6.01—Introduction to EECS I
Fall Semester, 2007

**Lecture 3 Notes**

# Object-Oriented Programming

In this lecture, we will look at a set of concepts that form the basis of modularity and abstraction in modern software engineering, leading up to object-oriented programming.

Here is our familiar framework for thinking about primitives and means of combination, abstraction, and capturing common patterns. In this lecture, we'll add ideas for abstracting and capturing common patterns in data structures, and ultimately achieving even greater abstraction and modularity by abstracting over data structures combined with the methods that operate on them.

| | Procedures | Data |
|---|---|---|
| Primitives | `+`, `*`, `==` | numbers, strings |
| Means of combination | `if`, `while`, `f(g(x))` | lists, dictionaries, objects |
| Means of abstraction | `def` | abstract data types, classes |
| Means of capturing common patterns | higher-order procedures | generic functions, classes |

## Data structures

We're going to do a running example in this lecture of keeping track of a bank account. The simplest imaginable strategy is to just write a file and store all the data for the bank account in the global environment for that module.

```
balance = 3834.22
interestRate = .0002
owner = "Monty Python"
ssn = "555121212"

def deposit(amount):
    global balance
    balance = balance + amount
```

We could add $100 to the account, and then check the balance like this:

```
>>> deposit(100)
>>> balance
3934.22
```

What if we wanted to add another customer to our bank? There's a really ugly solution involving new variables, `balance2, owner2`, etc., which would require a different `deposit` function for each customer. Yuck. Instead, we should group together the data for each individual customer, and build procedures that can operate on any account.

Data structures are organized ways of storing collections of primitive data elements. You are probably already familiar with arrays and lists. Here is a way to do it using lists.

```
a1 = [3834.22, .0002, "Monty Python", "555121212"]
a2 = [501222.10, .00025, "Ralph Reticulatus", "453129987"]

def deposit(account, amount):
    account[0] = account[0] + amount

>>> deposit(a1,100)
>>> a1[0]
3934.2199999999998
```

A nicer way to use lists (that makes it much more likely your program will actually be right, and that other people will be able to understand and/or modify it) is this:

```
def makeAccount(balance, interestRate, name, ssn):
    return [balance, interestRate, name, ssn]
def accountBalance(a):
    return a[0]
def setAccountBalance(a, b):
    a[0] = b

a1 = makeAccount(3834.22, .0002, "Monty Python", "555121212")
a2 = makeAccount(501222.10, .00025, "Ralph Reticulatus", "453129987")

def deposit(account, amount):
    setAccountBalance(account, accountBalance(account) + amount)

>>> deposit(a1,100)
>>> accountBalance(a1)
3934.2199999999998
```

Most languages have some sort of structure or record, which allows you to group together a collection of items that can be accessed by name; in Python, we can use dictionaries for this job.

```
a3 = {"balance": 3834.22,
      "interestRate": .0002,
      "owner": "Monty Python",
      "ssn": "555121212"}

def deposit(account, amount):
    account["balance"] = account["balance"] + amount

>>> deposit(a3,100)
>>> a3["balance"]
3934.2199999999998
```

There are a wide variety of interesting and complicated data structures that let you organize data efficiently, and can greatly affect the time it takes to do various operations on large collections of data. You can learn more about this in 6.006 and 6.046.

## Abstract data types

What if we needed to figure out how much money a person can borrow against their account? If the credit limit is a function of the account's current balance, we could write a procedure like this:

```
def creditLimit(account):
    return account["balance"] * 0.5
```

So, to get the credit limit of account `a3`, we'd say

```
>>> creditLimit(a3)
```

Another way to handle it would be to make the credit limit an explicit entry in the record, ratehr than recalculating it whenever it's needed. Then, we'd have to update it whenever we did a deposit (or other operation that changed the balance).

```
def deposit(account, amount):
    account["balance"] = account["balance"] + amount
    account["creditLimit"] = account["balance"] * 0.5
```

Then we'd get the credit limit by saying

```
a3["creditLimit"]
```

There's something ugly here about the fact that our representational choices (whether `creditLimit` is a dictionary key or a function) are being exposed to the user of the bank account. We can use the idea of an *abstract data type* (ADT) to show a consistent interface to the "clients" of our code. In fact, that's what we did with the second version of the list representation in the previous section. We define a set of procedures through which all interactions with the data structure are mediated. This set of procedures is often called an *application program interface* or API.

