

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.081—Introduction to EECS I
 Spring 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	<code>+, *, ==</code>	numbers, strings
Means of combination	<code>if, while, f(g(x))</code>	lists, dictionaries, objects
Means of abstraction	<code>def</code>	abstract data types, classes
Means of capturing common patterns	higher-order procedures	generic functions, classes

Environments

There is a fundamental idea in the operation of Python that we have to understand before delving into the main subject. It is the idea of *binding environment* (we'll just call them *environments*; they are also called *namespaces*, and sometimes *scopes*). An environment is a stored mapping between names and entities in a program. The entities can be all kinds of things: numbers, strings, procedures, objects, etc. In Python, the names are strings and environments are actually dictionaries, which we've already experimented with.

In Python, there are environments associated with each module (file) and one called `__builtin__` that has all the procedures that are built into Python. If you do

```
>>> import __builtin__
>>> dir(__builtin__)
```

you'll see a long list of names of things (like `'sum'`), which are built into Python, and whose names are defined in the builtin module. You generally don't have to type `import __builtin__`. You can try importing `math` and looking to see what names are defined there.

Another thing that creates a new environment is a function call: so, when you do something like:

```
def f(x):
    print x
```

```
>>> f(7)
```

at the moment when the `print` statement is about to be executed, there is a *local* environment that has the formal parameter `x` bound to actual parameter 7.

So, what happens when Python actually tries to evaluate `print x`? It takes the symbol `x` and has to try to figure out what it means. It starts by looking in the *local* environment, which is the one defined by the innermost function call. So, in the case above, it would look it up and find the value 7 and return it.

Now, consider this case:

```
def f(a):
    def g(x):
        print x, a
        return x + a
    return g(7)

>>> f(6)
```

What happens when it's time to evaluate `print x, a`? First, we have to think of the environments. The first call, `f(6)` establishes an environment in which `a` is bound to 6. Then the call `g(7)` establishes another environment in which `x` is bound to 7. So, when needs to print `x` it looks in the local environment and finds that it has value 7. Now, it looks for `a`, but doesn't find it in the local environment. So, it looks to see if it has it available in an *enclosing environment*; an environment that was enclosing this procedure *when it was defined*. In this case, the environment associated with the call to `f` is enclosing, and it has a binding for `a`, so it prints 6 for `a`. So, what does `f(6)` return? 13.

If you write a file that looks like this:

```
a = 6
def foo():
    print a

>>> foo()
6
```

When you evaluate `foo`, it won't be able to find it in the local environment, or in an enclosing environment created by another procedure definition. So, it will look in the *global environment*. The name `global` is a little bit misleading; it means the environment associated with the file. So, it will find a binding for `a` there, and print 6.

One way to remember how Python looks up names is the LEGB rule: it looks, in order, in the *Local*, then the *Enclosing*, then the *Global*, then the *Builtin* environments, to find a value for a name. As soon as it succeeds in finding a binding, it returns that one.

It's really important to understand this, in order to make sense of some of Python's behaviors, which would otherwise seem mysterious.

Data structures

Data structures are organized ways of storing collections of primitive data elements. You are probably already familiar with arrays and lists. Most languages have some sort of structure or record; in Python, we use dictionaries for such things.

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 file.

```
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
```

Why do we need the `global balance` statement in the `deposit` procedure? Because Python, whenever it sees an assignment to a variable inside a procedure makes a new entry in the environment of that procedure with the name of the variable. So, if we didn't have the `global balance` statement in there, it would make a new local variable called `balance`, and then get all confused when it couldn't find a value for it.¹ So, by saying `global balance`, we ask Python not to make a new local variable named `balance`, which means that when we refer to `balance` later on, it refers to the one defined in the global environment.

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. Yuch. Instead, we should group together the data for each individual customer, and build procedures that can operate on any account. 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]
```

¹Remember the LEGB rule: once there is a local variable with a particular name, it “shadows” any other variables with that name in other environments, so that they cannot be referred to.

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(a, accountBalance(a) + amount)

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

Dictionaries are a nice data structure for this.

