Lecture #15: OOP

Public-Service Announcement: Cal Hacks

"Cal Hacks is partnering with The Berkeley Forum to host a hackathon panelÂnext Monday at 6pm at Anna Head Alumnae Hall. Panelists include directors of Cal Hacks, co-founder of Teespring, Professor Armando Fox, and journalist Steven Leckart. Check out the Facebook event for more information!"

Public-Service Announcement: Outreach

Midterm Outreach

Computer Science Advisers will be emailing all students who scored below average on the first midterm. We will request an appointment to discuss your performance in the class and offer you resources.

Please keep an eye on your email!

377 Soda **379 Soda** **203 Cory** **205 Cory**






Recap of Object-Based Features

```
>>> class T:
...     _marked = False
...     def __init__(self, x): self._value = x
...     def value(self):       return self._value
...     def mark(self):        self._marked = True
...     @staticmethod
...     def setMark(x):        T._marked = x
```

Statements	T._marked	T._value	t1._marked	t1._value t1.value()	t2._marked	t2._value t2.value()
	False	<ERROR>				
t1 = T(3) t2 = T(5)	False	<ERROR>	False	3	False	5
t1.mark()	False	<ERROR>	True	3	False	5
T.setMark(0)	0	<ERROR>	True	3	0	5
t1.setMark([])	[]	<ERROR>	True	3	[]	5

Inheritance

- Classes are often conceptually related, sharing operations and behavior.
- One important relation is the *subtype* or *"is-a"* relation.
- Examples: A car is a vehicle. A square is a plane geometric figure.
- When multiple types of object are related like this, one can often define operations that will work on all of them, with each type adjusting the operation appropriately.
- In Python (like C++ and Java), a language mechanism called *inheritance* accomplishes this.

Example: Geometric Plane Figures

- Want to define a collection of types that represent polygons (squares, trapezoids, etc.).
- First, what are the common characteristics that make sense for all polygons?

```
class Polygon:
    def is_simple(self):
        """True iff I am simple (non-intersecting)."""
    def area(self): ...
    def bbox(self):
        """(xlow, ylow, xhigh, yhigh) of bounding rectangle."""
    def num_sides(self): ...
    def vertices(self):
        """My vertices, ordered clockwise, as a sequence
        of (x, y) pairs."""
    def describe(self):
        """A string describing me."""
```

- The point here is mostly to document our concept of Polygon, since we don't know how to implement any of these in general.

Partial Implementations

- Even though we don't know anything about Polygons, we can give default implementations.

```
class Polygon:
    def is_simple(self): raise NotImplemented
    def area(self): raise NotImplemented
    def vertices(self): raise NotImplemented
    def bbox(self):
        V = self.vertices()
        xlow, ylow =  xhigh, yhigh = V[0]
        for x, y in V[1:]:
            xlow, ylow = min(x, xlow), min(y, ylow),
            xhigh, yhigh = max(x, xhigh), max(y, yhigh),
        return xlow, ylow, xhigh, yhigh
    def num_sides(self): return len(self.vertices())
    def describe(self):
        return "A polygon with vertices {0}".format(self.vertices())
```

Specializing Polygons

- At this point, we can introduce simple (non-intersecting) polygons, for which there is a simple area formula.

```
class SimplePolygon(Polygon):
    def is_simple(self): return True
    def area(self):
        a = 0.0
        V = self.vertices()
        for i in range(len(V)-1):
            a += V[i][0] * V[i+1][1] - V[i+1][0]*V[i][1]
        return -0.5 * a
```

- This says that a `SimplePolygon` is a kind of `Polygon`, and that the attributes of `Polygon` are to be *inherited* by simple Polygon.
- So far, none of these Polygons are much good, since they have no defined vertices.
- We say that `Polygon` and `SimplePolygon` are *abstract types*.

