CS 106AX Programming Methodologies in JavaScript and Python Autumn 2026

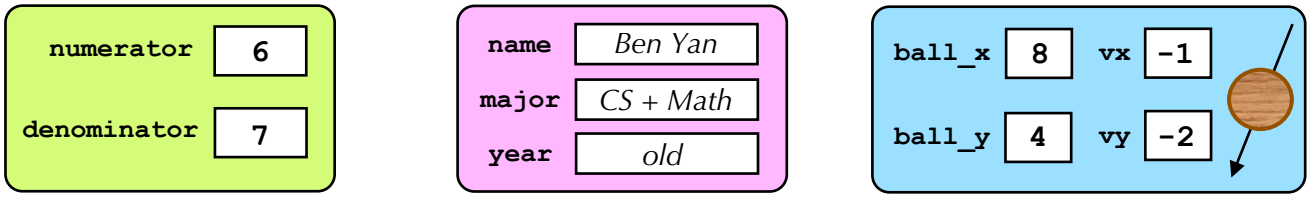
Lecture 18: Object-Oriented Programming & Abstraction

Written by Ben Yan, from materials by Jerry Cain

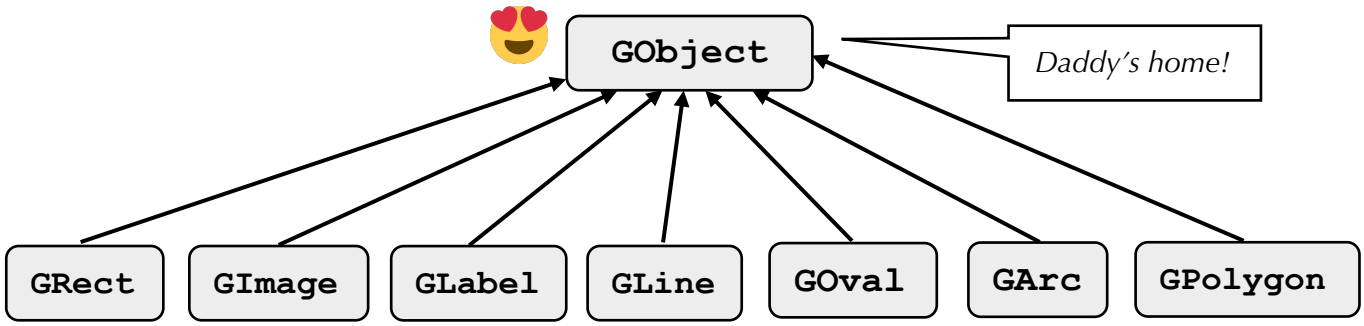
Object-Oriented Programming

Objected-oriented programming—typically abbreviated as OOP, like when you make a big mistake in your code or in life—is a programming paradigm concerned with the design of ***classes*** and ***objects***, as we covered last week. Invented in Norway in the 1960s, **object-orientation** is defined by two principles:

- **Encapsulation:** The principle that data values—and the methods that interact with them—should be integrated into a single unified, coherent object called an ***object***. A ***class*** templates an object.



- **Inheritance**: The notion that objects—and the classes that they represent—form hierarchies that enable new classes (***children***) to share or inherit behavior from other classes (***parent*** classes).



While it's helpful to understand why ***inheritance*** is important, **we'll emphasize *encapsulation* more in this course**. Today, we'll focus on using encapsulation to realize interesting data structures in Python ☺.

Rational Numbers Class

We define a class called `Rational` to **represent and encapsulate *rational numbers*, which are simply fractions, i.e. the quotient of two integers**. Rational numbers are useful in cases in which you need precise calculation with fractional values. Even when using floating-point numbers (`float`), decimals such as `0.1` are represented internally as ***approximations***. The rational number `1 / 10` is exact.

