CSC148-Section:L0301

Week#2-Monday

Instructed by

AbdulAziz Al-Helali

a.alhelali@mail.utoronto.ca

Office hours: Wednesday 11-1, BA2230.

Slides adapted from Professor Danny Heap and Jacqueline Smith slides winter17

Computer Science
UNIVERSITY OF TORONTO

Announcements

- Again Make sure your teach.cs account works
- If you are **unable to complete** a lab/tutorial **for any reason**, please *contact us as soon as possible* so that we can come to alternative arrangements.
  - Fill the form Special consideration
    - http://www.cdf.utoronto.ca/~csc148h/winter/specialConsideration.txt
  - Submit supporting documentation, together with this form, as an email: csc14818s@cs.toronto.edu

- Assignment # 1 is posted due 30th Jan.
- Lab #2 is posted

Outline

- Intro: Inheritance vs Composition
- Finalize Rational class
  - Managing attributes
  - Testing methods individually
  - float and __eq__ __lt___
  - special (aka magic)methods

Intro: Inheritance vs Composition

Composition

- Making use of other data types or objects of other classes
- E.g.
  - Point class uses objects of type float (x, y)
  - Rational class uses objects type int (num, denum)
  - Later we will build Square class using objects of type Point and Turtle

Inheritance

- A subclass inherits all attributes and methods (behavior) from superclass. Why?
  - to reuse code of existing class
- A subclass can extend, overload attributes and methods for a supercalss
- Subclass can be called child class
- Supper class can be called parent class

Example:

```
class Father:
    x: int = 10
    y: int = 20

    def m1(self) -> None:
        print('Father m1')

    def m2(self) -> None:
        print('Father m2')
```

Super class
Or
Parent class

```
class Son(Father):
    z: int = 30

    def m1(self) -> None:
        print('Son m1')
```

subclass
Or
child class

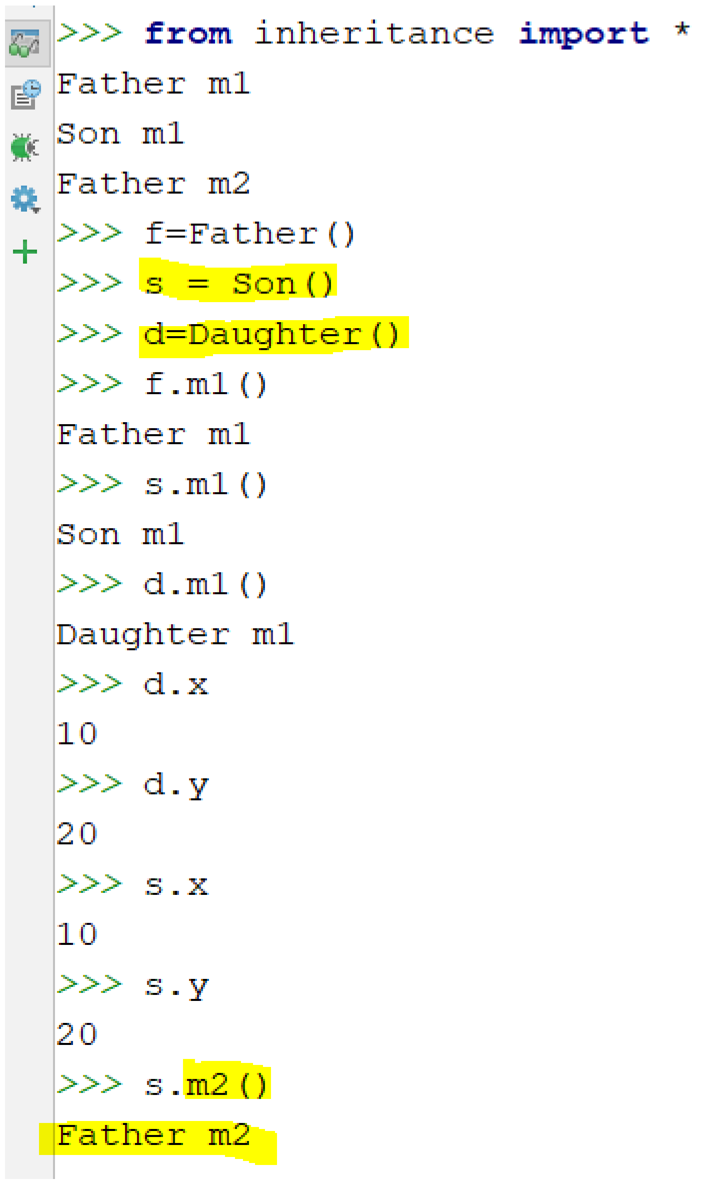
```
class Daughter(Father):
    z: int = 30

    def m1(self) -> None:
        print('Daughter m1')
```

