Lecture 9

- Introduction to Classes in Python
  (6.4 Fraction Class)

Definitions

Class is a *data type* (compound type) that has:
- attributes (variables, fields, properties), and
- behaviors (functions, methods)

First, we need to *define* a class. Class definitions can appear anywhere in a program, but they are usually near the beginning.

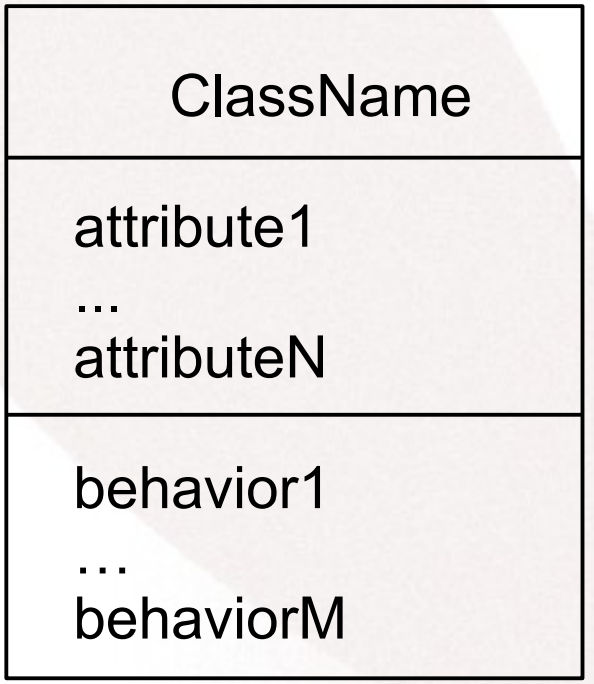
Definitions

Class is a *data type* (compound type) that has:
- attributes (variables, fields, properties), and
- behaviors (functions, methods)

First, we need to *define* a class. Class definitions can appear anywhere in a program, but they are usually near the beginning.

Syntax:

```
class <className>:
    <attribute1>
    ...
    <attributeN>

    def <behavior1>:
        <body1>
    ...

    def <behaviorM>:
        <bodyM>
```



class diagram

Definitions

Class is a *data type* (compound type) that has:
- attributes (variables, fields, properties), and
- behaviors (functions, methods)

First, we need to *define* a class. Class definitions can appear anywhere in a program, but they are usually near the beginning.

Syntax:

```
class <className>:
    <attribute1>
    ...
    <attributN>

    def <behavior1>:
        <body1>
    ...

    def <behaviorM>:
        <bodyM>
```

Then, we can use this *class* (new *data type*) by creating members of this type, that are called *instances* of the type or objects.

Creation of a new instance is called *instantiation*.

Fractions

Let's learn *how to define classes* on Fractions: we'll define a Fraction class and discuss the aspects of class definition during this process.

Fraction is an expression that indicates a quotient of two numbers $\frac{a}{b}$, where *a, b* are integers, and b $\neq$ 0

a is called *numerator*, and
b is called *denominator*

We can add, multiply, subtract, divide, take exponents, extract square roots, simplify fractions.

Fractions: Attributes

```
class Fraction:
    n = 0
    d = 1
```

or

```
class Fraction:
    n, d = 0, 1
```

Class Fraction has two attributes: n for *numerator*, and d for *denominator*.

Initially, their values are 0 and 1 (correspondingly), i.e. $\frac{n}{d}$

Let's see how can we use it:

```
def main():
    f1 = Fraction()
    f2 = Fraction()

    f1.n = 3
    f1.d = 5

    f2.n = 5
    f2.d = 12
```

Fractions: Attributes

```
class Fraction:
    n = 0
    d = 1
```

or

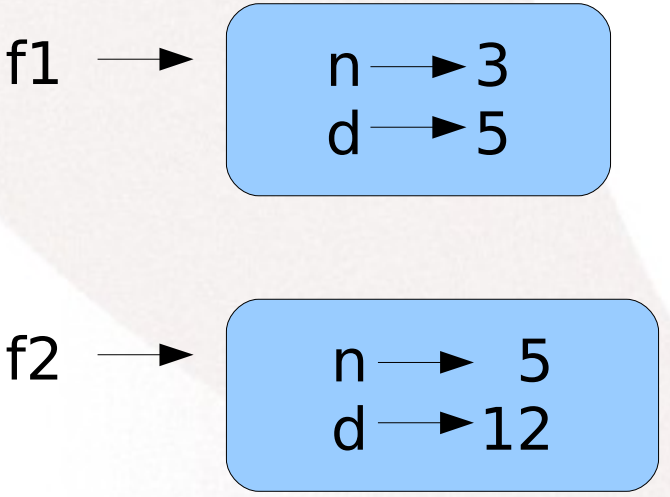
```
class Fraction:
    n, d = 0, 1
```

Class Fraction has two attributes: n for *numerator*, and d for *denominator*.

