Do Now:

- Before we start, we need to make a new directory to hold our programs for the upcoming topics:
  - Open a command prompt
  - `cd PreAPCS`
  - `cd PreAPCS-classcode`
  - `md Classes`

Python Types

- So far we have only used types that come preloaded with Python.
  - str (string)
  - int (integer)
  - float (floating point number)
  - bool (boolean, True/False)
- In this lesson, we will create our own type.

Rational Class

- Since Python comes with support for integers and floating point numbers, let's add support for rational numbers, i.e. fractions.
- First, let's create a new file called `rational.py` and save it in the folder you created in CS1-classcode called "Classes"
- We are going to create a `Rational` class (another word for type).
  - In Python, files (modules) are always named using lower_case, but classes are named using CapitalCamelCase.

Rational Class

- Enter the following code in rational.py

```
class Rational:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator
```

- Don’t worry about understanding every bit of this code.
- Over the next few lessons we’ll take time to understand each portion.

Class Basics

- To define a class, simply say `class` followed by the name you want the class to have.
- In order for us to make different members of the `Rational` class, we need to define a special function called `__init__`.
- The `__init__` function is automatically called when you create a `Rational` type.
- The first parameter for any `__init__` (short for initializer) function is `self`.
- The self parameter designates that our parameters are for this `Rational` right here – called this *instance*.
- An instance of a class is called an **object**.

Test Our `Rational` Class

- In order to test our Rational class, make a main function (below the class definition):

```
def main():
    x = Rational(3, 4)
    print(x)
if __name__ == "__main__":
    main()
```

Test Our `Rational` Class

- Now run the rational module and you should see something a little weird.
- It will say something like:

```
<__main__.Rational object at 0x03BB9950>
```

- The underlined number is a hexadecimal number that is uniquely assigned to each object you make with a class.
  - It's not very useful to us here, but this id number can be *very* useful in some situation.

`__str__` method

- The hexadecimal description of the object is te default string that is printed when you print an object.
- To fix this, we supply a `__str__` method inside the `Rational` class.
- Methods are like functions, but they belong to objects.

`__str__` method

- Update the Rational class then rerun the module:

```
class Rational:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator
    def __str__(self):
        return str(self.numerator) + "/"
           + str(self.denominstor)
```

Note: The return statement above should all be on one line.

Output: `3/4`

Important Note

- Any method that you want a `Rational` object to have must have `self` as the first parameter.

Improving `Rational`

- What if we put in the fraction 6/8?
- It should reduce to ¾.
- To do this, we need to divide each numerator and denominator by the greatest common divisor of the numerator and denominator.
- To do this, we will import the `math` module and use the `gcd` utility function.

Improving `Rational`

- Update the __init__ method and main function as shown:

```
def __init__(self, numerator, denominator):
    gcd = math.gcd(numerator, denominator)
    self.numerator = numerator // gcd
    self.denominator = numerator // gcd
def main():
    x = Rational(6,8)
    print(x)
    print(Rational(-6, 8))
    print(Rational(6, -8))
    print(Rational(-6, -8))
```

Definition: Method

- A function that belongs to an object is called a **method**.
- We will now add our first unique method.
- __str__ is a method too, but we were actually writing over a previously written method.

to_float

- Define the following method under our definition of __str__.
- ```
  def to_float(self):
      return self.numerator / self.denominator
  ```
- Notice that we must use the `self` keyword to ensure that we divide *this* `Rational` object's `numerator` and `denominator`.
  ```

Calling a method

- Recall that to call a function, we use `module_name.function_name()`.
- To call a method, we use `object_name.method_name()`.
- Update the main method:

```
def main():
    x = Rational(6,8)
    print(x)
    print(x.to_float())
```

Expected Output:

```
3/4
.75
```

More Methods

- Objects interact with each other through methods.
- For example, if we want to multiply Rational numbers, we need to make a multiply method.
- Multiplying rational numbers is pretty straightforward:
  - Multiply the numerators by each other and the denominators by each other.

Rational.multiply(other)

```
def multiply(self, other):
    return Rational(self.numerator * other.numerator,
                    self.denominator * other.denominator)
```

Note: the entire return statement should be on one line.

```
def main():
    x = Rational(6,8)
    y = Rational(1,5)
    product = x.multiply(y)
    print(product)
```

Expected output: `3/20`

Rational.add(other)

- Adding fractions is a little more complicated, but a neat trick is to use the criss-cross method:

$$\frac{a}{b} + \frac{c}{d} = \frac{a * d + c * b}{b * d}$$

- This is not usually reduced, but our Rational initializer reduces the result for us.
- Try to implement the add method on your own.

Rational.add(other)

```
def add(self, other):
    return Rational(self.numerator *
        other.denominator + other.numerator *
        self.denominator, self.denominator *
        other.denominator)
```

- Test with the following code in the main function:

```
x = Rational(6,8)
z = Rational(2,3)
sum = x.add(z)
print(sum)
```

- Expected output: `17/12`

Summary

- You added a class to Python to handle rational numbers.
- You overrided the __str__ method to make printing rational numbers more useful
- You added custom made instance methods to allow adding and multiplying rational numbers.

Intro to Classes - Exercises

1) Dividing a fraction by another fraction is almost like multiplying. To do it, you flip the numerator and denominator of the second fraction and then multiply the two fractions.

Add a divide method to the Rational class in the rational module. Flip the other fraction, then use the multiply method.

Important: part of this exercise is using the multiply method within the divide method.

Intro to Classes - Exercises

2) Substracting a fraction is also just a small adjustment. To subtract this fraction minus some other fraction, simply multiply the numerator by negative one and then call the add method.

   Implement the subtract method.

Intro to Classes - Exercises

3) Implement the following method:

```
Rational.compare_to(other)

  returns -1 if this Rational is less than other Rational.

  returns 0 if this Rational is equal to other Rational.

  returns 1 if this Rational is greater than other Rational.
```

Remember, while the method description does not show the self parameter, you must put it there!

Intro to Classes - Exercises

Answer the following two questions in comments at the top of your rational.py file.

4) What is the difference between a method and a function?

5) How is the manner in which you call a function and a method different?