subclass
Or
child class

```
f=Father()
s = Son()
f.m1()
s.m1()
s.m2()
```

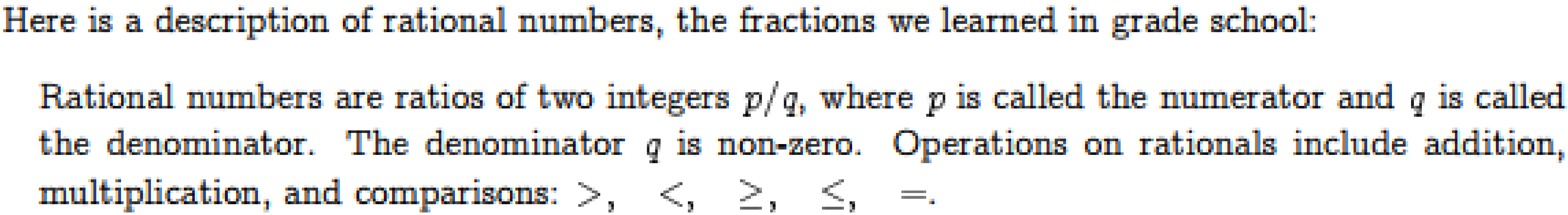
Son subclass is extending the attributes of Father

Son subclass is overriding m1() of Father



```
>>> from inheritance import *
Father m1
Son m1
Father m2
>>> f=Father()
>>> s = Son()
>>> d=Daughter()
>>> f.m1()
Father m1
>>> s.m1()
Son m1
>>> d.m1()
Daughter m1
>>> d.x
10
>>> d.y
20
>>> s.x
10
>>> s.y
20
>>> s.m2()
Father m2
```

Exercise: build Rational class



Here is a description of rational numbers, the fractions we learned in grade school:

> 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: $>$, $<$, $\geq$, $\leq$, $=$.

http://www.teach.cs.toronto.edu/~csc148h/winter/lecturedata/Danny/W1/rational_exercise.pdf

Exercise: build Rational class

- Rational
  - num: int
  - denum: int

__init__	called when >>> r1=Rational(2,3)
__eq__	called when >>> r1 == r2 also when we have list L=[r1,r2] >>> r1 in L
__str__	called when >>>print(r1)
__lt__	called when >>> r1 < r2

Special (aka magic)methods

- Python recognizes the names of special methods such as:

  __init__ , __eq__ , __ add__ , and __mul __ and

They have short-cuts (**aliases**) for them. E.g.:

__eq__ aliased with == and in

__ add__ aliased with +

__mul __ aliased with *

__lt __ aliased with < and allows sort() and sorted() methods to work

suppose you create a list of Rational, and then want to sort it, or check to see whether an equivalent element is in __lt __ lt and friends...

- For a full list of them check Python documentation:
  - https://docs.python.org/3/reference/datamodel.html#special-method-name

__eq__ aliased with in

```
>>> from rational_api_imp import *
>>> L=[Rational(1,2), Rational(5,3)]
>>> r=Rational(1,2)
>>> r in L
True
>>> L=[Rational(2,4),Rational(4,4)]
>>> r in L
True
```

float and __eq__

try this example with the __eq__ implementation that uses division.

```
>>> r4 = Rational(1, 3)
>>> r5 = Rational(1000000000000000001, 3000000000000000000)
>>> r4 == r5
```

or just try to do the division in console like:

```
>>>1/3 == 1000000000000000001/ 3000000000000000000
```

```
 True
```

then Use the console to see what is the float value for 1/3

```
>>>1/3
```

```
0.3333333333333333
```

compare that with value that you get for 1000000000000000001/ 3000000000000000000

```
>>>1000000000000000001/ 3000000000000000000
```

```
0.3333333333333333
```

due to rounding, we end up having the same result for two different rational numbers.

