# CSC108H Lecture 9

Dan Zingaro
OISE/UT

July 16, 2009

# Clickers: methods (method_clicker.py)

What is the output of the following program?

```
class Test(object):
  def __init__ (self, val = 0):
    self.x = val

  def get_value (self):
    return x

t = Test(8)
print t.get_value()
```

- A. 8
- B. 0
- C. Error, since `x` is not defined

# Object Features

So far, our objects can support

- Methods
- Per-instance and per-class variables
- Constructor: `__init__`
- String representation: `__str__`
- Object comparison: `__cmp__` (to support all comparisons) or `__eq__` (for equality)

# Operator Overloading

- If `p1` and `p2` are points, what does `p1+p2` do?
- It fails, because Python doesn't know how to "add" points
- We can overload the + operator, though, so that in addition to built-in types, + knows what to do with our objects
- In addition to notational convenience, implementing these operators makes our objects look and behave more like built-in types
- To support addition, we implement the `__add__` method; Python translates `p1 + p2` to `p1.__add__(p2)`

# Operator Overloading in Point (point5.py)

Notice how we create a new Point for the result of the addition.
It's unnatural for + to modify one of its two arguments (c.f. lists).

```
class Point(object):
  ...

  def __add__ (self, other):
    return Point (self.x + other.x, self.y + other.y)
```

# More on Attribute Search

- Assume we have a class `C` with class attribute `x`
- If we execute `a = C()`, the instance `a` has a link to the attribute `x`, in `C`
- The important thing is that it is a **link**: if we change `C.x`, it is reflected in `a.x`
- Python actually searches `a`, finds no `x`, so searches the class of `a` (which is `C`)
- But, as soon as we assign to `a.x`, `a` has a true attribute `x`, with no further link to the class' attribute

# More on Attribute Search (change_attr.py)

```
class C(object):
  x = 5

a = C()
b = C()
print a.x, b.x, C.x
C.x = 8
print a.x, b.x, C.x
a.x = 15
print a.x, b.x, C.x
C.x = 20
print a.x, b.x, C.x
```

# Inheritance

- Inheritance gives us the solution to the open-closed principle
- It lets us capture the commonalities of similar structures, while accounting for differences in individual cases
- Classes can specialize behavior of existing classes by **inheriting** from them, and overwriting existing attributes
- The class that inherits is called a subclass, and the class that is inherited from is its superclass
- Classes inherit all attribute names defined in their superclasses; instances of classes have access to all of the names reached upward through inheritance links
- This is a big difference from modules; to specialize a module, we have to copy-and-paste its text into a new module and change the code!

# Polygons

- Let's start with a class to represent arbitrary polygons (i.e. closed figures with three or more sides)
- We will store the vertices of the polygons as lists of points, using the `Point` class we have already defined
- We will include methods for translating (moving) a polygon, and calculating the perimeter
- In the general case, we calculate perimeter by summing the distances between each pair of vertices — and `Point` has a method for this

# Class Polygon (polygon.py)

```
class Polygon(object):
  '''arbitrary polygons'''

  def __init__ (self, points):
    self.vertices = points

  def perimeter (self):
    total = 0
    for i in range(len(self.vertices) - 1):
      total += self.vertices[i].distance (self.vertices[i + 1])
    total += self.vertices[-1].distance (self.vertices[0])
    return total

  def translate (self, a, b):
    for p in self.vertices:
      p.translate (a, b)
```

# Rectangles

- Now, assume we have to represent rectangles
- We could define them from scratch . . . or we can notice that they are a special type of polygon
- We can still translate a rectangle in the same way as in the general case, but we have a more efficient method for finding perimeter that we'd like to use
- We are capturing commonalities, but capitalizing on differences
- We have also added `s1` and `s2` to our rectangles: we can access these side-length attributes in rectangles, but not general polygons!
- Note: we do no error checking. If anything other than four points representing a rectangle is passed to the constructor, the object is in an invalid state

# Class Rectangle (rectangle.py)

```
from polygon import Polygon

class Rectangle(Polygon):
  '''rectangles'''

  def __init__ (self, points):
    '''Initialize rectangle from clockwise points.'''

    Polygon.__init__(self, points)
    self.s1 = self.vertices[0].distance(self.vertices[1])
    self.s2 = self.vertices[1].distance(self.vertices[2])

  def perimeter (self):
    return 2 * (self.s1 + self.s2)
```

# When (not) to use Inheritance

- Inheritance is the single most abused and misunderstood concept of OO
- When we define class `A` so that it inherits from `B`, we are making a claim that the objects of type `A` are a subset of those of type `B` (think subsets of sets)
- This is why our rectangles-and-polygons was a valid use of inheritance
- Here is a particularly bad idea
  - Cars are objects. People own cars. How about we inherit person from car to represent people-who-own-cars?
- This is an example of a has-a relationship, not an is-a relationship like inheritance
- To implement has-a, all we do is create an attribute (in person) of the thing it has (car); we call this **composition**

# Clickers: inheritance

Do you think that inheriting square from rectangle is a valid use of inheritance?

- A. Yes
- B. No

# Stacks

- Let's come up with a class to represent stacks
- A stack is a sequence of objects
- Objects are removed in the opposite order that they are inserted
- Last in, first out (LIFO), like stacking and taking out plates
- Object last inserted is at the top

# Stack Operations

- **push(o)** Add a new item to the top of the stack
- **pop()** Remove and return top item
- **peek()** Return top item
- **is_empty()** Test if stack is empty
- **size()** Return number of items in stack

# Stack Example

- Start with empty stack. The top will be at the right
- Push 5: [5]
- Push 8: [5, 8]
- Size of stack is now 2
- Pop: [5] ( and returns 8)

# Inheritance and Composition

- Stacks look a lot like lists: they are sequences, they are ordered, they are mutable ...
- Should we then inherit from `list`?
- They are **not** lists though: it should not be possible to execute methods like `append` or `reverse` on them
- Since a stack "is-not" a list, but it does "have-a" list (in our implementation), we use composition rather than inheritance

# Python Stack Class (stack.py)

```
class Stack(object):

  def __init__(self):
    self.items = []

  def push(self, o):
    self.items.append(o)

  def pop(self):
    return self.items.pop()

  def peek(self):
    return self.items[-1]

  def is_empty(self):
    return self.items == []

  def size(self):
    return len(self.items)
```