Now our accounts can be represented any way we want. We need to start by providing a way to create a new account. This is called a *constructor*.

```
def makeAccount(balance, rate, owner, ssn):
    return {"balance": balance,
            "interestRate": rate,
            "owner": owner,
            "ssn": ssn,
            "creditLimit": balance*0.5}

a4 = makeAccount(3834.22, .0002, "Monty Python", "555121212")
```

Then we need to add, to the previous example, a way of accessing information about the account.

```
def creditLimit(account):
    return account["creditLimit"]

def balance(account):
    return balance["creditLimit"]
```

Now, we get to the credit limit in the same way, `creditLimit(a4)`, as in the first representation, and nobody needs to know what's going on internally. This might seem like a lot of extraneous machinery, but in large systems, it will mean that you can easily change the underlying implementation of big parts of a system, and nobody else has to care.

## Generic functions

Our bank is getting bigger, and we want to have several different kinds of accounts. Now there is a monthly fee just to have the account, and the credit limit depends on the account type. Here's a new data structure and two constructors for the different kinds of accounts.

```
def makePremierAccount(balance, rate, owner, ssn):
    return {"balance": balance,
            "interestRate": rate,
            "owner": owner,
            "ssn": ssn,
            "type": "Premier"}

def makeEconomyAccount(balance, rate, owner, ssn):
    return {"balance": balance,
            "interestRate": rate,
            "owner": owner,
            "ssn": ssn,
            "type": "Economy"}

a5 = makePremierAccount(3021835.97, .0003, "Susan Squeeze", "558421212")
a6 = makeEconomyAccount(3.22, .00000001, "Carl Constrictor", "555121348")
```

The procedures for depositing and getting the balance would be the same as before. But how would we get the credit limit? We could have separate procedures for getting the credit limit for each different kind of account:

```
def creditLimitEconomy(account):
    return min(account['balance']*0.5, 20.00)
def creditLimitPremier(account):
    return min(account['balance']*1.5, 10000000)

>>> creditLimitPremier(a5)
4532753.9550000001
>>> creditLimitEconomy(a6)
1.6100000000000001
```

But doing this means that, no matter what you're doing with this account, you have to be conscious of what kind of account it is. It would be nicer if we could treat the account generically. We can, by writing one procedure that does different things depending on the account type. This is called a *generic* function.

```
def creditLimit(account):
    if account["type"] == "Economy":
        return minbalance*0.5, 20.00)
    elif account["type"] == "Premier":
        return min(balance*1.5, 10000000)
    else:
        return min(balance*0.5, 10000000)

>>> creditLimit(a5)
4532753.9550000001
>>> creditLimit(a6)
1.6100000000000001
```

In this example, we had to do what is known as *type dispatching*; that is, we had to explicitly check the type of the account being passed in and then do the appropriate operation. We'll see later in this lecture that Python has the ability to do this for us automatically.

## Encapsulated state

In the examples so far, we have made a distinction between *state* (that is, memory of the computation, stored in variables, lists, dictionaries, etc.), and the *procedures* that manipulate the state. In this section, we'll see that we can produce procedures that have *encapsulated state*; that is, procedures that have some variables associated directly with them, such that the variables can be accessed *only* through those procedures.