__lt __

n1*d2 < n2*d1 will not work if d2 is negative

e.g. 1/-4< 1/2 should be **True**

Using the above implementation:

1*2 < 1*-4 will return **False**

As a solution we can change < to > when we have negative numbers using **Funcational-IF in Python**

<exp1> If <condition> else <exp2>

If condition is True exp1 is evaluated if condition is False exp2 is evaluated

Solution will be:

n1*d2 < n2*d1 **if** d1*d2>0 **else** n1*d2 > n2*d1

__lt __

```
def __lt__(self, other: Any) -> bool:
    """
    Return whether Rational self is less than other.
    >>> Rational(3, 5).__lt__(Rational(4, 7))
    False
    >>> Rational(3, 5).__lt__(Rational(5, 7))
    True
    >>> Rational(1, 2).__lt__(Rational(3, 6))
    False
    >>> Rational(1, 2).__lt__(Rational(1, -4))
    False
    >>> Rational(1, -4).__lt__(Rational(1, 2))
    True
    """
    # return self.num * other.denum < other.num * self.denum
    # return self.num / self.denum < other.num / other.denum # wrong do not use it
    return (self.num * other.denum < other.num * self.denum
            if self.denum * other.denum > 0
            else self.num * other.denum > other.num * self.denum )
```

__lt __

Another way is to use this:

$$\boldsymbol{\frac{n1*d2}{d1*d2} < \frac{n2*d1}{d1*d2}}$$

Will float approximation generate errors?

Referring to Rational class

- In method headers use quotations
  - E.g.

```
def __mul__(self, other:  'Rational') -> 'Rational':
```

- In method or docstrings boy do not

```
"""
Return the product of Rational self and Rational other.
>>> print(Rational(3, 5).__mul__(Rational(4, 7)))
12 / 35
"""
return Rational(self.num * other.num, self.denum * other.denum)
```

__repr__

- Called when in console

```
>>> r1=Rational(1,2)
>>>r2=Rational(2,6)
>>> r1*r2
<rational_api_imp.Rational object at 0x000001E345107630>
```

To get something readable implement __repr__

  - Lazy way by just return self.__str__()
  - Return a string looks like
    - <rational_api_imp.Rational 2/12 >

__repr__ (Lazy way) ☺

```
def __repr__(self) -> str:
    """
    Return a string representation of Rational self.
    >>> r1=Rational(1,2)
    >>> r2=Rational(2,6)
    >>> r1*r2
    2 / 12
    """
    return self.__str__()
```

Managing attributes in Python

What if user put denum=0

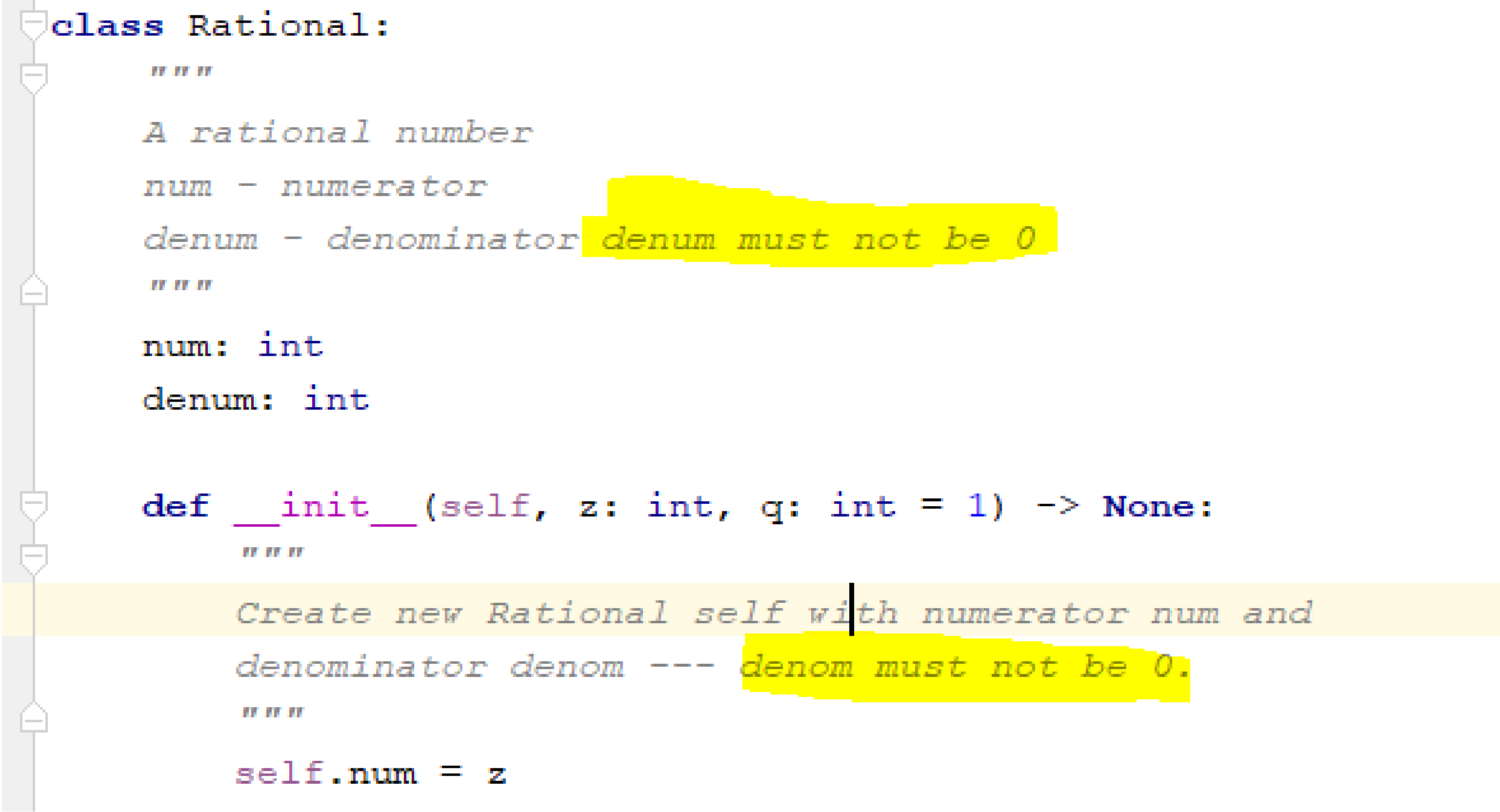
- Python provides 3 Ways to handle wrong input
  1. docstrings -- will use
  2. assert
  3. properties

