Lecture #16: Inheritance and Interfaces

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), language mechanisms called *inheritance* and *dynamic method selection* accomplish 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  # (see next slide)
    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())
```


Quick Aside: raise

- The statement raise NotImplemented is said to *raise an exception*.
- Usually used to signal an error ("*exceptional condition*").
- But sometimes used for regular programming (iterators later in this lecture).
- In place of NotImplemented, can use any subtype of the built-in class BaseException.
- And now, back to Polygons.

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

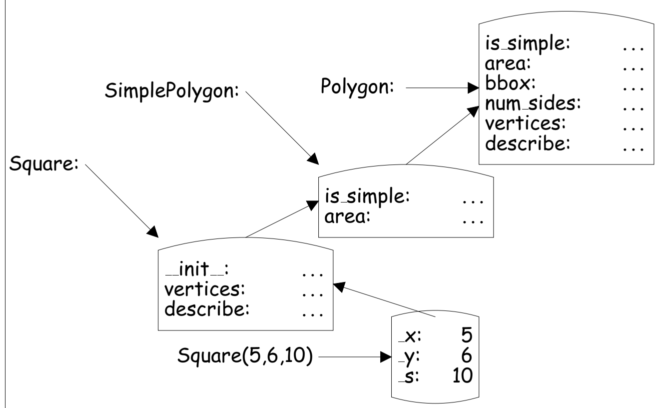
- Finally, a square is a type of simple Polygon:

```
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.

Inheritance Explained

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



Using Base Types

- Sometimes, we want an overriding method in a subtype to *augment* rather than totally replace an existing method.
- That means that we have to call the original version of the method within the overriding method somehow.
- Can't just do an ordinary method call on self, since that would cause infinite recursion.
- Fortunately, we can explicitly ask for the original version of the method by selecting from the class.

Example: "Memoization"

- Suppose we have

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

class FastEvaluator(Evaluator):
    def __init__(self):
        self.__memo_table = {}  # Maps arguments to results

    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.
- So we search for .value starting in Evaluator, as plain function.

super()

- Usually, when (as in FastEvaluator) we want to call a method we are overriding, we want to clearly mark this fact.
- So, the usual way to write FastEvaluator is like this:

```
class FastEvaluator(Evaluator):
    def __init__(self):
        self.__memo_table = {}  # Maps arguments to results

    def value(self, x):
        """A memoized value computation"""
        if x not in self.__memo_table:
            self.__memo_table[x] = super().value(x)
        return self.__memo_table[x]
```


Generic Programming

- Consider the function find:

```
def find(L, x, k):
    """Return the index in L of the kth occurrence of x (k>=0),
    or None if there isn't one."""
    for i in range(len(L)):
        if L[i] == x:
            if k == 0:
                return i
            k -= 1
```

- This same function works on lists, tuples, strings, and (if the keys are consecutive integers) dicts.
- In fact, it works for any list L for which len and indexing work as they do for lists and tuples.
- That is, find is *generic* in the type of L.

The Idea of an Interface

- In Python, this means any type that fits the following *interface*:

```
class SequenceLike:
    def __len__(self):
        """My length, as a non-negative integer."""

    def __getitem__(self, k):
        """My kth element, where 0 <= k < self.__len__()"""
```

(for which len(L) and L[...] are "*syntactic sugar*.")

- This is one way to describe an *interface*, which in a programming language consists of
  - A *syntactic specification* (operation names, numbers of parameters), and
  - A *semantic specification*—its meaning or behavior (given here by English-language comments.)
- Generic functions are written assuming only that their inputs honor particular interfaces.
- The fewer the assumptions in those interfaces, therefore, the more general (and reusable) the function.

Supertypes as Interfaces

- We call the types that a Python class inherits from its *supertypes* or *base types* (and the defined class, therefore, is a *subtype*).
- Good programming practice requires that we treat our supertypes as interfaces, and adhere to them in the subtypes.
- For example, were we to write

```
class MyQueue(SequenceLike):
   def __len__(self): ...
   def __getitem__(self, k): ...
```

then good practice says that MyQueue.__len__ should take a single parameter and return a non-negative integer, and that MyQueue.__getitem__ should accept an integer between 0 and the value of self.__len__()

- Python doesn't actually enforce either of these provisions; it's up to programmers to do so.
- Other languages (like C++, Java, or Ada) enforce the syntactic part of the specification.