Now it's time to remember what we know about how environments work. Here is a crucial case of the general rules we saw in the previous chapter. If a procedure creates a new procedure and returns it as a value, that procedure has attached to it *the environment that was enclosing it at the time it was defined.* It is this environment that will be used to look up names inside invocations of that procedure if the names are not bound in the local environment that was made when the procedure was called. Here is an example:

```
1 def makeSimpleAccount(initialBalance):
2     currentBalance = [initialBalance]
3     def deposit(amount):
4         currentBalance[0] = currentBalance[0] + amount
5     def balance():
6         return currentBalance[0]
7     return [deposit, balance]
```

The procedure `makeSimpleAccount` takes an initial balance as input, and returns a list of two procedures. These two procedures have references to `currentBalance`; that variable exists in the enclosing environment, and that environment will remain forever attached to those procedures, so that when they are called later on, they will refer to that same variable.

Why did we have to make `currentBalance` be a list of length 1, instead of just a number? This is an ugliness in Python. The problem is that, in the `deposit` procedure, if `currentBalance` is just a number, then we'll have a statement that looks like

```
    currentBalance = currentBalance + amount
```

and we'll get into all the trouble we talked about earlier, having to do with creating a new local variable, when we really wanted to refer to the one outside. Before, we fixed the problem by saying `global currentBalance`. Unfortunately, that won't work here, because the `currentBalance` we want to refer to is in an enclosing environment, not the global one. If `currentBalance` is a list, however, instead of to the variable `currentBalance`, we can modify the value of its first element. Therefore, the Python interpreter isn't tempted to create a new local variable. Sorry for the ugliness.

Now we can make an account, check the balance, add $20, and check the balance again.

```
>>> a7 = makeSimpleAccount(100)
>>> a7[1]()
100
>>> a7[0](20)
>>> a7[1]()
120
```

An account is represented here as a list of two procedures. The *state* of the account is stored in the enclosing environment of those procedures. What if we make a new account and do things to it?

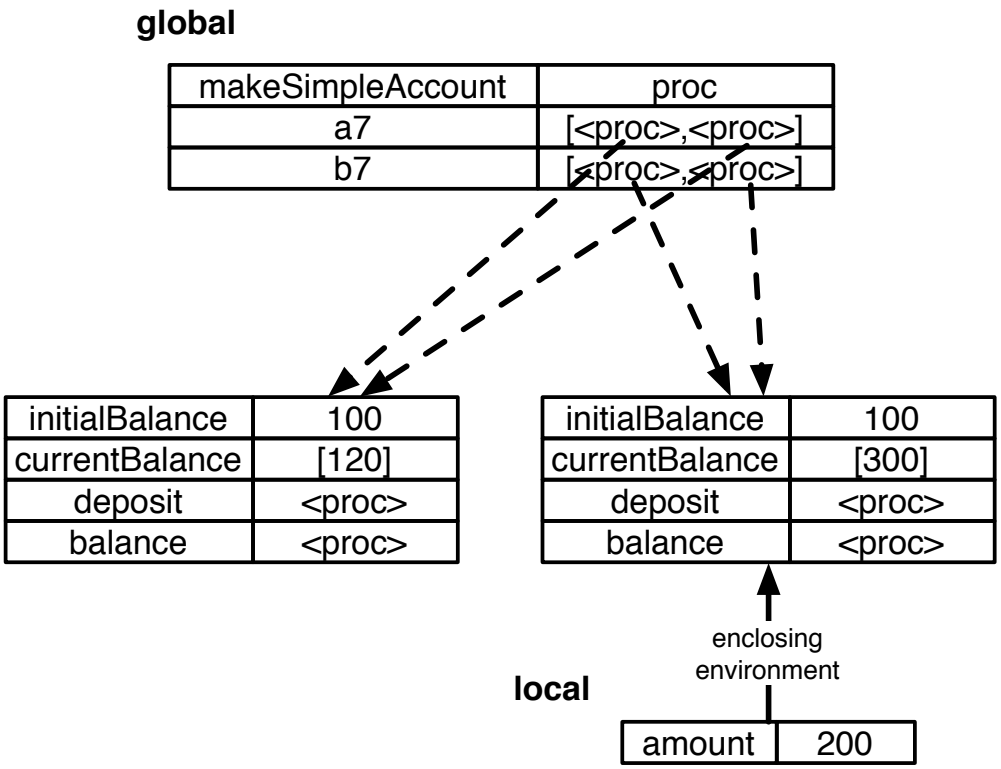


Figure 1: Binding environments that were in place when the deposit method is exectued on `b7`.

```
>>> b7 = makeSimpleAccount(100)
>>> b7[1]()
100
>>> b7[0](200)
>>> b7[1]()
300
>>> a7[1]()
120
```