1-Docstrings

- Warn the user about wrong input values in
- Class docstrings
- __init__ docstrings

- If a user put

0 in denum it is his fault



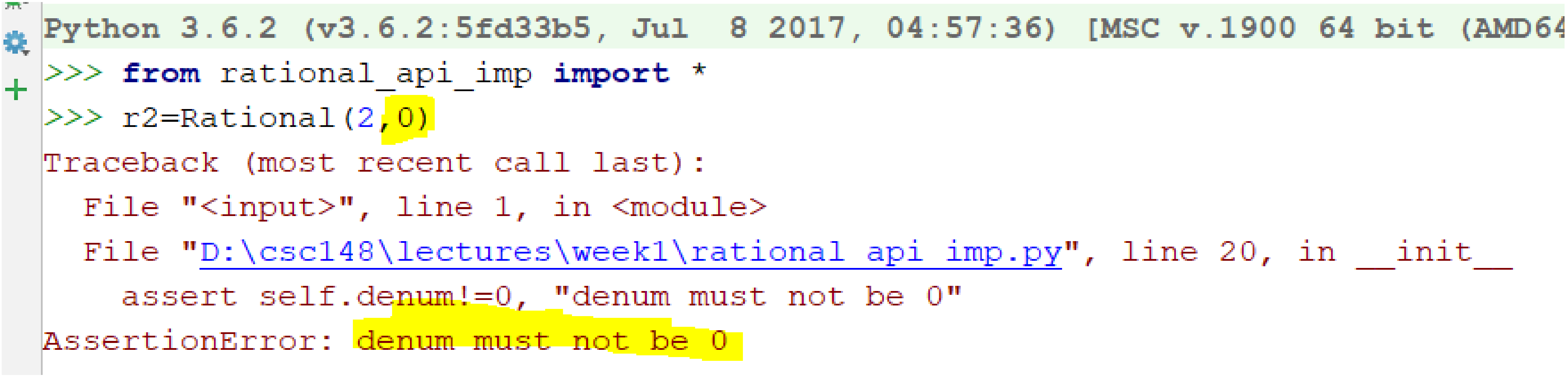
```
class Rational:
    """
    A rational number
    num - numerator
    denum - denominator denum must not be 0
    """
    num: int
    denum: int

    def __init__(self, z: int, q: int = 1) -> None:
        """
        Create new Rational self with numerator num and
        denominator denom --- denom must not be 0.
        """
        self.num = z
```

2- Assert

```
def __init__(self, num: int, denum: int) -> None:
    """
    Create new Rational self with numerator num and
    denominator denom --- denom must not be 0.
    """
    self.num = num
    self.denum = denum
    assert self.denum!=0, "denum must not be 0"
```

