CSC148 *Intro. to Computer Science*

Lecture 3: *designing classes, special methods, composition, inheritance, Stack, Sack*

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

- 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 (see ramp_up slides)
  - Introduction to object oriented design
  - Special methods
  - Manage attributes
  - Introduction to composition and inheritance
- Today
  - More on composition and inheritance
  - Inheriting, extending, and overriding
  - Stack and Sack ADTs

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

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?

More specific example

- Assume you are reading a project specification which is about defining, drawing, and animating some geometrical shapes …

- For now, assume it concerns only two shapes: **squares** and **right angled triangles**.

Square

Squares have four vertices (corners),
have a perimeter, an area, can move themselves
by adding an offset point to each corner, and can
draw themselves.

Right angled triangle

Right angled triangles have three vertices (corners),
have a perimeter, an area, can move themselves
by adding an offset point to each corner, and can
draw themselves.

Abstraction

- Obviously, we need to define two classes
  - Square and RightAngleTriangle

  before rushing to do so, let's rethink …

- Squares and RightAngleTriangles have something in common:
  - composed of some corners (points)
  - also, some common features (actions) are applicable to them: *perimeter*, *area*, *move*, *draw*
- That can be abstracted to a more general class, let's call it Shape

Shape class

- Develop the common features into an abstract class Shape
  - Points, perimeter, area
- Remember to follow the *class design recipe*
  - Read the project specification carefully
  - Define the class with a short description and some client code examples to show how to use it …
  - Develop API of all methods including the special ones, __eq__, __str__, …
    - Remember to follow the *function design recipe,* just don't implement it until your API is (almost) complete
  - Then, implement it

developing Shape API

```
from point import Point
from turtle import Turtle


class Shape:
    """
    A Shape shape that can draw itself, move, and
    report area and perimeter.

    === Attributes ===
    @param list[Point] corners: corners that define
    this Shape
    @param float area: area of this Shape
    @param float perimeter: perimeter of this Shape
    """


if __name__ == "__main__":
    import doctest
    doctest.testmod()
    s = Shape([Point(0, 0)])
```

developing Shape API

…

```
class Shape:
    """
    …
    """

    def __init__(self, corners):
        """
        Create a new Shape self with defined by its
        corners.

        @param Shape self: this Shape object
        @param list[Point] corners: corners that define
        this Shape
        @rtype: None
        """
        pass
```

API, then, implementation

- Continue with API of

  `__eq__(self, other)`
  `__str__(self)`

  `_set_perimeter(self)`
  `_get_perimeter(self)`
  `_set_area(self)`
  `_get_area(self)`

  `move_by(self, offset_point)`
  `draw(self)`

- Then, start implementing it; however …

Shape implementation

- So far, we implemented the common features of Square and RightAngleTriangle
- However, how about differences?
  - For instance, the area of a Square is calculated differently than that of a RightAngleTriangle
- In class Shape, do not implemented _set_area; instead, put a place-holder

```
def _set_area(self):
    # """
    # Set the area of Shape self to the Shape of
    # its sides.
    #
    # @type self: Shape
    # @rtype: None
    # """
    # impossible area to satisfy PyCharm...
    self._area = -1.0
    raise NotImplementedError("Set area in subclass!!!")

def _get_area(self):
    # """
    # Return the area of Shape self.
    #
    # @type self: Shape
    # @rtype: float
    #
    # >>> Shape([Point(1, 1), Point(2, 1), Point(2, 2), Point(1, 2)]).area
    # 1.0
    # """
    return self._area

# area is immutable --- no setter method in property
area = property(_get_area)
```

Inheritance

- So, we developed a super class that is abstract
  - it defines the common features of subclasses
  - but it's missing some features that must be defined in subclasses
- Square and RightAngleTriangle are two subclass examples of Shape from which inheriting the identical features

```
class Square(Shape): ...
class RightAngleTriangle(Shape): ...
```

- Develop Square and RightAngleTriangle
  - Remember to follow the recipes

developing Square

```
from shape import Shape

class Square(Shape):
    """
    A square Shape.
    """

if __name__ == '__main__':
    import doctest
    doctest.testmod()
    s = Square([Point(0, 0)])
```

developing Square

```
...
    def __init__(self, corners):
        """
        Create Square self with vertices corners.

        Assume all sides are equal and corners are square.

        Extended from Shape.

        @param Square self: this Square object
        @param list[Point] corners: corners that define this Square
        @rtype: None

        >>> s = Square([Point(0, 0), Point(1, 0), Point(1, 1), Point(0, 0)])
        """
        Shape.__init__(self, corners)
```

developing Square

```
...
    def _set_area(self):
        """
        Set Square self's area.

        Overrides Shape._set_area

        @type self: Square
        @rtype: float

        >>> s = Square([Point(0,0), Point(10,0),
                Point(10,10), Point(0,10)])
        >>> s.area
        100.0
        """
        self._area = self.corners[-1].distance(self.corners[0])**2
```