It's instructive to think about the various environments in detail here. When we do `b7[0](200)`, and line 4 of the code (the body of the `deposit` method) is being executed, it is evaluated in the environment shown in figure 1. The dashed lines show links to the enclosing environments that will be used when the associated procedures are called; it is crucial that these connections are preserved, so that when the procedures are called, they have access to the environment that enclosed them when they were defined.

Note that changes to one account don't affect the other account at all. Each time `makeSimpleAccount` is called, it makes a new enclosing environment, and a new `currentBalance` variable, which is only accessible to the pair of functions that are returned. Thus, we have connected these procedures directly with the state that they operate on; in addition, there is no way for other procedures to sneak in and modify that state.

The way we have to make a deposit to the account, using expressions like `b7[1](200)` is pretty unintuitive, though. We can take another approach:

```
def makeSimpleAccount(initialBalance):
    currentBalance = [initialBalance]
    def doIt(operation, amount = 0):
        if operation == "deposit":
            currentBalance[0] = currentBalance[0] + amount
        elif operation == "balance":
            return currentBalance[0]
        else:
            print "I don't know how to do operation ", operation
    return doIt

a8 = makeSimpleAccount(100)
a8("balance")
a8("deposit", 20)

a9 = makeSimpleAccount(200)
a9("balance")
a9("deposit", 100)
```

This is a more convenient interaction method, where we get to use names for the operations we want, instead of numbers. We will say that the `doIt` procedure *dispatches on* the operation we want to do to the bank account.

## Objects

We have now seen four important ideas: data structures, abstract data types, generic functions, and encapsulated state. These form the basis of *object-oriented programming* (OOP), which is a modern software methodology. Some people advocate a universal application of OOP methodology, but we believe that, as with all tools, it has appropriate applications, but should not be the only tool in our toolbox.

*Classes* and *objects* are, fundamentally, collections of procedures and data, attached to names in an environment. In practice, we define a class to describe a generic object type we'd like to model, like a bank account, and specify various data representations and procedures that operate on that data. Then, when we want to make a particular bank account, we can create an object that is an *instance* of the bank account class. Class instances (objects) are environments that contain all of the values defined in the class, which can then be specialized for the particular instance they are intended to represent.

Here's a *very* simple class, and a little demonstration of how it can be used.

```
class SimpleThing:
    pass

>>> x = SimpleThing()
>>> x
<__main__.SimpleThing instance at 0x85468>
>>> x.a = 4
>>> y = SimpleThing()
>>> y.a
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
AttributeError: SimpleThing instance has no attribute 'a'
>>> y.a = 10
>>> y.a
10
>>> x.a
4
```

A class definition creates a new environment, for the class. In this example, we don't actually add any bindings to that environment when we define the class. The `pass` statement is just used to say we don't want to specify anything about the class.

Whenever you define a class, you get a *constructor*, which will make a new instance of the class. In our example above, the constructor is the class name, for now just followed by a set of parentheses: `SimpleThing()`.[1] Now, to make an instance of the class, we say `SimpleThing()`, which returns a new object. This object also has an environment; it's not the same as the class environment, but its "enclosing" environment is the class environment. We can make new bindings in the object's environment by assigning values to new names, as in `x.a = 4`. To evaluate an expression of the form `name1.name2`, we first look `name1` up in the local environment, and then up the EGB chain; if the value of `name1` is not an object, then an error is generated. Then, the symbol `name2` is looked up in the returned environment. It is important to note that `name2` is *not* looked up in the local environment.

