CSC148 *Intro. to Computer Science*

Lecture 2: *designing classes, special methods, managing attributes; intro composition, inheritance*

Amir H. Chinaei, Winter 2016

Office Hours: W 16:00–17:45 BA4222

ahchinaei@cs.toronto.edu
http://www.cs.toronto.edu/~ahchinaei/

Course webpage:
http://www.cdf.toronto.edu/~csc148h/winter

Recall

- Labs start Thursday, Jan 21
  - Refer to the course web page for instructions, handouts, and many links to read
  - Do these, **prior** to go to the lab
- Use all resources available to you
  - Before it becomes too late!
  - What resources?
    - The course web page and its many hyperlinks!
    - Office Hours: W 4:00-5:45 BA422
    - The CS Help Center
    - Email ahchinaei @ cs.torotno.edu

Review

- So far
  - Recap of basic Python
    - refer to ramp_up slides in the course web page
  - Introduction to object oriented design
  - In particular, defining new compound data types ~ classes
  - Examples: Class Rectangle, Class Point
- Today
  - Special methods
  - Manage attributes
  - Introduction to composition and inheritance

Key terms

- *Class: (abstract/advanced/compound) data type*
  - It models a thing or concept (let's call it object), based on its common (or important) attributes and actions in a specific project
  - In other words, it bundles together attributes and methods that are relevant to each instance of those objects
- In OO world, objects are often **active** agents
  - In other words, actions are invoked on objects
  - E.g. you invoke an action on your phone to dial a number
  - E.g. you invoke an action on your alarm to wake you up
  - E.g. you invoke an action on your fridge to get you ice

Design roadmap--Step 1

- Before Start!:
  - Read the project specification carefully
  - In the specification:
    - frequent nouns may be good candidate for classes
    - properties of such nouns may be good candidates for attributes
    - actions of such nouns may be good candidates for methods
    - Keep in mind, that there are some **special** methods that are relevant to many classes

Point class(revisited)

- *Specification:*

In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. Points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin. Some of the typical operations that one associates with points might be calculating the distance of a point from the origin, or from another point, or finding a midpoint of two points, or asking if a point falls within a given rectangle or circle.

Point class(revisited)

- *Specification:*

In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. Points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin. Some of the typical operations that one associates with points might be calculating the distance of a point from the origin, or from another point, or finding a midpoint of two points, or asking if a point falls within a given rectangle or circle.

Point class(revisited)

- *Specification:*

In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. Points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin. Some of the typical operations that one associates with points might be calculating the distance of a point from the origin, or from another point, or finding a midpoint of two points, or asking if a point falls within a given rectangle or circle.

Point class(revisited)

- *Specification:*

In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. Points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin. Some of the typical operations that one associates with points might be calculating the distance of a point from the origin, or from another point, or finding a midpoint of two points, or asking if a point falls within a given rectangle or circle.

Design roadmap--Step 2

- Now, we can define a class API:
  1. choose a class name and write a brief description in the class docstring
  2. write some examples of client code that uses your class
  3. decide what services your class should provide as public methods, for each method declare an API (examples, type contract, header, description)
     - **refer to CSC108 function design recipe**
  4. decide which attributes your class should provide without calling a method, list them in the class docstring

Design roadmap-- Step 3

- Then, implement the class:
  1. body of special methods,
     `__init__`, `__eq__`, `__str__`
  2. body of other methods
     e.g. `distance`
  3. testing (more on this later)

Let's do it in PyCharm …

Rectangle class

A rectangle can be defined in many different ways. Here, assume a rectangle is defined by its top-left coordinates as well as its width and height. A rectangle is usually represented by a quadruple (x, y, w, h) where x and y represent the top-left coordinate, w represents the width, and h represents the height. For example, (10, 20, 300, 400) represents a rectangle that its top-left coordinate is located at point (10,20), its width is 300 and its height is 400. Some of the typical operations that one associates with rectangles might be translating the rectangle to the right, left, up, and down, or asking if two rectangles are conceptually equal, which means have same coordinate and size, or asking if a rectangle falls within another rectangle, etc.

Rectangle class

A rectangle can be defined in many different ways. Here, assume a rectangle is defined by its top-left coordinates as well as its width and height. A rectangle is usually represented by a quadruple (x, y, w, h) where x and y represent the top-left coordinate, w represents the width, and h represents the height. For example, (10, 20, 300, 400) represents a rectangle that its top-left coordinate is located at point (10,20), its width is 300 and its height is 400. Some of the typical operations that one associates with rectangles might be translating the rectangle to the right, left, up, and down, or asking if two rectangles are conceptually equal, which means have same coordinate and size, or asking if a rectangle falls within another rectangle, etc.

Rational class

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:

<> < ≤ > ≥ =

Recall: design roadmap

❖ Step 1: Read the project specification carefully

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:

<> < ≤ > ≥ =

Note: Python provides special methods:

`__ne__` `__lt__` `__le__` `__gt__` `__ge__` `__eq__`

Other special methods: `__init__` `__str__` `__add__` `__mul__` …

Recall: design roadmap

❖ Step 2: Define a class API:

1. choose a class name and write a brief description in the class docstring
2. write some examples of client code that uses your class
3. decide what services your class should provide as public methods, for each method declare an API (examples, type contract, header, description)
   – refer to CSC108 **function design recipe**
4. decide which attributes your class should provide without calling a method, list them in the class docstring

API: class definition & constructor

```
class Rational:
    """
    A rational number

    """

    def __init__(self, num, denom=1):
        """
        Create new Rational self with numerator num and
        denominator denom --- denom must not be 0.

        @type self: Rational
        @type num: int
        @type denom: int
        @rtype: None
        """
        pass
```

