CS61A Lecture 18

Wednesday, October 9th, 2019

Announcements

- Several deadlines pushed back.
- Online-only lecture.

Terminology

All objects have attributes, which are name-value pairs. Classes are objects, so they have attributes. Instance attributes are attributes specific to an instance. Class attributes are attributes of the class of an instance.

Where the set of all class attributes meets the set of all functions is methods.

According to the Python object system, functions are objects. Bound methods are also objects: a function that has its first parameter "self" already bound to an instance.

Dot expressions evaluate to bound methods for class attributes that are functions:

```
<instance>.<method_name>
```

Dot Expression Evaluation

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

1. Evaluate the `<expression>` to the left of a 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.

Attribute Assignment

Attribute assignment statements change the values that are bound to attribute names within an object or a class.

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression.

- If the object is an instance, then assignment sets an instance attribute.
- If the object is a class, then assignment sets a class attribute.

> **Instance Attribute Assignment vs. Class Attribute Assignment**
>
> ```
> Account.interest = 0.08
> ```

> This sets the `interest` of the class at 0.08.
>
> ```
> Account().interest = 0.04
> ```
>
> This sets the `interest` of that particular instance at 0.04.

Here's an example of how attribute assignment statements affect classes an objects.

```
class Account():
    interest = 0.02

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

First,

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04 #This shows an instance's class attributes are mutable.
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> jim_account.interest
0.08 #does not erase the instance value
>>> tom_account.interest
0.05
```

Inheritance

A feature of the object system in almost every programming language. A method for relating multiple related classes together, because sometimes classes don't operate in isolation.

A common use is two similar classes that differ in their degree of specialization.

The specialized case may have the same attributes as the general class, along with some special-case behavior.

```
class <name>(<base_class>):
    <suite>
```

Conceptually, the new subclass “shares” attributes with its base class. The subclass may override certain inherited attributes, and anything that isn’t stays the same.

Using inheritance, we implement a subclass by specifying its differences from the base class.

Example

A `CheckingAccount` is a specialized type of `Account`.

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest
0.01
```

The interest rate is lower, but deposits work the same way.

```
>>> ch.deposit(20)
20
>>> ch.withdraw(5)
14
```

Meanwhile, withdrawals incur a $1 fee. Most behavior, however, you notice, is shared with the base class `Account`. Here’s how to write this class.

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
```

Now, we need to redefine the `withdraw` method:

```
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

See that we can reuse the previous `withdraw` method, but since we are calling it on a class, and not on an instance, it does not act as a bound method, so we must supply `self`.

We don’t need to redefine `deposit`, or even `__init__`, as they can look up into the base class.

Base class attributes aren’t copied into subclasses! It automatically looks up.

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom') # Calls the __init__ of the base class
>>> ch.interest # Found in subclass
0.01
>>> ch.deposit(20) # Found in base class
20
```

```
>>> ch.withdraw(5) # Found in subclass
14
```

We could redefine `withdraw` in the subclass by copying the entire implementation found in `Account`, but the problem is any changes we made to the base class wouldn't be reflected in the subclass.

Object-Oriented Designing

Now, we'll talk about designing object-oriented programs, which is a discussion of the problems you will encounter, and some guidance on how to come to a useful solution.

Designing for Inheritance

- Don't repeat yourself; use existing implementations.
- Attributes that have been overridden are still accessible via class objects. (i.e. `Account.withdraw` still calls our older `withdraw` method.)
- Look up attributes on instances whenever possible. (i.e. `CheckingAccount.withdraw` should refer to `withdraw_fee` using `self.withdraw_fee`, rather than just 1, which would break data abstraction laws, or `CheckingAccount.withdraw_fee`, which doesn't allow for specialized checking accounts to have their own withdrawal fees.)

Inheritance and Composition

The other thing we should think about is when to use inheritance, and when to use composition.

Composition is when one object has another object as an attribute, which is also a common occurrence.

Object-oriented programming shines when we adopt the metaphor, and design our programs like how objects work in the real world.

Inheritance is best for representing is-a relationships, such as `CheckingAccount` being specific types of `Account`.

