CS98-52 Lecture 6

Wednesday, October 30th, 2019

Building our own Object system

One of the conclusions we had from Extra Lecture 4 is that we could build our list system, and then a dictionary system with what we call dispatch dictionary.

With dispatch dictionaries, we can now build our own object system! We don't need an object system to actually be built into the language:

```
def account(balance):

    def deposit(amount):
        nonlocal balance
        balance = balance + amount
        return balance

    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return "You're too poor!"
        balance = balance - amount
        return balance

    return {'deposit':deposit,'withdraw':withdraw}
```

This is kind of like an object system already! But we can't quite yet do all the things an object system has. These are the specifications we want:

- Above the abstraction barrier:
  - Objects have local sytate and interact via message passing.
  - Objects are instantiated by classes, which are also objects.
  - Classes may inherit from other classes to share behavior.
  - Mechanics are governed by evaluation procedures
- Below the barrier:
  - Each object has a mutable dictionary of attributes
  - Attribute look-up for instances is a function
  - Attribute look-up for classes is another function
  - Object instantiation is another function

Here are the things we need:

- Object instantiation and initialization
- Attribute lookup and assignment
- Method invocation
- Inheritance

Not important:

- Dot notation
- Multiple inheritance (because it's rare)
- Introspection

Here's a small preview of our new object system:

```
>>> a = Account['new']('Jill')
>>> a['get']('deposit')(40)
40
>>> a['set']('holder') = 'Jack'
```

Instances

Instances are dispatch dictionaries with messages `get` and `set`. Attributes stored in a local dictionary called attributes. Here's how we design the instance system:

```
def make_instance(cls):
    """Return a new object instance."""

    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value,instance)

    def sets_value(name):
        attributes[name] = value

    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance
```

We have to fill the `bind_method` function ourself to implement bound methods in our object system!

```
def bind_method(value,instance):
    """Return value or a bound method if value is callable."""
    if callable(value): # A built-in function that returns whether
                        # something is callable (function, class etc.)
        def method(*args):
            return value(instance,args)
        return method
    else:
        return value
```

> **Asterisk notation**
> Asterisk notation is used to specify that a function that a non-specific number of arguments.
>
> ```
> >>> def f(*a):
> ```

```
...     return a
>>> f(1,2,3,4)
(1,2,3,4)
>>> s = (2,7)
>>> pow(*s)
128
```

We now have much of the object system we need. We just need to build the class itself!

Classes

Here's a small preview of our implementation:

```
>>> Account.keys()
['get', 'set', 'new']
>>> Account['new']('John')
```

So how do we build the class function?

```
def make_class(attributes, base_class=None):

    def get_value(name):

        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)

        def set_value(name,value):
            attributes[name] = value

        def new(*args):
            return init_instance(cls,*args)

    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls
```

Notice we need to define `init_instance`:

```
def init_instance(cls,*args):
    instance = make_instance(cls)
    init = cls['get']('__init__')
    if init is not None:
        init(instance,*args)
    return instance
```

> **Python and `locals()`**
> Python manages to keep track of the names used in a local frame, and we can get them using the `locals()` function:
>
> ```
> >>> def f(x,y):
> ```

```
...     z = 12
...     return locals()
>>> f(2,3)
{'x': 2, 'y': 3, 'z': 12}
>>> def h(x):
...     def hh(y):
...         nonlocal x
...         return locals()
...     return hh
>>> h(7)(8)
{'y': 8, 'x': 7} #OUR ENVIRONMENT DIAGRAMS WERE WRONG
```

Now, we should focus on making our account class?

```
def make_account_class():

    interest = 0.02

    def __init__(self,holder):
        self['set']('holder')=holder
        self['set']('balance')=0

    def deposit(self,amount):
        balance = self['get']('balance')
        balance += amount
        return balance

    def withdraw(self,amount):
        balance = self['get']('balance')
        if amount > balance:
            return 'You're too poor!'
        balance -= amount
        return balance
```

And finally, how do we build inheritance? We already saw this in our class function:

```
def make_class(attributes, base_class=None):
    ...
```

So how would we make our CheckingAccount class from the previous lecture?

```
def make_checking_account_class():
    withdraw_fee = 1

    def withdraw(self,amount):
        fee = self['get']('withdraw_fee')
        return Account['get']('withdraw')(self, amount + fee)

    return make_class(locals(), Account)
```

See the base class identifier? That's how we built inheritance into our custom object system!

And there you go! Over the past two lectures, we've gone from having only functions, to building pairs, using that to build lists, and using that to build dictionaries, and now using dictionaries to build our object system! (We used Python's dictionary implementation here, but we could've used our own)

Relationship to the Python Object System

We spent two lectures talking about the object system in 61A, but we just rebuilt the entire object system in thirty minutes using code! It's almost easier to do it this way...

We didn't quite cover everything. We did do these things:

- Object attributes are stored as dictionaries.
  - In Python, you could type `Account.__dict__.keys()` to see all the values the class is storing.
- There are some magic methods that we haven't quite covered (for example the `__str__` and `__repr__` methods), but otherwise we covered the basics of the object system.