Object-Oriented Programming

CMPT 145

The Rectangle Class

- An example of a mutable object
- Object methods will change the attributes

Rectangle initialization

```
1 class Rectangle(object):
2     def __init__(self, width, length, x, y):
3         self.length = length
4         self.width = width
5         self.x = x
6         self.y = y
```

Working with Rectangle object instances

```
1  if __name__ == '__main__':
2
3      small = Rectangle(5, 5, 0, 0)  #  5x5  at (0,0)
4      big = Rectangle(10, 10, 0, 0)     # 10x10 at (0,0)
5
6      # move them
7      small.x = 10
8      small.y = -5
9
10     big.x = 15
11     big.y = 0
```

Advantages? Disadvantages?

Methods might be more readable

```
1  class Rectangle(object):
2      # ... __init__ as above
3
4      def move_by(self, by_x, by_y):
5          self.x += by_x
6          self.y += by_y
7
8      def move_to(self, to_x, to_y):
9          self.x = to_x
10         self.y = to_y
11
12 if __name__ == '__main__':
13     # ... as above
14
15     # move them
16     small.move_by(10, -5)
17
18     big.move_to(15, 0)
```

Thinking about public access

- Can public access to the Rectangle attributes cause errors?
- Do methods provide convenience to programmers?

The Square Class

- An example of a mutable object
- Object methods will change the attributes

Square initialization

```
1  class Square(object):
2      def __init__(self, side, x, y):
3          self.side = side
4          self.x = x
5          self.y = y
6
7      def move_by(self, by_x, by_y):
8          self.x += by_x
9          self.y += by_y
10
11     def move_to(self, to_x, to_y):
12         self.x = to_x
13         self.y = to_y
```

The Circle Class

- An example of a mutable object
- Object methods will change the attributes

Square initialization

```
1  class Circle(object):
2      def __init__(self, r, x, y):
3          self.radius = r
4          self.x = x
5          self.y = y
6
7      def move_by(self, by_x, by_y):
8          self.x += by_x
9          self.y += by_y
10
11     def move_to(self, to_x, to_y):
12         self.x = to_x
13         self.y = to_y
```

Rectangles, Circles, and Squares! Oh my!

- Lots of code copying: `move_to()` and `move_by()` are the same.
- Bad!
- A Square is a special kind of Rectangle.
- A Circle is less similar.
- All shapes have certain things in common, e.g., their position.
- All shapes have differences, e.g., the calculations for area.

Redesign using Inheritance: A Base Class

```
1  class Shape(object):
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5          self._shapestr = 'shape'
6
7      def move_by(self, by_x, by_y):
8          self.x += by_x
9          self.y += by_y
10
11     def move_to(self, to_x, to_y):
12         self.x = to_x
13         self.y = to_y
```

The Shape class captures what it means for shapes to have a position.

Redesign using Inheritance: A Sub Class

```
1 class Rectangle(Shape):
2     def __init__(self, width, length, x, y):
3         Shape.__init__(self, x, y)
4         self.length = length
5         self.width = width
6         self._shapestr = 'rectangle'
7
8     def area(self):
9         return self.length * self.width
```

The Rectangle class inherits attributes from Shape:

- Attributes `self.x`, `self.y`
- Methods `move_to()`, `move_by()`

These are initialized by Line 3.

Redesign using Inheritance: A Sub Class

```
1 class Circle(Shape):
2     def __init__(self, radius, x, y):
3         Shape.__init__(self, x, y)
4         self.radius = radius
5         self._shapestr = 'circle'
6
7     def area(self):
8         return Math.pi * self.radius ** 2
```

The Circle class inherits attributes from Shape:

- Attributes `self.x`, `self.y`
- Methods `move_to()`, `move_by()`

These are initialized by Line 3.

Redesign using Inheritance: A Sub Class

```
1 class Square(Rectangle):
2     def __init__(self, side, x, y):
3         Rectangle.__init__(self, side, side, x, y)
4         self._shapestr = 'square'
```

The Square class inherits attributes from Rectangle:

- Attributes `self.x`, `self.y`, `self.length`, `self.width`,
- Methods `move_to()`, `move_by()`, `area()`

These are initialized by Line 3.