2- Assert

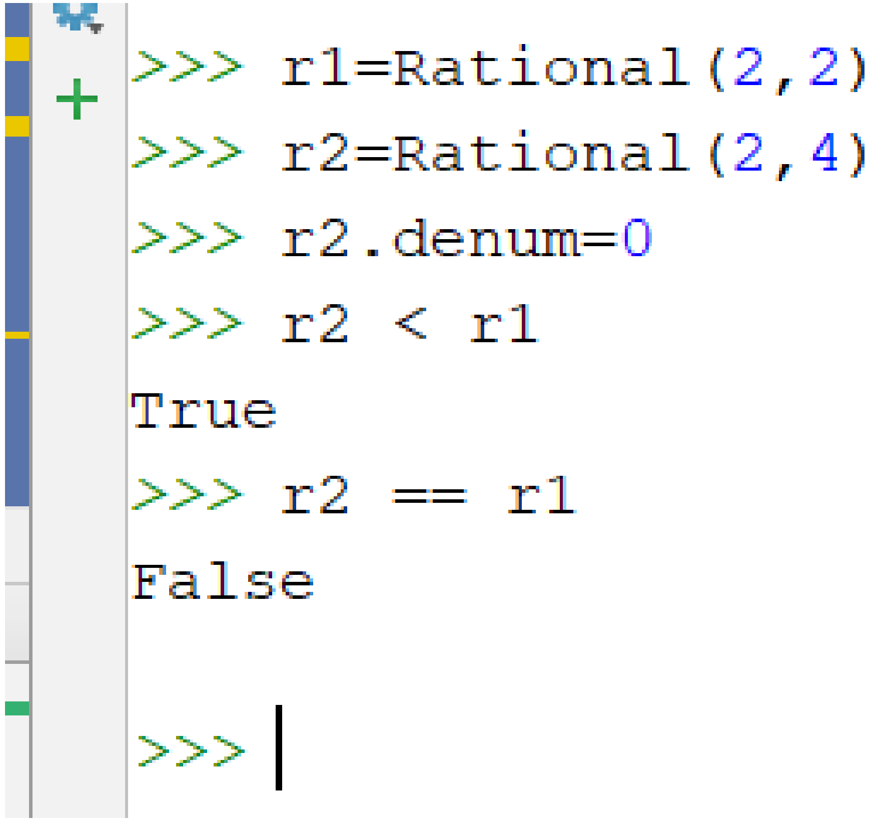
- If you try to create rational with denum=0 you get error



```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64
>>> from rational_api_imp import *
>>> r2=Rational(2,0)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "D:\csc148\lectures\week1\rational_api_imp.py", line 20, in __init__
    assert self.denum!=0, "denum must not be 0"
AssertionError: denum must not be 0
```

2- Assert

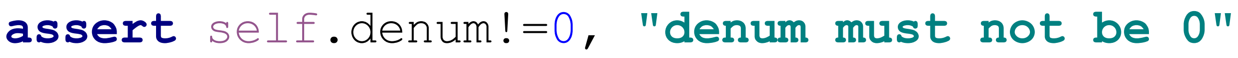
What happens if do the following



```
>>> r1=Rational(2,2)
>>> r2=Rational(2,4)
>>> r2.denum=0
>>> r2 < r1
True
>>> r2 == r1
False

>>> |
```

2- Assert

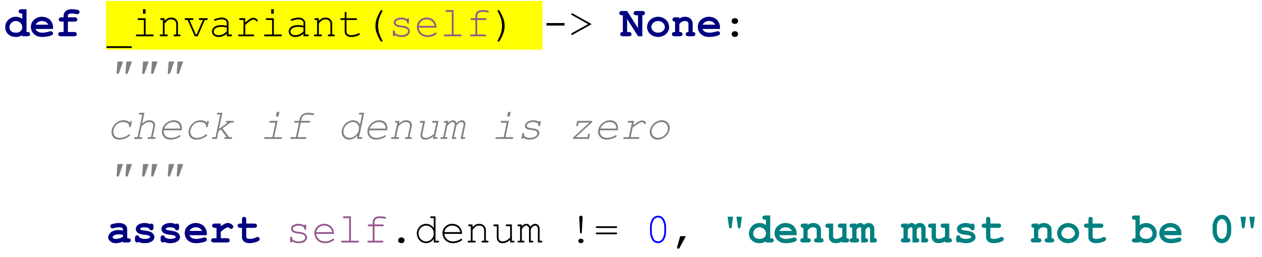
- We need to put assert in every public method that uses denum



```
assert self.denum!=0, "denum must not be 0"
```