We will sometimes call `a` an *attribute* of `x`. If we make a new instance, say `y` of the class `SimpleThing`, and try to evaluate `y.a`, we will get an error, because that attribute is defined in `x`, but not in `y`. Now, we can add attribute `a` to `y`, it is unrelated to the `a` in `x`, which we demonstrate above by changing the value in object `y` and showing that the value doesn't change in object `x`. Some of you may have experience with Java, which is much more rigid about what you can do with objects than Python is, and which requires you to state, in the class definition, all the attributes the object will have. In Python, it is fine to add attributes to objects on the fly, as we've done here.

Here is an example to illustrate the definition and use of *methods*, which are procedures that are named in the class environment and that have a special connection to instances of the class.

```
1 class Square:
2     dim = 6
3     def getArea (self):
4         return self.dim * self.dim
5     def setArea (self, area):
6         self.dim = area**0.5
```

This class is meant to represent a square. Squares need to store, or remember, their dimension. So, we define a dimension `dim` in the class, and assign it initially to be 6 (we'll be smarter about this in the next example). Now, we define a method `getArea` that is intended to return the area of the square. There are a couple of interesting things here.

Like all methods, `getArea` has an argument, `self`, which will stand for the object that this method is supposed to operate on.[2] Now, remembering that objects are environments, the way we can find

[1] A note on style. It is useful to adopt some conventions for naming things, just to help your programs be more readable. We've used the convention that variables and procedure names start with lower case letters and that class names start with upper case letters. And we try to be consistent about using something called "camel caps" for writing compound words, which is to write a compound name with the successiveWordsCapitalized. An alternative is_to_use_underscores.

[2] The argument doesn't have to be named `self`, but this is a common convention.

the dimension of the square is by looking up the name `dim` in this square's environment, which is bound to the formal parameter `self` inside the method.

We define another method, `setArea`, which will set the area of the square to a given value. In order to change the square's area, we have to compute a new dimension and store it in the `dim` attribute of the square object.

Now, we can experiment with instances of class `Square`.

```
>>> s = Square()
>>> s.getArea()              # Note that it has no arguments in this form
36
>>> Square.getArea(s)
36
>>> s.dim
6
>>> s.setArea(100)
>>> s.dim
10.0
```

We make a new instance using the constructor, and ask for its area by writing `s.getArea()`. This is the standard syntax for calling a method of an object, but it's a little bit confusing because its argument list doesn't really seem to match up with the method's definition (which had one argument). A style that is less convenient, but perhaps easier to understand, is this: `Square.getArea(s)`. Remembering that a class is also an environment, with a bunch of definitions in it, we can see that the interpretation of this statement starts with the class environment `Square` and looks up the name `getArea`. This gives us a procedure of one argument, as we defined it, and then we call that procedure on the object `s`. It is fine to use this syntax, if you prefer, but you'll probably find the `s.getArea()` version to be more convenient. One way to think of it is that we are asking the object `s` to perform its `getArea` method on itself (and the Python interpreter automatically supplying the object as the `self` argument).

When we first call `s.getArea()`, the situation is as shown in figure 2. The body of the `getArea` method is evaluated in an environment with no enclosing environment. However, that local environment (which was made when the method was called) has the name `self` bound to the instance on which it was called. So, now, when the interpeter evaluates `self.dim` in that environment, it looks up `self`, and gets the object. Then, it looks for the name `dim` in that instance's environment. It doesn't find the name there, so it looks in the enclosing (class) environment, and finds the name `dim` there, with the value 6.

Now, when we call `s.setArea()`, things become a bit different; the situation just after line 6 is executed is shown in figure 3. When we assign to `self.dim`, instead of changing the value of `dim` in the class environment, it adds a new name `dim` to the object's environment, and changes it to have value 10. This is very important. If we were to make a brand new square, and query its `dim` value, we would get 6.