Duck Typing

- A *statically typed language* (such as Java) requires that you specify a type for each variable or parameter, one that specifies all the operations you intend to use on that variable or parameter.
- To create a generic function, therefore, your parameters' types must be subtypes of some particular interface.
- You can do this in Python, too, but it is not a requirement.
- In fact, our find function will work on any object that responds appropriately to __len__ and __getitem__, regardless of the object's type.
- This property is sometimes called *duck typing:* "This parameter must be a duck, and if it walks like a duck and quacks like a duck, we'll say it *is* a duck."
- In sum, an explicit supertype is not required in Python to get the benefits of generic programming, but it can help document what we're doing.

Consequences of Good Practice

- If we obey the supertype-as-interface guideline, then we can pass any object that has a subtype of SequenceLike to find and expect it to work.
- This fact is an example of what is called the *Liskov Substitution Principle,* after Prof. Barbara Liskov of MIT, who is generally credited with enunciating it.

Interface as Documentation

- The interface (especially its documentation comments) provides a *contract* between clients of the interface and its subtypes—implementations of the interface:

  > "I, the implementor, agree that all the subclasses I define will conform to the signature and comments in this interface, as long as you, the client, obey any restrictions specified in the interface."

- Since Python does not check or enforce the consistency of supertypes and subtypes, use of the guideline is a matter of individual discipline.
- Enforced or not, the interface type provides a convenient place to document the contract.
- But even when using duck typing, good practice requires that we document the assumptions made by the implementor about parameters to methods (what methods they have, in particular).

Example: The __repr__ Method

- When the interpreter prints the value of an expression, it must first convert that value to a (printable) string.
- To do so, it calls the __repr__() method of the value, which is supposed to return a string that suggests how you'd create the value in Python.

```
>>> "Hello"
'Hello'
>>> print(repr("Hello"))
'Hello'
>>> repr("Hello")    # What does the interpreter print?
```

- (As a convenience, the built-in function repr(x) calls x.__repr__.)
- User-defined classes can define their own __repr__ method to control how the interpreter prints them (see HW#6).

Example: The __str__ Method

- When the print function prints a value, it calls the __str__() method to find out what string to print.
- The constructor for the string type, str, does the same thing.
- Again, you can define your own __str__ on a class to control this behavior. (The default is just to call __repr__)

```
>>> class rational:
...     def __init__(num, den): ...
...     def __str__(self):
...         if self.numer() == 0: return "0"
...         elif self.denom() == 1: return str(self.numer())
...         else: return "{0}/{1}".format(self.numer(), self.denom())
...
>>> rational(3,4)
3/4
>>> rational(5, 1)
5
```


Iterators

- In the homework, we introduce the notion of *iterators*, another use of duck typing.
- The **for** statement is actually a generic control construct with the following meaning:

```
for x in C:          tmp_iter = C.__iter__()
    S                try:
                         while True:
                             x = tmp_iter.__next__()
                             S
                     except StopIteration:
                         pass
```

- The __next__ method can use raise StopIteration statement to cause the loop to exit.
- Types that implement __iter__ are called *iterable*, and those that implement __next__ are *iterators*.
- As usual, the builtin functions iter(x) and next(x) are defined to call x.__iter__() and x.__next__().

Problem: Reconstruct the range class

- Want Range(1, 10) to give us something that behaves like a Python range, so that this loop prints 1–9:

```
for x in Range(1, 10):
    print(x)
```

```
class Range:
    def __init__(self, low, high):
        self._low = low
        self._high = high
    def __iter__(self):
        return RangeIter(self)
```

```
class RangeIter:
    def __init__(self, limits):
        self._bound = limits._high
        self._next = limits._low

    def __next__(self):
        if self._next >= self._bound:
            raise StopIteration
        else:
            self._next += 1
            return self._next
```