Meanwhile, composition is best for representing has-a relationships, such as a bank having a collection of bank accounts it manages, so the list of accounts is an attribute. `Account` won't inherit from a bank.

```
class Bank:
    """A bank *has* accounts
    >>> bank = Bank()
    >>> john = bank.open_account('John', 10)
    >>> jack = bank.open_account('Jack',5,CheckingAccount)
    >>> john.interest
    0.02
    >>> jack.interest
    0.01
    >>> bank.pay_interest()
    >>> john.balance
    10.2
    """
```

Well, so how do we implement `Bank` ? First,we note it takes no arguments to create one, and it has to remember what accounts it has.

```
    def __init__(self):
        self.accounts = []
```

Now, we start implementing its methods.

```
    def open_account(self,holder,amount,kind=Account):
        account = kind(holder)
        account.deposit(amount)
        self.accounts.append(account)
        return account
```

We create the account by calling `kind` and binding `holder` to its name. Next, we have a starting balance, and we also have to remember that this is an account at our bank.

And finally, our bank has to be able to pay interest, a function that takes no arguments.

```
    def pay_interest(self):
        for a in self.accounts:
            a.deposit(a.balance * a.interest)
```

Are we finished? No, our bank should have some protection so that it shouldn't go bankrupt.

```
    def too_big_to_fail(self):
        return len(self.accounts)>=2
```

Attribute Lookup Practice

We can practice our understanding of objects, classes and inheritance. We solve problems like WWPD by using an environment diagram that has modified rules.

When an instance is created on a class that has no `__init__` , the instance never has instance attributes.

Multiple Inheritance

This occurs when a subclass has multiple base classes.

Let's create a new type of account:

```
class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount -self.deposit_fee)
```

A class may inherit from multiple base classes in Python. Let's say a clever bank executive creates a new type of account:

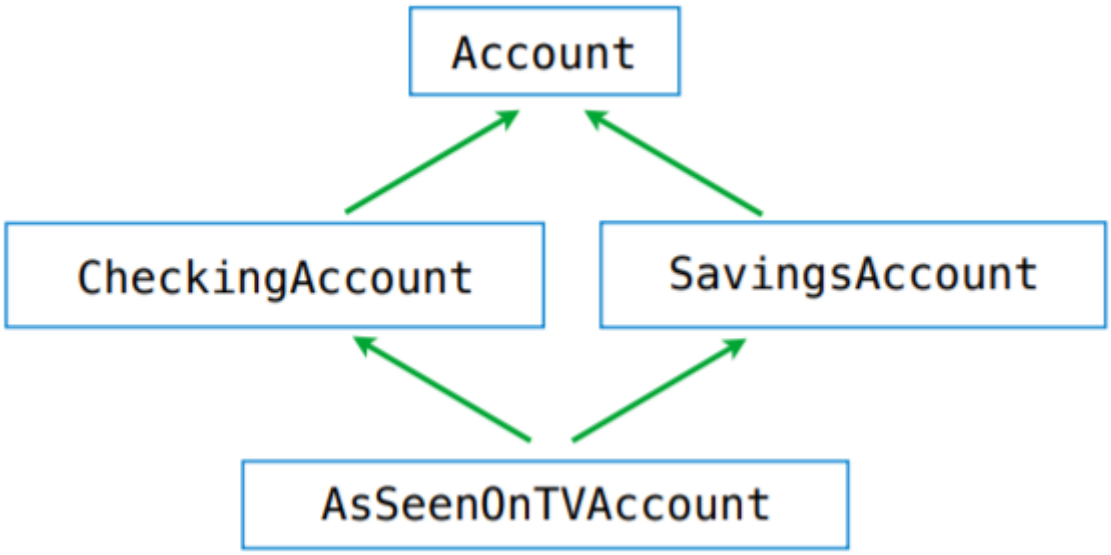
- Low interest rate of 1%
- A $1 fee for withdrawals
- A $2 fee for deposits
- A free dollar when you open your account.

```
class AsSeenOnTVAccount(CheckingAccount,SavingsAccount):
    def __init__(self,account_holder):
        self.holder = account_holder
        self.balance = 1
```

What happens when I create this account?

```
>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

It's easy to see the order in which the inheritance is inherited. The important thing you should remember is you should look at the subclass before the base class.



Complicated Inheritance

Multiple inheritance tends to make programs very complicated, so you should only use it very sparingly, to make your programs clear and readable.