Discussion summary

- A Shape is a composition of some Points
- Square and RightAngleTriangle inherit from Shape
  - They inherit the perimeter, area, move and draw from Shape
  - They (slightly) extend the constructor of Shape
  - They override the _set_area of Shape

==========

- The client code can use subclasses Square and RightAngleTriangle, to construct different objects (instances), get their perimeter and area, move them around, and draw them
- What other subclasses can inherit from Shape?

Final note

- Don't maintain documentation in two places, e.g. superclass and subclass, unless there's no other choice:
  - Inherited methods, attributes
    - no need to document again
  - extended methods
    - document that they are extended and how
  - overridden methods, attributes
    - document that they are overridden and how

Let's move on to another case

Stack definition

A stack contains items of various sorts. New items are added on to the top of the stack, items may only be removed from the top of the stack. It's a mistake to try to remove an item from an empty stack, so we need to know if it is empty. We can tell how big a stack is.

Stack definition

A stack contains items of various sorts. New items are added on to the top of the stack, items may only be removed from the top of the stack. It's a mistake to try to remove an item from an empty stack, so we need to know if it is empty. We can tell how big a stack is.

developing Stack API

```
class Stack:
    """
    Last-in, first-out (LIFO) stack.
    """


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

developing Stack API

```
class Stack:
    …
    def __init__(self):
        """
        Create a new, empty Stack self.

        @param Stack self: this stack
        @rtype: None
        """
        pass

    def add(self, obj):
        """
        Add object obj to top of Stack self.

        @param Stack self: this Stack
        @param Any obj: object to place on Stack
        @rtype: None
        """
        pass
```

developing Stack API

```
class Stack:
    …
    …

    def remove(self):
        """
        Remove and return top element of Stack self.

        Assume Stack self is not empty.

        @param Stack self: this Stack
        @rtype: object

        >>> s = Stack()
        >>> s.add(5)
        >>> s.add(7)
        >>> s.remove()
        7
        """
        pass
```

developing Stack API

```
class Stack:
    …
    …
    …

    def is_empty(self):
        """
        Return whether Stack self is empty.

        @param Stack self: this Stack
        @rtype: bool
        """
        pass


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Sack (bag) definition

sack contains items of various sorts. New items are added on to a random place in the sack, so the order items are removed from the sack is completely unpredictable. It's a mistake to try to remove an item from an empty sack, so we need to know if it is empty. We can tel l how big a sack is

Sack (bag) definition

sack contains items of various sorts. New items are added on to a random place in the sack, so the order items are removed from the sack is completely unpredictable. It's a mistake to try to remove an item from an empty sack, so we need to know if it is empty. We can tell how big a sack is

developing Sack API

```
class Sack:
    """
    A Sack with elements in no particular order.
    """


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

developing Sack API

```
class Sack:
    …
    def __init__(self):
        """
        Create a new, empty Sack self.

        @param Sack self: this sack
        @rtype: None
        """
        pass

    def add(self, obj):
        """
        Add object obj to some random location of Sack self.

        @param Sack self: this Sack
        @param Any obj: object to place on Sack
        @rtype: None
        """
        pass
```

developing Sack API

```
class Sack:
    …
    …

    def remove(self):
        """
        Remove and return some random element of Sack self.

        Assume Sack self is not empty.

        @param Sack self: this Sack
        @rtype: object

        >>> s = Sack()
        >>> s.add(7)
        >>> s.remove()
        7
        """
        pass
```

```
class Sack:
    …
    …
    …

    def is_empty(self):
        """
        Return whether Sack self is empty.

        @param Sack self: this Sack
        @rtype: bool
        """
        pass


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Compare Slides 24-27 with 30-33

What are the similarities and the differences?

Implementation thoughts

- The public interface should be constant, but inside we could implement Stack and Sack in various ways
  - Use a python list, which has certain methods that can be used in certain ways to be useful for Stack or Sack needs.
  - Use a python dictionary, with integer keys 0, 1, ... , keeping track of the indexes in certain ways to satisfy Stack or Sack needs

Next

- How Stack and Sack can be abstracted to a more general Container
- More on *testing*
- …