6.01: Introduction to EECS I

Primitives, Combination, Abstraction, and Patterns

February 8, 2011

PCAP Framework for Managing Complexity

Python has features that facilitate modular programming.

- **def** combines operations into a procedure and binds a name to it
- **lists** provide flexible and hierarchical structures for data
- **variables** associate names with data
- **classes** associate data (attributes) and procedures (methods)

<table>
<thead>
<tr>
<th>procedures</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitives</td>
<td>+, *, ==, !=</td>
</tr>
<tr>
<td>Combination</td>
<td>if, while, ( f(g(x)) )</td>
</tr>
<tr>
<td>Abstraction</td>
<td><strong>def</strong></td>
</tr>
<tr>
<td>Patterns</td>
<td>higher-order procedures</td>
</tr>
<tr>
<td></td>
<td>super-classes, sub-classes</td>
</tr>
</tbody>
</table>

Programming Styles for Managing Complexity

Structure of program has significant effect on its modularity.

- **Imperative** (procedural) programming
- focus on step-by-step instructions to accomplish task
- organize program using structured conditionals and loops

- **Functional** programming
- focus on procedures that mimic mathematical functions, producing outputs from inputs without side effects
- functions are **first-class objects** used in data structures, arguments to procedures, and can be returned by procedures

- **Object-oriented** programming
- focus on collections of related procedures and data
- organize programs as hierarchies of related classes and instances

Example Program

Task: Find a sequence of operations (either increment or square) that transforms the integer \( i \) (initial) to the integer \( g \) (goal).

Example: applying the sequence

```
increment increment increment square
```
to 1 yields 16

apply **increment** to 1 → 2
apply **increment** to 2 → 3
apply **increment** to 3 → 4
apply **square** to 4 → 16

Check Yourself

What is the minimum length sequence of increment and square operations needed to transform 1 to 100?

1: <4  2: 4  3: 5  4: 6  5: >6
6.01: Introduction to EECS I

**Imperative (Procedural) Programming**

Solve the previous problem by writing an imperative program to step through all possible sequences of length 1, 2, 3, ...

def increment(n):
    return n+1

def square(n):
    return n**2

def findSequence(initial,goal):
    # construct list of "candidates" of form ('1 increment increment', 3)
    candidates = [(str(initial), initial)]
    # loop over sequences of length "i" = 1, 2, 3, ...
    for i in range(1,goal-initial+1):
        newCandidates = []
        # construct each new candidate by adding one operation to prev candidate
        for (a,r) in [('1 increment','increment'), ('1 square','square')]:
            newCandidates.append((a+r(result)))
        print i,' ',newCandidates[-1]
        if newCandidates[-1] == goal:
            return newCandidates[-1]
        candidates = newCandidates

answer = findSequence(1,100)
print 'answer =', answer

4 : ('1 square increment square increment', 5)
4 : ('1 square increment square increment', 16)
4 : ('1 square increment increment square increment', 3)
4 : ('1 square increment square increment', 4)
4 : ('1 square increment square increment', 2)
4 : ('1 square increment increment square increment', 1)
5 : ('1 increment increment increment increment increment increment', 6)
5 : ('1 increment increment increment increment increment square increment', 25)
5 : ('1 increment increment increment increment square increment', 11)
5 : ('1 increment increment increment square increment increment square increment', 100)

answer = ('1 increment increment increment square increment square increment', 100)

**Functional Programming**

This version focuses on functions as primitives.

def apply(opList,arg):
    if len(opList)==0:
        return arg
    else:
        return apply(opList[1:],opList[0](arg))

def addLevel(opList,fctList):
    return [op[fct] for fct in fctList for op in opList]

def findSequence(initial,goal):
    opList = [(i,goal-initial+i)]
    for i in range(1,goal-initial+1):
        opList = addLevel(opList,[increment,square])
    for seq in opList:
        if apply(seq,initial)==goal:
            return seq

answer = findSequence(1,100)
print 'answer =', answer

 voyeur = [function increment at 0xb777f34>, <function increment at 0xb777f34>, <function square at 0xb777f224>, <function square at 0xb777f224>]