How inheritance works

- The `self` object is an `object`.
- Python knows it can have attributes and methods.
- Inheritance is like a tree:
  - A class can have children, called sub-classes.
  - A class can have a parent, called a super-class.
  - The `object` class is the root of the inheritance tree.
- A method call is directed to the object; it will respond if it can.
- If an object's class does not contain the method, the Python will try the object's parent class.

Looking again at class definitions

```
1 class Shape(object):
2     # ...
```

- The Shape class refers directly to `object`
- `object` is the name of Python's object class
- This implies that the Shape class inherits the abilities of `objects`

The Currency Class

- An example of an immutable object
- Object methods will create new object instances

Currency initialization

```
1 class Currency(object):
2     def __init__(self, dollars, cents):
3         self.__dollars = dollars
4         self.__cents = cents
```

- Private attributes cannot be accessed outside the Currency class.

Working with Currency object instances

```
1 if __name__ == '__main__':
2
3     movie_price = Currency(10, 50)
4     pop_corn_price = Currency(7, 95)
5     drink_price = Currency(5, 95)
6     # ...
```

Advantages? Disadvantages?

Adding functionality to Currency objects

```
1  class Currency(object):
2      # ...
3      def add(self, other):
4          """
5          Add two Currency values together,
6          producing a new Currency value
7          :param other: A Currency value
8          :return: A Currency value
9          """
10         cents = 100 * (self.__dollars + other.__dollars) \
11                 + (self.__cents + other.__cents)
12         return Currency(cents//100, cents % 100)
```

Working with Currency object instances

```
1  if __name__ == '__main__':
2
3      movie_price = Currency(10, 50)
4      pop_corn_price = Currency(7, 95)
5      drink_price = Currency(5, 95)
6
7      sub_total = Currency(0, 0)
8      sub_total = sub_total.add(movie_price)
9      sub_total = sub_total.add(pop_corn_price)
10     sub_total = sub_total.add(drink_price)
```

Adding functionality to Currency objects

```
1  class Currency(object):
2      # ...
3      def to_string(self):
4          return '$'+str(self.__dollars)+'.'+str(self.__cents)
5
6
7  if __name__ == '__main__':
8      # ...
9      print('Before taxes:', sub_total.to_string())
```

Adding functionality to Currency objects

```
1  class Currency(object):
2      # ...
3      def multiply(self, rate):
4          cents = 100*self.__dollars + self.__cents
5          cents_f = cents * rate
6          cents_i = int(cents_f)
7          result = Currency(cents_i//100, cents_i % 100)
8          return result
9
10 if __name__ == '__main__':
11     # ...
12     taxes = sub_total.multiply(0.08)
13     total = sub_total.add(taxes)
14     print('Total:', total.to_string())
```

Looking again at class definitions

```
1 class Currency(object):
2     # ...
```

- The Currency class refers directly to `object`
- `object` is the name of Python's object class
- This implies that the Currency class inherits the abilities of `objects`
- Object oriented programming with inheritance is covered extensively in CMPT 270.
- Just a hint here!

The `str` function

- Can be applied to any Python data value
- Returns a string that represents the value
- One ability that every `object` has is to represent a value as a string.
- Our Currency class has a `to_string()` method.
- We can use inheritance to connect to the `str()` function.

Adding functionality to Currency objects

```
1  class Currency(object):
2      # ...
3      def __str__(self):
4          return '$'+str(self.__dollars)+'.'+str(self.__cents)
5
6  if __name__ == '__main__':
7      # ...
8          print('Total Cost:', str(total))
```

How inheritance works for `str`

- We cannot define `str` as it would shadow Python's built-in
- Python's `str` calls the value's `__str__()` method.

```
1 def str(obj):
2     # not literally, but something like this
3     return obj.__str__()
```

- If a class defines `__str__()` it gets used
- If a class doesn't define `__str__()` the call goes to the parent class.