Here's a version of the square class that has a special `initialization` method (and no `dim` defined at the class level):

```
class Square1:
    def __init__(self, initialDim):
        self.dim = initialDim

    def getArea (self):
```

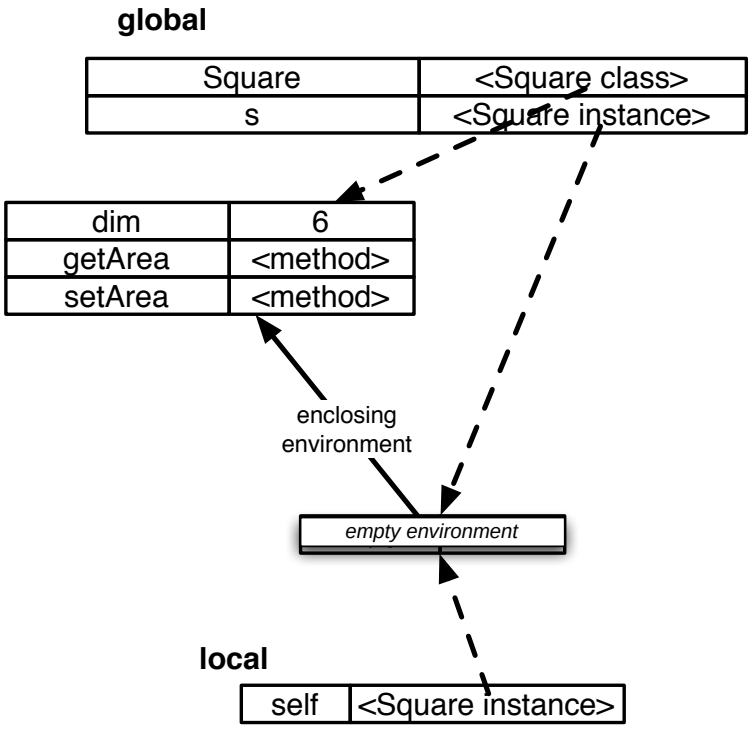


Figure 2: Environment while evaluating statements inside the first call to `s.getArea()`. Dashed arrows show the environments associated with instances and classes.

```
        return self.dim * self.dim

    def setArea (self, area):
        self.dim = area**0.5

    def __str__(self):
        return "Square of dim " + str(self.dim)
```

Whenever the constructor for a class is called, Python looks to see if that class has a method named `__init__` and calls it, with the newly constructed object as the first argument and the rest of the arguments from the constructor added on. So, we could make two new squares by doing

```
>>> s1 = Square1(10)
>>> s1.dim
10
>>> s1.getArea()
100
>>> s2 = Square1(100)
>>> s2.getArea()
10000
>>> print s1
Square of dim 10
```

Now, instead of having an attribute `dim` defined at the class level, we create it inside the initialization method. The initialization method is a method like any other; it just has a special name. Note that

