Inheritance in OOP

When working in the OOP paradigm, we often find that different abstract data types are related. In particular, we find that similar classes differ in their amount of specialization. Two classes may have similar attributes, but one represents a special case of the other.

For example, we may want to implement a checking account, which is different from a standard account. A checking account charges an extra $1 for each withdrawal and has a lower interest rate. Here, we demonstrate the desired behavior.

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # withdrawals decrease balance by an extra charge
14
```

A `CheckingAccount` is a specialization of an `Account`. In OOP terminology, the generic account will serve as the base class of`CheckingAccount`, while `CheckingAccount` will be a subclass of `Account`. (The terms *parent class* and *superclass* are also used for the base class, while *child class* is also used for the subclass.)

A subclass *inherits* the attributes of its base class, but may *override* certain attributes, including certain methods. With inheritance, we only specify what is different between the subclass and the base class. Anything that we leave unspecified in the subclass is automatically assumed to behave just as it would for the base class.

Inheritance also has a role in our object metaphor, in addition to being a useful organizational feature. Inheritance is meant to represent *is-a* relationships between classes, which contrast with *has-a* relationships. A checking account *is-a* specific type of account, so having a `CheckingAccount` inherit from `Account` is an appropriate use of inheritance. On the other hand, a bank*has-a* list of bank accounts that it manages, so

neither should inherit from the other. Instead, a list of account objects would be naturally expressed as an instance attribute of a bank object.

2.5.6 Using Inheritance

We specify inheritance by putting the base class in parentheses after the class name. First, we give a full implementation of the`Account` class, which includes docstrings for the class and its methods.

```
>>> class Account(object):
        """A bank account that has a non-negative
balance."""
        interest = 0.02
        def __init__(self, account_holder):
            self.balance = 0
            self.holder = account_holder
        def deposit(self, amount):
            """Increase the account balance by amount and
return the new balance."""
            self.balance = self.balance + amount
            return self.balance
        def withdraw(self, amount):
            """Decrease the account balance by amount and
return the new balance."""
            if amount > self.balance:
                return 'Insufficient funds'
            self.balance = self.balance - amount
            return self.balance
```

A full implementation of `CheckingAccount` appears below.

```
>>> class CheckingAccount(Account):
        """A bank account that charges for
withdrawals."""
        withdraw_charge = 1
        interest = 0.01
        def withdraw(self, amount):
            return Account.withdraw(self, amount +
self.withdraw_charge)
```

Here, we introduce a class attribute `withdraw_charge` that is specific to the `CheckingAccount` class. We assign a lower value to the `interest` attribute. We also define a new `withdraw` method to override the behavior defined in the `Account`class. With no further statements in the class suite, all other behavior is inherited from the base class `Account`.

```
>>> checking = CheckingAccount('Sam')
>>> checking.deposit(10)
10
>>> checking.withdraw(5)
4
>>> checking.interest
0.01
```

The expression `checking.deposit` evaluates to a bound method for making deposits, which was defined in the `Account`class. When Python resolves a name in a dot expression that is not an attribute of the instance, it looks up the name in the class. In fact, the act of "looking up" a name in a class tries to find that name in every base class in the inheritance chain for the original object's class. We can define this procedure recursively. 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.

In the case of `deposit`, Python would have looked for the name first on the instance, and then in the `CheckingAccount` class. Finally, it would look in the `Account` class, where `deposit` is defined. According to our evaluation rule for dot expressions, since`deposit` is a function looked up in the class for the `checking` instance, the dot expression evaluates to a bound method value. That method is invoked with the argument `10`, which calls the deposit method with `self` bound to the `checking` object and`amount` bound to `10`.

The class of an object stays constant throughout. Even though the `deposit` method was found in the `Account` class, `deposit` is called with `self` bound to an instance of `CheckingAccount`, not of `Account`.

Calling ancestors. Attributes that have been overridden are still accessible via class objects. For instance, we implemented the`withdraw` method of `CheckingAccount` by calling the `withdraw` method of `Account` with an argument that included the`withdraw_charge`.

Notice that we called `self.withdraw_charge` rather than the equivalent `CheckingAccount.withdraw_charge`. The benefit of the former over the latter is that a class that inherits from `CheckingAccount` might override the withdrawal charge. If that is the case, we would like our implementation of `withdraw` to find that new value instead of the old one.

Object Abstractions. It is extremely common in object-oriented programs that different types of objects will share the same attribute names. An *object abstraction* is a collection of attributes and conditions on those attributes. For example, all accounts must have `deposit` and `withdraw` methods that take numerical arguments, as well as a `balance` attribute. The classes`Account` and `CheckingAccount` both implement this abstraction. Inheritance specifically promotes name sharing in this way.

The parts of your program that use objects (rather than implementing them) are most robust to future changes if they do not make assumptions about object types, but instead only about their attribute names. That is, they use the object abstraction, rather than assuming anything about its implementation. For example, let us say that we run a lottery, and we wish to deposit $5 into each of a list of accounts. The following implementation does not assume anything about the types of those accounts, and therefore works equally well with any type of object that has a `deposit` method:

```
>>> def deposit_all(winners, amount=5):
        for account in winners:
            account.deposit(amount)
```

The function `deposit_all` above assumes only that each `account` satisfies the account object abstraction, and so it will work with any other account classes that also implement this abstraction. Assuming a particular class of account would violate the abstraction barrier of the account object abstraction. For example, the following implementation will not necessarily work with new kinds of accounts:

```
>>> def deposit_all(winners, amount=5):
        for account in winners:
            Account.deposit(account, amount)
```

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#inheritance