**Imperative (Procedural) Programming**

This imperative version of the program has three levels of looping.

def findSequence(initial,goal):
    # construct list of "candidates" of form ('1 increment increment', 3)
    candidates = [(str(initial), initial)]
    # loop over sequences of length "i" = 1, 2, 3, ...
    for i in range(1,goal-initial+1):
        newCandidates = []
        # construct each new candidate by adding one operation to prev candidate
        for (a,r) in [('1 increment','increment'), ('1 square','square')]:
            newCandidates.append((a+r(result)))
        print i,' ',newCandidates[-1]
        if newCandidates[-1] == goal:
            return newCandidates[-1]
        candidates = newCandidates

answer = findSequence(1,100)
print 'answer =', answer

4 : ('1 increment increment increment increment increment', 2)
4 : ('1 increment increment increment increment', 3)
4 : ('1 increment increment increment square increment', 4)
4 : ('1 increment increment increment square', 5)
4 : ('1 increment increment square increment', 6)
4 : ('1 increment increment square increment square', 7)
4 : ('1 increment increment square increment square', 256)
4 : ('1 increment increment square increment increment square increment', 11)
4 : ('1 increment increment square increment increment square increment', 16)
4 : ('1 increment increment square increment increment square increment', 9)
4 : ('1 increment increment square increment increment square increment', 10)
4 : ('1 increment increment square increment increment square increment', 16)
4 : ('1 increment increment square increment increment square increment', 81)
4 : ('1 increment increment square increment increment square increment', 16)
4 : ('1 increment increment square increment increment square increment', 256)
4 : ('1 increment increment square increment increment square increment', 9)

**Functional Programming**

The answer is now a list of functions.

def apply(opList,arg):
    if len(opList)==0:
        return arg
    else:
        return apply(opList[1:],opList[0](arg))

def addLevel(opList,fctList):
    return [op[fct] for fct in fctList for op in opList]

def findSequence(initial,goal):
    opList = [(i,goal-initial+i)]
    for i in range(1,goal-initial+1):
        opList = addLevel(opList,[increment,square])
    for seq in opList:
        if apply(seq,initial)==goal:
            return seq

answer = findSequence(1,100)
print 'answer =', answer

 voyeur = [function increment at 0xb777f34>, <function increment at 0xb777f34>, <function square at 0xb777f224>, <function square at 0xb777f224>]

This approach is straightforward, but nested loops can be confusing.
Challenge is to get the indices right.
Functional Programming

The functions `apply` and `addLevel` are easy to check.

```python
def apply(opList, arg):
    if len(opList) == 0:
        return arg
    else:
        return apply(opList[1:], opList[0](arg))

def addLevel(opList, fctList):
    return [x + [y] for y in fctList for x in opList]
```

```python
>>> apply([], 7)
7
>>> apply([increment], 7)
8
>>> apply([square], 7)
49
>>> apply([increment, square], 7)
64
>>> addLevel([[increment]], [increment, square])
[<function increment at 0xb7480aac>,
 <function increment at 0xb7480aac>]
```

Greater modularity reduces complexity and simplifies debugging.

Recursion

Express solution to problem in terms of simpler version of problem.

Example: raising a number to a non-negative integer power

\[ b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n - 1} & \text{if } n > 0 
\end{cases} \]

Functional notation:

\[ f(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 b f(n - 1) & \text{if } n > 0 
\end{cases} \]

Python implementation:

```python
def exponent(b, n):
    if n == 0:
        return 1
    else:
        return b * exponent(b, n - 1)
```

Invoking `exponent(2, 6)` generates 6 more invocations of `exponent`.

```python
def exponent(b, n):
    if n == 0:
        return 1
    elif n % 2 == 1:
        return b * exponent(b, n - 1)
    else:
        return exponent(b, n / 2) ** 2
```

Recursive Exponentiation

Invoking `exponent(2, 6)` generates 6 more invocations of `exponent`.

```python
def exponent(b, n):
    if n == 0:
        return 1
    else:
        return b * exponent(b, n - 1)
```

```python
def fastExponent(b, n):
    if n == 0:
        return 1
    elif n % 2 == 1:
        return b * fastExponent(b, n - 1)
    else:
        return fastExponent(b, n / 2) ** 2
```