```
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"]
```

There are whole worlds of interesting and complicated data structures that let you organize data efficiently. Take 6.046 for details.

Abstract data types

Now, let's say we need to figure out how much money the person can borrow against this account. The credit limit is a function of the current balance. We could write it like this:

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

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

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

Another way to handle it would be to make the credit limit a field in the record. 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 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 interaction 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 account["balance"]
```

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": interestRate,
            "owner": owner,
            "ssn": ssn,
            "type": "Premier"}

def makeEconomyAccount(balance, rate, owner, ssn):
    return {"balance": balance,
            "interestRate": interestRate,
            "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(balance*0.5, 20.00)
def creditLimitPremier(account):
    return min(balance*1.5, 10000000)

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

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 min(balance*0.5, 20.00)
    elif account["type"] == "Premier":
        return min(balance*1.5, 10000000)
    else:
        return min(balance*0.5, 10000000)

>>> creditLimit(a5)
>>> creditLimit(a6)
```

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* or 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, that have some variables associated directly with them, and 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 learned above. 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. Here is an example:

```
def makeSimpleAccount(initialBalance):
    currentBalance = [initialBalance]
    def deposit(amount):
        currentBalance[0] = currentBalance[0] + amount
    def balance():
        return currentBalance[0]
    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. So, there's no way to assign to a variable that's defined in an enclosing scope. It's a real bummer. If we make a list, however, we're never assigning to the variable `currentBalance`, but rather assigning to its first element. Therefore, it's never 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
```

What if we make a new account and do things to it?

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

Note that it doesn't affect our previous 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.

There is something pretty ugly about the way we have to make a deposit to the account, 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. 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, state, and generic functions. These form the basis of *object-oriented programming* (OOP), which is a modern software methodology. Some people advocate a very rigid application of narrow OOP methodology, but we

believe that, as with all tools, it has appropriate applications, but should not be the only tool in our toolbox.

An *object* is a collection of procedures and data, attached to names in an environment. A *class* is essentially the same thing, at a formal level. 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, but that 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:
    a = 6

>>> x = SimpleThing()
>>> x
<__main__.SimpleThing instance at 0x85468>
>>> x.a
6
>>> y = SimpleThing()
>>> y.a
6
>>> y.a = 10
>>> y.a
10
>>> x.a
6
```

To define a class, you start with a class statement, and then a set of indented assignments and definitions. Each assignment makes a new variable within 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 `SimpleThing()`.² When we make a new instance of `SimpleThing`, we get an object. We can look at the value of attribute `a` of the object `x` by writing `x.a`. An object is an environment, and this is the syntax for looking up name `a` in environment `x`. If we make a new instance of the class, it has a separate copy of the `a` attribute, which we demonstrate above by changing the value in object `x` and showing that the value doesn't change in object `y`.

Some of you may have experience with Java, which is much more rigid about what you can do with objects than Python is. In Python, you can add attributes to objects on the fly. So, we could continue the previous example with:

```
>>> x.newAttribute = "hi"
```

²A 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`.

and there would be no problem.

Here's another example to illustrate the definition and use of *methods*, which are procedures whose first argument is the object, and that can be accessed via the object.

```
class Square:
    dim = 6

    def getArea (self):
        return self.dim * self.dim

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

This class is meant to represent a square. Squares need to store, or remember, their dimension, so we make an attribute for it, 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.³ Now, remembering that objects are environments, the way we can find the dimension of the square is by looking up the name `dim` in this square's environment, which was passed into this method as the object `self`.

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()
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 it 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`.

³The argument doesn't have to be named `self`, but this is a standard convention.

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 as asking the object `s` to perform its `getArea` method on itself.

Here's a version of the square class that has a special `initialization` method.

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

    def getArea (self):
        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 there is a method called `__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 it's crucial that we write `self.dim = initialDim` here, and not just `dim = initialDim`. All the usual rules about environments apply here. If we wrote `dim = initialDim`, it would make a local variable called `dim`, but that variable would only exist 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.

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 a premier account have a larger credit limit. 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, there's something pretty ugly about this last move. 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, abstract, and reuse.

Inheritance lets us make a new class that's like an old class, but with some parts overridden. 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*, 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
```

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 OO features of Python help us achieve useful software-engineering mechanisms.

1. **Data structure:** Object is a dictionary
2. **ADT:** Methods provide abstraction of implementation details
3. **State:** Object dictionary is persistent
4. **Generic functions:** Method name looked up in object
5. **Inheritance:** Easily make new related classes