- OR to reuse your code:

1- Create a new method that uses assert as follows:



```
def _invariant(self) -> None:
    """
    check if denum is zero
    """
    assert self.denum != 0, "denum must not be 0"
```

2- call this method in

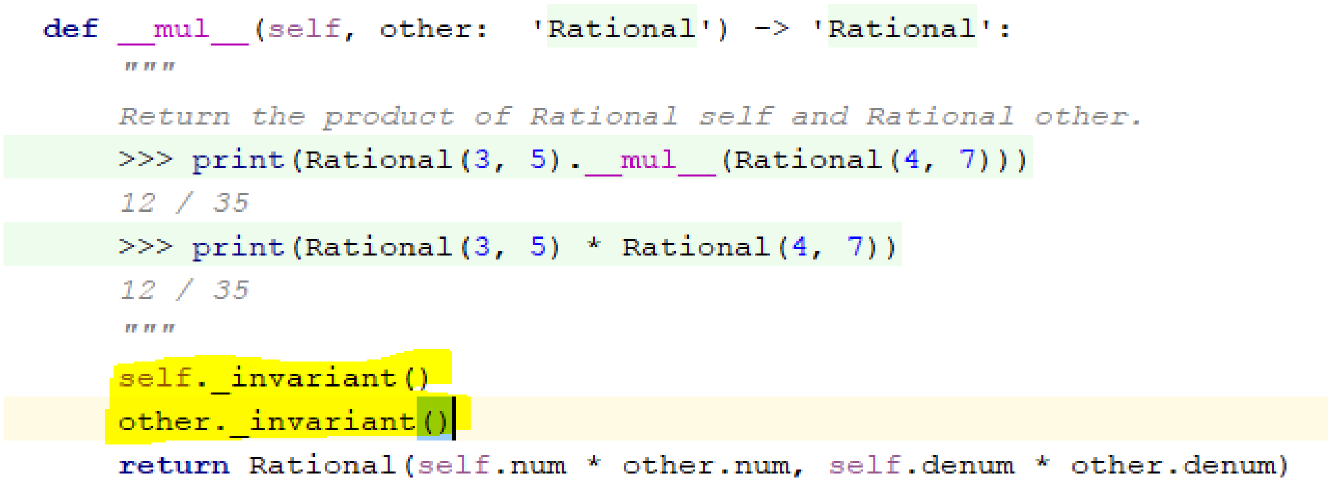
all other public methods

(__init__

__add__

__mult__

etc)



```
def __mul__(self, other: 'Rational') -> 'Rational':
    """
    Return the product of Rational self and Rational other.
    >>> print(Rational(3, 5).__mul__(Rational(4, 7)))
    12 / 35
    >>> print(Rational(3, 5) * Rational(4, 7))
    12 / 35
    """
    self._invariant()
    other._invariant()
    return Rational(self.num * other.num, self.denum * other.denum)
```

3-Property

- Python make public attributes **directly accessible** (no accessors, aka getters/setters) through class name or object name. e.g:
    - Rational.num
    - r.num
- Other programming languages **like Java** have the concept of public an private attributes where private attributes can not be accessed by calling the attribute name they have to be called through special methods called getters and setters.
    - E.g.
    - In Java if you define attribute as private num
    - Then you can not use r.num to access it you must use r.get_num() where a get_num() is a method that returns num
- You can indicated that an attribute is private by placing under score (_) in front of its name like: _num
    - This will not prevent users from accessing it by saying **r._num** but tells them that **they should not** because Python does not hide attributes and expects users to use them properly.
- **Python solution** is to use **property** to delegate the management of public attributes behind the scenes
    - **90% (about) you will not use it**

3-Property

- Suppose that client code written by billions of developers uses Rational, but some of them complain that that class doesn’t protect them from silly mistakes like supplying non-integers for the numerator or denominator, or even zero for the denominator. . .

- After you have **already shipped** class Rational, you can write methods _get_num, _set _num, _ get _denom, and _set_denom, and then use **property** to have Python use these functions whenever it sees num or denom

Managing attributes in Python

- Python provides 3 Ways to handle wrong input
  1. Docstrings -- you have to use it in this course
  2. Assert -- you have to use it in this course
  3. Properties – optional
     - Read about it in the course notes
       - http://www.teach.cs.toronto.edu/~heap/148/W16/148notes.pdf