Initially, their values are 0 and 1 (correspondingly), i.e. $\frac{0}{1}$

Let's see how can we use it:

```
def main():
    f1 = Fraction()
    f2 = Fraction()

    f1.n = 3
    f1.d = 5

    f2.n = 5
    f2.d = 12
```



Fractions: Attributes

two options:

- each instance of a class has its own attributes values (we'll see it later, with self.___)

- class-level attributes
  i.e. attributes that are shared by all instances of the class.

Any identifier that is introduced within the body of the class definition, yet outside any of the methods is considered to have class-level scope.

```
class Fraction:
    n = 0
    d = 1
```

```
class Fraction:
    n, d = 0, 1
```

class-level attributes

See example in class-level-attrExample.py

Fraction: Behaviors (Methods) $\frac{n}{d}$

The previous definition of the Fraction class is not good.

1. we don't want all instances of class Fraction have the same value for numerator and denominator,

2. it is better to have a separate function (method, behavior) that:

- will take care of default value of the fraction, and
- will allow a user to create a particular fraction in one statement.

```
class Fraction:
    def _ _init_ _(self, n=0, d=1):
```

constructor – a member function

(*each time a caller instantiates a fraction, this method is automatically called by Python*)

Fraction: Behaviors (Methods) $\frac{n}{d}$

The previous definition of the Fraction class is not good.

1. we don't want all instances of class Fraction have the same value for numerator and denominator,

2. it is better to have a separate function (method, behavior) that:

- will take care of default value of the fraction, and
- will allow a user to create a particular fraction in one statement.

```
class Fraction:
    def _ _init_ _(self, n=0, d=1):
```

implicit parameter

(serves internally to identify the particular instance being constructed;
allows us to access members of this instance using the standard syntactic form *object.membername*)

Fraction: Behaviors (Methods) $\frac{n}{d}$

The previous definition of the Fraction class is not good.

1. we don't want all instances of class Fraction have the same value for numerator and denominator,

2. it is better to have a separate function (method, behavior) that:

- will take care of default value of the fraction, and
- will allow a user to create a particular fraction in one statement.

```
class Fraction:
    def _ _init_ _(self, n=0, d=1):
```

default value of *numerator* is 0
(optional parameter)

default value of *denominator* is 1
(optional parameter)

Fraction: Behaviors (Methods) $\frac{n}{d}$

```
class Fraction:
    def _ _init_ _(self, n=0, d=1):
        self._n = n  initially, numerator is n (if user initialized it)
        self._d = d  initially, den. is d (if user initialized it)
```

comment:
without use of self, both _n and _d will be local variables (*within function's body*);

with **self**, both _n and _d become parts of the object's internal representation

Fraction: Behaviors (Methods) $\frac{n}{d}$

```
class Fraction:
    def _ _init_ _(self, n=0, d=1):
        self._n = n
        self._d = d
```

Now we can use the class this way:

```
def main():
    f1 = Fraction(3,5)
    f2 = Fraction(5,12)
```

Fraction: Behaviors (Methods $\frac{n}{d}$)

```
class Fraction:
    def __init__(self, n=0, d=1):
        self._n = n
        self._d = d
```

method's signature

Now we can use the class this way:

```
def main():
    f1 = Fraction(3,5)
    f2 = Fraction(5,12)
```

Fraction Class

What can we usually do with fractions?

- addition, $\frac{a}{b}+\frac{c}{d}=\frac{ad+cb}{bd}$
- subtraction, $\frac{a}{b}-\frac{c}{d}=\frac{ad-cb}{bd}$
- multiplication, $(\frac{a}{b})*(\frac{c}{d})=\frac{ac}{bd}$
- division, $(\frac{a}{b})\div(\frac{c}{d})=\frac{ad}{bc}$
- power, $(\frac{a}{b})^{n}$
- simplify (bring to lowest terms), $\frac{12}{32}=\frac{3}{8}$
- and so on

Mutable-Immutable Objects

There are two types of objects:

- immutable objects
  (once constructed, can no longer be modified)

- mutable objects
  (can be modified after construction)

1. Let's define class Fraction to construct *immutable objects*, i.e. once a fraction is created, it cannot be modified.

2. Moreover, let's reduce it to lowest terms immediately, i.e. fraction $\frac{25}{35}$ will be immediately reduced to $\frac{5}{7}$

- it is less work with a simplified fraction in the future: during multiplication, division and so on.

