Lecture #14: OOP

Some Useful Annotations: `@staticmethod`

- We saw annotations earlier, as examples of higher-order functions.
- For classes, Python defines a few specialized methods.
- The `@staticmethod` annotation denotes a class method (i.e., ordinary function), which does not apply to any particular object.

```
class Account:
    total_deposits = 0

@staticmethod
def total_deposits():
    # No 'self' needed.
    return Account.total_deposits
```

- Now we can write:
  ```
  acct = Account(...)
  acct.total_deposits()  # Total deposits in bank.
  Account.total_deposits()  # Ditto
  ```

Some Useful Annotations: `@property`

- I've said that generally, method calls are the preferred way for clients to access an object (rather than direct access to instance variables).
- This practice allows the class implementor to hide details of implementation.
- Still, it's cumbersome to have to say, e.g., `aPoint.getX()` rather than `aPoint.x`, and `aPoint.setX(v)` rather than `aPoint.x = v`.
- To alleviate this, Python introduced the idea of a property object.
- When a property object is an attribute of an object, it calls a function when it is fetched from its containing object by dot notation.
- The property object can also be defined to call a different function on assignment to the attribute.
- Attributes defined as property objects are called computed or managed attributes.

```
class rational:
    def __init__(self, num, den):
        g = gcd(num, den)
        self.num, self.den = num/g, den/g

    def getNumer(self):
        return self.num

    def setNumer(self, val):
        self.num = val / gcd(val, self.den)

    numer = property(getNumer, setNumer)
    # Alternatively, numer = property(TMPNAME).setter(TMPNAME)
```

- As a result:
  ```
  >>> a = rational(3, 4)
  >>> a.numer
  3
  >>> a.numer = 5
  >>> a.numer
  5
  ```

Properties (Short Form)

The built-in `property` function is also a decorator:

```
class rational:
    ...
    @property
    def numer(self):
        return self.num
    # Equivalent to
    # def TMPNAME(self): return self.num
    # numer = property(TMPNAME)
    # where TMPNAME is some identifier not used anywhere else.

    @numer.setter
    def numer(self, val):
        # Equivalent to
        # def TMPNAME(self, val): self.num = val / gcd(val, self.den)
        # numer = numer.setter(TMPNAME)
```

This is a bit obscure, but the idea is that every property object has a setter method that turns out a new property object that governs both getting and setting of a value.

Recap of Object-Based Features

```
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

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

- Even though we don’t know anything about Polygons, we can give default implementations.
- At this point, we can introduce simple (non-intersecting) polygons.
- Finally, a square is a type of simple Polygon.

A Concrete Type

We say that Polygon and SimplePolygon are abstract types.

For now, most of those polygons are not much good, since they have no attributes or behavior that we have defined. In a moment, we’ll define an area formula for which these are a simple and accurate approximation.

At this point, we can introduce simple (non-intersecting) polygons.

### Specializing Polygons

- A square is a type of simple Polygon:

### Partial Implementations

- Don’t have to define area, etc., since the defaults work.
- Overview:

### (Simple) Inheritance Explained

- Inheritance (in Python) works like nested environment frames.
Do You Understand the Machinery?

```python
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()
```

Last modified: Mon Feb 27 15:56:12 2017

Multiple Inheritance

- A class describes some set attributes.
- One can imagine assembling a set of attributes from smaller clusters.
- 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` method, which is what `print` calls to print things.

Multiple Inheritance Example

```python
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.

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

Super

- Sometimes we just want to add to or use the behavior of our parent.

```python
class Transformer:
    def mogrify(self):
        """Do something"""
    
    def count(self):
        return self.count
```

Example: "Memoization"

```python
class Evaluator:
    def value(self, x):
        some expensive computation that depends only on x

class FastEvaluator(Evaluator):
    def __init__(self): self.memo_table = {}
    def value(self, x):
        """A memoized value computation"""
        if x not in self.memo_table:
            self.memo_table[x] = Evaluator.value(self, x)
        return self.memo_table[x]
```

- FastEvaluator.value must call the .value method of its base (super) class, but we can't just say self.value(x), since that gives an infinite recursion.

```python
"""Memoization"
```