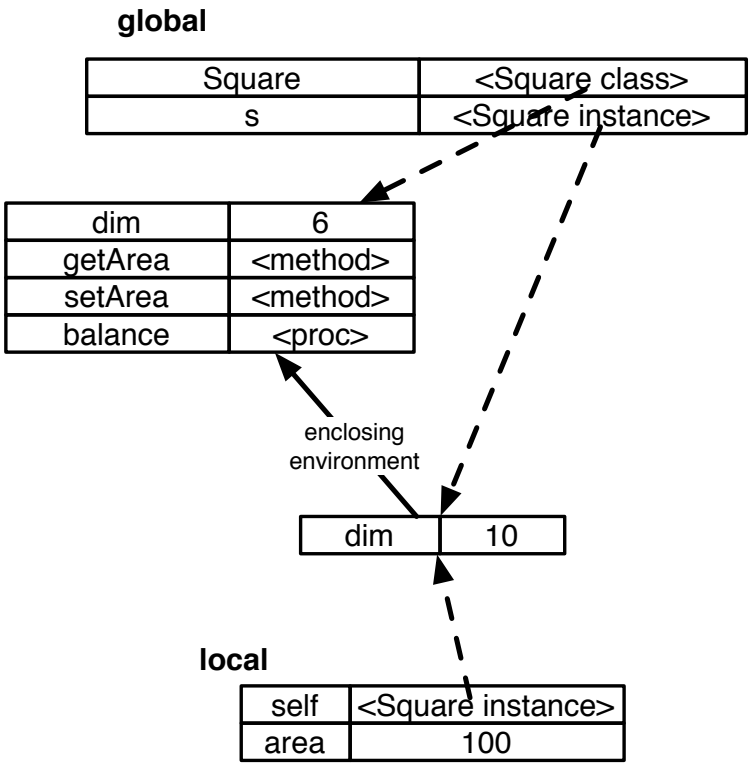


Figure 3: Environment while evaluating statements inside the call to `s.setArea(100)`. Dashed arrows show the environments associated with instances and classes.

it's crucial that we write `self.dim = initialDim` here, and not just `dim = initialDim`. All the usual rules about environments apply here. Remember that methods have no enclosing environment when they are executed. If we wrote `dim = initialDim`, it would make a local variable called `dim`, but that variable would exist only during the execution of the `__init__` procedure. To make a new attribute of the object, it needs to be stored in the environment associated with the object, which we access through `self`.

Our class `Square1` has another special method, `__str__`. This is the method that Python calls on an object whenever it needs to find a printable name for it. By default, it prints something like `<__main__.Square1 instance at 0x830a8>`, but for debugging, that can be pretty uninformative. By defining a special version of that method for our class of objects, we can make it so when we try to print an instance of our class we get something like `Square of dim 10` instead. We've used the Python procedure `str` to get a string representation of the value of `self.dim`. You can call `str` on any entity in Python, and it will give you a more or less useful string representation of it. Of course, now, for `s1`, it would return `'Square of dim 10'`. Pretty cool.

Okay. Now we can go back to running the bank.

```
class Account:
    def __init__(self, initialBalance):
        self.currentBalance = initialBalance
    def balance(self):
        return self.currentBalance
    def deposit(self, amount):
        self.currentBalance = self.currentBalance + amount
    def creditLimit(self):
        return min(self.currentBalance * 0.5, 10000000)

>>> a = Account(100)
>>> b = Account(1000000)

>>> Account.balance(a)
100
>>> a.balance()
100
>>> Account.deposit(a, 100)
>>> a.deposit(100)
>>> a.balance()
300
>>> b.balance()
1000000
```

We've made an `Account` class that maintains a balance as state. There are methods for returning the balance, for making a deposit, and for returning the credit limit. These methods hide the details of the internal representation of the object entirely, and each object encapsulates the state it needs. One thing to note about Python, in constrast to Java, for example, is that it's still possible to go and look at the values of attributes and methods in all the classes, if you try hard enough. (For instance, you can do `dir(x)` where `x` is any object or class, to see the names that are bound in its environment, and then go look at them.

If we wanted to define another type of account, we could do it this way:

```
class PremierAccount:
    def __init__(self, initialBalance):
```

```
        self.currentBalance = initialBalance
    def balance(self):
        return self.currentBalance
    def deposit(self, amount):
        self.currentBalance = self.currentBalance + amount
    def creditLimit(self):
        return min(self.currentBalance * 1.5, 10000000)
>>> c = PremierAccount(1000)
>>> c.creditLimit()
1500.0
```

This will let people with premier accounts have larger credit limits. And, the nice thing is that we can ask for its credit limit without knowing what kind of an account it is, so we see that objects support generic functions, as we spoke about them earlier.

However, this solution is still not satisfactory. In order to make a premier account, we had to repeat a lot of the same definitions as we had in the basic account class. That violates our fundamental principle of laziness: never do twice what you could do once; instead, abstract and reuse.

