MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Course notes for Week 4

1 State, objects, and abstraction

Last week we introduced object-oriented programming, motivating classes because they provide a convenient way of organizing the procedures and data associated with an abstract data type. This week, we'll look at some other important abstractions strategies for computer programs and see how OOP can help us with them, as well.

In the context of our table and the PCAP framework, we will pay special attention to generic functions and inheritance in OOP, which give us methods for capturing common patterns in data (and the procedures that operate on that data).

	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, inheritance

1.1 Objects and state

We saw, last week, how to define a convenient bank-account object that implements an ADT for a bank account.

```
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)
```

We can use this ADT to create and manage several bank accounts at once:

```
>>> 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
>>> a.balance()
300
```

The `Account` class contains the procedures that are common to all bank accounts; the individual objects contain state in the values associated with the names in their environments. That state is *persistent*, in the sense that it exists for the lifetime of the program that is running, and doesn't disappear when a particular method call is over.

1.2 Generic functions

Now, imagine that the bank we're running 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. Let's see how it would work to use dictionaries to store the data for our bank accounts. 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 for both kinds of accounts. 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 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)
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.

1.3 Classes and inheritance

If we wanted to define another type of account as a Python class, 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 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
```

This is actually implemented by setting the subclass's enclosing environment to be the superclass's environment. Figure 1 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 look-up process are the ones we know from the rest of Python.

The fact that objects know what class they were derived from allows us to ask each object to do the operations appropriate to it, without having to take specific notice of what class it is. Procedures that can operate on objects of different types or classes without explicitly taking their types or classes into account are called *polymorphic*. Polymorphism is a very powerful method for capturing common patterns.

There is a lot more to learn and understand about object-oriented programming; we have just seen 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 overridden attributes.

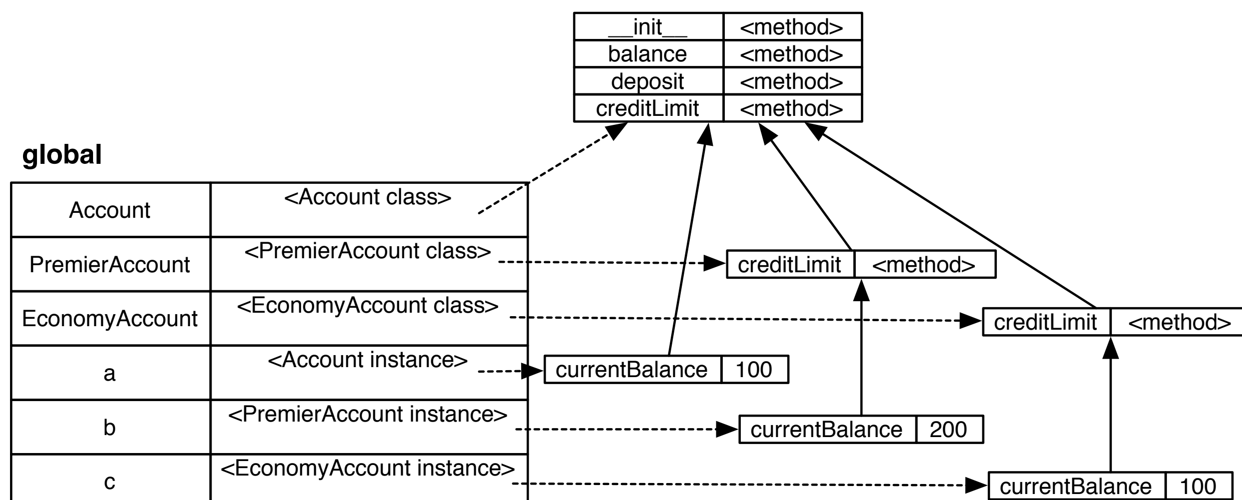

Figure 1: The `Account` class, two subclasses, and three instances. Dashed arrows show the environments associated with instances and classes. Solid arrows show enclosing environments.

2 Combination and abstraction of state machines

Last week, we studied the definition of a primitive state machine, and saw a number of examples. State machines are useful for a wide variety of problems, but specifying them using state transition tables or even more general functions ends up being quite tedious. Ultimately, we'll want to build large state-machine descriptions compositionally: by specifying primitive machines and then combining them into more complex systems. We'll start, here, by looking at ways of combining state machines.

2.1 Machine Composition

We can apply our PCAP (primitive, combination, abstraction, pattern) methodology here, to build more complex SMs out of simpler ones. Figure 2 sketches three types of SM composition: serial, parallel, and feedback.

2.1.1 Serial composition

In serial composition, we take two machines and use the output of the first one as the input to the second. The result is a new composite machine, whose input vocabulary is the input vocabulary of the first machine and whose output vocabulary is the output vocabulary of the second machine. It is, of course, crucial that the output vocabulary of the first machine be the same as the input vocabulary of the second machine.