Here's a draft of a constructor for the `Rational` class, which initializes a new `Rational` object.

```
class Rational:
    def __init__(self, num, den = 1):
        """ Creates a new Rational object from num and den. """
        self._num = num
        self._den = den
```

Note that the user supplies the numerator **num** and denominator **den** as input. The **den** is set to **1** by default if not explicitly specified, e.g., **Fraction(8)** represents the value **8 / 1**, or the integer **8**.

```
quarter = Fraction(1, 4)
half = Fraction(50, 100)
numPlanets = Fraction(8)
pi = Rational(314, 100)      # an approximation, don't come after me math majors!
```

However, we may want a more involved and sophisticated constructor—to make life easier down the road—which checks that the **num** and **den** attributes of the rational number obey certain rules. That is:

- **The denominator must be greater than 0**. A denominator of **0** (division by zero) is verboten.
- **The value zero is always represent as 0 / 1**, with numerator **0** and denominator **1**.
- **The fraction is always reduced to the lowest terms**, e.g., **75 / 100** is the same as **3 / 4**.

A revised constructor appears below. Note that an error is raised if the denominator is supplied to be **0**.

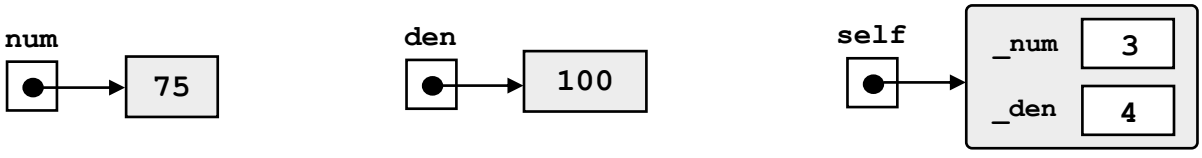
```
class Rational:
    def __init__(self, num, den = 1):
        """ Creates a new Rational object from num and den. """
        if den == 0:
            raise ValueError("Illegal denominator value")

        if num == 0:
            den = 1

        elif den < 0: # ensure that denominator is always positive, even if fraction is negative
            den = -den
            num = -num

        gcd = math.gcd(abs(num), den) # reduce fraction to lowest possible terms
        self._num = num // gcd
        self._den = den // gcd
```

To reduce the fraction, **we divide the num and den by their GCD (greatest common denominator),** which is computed using the **math.gcd** method from the Python **math** library. For instance, **gcd(75, 100) = 25**, which is why the reduced form has **num = 75 / 25 = 3** and **den = 100 / 25 = 4**.



To make printing easier for our next exercise, we'll also override the **__str__** method of this class.

```
    def __str__(self):
        """ Returns the string representation of this Rational object. """
        if self._den == 1:
            return str(self._num) # just print the integer value
        else:
            return str(self._num) + "/" + str(self._den)
```

The below excerpt demonstrates the creation of two fractional objects from this **Rational** class.

Python Interpreter

```
> a = Rational(1, 2)
> b = Rational(2)

> str(a)
1/2

> str(b)
2
```

Rational Class: Arithmetic Operators

Next, we'll implement methods to support **fraction addition, multiplication, subtraction, and division**. **These operations, when applied to two Rational objects, are specified using the *receiver syntax***, i.e.

```
r1.add(r2)
```

where the first operand, **r1**, is the receiver object; the other operand, **r2**, is passed in as an argument. Furthermore, if we recall the formula for the sum of two fractions, a / b and c / d, it should look like:

Addition

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

We write the following **add** method, which creates a new **Rational** object to encapsulate this sum.

```
class Rational:
      def add(self, r):
         return Rational(self._num * r._den + self._den * r._num,
                         self._den * r._den)
```

If we use floating-point arithmetic—without **Rational** objects—we get an irritatingly imprecise sum.

Python Interpreter

```
> 1/2 + 1/3 + 1/6
0.9999999999999
```

With **Rational** arithmetic, we achieve the exact computation, as accomplished by the function below.

```
def RationalSum():
      a = Rational(1, 2)
      b = Rational(1, 3)
      c = Rational(1, 6)

      total = a.add(b).add(c)
      print("1/2 + 1/3 + 1/6 = " + str(total))
```

RationalSum

```
1/2 + 1/3 + 1/6 = 1
```

Similarly, we can incorporate the formulas for fraction subtraction, multiplication, and division ☺.