A Concrete Type

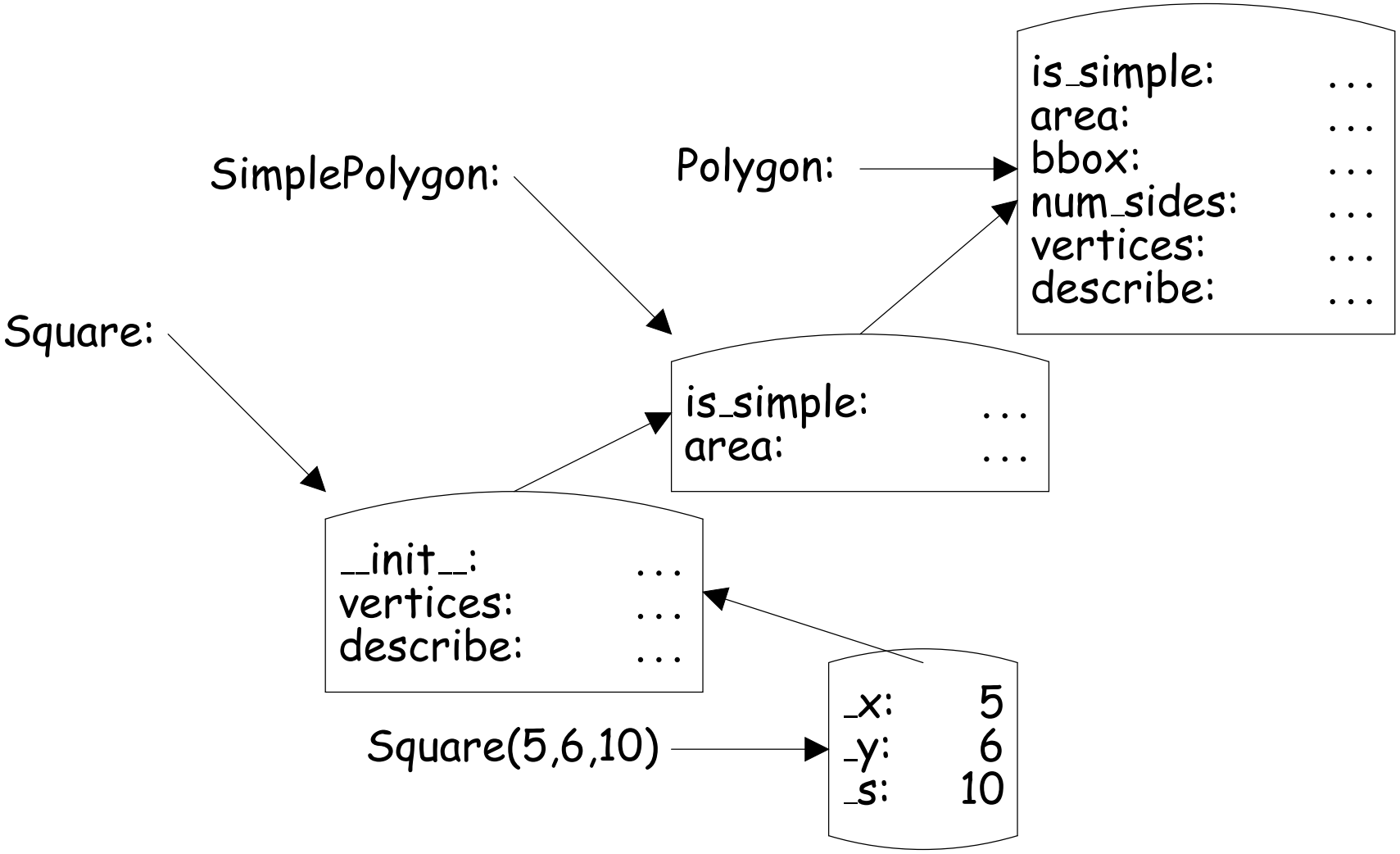
- Finally, a square is a type of simple Polygon:

```
class Square(SimplePolygon):
    def __init__(self, xll, yll, side):
        """A square with lower-left corner at (xll,yll) and
        given length on a side."""
        self._x = xll
        self._y = yll
        self._s = side
    def vertices(self):
        x0, y0, s = self._x, self._y, self._s
        return ((x0, y0), (x0, y0+s), (x0+s, y0+s),
                (x0+s, y0), (x0, y0))
    def describe(self):
        return "A {0}x{0} square with lower-left corner ({1},{2})" \
               .format(self._s, self._x, self._y)
```

- Don't have to define `area,`, etc., since the defaults work.
- We chose to *override* `describe` to give a more specific description.

(Simple) Inheritance Explained

- Inheritance (in Python) works like nested environment frames.



Do You Understand the Machinery?

```
>>> class Parent:
...     def f(s):   # No, you don't have to call it 'self'!
...         print("Parent.f")
...     def g(s):
...         s.f()

>>> class Child(Parent):
...     def f(me):
...         print("Child.f")

>>> aChild = Child()
>>> aChild.g()
# What does Python print?
```

Multiple Inheritance

- A class describes some set attributes.
- One can imagine *assembling* a set of attributes from smaller clusters of related attributes.
- For example, many kinds of object represent some kind of *collection of values* (e.g., lists, tuples, files).
- Built-in kinds of collection have specialized functions representing them as strings (so lists print as `[ ... ]`).
- When we introduce our own notion of collection, we can do this as well, by writing a suitable `__str__(self)` method, which is what `print` calls to print things.
- Many of these methods are similar; perhaps we can consolidate.

Multiple Inheritance Example

```
class Printable:
    """A mixin class for creating a __str__ method that prints
    a sequence object.  Assumes that the type defines __getitem__."""
    def left_bracket(self):
        return type(self).__name__ + "["
    def right_bracket(self):
        return "]"

    def __str__(self):
        result = self.left_bracket()
        for i in range(len(self) - 1):
            result += str(self[i]) + ", "
        if len(self) > 0:
            result += str(self[-1])
        return result + self.right_bracket()
```

Multiple Inheritance Example

- I define a new kind of "sequence with benefits" and would like a distinct way of printing it.

```
class MySeq(list, Printable):
    ...
```

- `MySeqs` will print like

```
MySeq[1, 2, 3]
```

Super

- Sometimes we just want to add to or use the behavior of our parent.
- For example, suppose we have a class that mogrifies:

```
class Transformer:
    def mogrify(self):
        """Do something"""
```

- We want another type that counts how many time `mogrify` is called:

```
class CountedTransformer(Transformer):
    """A Transformer that counts the number of calls to its
    mogrify method."""
    _count = 0

    def mogrify(self):
        _count += 1
        Transformer.mogrify(self) # Calls Transformer's method
        # Or super().mogrify()
    def count(self):
        return self._count
```