Recall

- Use all resources available to you
  - Before it becomes too late!
- What resources?
  - Office Hours: R 10-12 BA4222
  - The course web page and its many hyperlinks!
  - The CS Help Centre
  - The course forum
  - The TAs mailing list: csc148ta @ cdf.toronto.edu
  - Email achchinaei@cs.toronto.edu
- Note:
  - Today, May 26, the Bahen building is among some other buildings that will be closed from ~9pm to May 27, due to a power turn-off. It will NOT affect our lecture though.

Review

- So far
  - Recap of basic Python (see ramp_up slides)
  - Introduction to OO D/P
  - Special methods
  - Manage attributes
  - Introduction to composition and inheritance
- Today
  - Managing attributes
  - 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

- Encapsulation
  - Hiding instance attributes from clients
    - by making them private
  - Pythonic way of thinking of attributes is
    - to leave them public
  - However, if you wish, you can make them kind of private
    - and use getters, setters, and properties to access them
  - This is useful when you think their implementation can change in future—without changing their interface

Encapsulation (by getters, setters, properties)

- Recall the Rational class:
  - We had two public attributes there:
    - num and denom
  - Let's see how we can make them kind of private

class Rational:

  A rational number

  Public Attributes:

  :type num: int
  the numerator of the rational number

  :type denom: int
  the denominator of the rational number
Encapsulation (by getters, setters, properties)

- Recall the Rational class:
  - We had two public attributes there:
    - num and denom
  - Let's see how we can make them kind of private

```python
class Rational:
    """A rational number""
    # Public Attributes:
    # ==================
    #:type num: int
    #   the numerator of the rational number
    #:type denom: int
    #   the denominator of the rational number
```

**Getter to manage _num**

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

**Setter to manage _num**

```python
def _set_num(self, num):
    # Set numerator of Rational self to num.
    #:type num: int
    #:rtype: None
    if num == 0:
        raise Exception("Zero denominator!")
    else:
        self._num = int(num)
    num = property(_get_num, _set_num)
```

**Getter to manage _denom**

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

**Setter to manage _denom**

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

OOP Features

- Encapsulation
  - So, num and denom are now managed attributes,
    - kind of private!
  - clients should not use them directly
  - If you want to make an attribute read-only, do not provide the setter for it.
  - If you want to make an attribute really private, use __ as its name prefix, but not as its name postfix
Let's move on to other OOP Features

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) easily 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.

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 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 RightAngledTriangle
  before rushing to do so, let's rethink ...

- Squares and RightAngledTriangles 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
Designing Classes

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 RightAngledTriangle
- However, how about differences?
  - For instance, the area of a Square is calculated differently than that of a RightAngledTriangle
  - In class Shape, do not implement _set_area; instead, put a place-holder

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 RightAngledTriangle are two subclass examples of Shape from which inheriting the identical features
  - class Square(Shape): ...
  - class RightAngledTriangle(Shape): ...
- Develop Square and RightAngledTriangle
  - Remember to follow the recipes
from shape import Shape

class Square(Shape):
    A square Shape.

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

def __init__(self, corners):
    Create Square self with vertices corners.
    Assume all sides are equal and corners are square.
    Extended from Shape.
    param corners: corners that define this Square
    :type corners: list[Point]
    >>> s = Square([Point(0, 0), Point(1, 0), Point(1, 1), Point(0, 1)])
    >>> s.area
    1.0

    self._area = self.corners[-1].distance(self.corners[0])**2

Discussion summary

- A Shape is a composition of some Points
- Square and RightAngledTriangle 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 RightAngledTriangle, 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 topic
A stack contains items of various sorts. New items are added onto 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.

```python
class Stack:
    # Last-in, first-out (LIFO) stack.
    def __init__(self):
        # Create a new, empty Stack self.
        pass
    def add(self, obj):
        # Add object obj to top of Stack self.
        # param obj: object to place on Stack
        # rtype: None
        pass
    def remove(self):
        # Remove and return top element of Stack self.
        # Assume Stack self is not empty.
        # rtype: object
        >>> s = Stack()
        >>> s.add(5)
        >>> s.add(7)
        >>> s.remove()
        7
    def is_empty(self):
        # Return whether Stack self is empty.
        # rtype: bool
        pass
if __name__ == "__main__":
    import doctest
doctest.testmod()
```

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.

```python
class Sack:
    """A Sack with elements in no particular order."""
    if __name__ == "__main__":
        import doctest
        doctest.testmod()
        s = Sack()
        s.add(5)
```

defining Sack API

```python
class Sack:
    __init__(self):
        """Create a new, empty Sack self."""
        pass
    add(self, obj):
        """Add object obj to some random location of Sack self.
        :param obj: object to place on Sack
        :type obj: Any
        :rtype: None
        """
        pass
```

defining Sack API

```python
class Sack:
    remove(self):
        """Remove and return some random element of Sack self.
        Assume Sack self is not empty.
        :param Sack self: this Sack
        :type: object
        :rtype: object
        """
        pass
```

defining Sack API

```python
class Sack:
    is_empty(self):
        """Return whether Sack self is empty.
        :type: bool
        :rtype: None
        """
        pass
```

defining Sack API

```python
if __name__ == "__main__":
    import doctest
    doctest.testmod()
    s = Sack()
    s.add(5)
```
Compare Slides 33-36 with 39-42

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
- ...