3. Also, we will take care of the case, when user is trying to create a fraction with denominator 0.

Fraction Class - Constructor

```
class Fraction:

    def __init__(self, n = 0, d = 1):
        """Fraction(value1,value2) -> value1 / value 2

           constructor of Fraction class takes two optional
           parameters: numerator and denominator"""

        if d==0: #fraction is undefined, if denominator is 0
           self._n=0
           self._d=0

        else : # reducing to lowest terms
           factor = gcd(abs(n),abs(d)) # getting gcd(num,denom)
           self._n = n/factor
           self._d = d/factor
```

Fraction Class – First try to use it

Let's try to use what we developed so far:

```
def main():

    f1=Fraction(10,14)
    f2=Fraction(4,16)

    print("We have two fractions:", f1, "and", f2)

main()
```

This is what we will see (or similar) when we run fraction_class1.py:

```
We have two fractions: <__main__.Fraction instance at
0x01C57B70> and <__main__.Fraction instance at 0x01C57C88>
```

By default, the interpreter prints the above representation, since it doesn't know how objects (information about objects) of class Fraction should be displayed.

Fraction Class - __str__ method

Let's fix it: customize method _ _str_ _ which is responsible for displaying/printing of objects

```
def __str__(self):
    return str(self._n)+'/'+str(self._d)
```

From now on, the fraction $\frac{3}{5}$ will be displayed as 3 / 5

- it is called Polymorphism
(ability to perform an operation differently depending on the specific context, i.e. object of which class invokes this method)

- methods have the same name, same number of parameters, but belong to different classes

see fraction_class2.py

Operator Overloading

There is also a notion of Operator Overloading.

- it is when functions/behaviors/methods have the same name (and sometimes belong to the same class) but have different number of parameters/arguments.

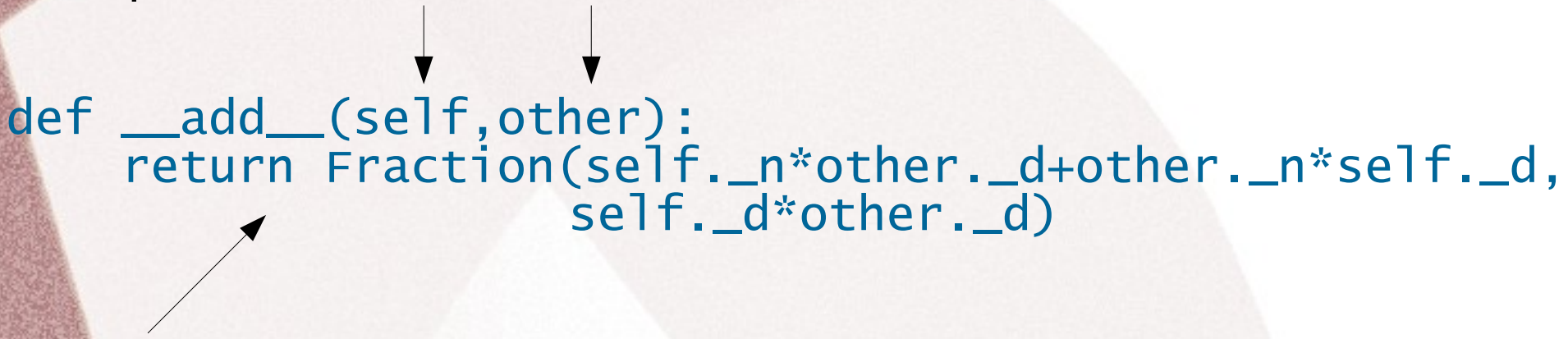
Example:

```
def average(a,b):
    return (a+b)/2.0

def average(a,b,c):
    return (a+b+c)/3.0

def average(a,b,c,d):
    return (a+b+c+d)/4.0
```

Fractions - Addition

Let's define the addition method for fractions: $\frac{a}{b}+\frac{c}{d}=\frac{ad+cb}{bd}$

two parameters: two fractions



```
def __add__(self,other):
    return Fraction(self._n*other._d+other._n*self._d,
                    self._d*other._d)
```

construct/instantiate a new Fraction object, which is the result of addition of two fractions.