Recalling the *delay* machine from last week, let's see what happens if we make the serial composition of two delay machines. Let m_1 be a delay machine with initial value $init_1$ and m_2 be a delay machine with initial value $init_2$. Then $serialCompose(m_1, m_2)$ is a new state machine, constructed by making the output of m_1 be the input of m_2. Now, imagine we feed a sequence of values,

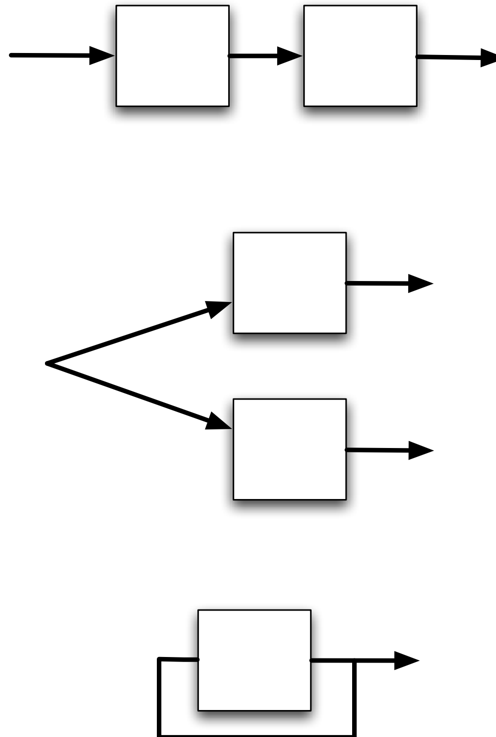

Figure 2: Serial, parallel, and feedback compositions of state machines.

$3, 2, 5, 6$, into the composite machine, m. What will come out? Let's try to understand this by making a table of the states and values at different times:

time	m_1 input	m_1 state	m_1 output	m_2 input	m_2 state	m_2 output
0	3	$init_1$	$init_1$	$init_1$	$init_2$	$init_2$
1	2	3	3	3	$init_1$	$init_1$
2	5	2	2	2	3	3
3	6	5	5	5	2	2
4	—	6	6	6	5	5

Another way to think about state machines and composition is as follows. Let the input to m_1 at time t be called $I_1[t]$ and the output of m_1 at time t be called $O_1[t]$. Then, we can describe the workings of the machine in terms of an equation:

$$O_1[t] = I_1[t-1] \quad \text{, for all values of } t > 0;$$

that is, that the output value at some time t is equal to the input value at the previous time step. You can see that in the table above. The same relation holds for the input and output of m_2:

$$O_2[t] = I_2[t-1] \quad \text{for all values of } t > 0.$$

Now, since we have connected the output of m_1 to the input of m_2, we also have that $I_2[t] = O_1[t]$ for all values of t. This lets us make the following derivation:

$$O_2[t] = I_2[t-1]$$

$$\begin{aligned} &= O_1[t-1] \\ &= I_1[t-2] \end{aligned}$$

This makes it clear that we have built a "delay by two" machine, by serially composing two single delay machines.

As with all of our systems of combination, we will be able to form the serial composition not only of two primitive machines, but of any two machines that we can make, through any set of compositions of primitive machines.

2.1.2 Parallel composition

In parallel composition, we take two machines and run them "side by side". They both take the same input, and the output of the composite machine is the pair of outputs of the individual machines. The result is a new composite machine, whose input vocabulary is the same as the input vocabulary of the component machines and whose output vocabulary is pairs of elements, the first from the output vocabulary of the first machine and the second from the output vocabulary of the second machine.

2.1.3 Feedback composition

Another important means of combination that we will make much use of later is the feedback combinator, in which the output of a machine is fed back to be the input of the same machine at the next step. The first value that is fed back is the output associated with the initial state of the machine which is being operated upon. It is crucial that the input and output vocabularies of the machine that is being operated on are the same (because the output at step t will be the input at step $t + 1$). Because we have fed the output back to the input, this machine doesn't consume any inputs; but we will treat the feedback value as an output of this machine.

Here is an example of using feedback to make a machine that counts.

We can start with a simple machine, an incrementer, that takes a number as input and returns that same number plus 1 as the output. It has very limited memory. Here is its formal description:

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \\ O &= \textit{numbers} \\ t(s, i) &= i + 1 \\ o(s) &= s \\ s_0 &= 0 \end{aligned}$$

So, if you feed this machine the stream of inputs $4, 9, 2, 7$, you will get the stream of outputs $0, 5, 10, 3, 8$. As with all of our state machines, the output depends on the input from the previous time step, so it does have memory in that sense, but it doesn't remember much. We can us it to build a machine with persistent memory by feeding the output of the incrementer back to be its input.

To make a counter, which *does* need memory, we do a feedback operation on the incrementer, connecting its output up to its input. More formally, if the input to the incrementer at time t