Fast Exponentiation

There is a straightforward way to speed this process:

If \( n \) is even, then square the result of raising \( b \) to the \( n/2 \) power.

\[ b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n - 1} & \text{if } n \text{ odd} \\
 \left( b^{n/2} \right)^2 & \text{otherwise} 
\end{cases} \]

Functional notation:

\[ f(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 b f(n - 1) & \text{if } n \text{ odd} \\
 \left( f(n/2) \right)^2 & \text{otherwise} 
\end{cases} \]

Number of invocations increases in proportion to \( n \) (i.e., linearly).
Check Yourself

```python
def fastExponent(b,n):
    if n==0:
        return 1
    elif n%2==1:
        return b*fastExponent(b,n-1)
    else:
        return fastExponent(b,n/2)**2
```

How many invocations of `fastExponent` is generated by `fastExponent(2,10)`?

1. 10
2. 8
3. 7
4. 6
5. 5

Functional approach is “expressive.”

Back to the Earlier Example

Task: Find a sequence of operations (either `increment` or `square`) that transforms the integer `i` (initial) to the integer `g` (goal).

**Imperative** (procedural) approach ✓

**Functional** approach ✓

**Object-oriented** approach

---

Towers of Hanoi

Transfer a stack of disks from post A to post B by moving the disks one-at-a-time, without placing any disk on a smaller disk.

```python
def Hanoi(n,A,B,C):
    if n==1:
        print 'move from ' + A + ' to ' + B
    else:
        Hanoi(n-1,A,C,B)
        Hanoi(1,A,B,C)
        Hanoi(n-1,C,B,A)
```

Recursive solution is “expressive” (also simple and elegant).

OOP

Represent all possible sequences in a tree.

Define an object to represent each of these “nodes”:

```python
class Node:
    def __init__(self,parent,action,answer):
        self.parent = parent
        self.action = action
        self.answer = answer
    def path(self):
        if self.parent == None:
            return [(self.action, self.answer)]
        else:
            return self.parent.path() + [(self.action, self.answer)]
```

---

Programming Styles for Managing Complexity

Task: Find a sequence of operations (either `increment` or `square`) that transforms the integer `i` (initial) to the integer `g` (goal).

**Imperative** (procedural) approach
- structure of search was embedded in loops

**Functional** approach
- structure of search was constructed in lists of functions

**Object-oriented** approach
- structure of search was constructed from objects

Structure of program has significant effect on its modularity.

Now consider abstractions at even higher levels.

---

Focus on constructing objects that represent pieces of the solution.

More later, when we focus on effective search strategies.
Controlling Processes

Programs that control the evolution of processes are different.

Examples:
- bank accounts
- graphical user interfaces
- controllers (robotic steering)

We need a different kind of abstraction.

State Machines

Organizing computations that evolve with time.

On the $n$th step, the system
- gets input $i_n$
- generates output $o_n$ and
- moves to a new state $s_{n+1}$

Output and next state depend on input and current state

Explicit representation of stepwise nature of required computation.

State Machines

Example: Turnstile

Inputs = \{coin, turn, none\}
Outputs = \{enter, pay\}
States = \{locked, unlocked\}

$nextState(s, i) = \begin{cases} 
\text{unlocked} & \text{if } i = \text{coin} \\
\text{locked} & \text{if } i = \text{turn} \\
\text{otherwise} & 
\end{cases}$

$s_0 = \text{locked}$

State-transition Diagram

Graphical representation of process.
- Nodes represent states
- Arcs represent transitions: label is input / output

Turn Table

Transition table.