Fractions - Subtraction

Let's define the subtraction method for fractions:

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - cb}{bd}$$

```
def __sub__(self,other):
    return Fraction(self._n*other._d-other._n*self._d,
                    self._d*other._d)
```

Fractions - Multiplication

Let's define the multiplication method for fractions:

$$(\frac{a}{b})*(\frac{c}{d})=\frac{ac}{bd}$$

```
def __mul__(self,other):
    return Fraction(self._n*other._n,self._d*other._d)
```

Fractions - Division

Let's define the division method for fractions:

$$(\frac{a}{b}) \div (\frac{c}{d}) = \frac{ad}{bc}$$

```
def __div__(self,other):
    return Fraction(self._n*other._d,self._d*other._n)
```

Fractions - Exponent

Let's define the exponent method:

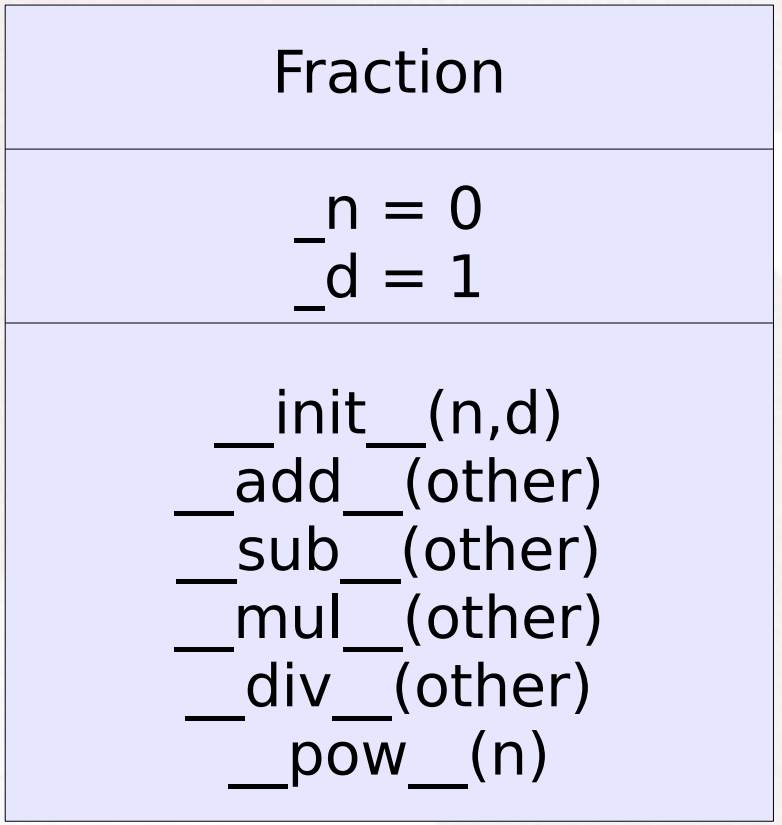
$$(\frac{a}{b})^n = \frac{a^n}{b^n}$$

we use method pow(x,y) which produces x^y

```
def __pow__(self,n):
    return Fraction(pow(self._n,n),pow(self._d,n))
```

see program fraction_class3.py

Fraction Class Diagram



Fractions - Exponent

What didn't we take into consideration?

Fractions - Exponent

What didn't we take into consideration?

Negative Fractions!

Accommodating Negative Fractions

What do we need to update/change to accommodate negative fractions and do we need to change anything at all?

Where to store a sign of a fraction (in denominator or numerator or as a separate attribute)?

Which methods should we modify?

Homework assignment

1. Update/modify our fraction class to accommodate negative numbers

2. problem 6.1 on page 231 (has answer at the end of the book)

3. problem 6.4 on page 232

In-class assignment

1. Create/define a class myFirstClass, using the following class diagram:

MyFirst Class
a = 5 b = -10
getX() returns the value of a getY() returns the value of b setX(value) changes values of a to value setY(value) changes value of b to value

2. Write main() method/function that does the following:
- creates *two instances* of class myFirstClass (using default values of a and b, for each instance), then
- changes the values of a and b to 12 and 23 correspondingly in the first instance, and
- changes the values of a and b to 16 and -100 in the second instance.