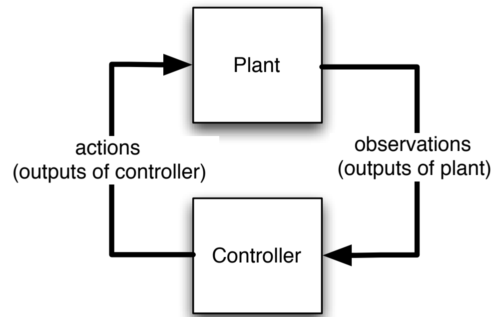

Figure 3: Two machines connected together, as for a simulator.

is $I[t]$ and the output of the incrementer at time i is $O[t]$, then the definition of the incrementer implies that $O[t] = 1 + I[t-1]$. Now, doing the feedback operation means that $I[t] = O[t]$, so that $O[t] = 1 + O[t-1]$. This is a recursive relationship, which bottoms out at $O[0]$, which is the output associated with the initial state of the original machine. We'll be spending time over the next few weeks studying machines whose behavior is defined by equations of this basic kind.

2.2 Plants and controllers

One common situation in which we combine machines is to simulate the effects of coupling a controller and a so-called "plant". A plant is a factory or other external environment that we might wish to control. In this case, we connect two state machines so that the output of the plant (typically thought of as some sensory observations) is input to the controller, and the output of the controller (typically thought of as some actions) is input to the plant. This is shown schematically in figure 3. For example, that's what happens when you build a Soar brain that interacts with the robot: the robot (and the world it is operating in) is the "plant" and the brain is the controller. We can build a coupled machine by first connecting the machines serially and then using feedback on that combination.

As a concrete example, let's think about a robot driving straight toward a wall. It has a distance sensor that allows it to observe the distance to the wall at time t, $d[t]$, and it desires to stop at some distance $d_{desired}$. The robot can execute velocity commands, and we program it to use this rule to set its velocity at time t, based on its most recent sensor reading:

$$v[t] = K(d_{desired} - d[t-1]) \ .$$

This controller can also be described as a state machine, whose input sequence is the values of d and whose output sequence is the values of v.

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \\ O &= \textit{numbers} \\ t(s,i) &= K(d_{desired} - i) \\ o(s) &= s \\ s_0 &= d_{init} \end{aligned}$$

Now, we can think about the "plant"; that is, the relationship between the robot and the world. The distance of the robot to the wall changes at each time step depending on the robot's forward velocity and the length of the time steps. Let δT be the length of time between velocity commands issued by the robot. Then we can describe the world with the equation:

$$d[t] = d[t-1] - \delta T v[t-1] \ .$$

This system can be described as a state machine, whose input sequence is the values of the robot's velocity, v, and whose output sequence is the values of its distance to the wall, d.

Finally, we can couple these two systems, as for a simulator, to get a single state machine with no inputs. We can observe the sequence of internal values of d and v to understand how the system is behaving.

State machines are such a general formalism, that a huge class of discrete-time systems can be described as state machines. The system of defining primitive machines and combinations gives us one discipline for describing complex systems. It will turn out that there are some systems that are conveniently defined using this discipline, but that for other kinds of systems, other disciplines would be more natural. As you encounter complex engineering problems, your job is to find the PCAP system that is appropriate for them, and if one doesn't exist already, invent one.

State machines are such a general class of systems that although it is a useful framework for implementing systems, we cannot generally analyze the behavior of state machines. That is, we can't make much in the way of generic predictions about their future behavior, except by running them to see what will happen.

Next week, we will look at a restricted class of state machines, whose state is representable as a bounded history of their previous states and previous inputs, and whose output is a linear function of those states and inputs. This is a *much* smaller class of systems than all state machines, but it is nonetheless very powerful. The important lesson will be that restricting the form of the models we are using will allow us to make stronger claims about their behavior.