CS61A Lecture 17

Monday, October 7th, 2019

Announcements

- Ants project released today, due next Thursday.
  - Checkpoint on Monday, submit by next Wednesday for early submission bonus point.

Object-Oriented Programming

- Combines concepts of mutable functions and data abstraction to reinforce the idea that modern programs are more than just a long list of instructions, but instead objects interacting with one another in certain ways.

We already know all values are objects in Python, but we'll begin to understand what that really means.

OOP is a method for organizing modular programs, which means you define each piece, without worrying about the other pieces, then they all fit barriers.

Here's what we do need to worry about:

- Abstraction barriers
- Bundling together information and related behavior.

OOP is also a metaphor for computation using distributed state:

- Each object has its own local state. When we want to know what's going on, we have to inpsect each object to figure out its state.
- Each object also knows how to manage its own local state, based on method calls, so when we call the method, it might update the state, and the object knows how to update that state itself.
- Method calls are messages passed between objects.
- Several objects may be instances of a common type, and the different types may relate to each other.

OOP has specialized syntax and vocabulary to uspport this metaphor.

We'll see examples of all this stuff, but these are the basic ideas of OOP you should remember.

> **Example:**
> John wants to transfer $10 from his account to Steven's. John, the object, sends a message to his account, the object to withdraw $10. The account object worries about updating balance, all John needs to know is that $10 was taken out of his account. John the object then sends a message to Steven, telling it that it receives $10, and again, Steven's bank account, the object, worries about updating its balance.
>
> The objects may have other messages coming in and out of them as well.

Classes

Objects are organized according to classes. A class serves as a template for its instances, and each object is an instance of some class.

We've seen built-in classes along the way, but now we're going to start defining our own.

The idea is that all bank accounts have a balance and account holder; the `Account` class should add those attributes to each newly created instance.

Here's what creating an object is going to be like:

```
>>> a = Account('Jim')
>>> a.holder
Jim
>>> a.balance
0
>>> a.deposit(10)
10
>>> a.withdraw(5)
5
>>> a.balance
5
```

We could have theoretically done this with our previous knowledge, defining each object with its own deposit and withdraw functionality, but what we want to do is define a class where these properties can be shared by all objects of the same class, to make sure they behave identically.

Class Statements

Class statements let you create any type of data you want. A class statement looks like this:

```
class <name>:
    <suite>
```

A class statement creates a new class and binds that class to in the first frame of the current environment.

Assignment and def statements in create attributes of the class, and not names in frames.

```
>>> class Clown:
...     nose = 'big and red'
...     def dance():
...         return 'No thanks'
>>> Clown.nose
'big and red'
>>> Clown.dance()
'No thanks'
>>> Clown
<class '__main__.Clown'>
```

Object Construction

Idea: All bank accounts have a balance and an account holder; the 'Account' class should add those attributes to each of its instances.

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created.
2. The `__init__` method the class is called with the new object as its first argument, named `self`, along with additional arguments provided in the call expression.
   class Account:
   def **init**(self, account_holder):
   self.balance = 0
   self.holder = account_holder

The object, when passed in, is bound to the name `self`, and whatever else is passed into the arguments of the definition.

And now we have something useful, not just a blank slate but something that has a holder, and a balance.

`__init__` is a special name for a constructor, and is always used.

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('Jim')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Identity operators `is` and `is not` test if two expressions evaluate to the same object.

Binding an object to a new name using assignment does not create a new object.

```
>>> c = a
>>> c is a
True
```

Methods

The last thing we need to know to finish implementing a class is methods, and methods are the messages that classes receive and send.

Methods are defined in the suite of a class statemenet.

```
class Account:
    def __init__(self, account_holder):
```

```
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Now it's important to pay careful attention to what's going on. `self` is a name that's going to refer to an instance of the `Account` class that it's being deposited into.

When we change the value of an assignment using this kind of statement, what we're doing is that we're changing its value in that instance.

```
def deposit(self, amount):
    if amount > self.balance:
        return 'Insufficient funds'
    self.balance = self.balance - amount
    return self.balance
```

These `def` statements create function objects as always. There's no new rule for executing a `def` statement. But instead, their names aren't bound to a particular name, but as attributes of the class.

Invoking Methods

Now that we've defined our methods, we can invoke them. All invoked methods have access to the object via the `self` parameter, so that they can all access and manipulate to the object's state.

```
class deposit(self,amount):
    self.balance = self.balance + amount
    return self.balance
```

We define `deposit` with two arguments, and dot notation automatically supplies the first argument to a method.

```
>>> tom = Account('Tom')
>>> tom.deposit(100)
100
```

The method is invoked with one argument, binding `self` to `tom` , and `100` to `amount` , but the function is otherwise executed like any other function.

Dot Notation

Objects receive messages via dot notation, which can access attributes of its instance **or** its class. For example, `balance` is something unique to each instance, but `withdraw` is shared among all instances.

```
<expression>.<name>
```

The `<expression>` can be any valid Python expression, and `<name>` has to be a simple name. It evaluates to the value of the attribute looked up by `<name>` in the object that is the value of the expression.

The basic idea is that you first look to the instance, and whether the name is bound there, and if not, you look up the class definition.

Attributes

Attributes is just data stored within an instance or the class itself. We can access them with dot notation, or there is a built-in expression that does the same thing.

```
>>> john = Account('John')
>>> john.balance
0
>>> getattr
<built-in function getattr>
>>> getattr(john, 'balance')
0
>>> john.deposit(100)
100
>>> getattr(john, 'balance')
100
>>> hasattr(john, 'blnce')
False
```

Accessing Attributes

We can either use dot notation or use `getattr` to look up an attribute using a string. Both look up a name in the same way, they are just written differently.

Looking up an attribute name in an object may return:

- One of its instance attributes, or
- One of the attributes of its class

Methods and Functions

A method is an attribute that's a function.

Python distinguishes between:

- A function, which we've been creating for a long time, and
- A bound method, which couple together a function and the object on which that method will be invoked.

```
>>> type(Account.deposit)
<class 'function'>
>>> type(tom.deposit)
<class 'method'>
```

This means we've taken the deposit function and bound it together to the `tom` name.

We can access attributes by doing this:

```
>>> Account.deposit(tom, 1001)
1011
>>> tom.deposit(1001)
2011
```

The difference is that we no longer have to bind the name `self` to the argument.

Looking Up Attributes by Name

Dot expression evaluation rules:

1. Evaluate the expression to the left of the dot, which yields the object of the dot expression.
2. `<name>` is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If not, `<name>` is looked up in the class, which yields a class attribute value.
4. That value is returned unless it is a function, in which case a bound method is returned instead.

Class Attributes

There are attributes of an instance, and there are attributes of a class. So far, they have all been methods, but they are not necessarily always methods. Class attributes can be shared values.

For example,

```
class Account:

    interest = 0.02

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

So now,

```
>>> tom = Account('Tom')
>>> tom.interest
0.02
```

This `interest` value is not "copied in" from the class, but it is just looked up, which means if you mutate its value, it will be changed across all instances of that class.

Review

Sp18 MT2, Q4(b) `combo`

When you’re trying to evaluate the smallest option, or print them all out, or just in general choosing between many different possible solutions, that’s a tree recursion problem.

That means looking for base cases, for recurisve cases, and hwo to combine the results.