Subtraction	$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$
Multiplication	$\frac{a}{b} * \frac{c}{d} = \frac{ac}{bd}$
Division	$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$

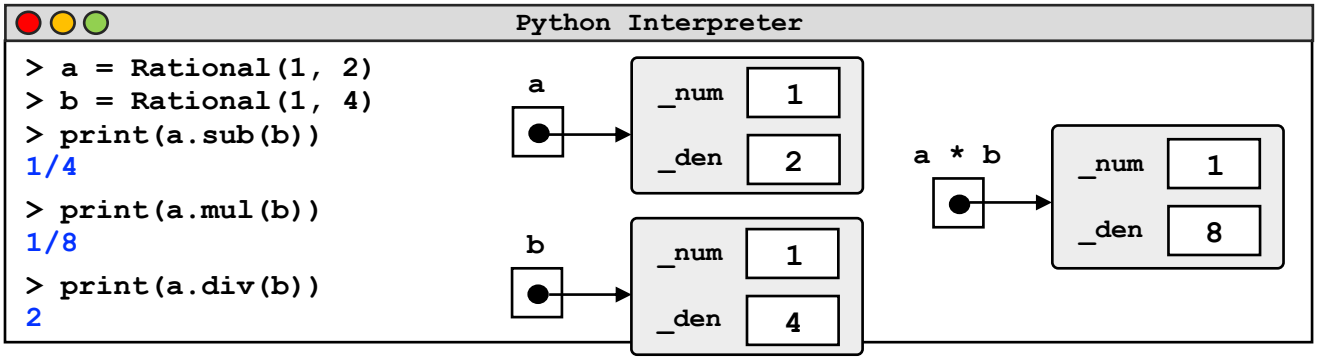
Each of these arithmetic methods synthesize a new **`Rational`** object to encapsulate the object.

```
class Rational:
    def sub(self, r):
        """ Creates a new Rational by subtracting r from self. """
        return Rational(self._num * r._den - self._den * r._num,
                        self._den * r._den)

    def mul(self, r):
        """ Creates a new Rational by multiplying self by r. """
        return Rational(self._num * r._num, self._den * r._den)

    def div(self, r):
        """ Creates a new Rational by dividing self by r. """
        return Rational(self._num * r._den, self._den * r._num)
```

The below excerpt uses each of the three **`Rational`** methods defined above. Note that **`print`** displays the string (**`str`**) form of the new **`Rational`** object, as created and returned by each method.



Operator Overloading

The receiver syntax used in the **`RationalSum`** program, while functionally permissible, makes the program an eyesore to read, especially to those unfamiliar with the jargon and technicalities of OOP.

The program would be so much clearer and legible if you could replace **`a.add(b).add(c)`** with the more familiar **`a + b + c`**, right? ☺ Recall that **`a`**, **`b`**, **`c`** are all references to unique **`Rational`** objects.

Unlike Java and JavaScript, Python enthrallingly allows you to do just that. Each standard operator (e.g. **`+`**, **`*`**, **`<`**, **`==`**) is associated with a special method name (e.g., **`__add__`**, **`__truediv__`**, **`__lt__`**, or **`__eq__`**), which specifies how that operator should be implemented for the defining Python class.

This procedure has the cool name ***operator overloading***, and it's super fun! As a relevant example, you can redefine addition, e.g., **`r1 + r2`**, for the **`Rational`** class by implementing the **`__add__`** method.

```
class Rational:
      def __add__(self, rhs):
            self.add(rhs) # calls the add method we just implemented!
```

With this method defined, we can now directly use the **+** operator to add **`Rational`** objects together!