<table>
<thead>
<tr>
<th>state</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>locked</td>
<td>none</td>
<td>pay</td>
</tr>
<tr>
<td>unlocked</td>
<td>coin</td>
<td>enter</td>
</tr>
<tr>
<td>locked</td>
<td>coin</td>
<td>enter</td>
</tr>
<tr>
<td>unlocked</td>
<td>turn</td>
<td>pay</td>
</tr>
<tr>
<td>locked</td>
<td>turn</td>
<td>pay</td>
</tr>
<tr>
<td>locked</td>
<td>coin</td>
<td>enter</td>
</tr>
<tr>
<td>locked</td>
<td>coin</td>
<td>enter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>state</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>locked</td>
<td>none</td>
<td>pay</td>
</tr>
<tr>
<td>1</td>
<td>locked</td>
<td>coin</td>
<td>enter</td>
</tr>
<tr>
<td>2</td>
<td>unlocked</td>
<td>none</td>
<td>turn</td>
</tr>
<tr>
<td>3</td>
<td>locked</td>
<td>turn</td>
<td>pay</td>
</tr>
<tr>
<td>4</td>
<td>locked</td>
<td>coin</td>
<td>enter</td>
</tr>
<tr>
<td>5</td>
<td>locked</td>
<td>coin</td>
<td>enter</td>
</tr>
<tr>
<td>6</td>
<td>locked</td>
<td>coin</td>
<td>enter</td>
</tr>
</tbody>
</table>

State Machines

The state machine representation for controlling processes
- is simple and concise
- separates system specification from looping structures over time
- is modular

We will use this approach in controlling our robots.
State Machines in Python

Represent common features of all state machines in the SM class. Represent kinds of state machines as subclasses of SM. Represent particular state machines as instances.

Example of hierarchical structure

SM Class: All state machines share some methods:
• start(self) – initialize the instance
• step(self, input) – receive and process new input
• transduce(self, inputs) – make repeated calls to step

Turnstile Class: All turnstiles share some methods and attributes:
• startState – initial contents of state
• getNextValues(self, state, inp) – method to process input

Turnstile Instance: Attributes of this particular turnstile:
• state – current state of this turnstile

SM Class

The generic methods of the SM class use startState to initialize the instance variable state. Then getNextValues is used to process inputs, so that step can update state.

class SM:
    def start(self):
        self.state = self.startState
    def step(self, inp):
        (s, o) = self.getNextValues(self.state, inp)
        self.state = s
        return o
    def transduce(self, inputs):
        self.start()
        return [self.step(inp) for inp in inputs]

Note that getNextValues should not change state. The state is managed by start and step.

Turnstile Class

All turnstiles share the same startState and getNextValues.

class Turnstile(SM):
    startState = 'locked'
    def getNextValues(self, state, inp):
        if inp == 'coin':
            return ('unlocked', 'enter')
        elif inp == 'turn':
            return ('locked', 'pay')
        else:
            return ('locked', 'pay')

Accumulator

class Accumulator(SM):
    startState = 0
    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
Check Yourself

```python
>>> a = Accumulator()
>>> a.start()
>>> a.step(7)
>>> b = Accumulator()
>>> b.start()
>>> b.step(10)
>>> a.step(-2)
>>> print a.state, a.getNextValues(8, 13), b.getNextValues(8, 13)
```

What will be printed?

1: 5 (18, 18) (23, 23)
2: 5 (21, 21) (21, 21)
3: 15 (18, 18) (23, 23)
4: 15 (21, 21) (21, 21)
5: none of the above

State Machine Combinators

State machines can be combined for more complicated tasks.

**Cascade(M1, M2)**
```
input: [i1, i2]  output: [o1, o2]
M1  M2
```

**Parallel(M1, M2)**
```
input: [i1, i2]  output: [o1, o2]
M1  M1
```

**Feedback(M1)**
```
input: [i1, i2]  output: [o1, o2]
M1  M1
```

Check Yourself

```python
>>> a = Accumulator()
>>> b = Accumulator()
>>> c = Cascade(a, b)
>>> print c.transduce([7, 3, 4])
```

What will be printed?

1: [7, 3, 4]
2: [7, 10, 14]
3: [7, 17, 31]
4: [0, 7, 17]
5: none of the above

This Week

**Software lab:** Practice with simple state machines

**Design lab:** Controlling robots with state machines

**Homework 1:** Symbolic calculator