MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.099—Introduction to EECS I
Fall Semester, 2006

Lecture 3

- Environments
- Data structures: group values of different kinds together
- Abstract data types: hide the representation details; present a consistent picture
- Generic functions: operate on different data types in same patterns
- Encapsulating state: wrap memory up inside a procedure
- Objects = data structures + ADTs + generic functions + state
- Inheritance: themes and variations

Framework for Abstraction

	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

Environments

Environments (or namespaces) are mappings from names to values. What makes a new environment: file (module), def

Def adds entries in the new environment for the names of its arguments

The first assignment to a variable in an environment makes a new entry in that environment for that variable

When we need to know the value of a symbol, look in the textually enclosing environments, from inside to outside.

```
print max   # Built-in
max = 1
print max   # Global (module)
def foo (max):
    print max   # Encapsulating
    def bar (c):
        max = 2
        print max   # Local
    bar (0)
foo (10)
```

Data structures

Make a bank account with global variables

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

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

To add money, say

```
deposit (100)
```

To check the balance, say

```
balance
```

Why do we need `global balance`? Otherwise, it would want to make a new local variable called `balance`

But what to do when we need to add another customer? Make a data structure. Here are some examples in Python 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
```

We could do a deposit by saying

```
deposit (a1, 100)
```

and check our balance

```
a1[0]
```

Dictionaries are a nice data structure for this. Some other programming languages call them records or structures.

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

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

We could still do a deposit by saying

```
deposit (a3, 100)
```

and check our balance

```
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.

Define a set of procedures through which all interaction with the data structure are mediated. Sometimes called an API (application program interface).

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, as in the first representation, and nobody needs to know what's going on internally.

```
creditLimit(a4)
```

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. And the credit limit depends on the account type. Here's a new data structure and two constructors

```
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)
```

Encapsulated state

If you return a procedure as a value, it brings the current environment along with it!

So, we can do this:

```
def makeSimpleAccount (initialBalance):
    currentBalance = initialBalance
    def deposit (amount):
        global currentBalance
        currentBalance = currentBalance + amount
    def balance ():
        return currentBalance
    return (deposit, balance)
```

Why do we need `global currentBalance`? Because otherwise, when we try to assign a value to currentBalance in the local environment of deposit, it makes a new local variable called currentBalance that ”shadows” the currentBalance variable inside makeSimpleAccount.

Now we can do:

```
a7 = makeSimpleAccount (100)
a7[1]()   # check the balance
a7[0](20)   # deposit 20
a7[1]()   # check it again
```

Here's another way:

```
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)
```

a9 and a8 have encapsulated or ”closed over” their own copies of the environment. There is no interaction between them. This is what was going on with memoize.

Objects

Object-oriented programming is built out of these four ideas. Objects = data structures + ADTs + state + generic functions

A **class** is an environment with a bunch of procedures (called methods) defined in it.

It can also have other values, but we'll ignore that for now.

Each class defines a constructor `className()`

Methods are just procedures whose first argument is an object. We call it `self` by convention (could use any name).

Whenever the constructor is called, it looks for 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.

An object is really nothing but a namespace. You can think of it as a dictionary where we look up names. Whenever you want to get something out of an object, you need to say `object.thing`.

To apply some method from a class to an object and another argument, we can say:
`class.methodname(object, arg)`
This looks up `methodname` in the environment that belongs to `class`, gets back a procedure, and calls it with `object` as the first argument and `arg` as the second.

Another way to do this, which is more common and compact and often very convenient, is to say
`object.methodname(arg)`
You can think of it as asking the object to perform its method on the object using the argument.

Inside a method, whenver you want to refer to a variable that belongs to a method, you need to say `self.var`. If you say simply `var = 2`, it will make a new variable in the environment for the method. If you say `self.var = 2`, it will look to see if the object `self` already has a variable named `var`, and if so, it will change its value to 2. If not, it makes a new variable called `var` and assigns it to have value 2.

```
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)
a.balance()

Account.deposit(a, 100)
a.deposit(100)

b.balance()
```

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

```
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)
```

```
b = PremierAccount(100)
c = EconomyAccount(100)
a.creditLimit()
b.creditLimit()
c.creditLimit()
```

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()
```

Object-Oriented Programming

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