Python Interpreter

```
> a = Rational(1, 2)
> b = Rational(1, 4)
> print(a + b)   # instead of having to write a.add(b) !
3/4
```

Although this simple implementation works, it's a better user experience for clients if one can mix types in an expression, e.g., **`Rational(1, 2) + 3`**, which should evaluate to fraction **`Rational(7, 2)`**. Here's a revised version that **converts the right-hand side from `int` type to `Rational` type if needed**.

```
      def __add__(self, rhs):
            """ Invoked when lhs (left hand side) is Rational object."""
            if type(rhs) is int:
                  return self.add(Rational(rhs))

            elif type(rhs) is Rational:
                  return self.add(rhs)

            else:
                  return NotImplemented
```

If the right-hand side is neither **`int`** nor **`Rational`** (e.g., a string type), we return the built-in constant **`NotImplemented`**, to indicate this **+** operation is not supported for the provided pair of types ☹.

Do note that **`__add__`** can define mixed-type addition **only if the `Rational` operand appears to the left of the + operator**. If the **`Rational`** operand can appear on the right, e.g., **`2 + Rational(3,4)`**, or **`3 + Rational(14, 100)`**, we also need to define the right-add method **`__radd__`** as well.

Here's an implementation of **`__radd__`**, so that we've now overloaded the **+** in both directions ☺.

```
      def __radd__(self, lhs):
            """ Invoked when rhs (right hand side) is Rational, but lhs (left hand side) is not."""
            if type(lhs) is int:
                  return Rational(lhs).add(self)

            elif type(lhs) is Rational:
                  return lhs.add(self)

            else:
                  return NotImplemented
```

You can now accomplish all the following tricks in Python using the **`Rational`** class, ta-da!

Python Interpreter

```
> pumpkin = Rational(14, 100)
> pie = 3
> print(pumpkin + pie)       # adds Rational object to integer
157/50

> print(pie + pumpkin)       # adds integer to Rational object
157/50

> print(pumpkin + pumpkin)   # adds Rational object to Rational object
7/25

> print(pie + pie)      # adds integer to integer
6
```



Note that **`pumpkin + pie`** enlists **`pumpkin`** (the **`Rational`** object on the left side) to use **`__add__`**, whereas **`pie + pumpkin`** enlists **`pumpkin`** (now on the right) to use **`__radd__`**. In both cases, we bake an equivalent object from **`pie = 3`**, i.e. **`Rational(3, 1)`**, before adding the fractions together.

Operator Methods in Python

If you're curious just how many operators you can overload in Python, here's a whole galaxy of them! ☺

Left Hand Side	Right Hand Side	Description
`__add__`	**`__radd__`**	**Addition**: Redefines the **+** operator
`__sub__`	**`__rsub__`**	**Subtraction**: Redefines the **–** operator
`__mul__`	**`__rmul__`**	**Multiplication**: Redefines the ***** operator
`__truediv__`	**`__rtruediv__`**	**Division**: Redefines the **/** operator
`__floordiv__`	**`__rfloordiv__`**	**Integer Division**: Redefines the **//** operator
`__mod__`	**`__rmod__`**	**Remainder**: Redefines the **%** operator
`__pow__`	**`__rpow__`**	**Exponentiation**: Redefines the ****** operator
`__neg__`	*(not applicable)*	**Negation**: Redefines the unary **–** operator
`__eq__`	*(symmetric)*	**Equality**: Redefines the **==** operator
`__ne__`	*(symmetric)*	**Non-Equality**: Redefines the **!=** operator
`__lt__`	*(inferred from **>**)*	**Less Than**: Redefines the **<** operator
`__gt__`	*(inferred from **<**)*	**Greater Than**: Redefines the ****** operator
`__le__`	*(inferred from **>=**)*	**Less Than or Equal to**: Redefines the **<=**
`__ge__`	*(inferred from **<=**)*	**Greater Than or Equal to**: Redefines the **>=**

Operator Overloading for `Rational`, Part Deux

Here's an overhaul of three other operators for the **`Rational`** class, namely ***subtraction***, ***multiplication***, and true, decimal ***division***. You won't be surprised to learn that each are implemented very similarly to the **`__add__`** method, which overloads ***addition*** **when a `Rational` object is on the left-hand side**.

