None:
        self.x, self.y = x, y

    def set_x(self, x: float) -> None:
        assert 0 <= x <= 1000, "x should be between 0 and 1000"
        self.__x = x

    def get_x(self) -> Union[int|float]:
        return self.__x

Change of requirement:

    X will only have values between 0 and 1000
Example Property

```python
import Point

class Point:
    def __int__(self, x: Union[int, float], y: Union[int, float]) -> None:
        self.x, self.y = x, y

    def set_x(self, x: float) -> None:
        assert 0 <= x <= 1000, \"x should be \"between 0 and 1000\"
        self.__x = x

    def get_x(self) -> Union[int, float]:
        return self.__x

def __init__(self, x: Union[int, float], y: Union[int, float]) -> None:
    self.set_x(x)
    self.y = y

def set_x(self, x: float) -> None:
    assert 0 <= x <= 1000, \"x should be \"between 0 and 1000\"
    self.__x = x

def get_x(self) -> Union[int, float]:
    return self.__x
```

Have to change HUGEEEE number of client code lines
```python
class Point:

def __init__(self, x: Union[int, float], y: Union[int, float]) -> None:
    self.set_x(x)
    self.y = y

def set_x(self, x: float) -> None:
    assert 0 <= x <= 1000, "x should be between 0 and 1000"
    self.__x = x

def get_x(self) -> Union[int, float]:
    return self.__x

def _set_x(self, x: float) -> None:
    assert 0 <= x <= 1000, "x should be between 0 and 1000"
    self.x = x

def _get_x(self) -> Union[int, float]:
    return self.x

x = property(_get_x, _set_x)
```
managing attributes num and denom in Rational

● Protect from silly mistakes like
  ○ supplying non-integers for the numerator or denominator, or
  ○ zero for the denominator.

Onto PyCharm
Composition and Inheritance
Composition Example

```python
>>> class Math:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def add(self):
        return self.x + self.y
    def subtract(self):
        return self.x - self.y

>>> class Math2:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def multiply(self):
        return self.x * self.y
    def divide(self):
        return self.x / self.y
```

Need a Class Math3 that calculates the Power AND has the ability to add, subtract, multiply and divide
Composition Example

```python
>>> class Math:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def add(self):
        return self.x + self.y
    def subtract(self):
        return self.x - self.y

>>> class Math2:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def multiply(self):
        return self.x * self.y
    def divide(self):
        return self.x / self.y

>>> class Math3:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.m1 = Math(x, y)
        self.m2 = Math2(x, y)
    def power(self):
        return self.x ** self.y
    def add(self):
        return self.m1.add()
    def subtract(self):
        return self.m1.subtract()
    def multiply(self):
        return self.m2.multiply()
```
Composition: Shapes

Use existing types inside new user-defined types

We will use the Point class type inside Square

We will use the Turtle class type inside Square

Let’s see that in details
Example

Say we want to implement class Square:

*Squares have four vertices (corners), have a perimeter, an area, can move themselves by adding an offset to each corner, and can draw themselves.*
Squares have four vertices (corners), have a perimeter, an area, can move themselves by adding an offset to each corner, and can draw themselves.
Composition

We need:

- Ability to draw a Square => each Square needs a Turtle
- Vertices, aka Points => need Point to represent corners
  - We also get the Point’s “abilities”: to move by an offset, to calculate a distance, etc.
- Composition allows us to avoid writing code to duplicate the abilities of Turtle and Points

Implementation in pycharm