## Inheritance

*Inheritance* lets us make a new class that's like an old class, but with some parts overridden or new parts added. When defining a class, you can actually specify an argument, which is another class. You are saying that this new class should be exactly like the *parent class* or *superclass*, but with certain definitions added or overridden. So, for example, we can say

```
class PremierAccount(Account):
    def creditLimit(self):
        return min(self.currentBalance * 1.5, 10000000)

class EconomyAccount(Account):
    def creditLimit(self):
        return min(self.currentBalance*0.5, 20.00)

>>> a = Account(100)
>>> b = PremierAccount(100)
>>> c = EconomyAccount(100)
>>> a.creditLimit()
100.0
>>> b.creditLimit()
150.0
>>> c.creditLimit()
20.0
```

This is like generic functions! But we don't have to define the whole thing at once. We can add pieces and parts as we define new types of accounts. And we automatically inherit the methods of our superclass (including `__init__`). So we still know how to make deposits into a premier account:

```
>>> b.deposit(100)
>>> b.balance()
200
```

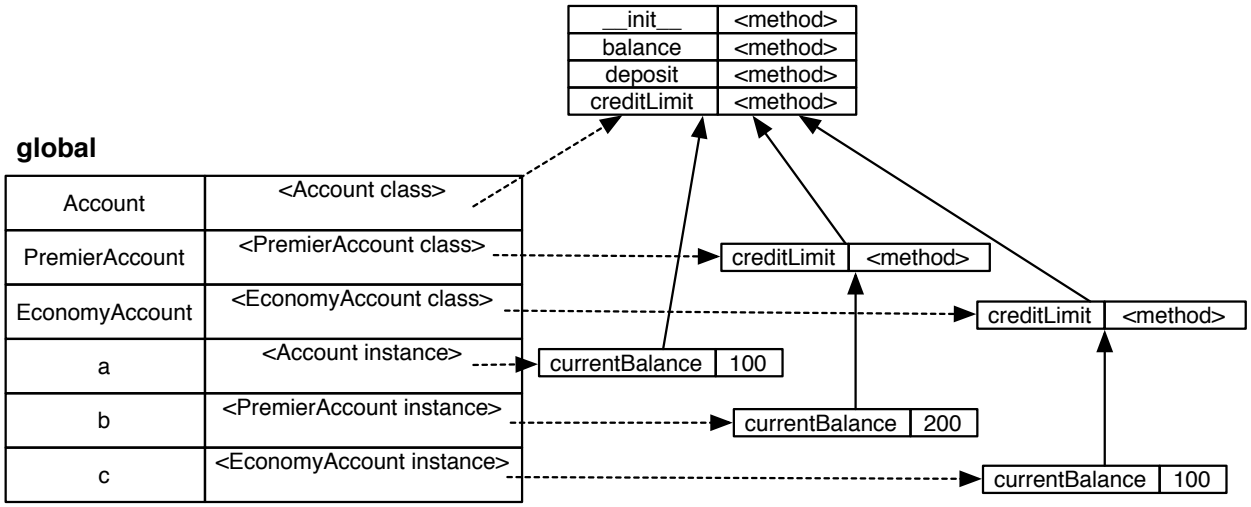


Figure 4: The `Account` class, two subclasses, and three instances. Dashed arrows show the environments associated with instances and classes. Solid arrows show enclosing environments.

This is actually implemented by setting the subclass's enclosing environment to be the superclass's environment. Figure 4 shows the environments after we've executed all of the account-related statements above. You can see that each class and each instance is an environment, and that superclasses enclose subclasses and classes enclose their instances. So, as a consequence of the rules for looking up names in environments, when we ask a `PremierAccount` instance for its `creditLimit` method, it is found in the enclosing class environment; but when we look for the `deposit` method, it is found in the superclass's environment. Once we know how the environment structure is set up, the details of the lookup process are the ones we know from the rest of Python.

There is a lot more to learn and understand about object-oriented programming; this was just the bare basics. But here's a summary of how the object-oriented features of Python help us achieve useful software-engineering mechanisms.

1. **Data structure**: Objects contain a dictionary mapping attribute names to values.
2. **Abstract data types**: Methods provide abstraction of implementation details.
3. **State**: Object attributes are persistent.
4. **Generic functions**: Method names are looked up in object's environment
5. **Inheritance**: Can easily make new related classes, with added or overriden attributes.