Sign In

- Website: `https://goo.gl/forms/FzHSa5INKlavWIJC3`
- Enter the word of the day in the appropriate slot.

Lecture #5: Objects and Object Metaphors

- **Problem:** How to implement the behavior of classes and objects?
- Main points to get:
  - Objects have *attributes.*
  - An object may *inherit* an attribute from its class if it does not define that attribute itself.
  - Likewise, a class may inherit attributes from its parent class(es).
  - In Python, attributes of objects and classes may be added dynamically.
  - In Python, attributes of objects that are methods become *bound methods* when fetched from the object's class.

Python's Representation

- In essence, Python uses a look-up technique that parallels the inheritance diagrams we've shown.
- Think of objects as tuples (D, C) where
  - D is a dictionary, `x.__dict__`,
  - C is a class `x.__class__`

Fetching a Value

- Fetching an attribute now looks like this:

```
def fetch(obj, attr):
    """Fetch the attribute named ATTR from OBJ (an object or
    class)."""
    d, c = obj      # Get the pieces of the object
    if attr in d:
        return d[attr]
    if c is the type of all classes:
        for cl in d['__mro__']: # What's this?
            if attr in cl[0]:   # The dictionary part
                return cl[0][attr]
        raise AttributeError
    else:
        v = fetch(c, attr)
        if v is a method:
            return make_bound_method(obj, v)
        else:
            return v
```

The Method Resolution Order

- When a class has base classes (as is true of all classes but `object`), the order in which they (and their ancestors) are searched matters, because several may define the same attribute.
- This order is called the *method resolution order*.
- It's simple when all classes have one (or zero) base classes:
  - Search the class.
  - Recursively search its base class, if any.
- Things get interesting when there is multiple inheritance, because the same ancestor class can turn up several times when search up through the inheritance chain.
- For more on this issue (and for how Python actually computes its MRO), search for "Python MRO" (or take CS164!).

Less Dynamic Approaches

- Python programs can be slow compared to algorithmically equivalent Java or C++ programs in part because of Python's very dynamic object system.
- Languages such as Java or C++ do not allow the dynamic introduction of new methods.
- This allows considerable speed optimizations.

Virtual Tables

- Rather than dictionaries, Java and C++ use (in effect) lists.
- Instead of searching for strings during execution, the compiler assigns a number to each attribute, and uses list indexing.
- All objects contain all their possible instance variables (new ones are never added) and these are at precomputed indices.
- Each class is represented by a *virtual table* or *dispatch table* containing both methods defined in that class and those inherited from its base classes (also indexed by attribute number), plus other information about the class.
- All objects contain a pointer to the virtual table for their class.

Example

```
class A:
    x = 0                           # Index 1
    def f(self): print("A.f")       # Index 1
    def g(self): print("A.g")       # Index 2

class B(A):
    y = 1                           # Index 2
    def f(self): print("B.f")       # Index 1
    def h(self): print("B.h")       # Index 3

aB = B()
aB.g()
z = aB.x

# Is represented by

A = [ [object], lambda self: print("A.f"), lambda self: print("A.g")]
B = [ [A], lambda self: print("B.f"), A[2], lambda self: print("B.h") ]
aB = [B, 0, 1]      #  Assume aB.x, aB.y initialized from the classes
aB[0][2](aB)
z = aB[1]
```

Nouns as Objects: Constraints

[See .py file]