Note that methods **`sub`**, **`mul`**, **`div`** are already written to back operations with both **`Rational`** objects. We're extending them to support (1) mixed types, and (2) traditional operator syntax (e.g., **`+`**, **`-`**, **`*`**, **`%`**).

```
class Rational:
    """ Previously implemented methods of this class omitted for conciseness! """

    def __sub__(self, rhs):
        if type(rhs) is int:
            return self.sub(Rational(rhs))
        elif type(rhs) is Rational:
            return self.sub(rhs)
        else:
            return NotImplemented

    def __mul__(self, rhs):
        if type(rhs) is int:
            return self.mul(Rational(rhs))
        elif type(rhs) is Rational:
            return self.mul(rhs)
        else:
            return NotImplemented

    def __truediv__(self, rhs):
        if type(rhs) is int:
            return self.div(Rational(rhs))
        elif type(rhs) is Rational:
            return self.div(rhs)
        else:
            return NotImplemented
```

```
> Rational(7,2) - 3
Rational(1, 2)

> Rational(1, 2) -
Rational(1, 3)
Rational(1, 6)
```

```
> Rational(1,2) * 3
Rational(3, 2)

> Rational(1, 2) *
Rational(2, 3)
Rational(1, 3)
```

```
> Rational(6, 7) / 6
Rational(1, 7)

> Rational(3, 4) /
Rational(3, 2)
Rational(1, 2)
```

The operators for when the **`Rational`** object appears on the ***right-hand side*** are implemented similarly.

```
    def __rsub__(self, lhs):
        if type(lhs) is int:
            return Rational(lhs).sub(self)
        elif type(lhs) is Rational:
            return lhs.sub(self)
        else:
            return NotImplemented
```

```
> 3 - Rational(5,2)
Rational(1, 2)

> "hi" - Rational(1,2)
NotImplemented
```

```
def __rmul__(self, lhs):
    if type(lhs) is int:
        return Rational(lhs).mul(self)

    elif type(lhs) is Rational:
        return lhs.mul(self)

    else:
        return NotImplemented

def __rtruediv__(self, lhs):
    if type(lhs) is int:
        return Rational(lhs).div(self)

    elif type(lhs) is Rational:
        return lhs.div(self)

    else:
        return NotImplemented
```

```
> 6 * Rational(1,7)
Rational(6, 7)

> "hi" * Rational(1,7)
NotImplemented
```

```
> 3 / Rational(1,2)
Rational(6, 1)

> "hi" / Rational(1,2)
NotImplemented
```

We can also implement the ***equals*** operator, noting two fractions, **a/b** and **c/d**, are identical in value if

$$\frac{a}{b} = \frac{c}{d} \leftrightarrow \quad \boxed{a * d = b * c}$$

We overload **__eq__** as follows, noting that other relational operators can be coded near identically.

```
def __eq__(self, rhs):
    if type(rhs) is int:
        return self.__eq__(Rational(rhs))

    elif type(rhs) is Rational:
        return (self._num * rhs._den) == (self._den * rhs._num)

    else:
        return False
```

For example, the expression **Rational(1, 2) == Rational(3, 6)** would evaluate to **True**.

Fraction Sum Program

One application of the **Rational** class is to **calculate the sum of a series of fractions**, while avoiding the accumulation of errors that would result from repeatedly adding floating-point decimals together ☺.

For example, the below function computes the sum: **1 / 1 + 1 / 2 + 1 / 3 + ... + 1 / n**.

```
def HarmonicSeries(n):
    sum = 0
    for denom in range(1, n + 1):
        term = Rational(1, denom)
        sum += term
    return sum
```

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} = \frac{137}{60}$$

As a sample function call, **HarmonicSeries(5)** outputs the fraction object **Rational(137, 60)**.