API: other methods (==)

```python
def __eq__(self, other):
    """
    Return whether Rational self is equivalent to other.

    @type self: Rational
    @type other: Rational | Any
    @rtype: bool

    >>> r1 = Rational(3, 5)
    >>> r2 = Rational(6, 10)
    >>> r3 = Rational(4, 7)
    >>> r1 == r2
    True
    >>> r1.__eq__(r3)
    False
    """
    pass
```


API: other methods (str())

```python
def __str__(self):
    """
    Return a user-friendly string representation of
    Rational self.

    @type self: Rational
    @rtype: str

    >>> print(Rational(3, 5))
    3 / 5
    """
    pass
```


API: other methods (<)

```python
def __lt__(self, other):
    """
    Return whether Rational self is less than other.

    @type self: Rational
    @type other: Rational | Any
    @rtype: bool

    >>> Rational(3, 5).__lt__(Rational(4, 7))
    False
    >>> Rational(3, 5).__lt__(Rational(5, 7))
    True
    """
    pass
```


API: other methods (*)

```python
def __mul__(self, other):
    """
    Return the product of Rational self and Rational other.

    @type self: Rational
    @type other: Rational
    @rtype: Rational

    >>> print(Rational(3, 5).__mul__(Rational(4, 7)))
    12 / 35
    """
    pass
```


API: other methods (+)

```python
def __add__(self, other):
    """
    Return the sum of Rational self and Rational other.

    @type self: Rational
    @type other: Rational
    @rtype: Rational

    >>> print(Rational(3, 5).__add__(Rational(4, 7)))
    41 / 35
    """
    pass
```


… design roadmap

- Continue to develop API for all other methods
- Then, Step 3: Develop the implementation

imp: class definition & constructor

```
class Rational:
    """
    A rational number

    """

    def __init__(self, num, denom=1):
        """
        Create new Rational self with numerator num and
        denominator denom --- denom must not be 0.

        @type self: Rational
        @type num: int
        @type denom: int
        @rtype: None
        """
        self.num, self.denom = int(num), int(denom)
```


imp: other methods (==)

```
def __eq__(self, other):
    """
    Return whether Rational self is equivalent to other.

    @type self: Rational
    @type other: Rational | Any
    @rtype: bool

    >>> r1 = Rational(3, 5)
    >>> r2 = Rational(6, 10)
    >>> r3 = Rational(4, 7)
    >>> r1 == r2
    True
    >>> r1.__eq__(r3)
    False
    """
    return (type(self) == type(other) and
        self.num * other.denom == self.denom * other.num)
```


imp: other methods (str())

```
def __str__(self):
    """
    Return a user-friendly string representation of
    Rational self.

    @type self: Rational
    @rtype: str

    >>> print(Rational(3, 5))
    3 / 5
    """
    return "{} / {}".format(self.num, self.denom)
```


imp: other methods (<)

```
def __lt__(self, other):
    """
    Return whether Rational self is less than other.

    @type self: Rational
    @type other: Rational | Any
    @rtype: bool

    >>> Rational(3, 5).__lt__(Rational(4, 7))
    False
    >>> Rational(3, 5).__lt__(Rational(5, 7))
    True
    """
    return self.num * other.denom < self.denom * other.num
```


imp: other methods (*)

```
def __mul__(self, other):
    """
    Return the product of Rational self and Rational other.

    @type self: Rational
    @type other: Rational
    @rtype: Rational

    >>> print(Rational(3, 5).__mul__(Rational(4, 7)))
    12 / 35
    """
    return Rational(self.num * other.num,
                    self.denom * other.denom)
```


imp: other methods (+)

```
def __add__(self, other):
    """
    Return the sum of Rational self and Rational other.

    @type self: Rational
    @type other: Rational
    @rtype: Rational

    >>> print(Rational(3, 5).__add__(Rational(4, 7)))
    41 / 35
    """
    return Rational(self.num * other.denom +
                    other.num * self.denom,
                    self.denom * other.denom)
```

What if the *denominator* is 0?

Getters, setters and properties

```
def _get_num(self):
    # """
    # Return numerator num.
    #
    # @type self: Rational
    # @rtype: int
    #
    # >>> Rational(3, 4)._get_num()
    # 3
    # """
    return self._num
```


Getters, setters and properties

```
def _set_num(self, num):
    # """
    # Set numerator of Rational self to num.
    #
    # @type self: Rational
    # @type num: int
    # @rtype: None
    # """
    self._num = int(num)

num = property(_get_num, _set_num)
```


Getters, setters and properties

```
def _get_denom(self):
    # """
    # Return denominator of Rational self.
    #
    # @type self: Rational
    # @rtype: int
    #
    # >>> Rational(3, 4)._get_denom()
    # 4
    # """
    return self._denom
```


Getters, setters and properties

```
def _set_denom(self, denom):
    # """
    # Set denominator of Rational self to denom.
    #
    # @type self: Rational
    # @type denom: int
    # @rtype: None
    # """
    if denom == 0:
        raise Exception("Zero denominator!")
    else:
        self._denom = int(denom)

denom = property(_get_denom, _set_denom)
```


Introduction to OOP features

- Composition and Inheritance
  - A rectangle has some vertices (points)
  - A triangle has some vertices (points)
  - A triangle is a shape
  - A rectangle is a shape
- has_a vs is_a relationship
- a shape has a perimeter
  - A rectangle can inherit the perimeter from a shape
  - A triangle too
- a shape has an area
  - Can be area of a rectangle or triangle abstracted to the shape level?