Week 8

Sun Jun

with slides from Stanley
Composition is a powerful way to build complex systems.

**PCAP** framework to manage complexity.

<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>def</td>
</tr>
<tr>
<td>Patterns</td>
<td>high-order procedures</td>
</tr>
</tbody>
</table>

We will develop compositional representations throughout.
- software systems, signals and systems, circuits
- (if we have time) probability and planning
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>def</td>
</tr>
<tr>
<td>Patterns</td>
<td>high-order procedures</td>
</tr>
<tr>
<td></td>
<td>numbers, booleans, strings</td>
</tr>
<tr>
<td></td>
<td>lists, dictionaries, objects, classes</td>
</tr>
<tr>
<td></td>
<td>classes</td>
</tr>
<tr>
<td></td>
<td>super-classes, sub-classes</td>
</tr>
</tbody>
</table>
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 $S_n$

Output and next state depend on input and current state

Explicit representation of stepwise nature of required computation.
Vending Machine

100/(0, coke, 0)

50/(50,--,0)

50/(0, coke, 0)

100/(0, coke, 50)
Example: Turnstile

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

nextState(s, i) = unlocked if \(i = \text{coin}\)
nextState(s, i) = locked if \(i = \text{turn}\)
nextState(s, i) = s otherwise

output(s, i) = enter if nextState(s, i) = unlocked
output(s, i) = pay otherwise

\(S_0 = \text{locked}\)
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>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>locked</td>
<td>locked</td>
<td>unlocked</td>
<td>unlocked</td>
<td>locked</td>
<td>locked</td>
<td>unlocked</td>
</tr>
<tr>
<td>input</td>
<td>none/pay</td>
<td>coin/enter</td>
<td>none/enter</td>
<td>turn/pay</td>
<td>turn/pay</td>
<td>coin/enter</td>
<td>coin/enter</td>
</tr>
<tr>
<td>output</td>
<td>none/pay</td>
<td>coin/enter</td>
<td>none/enter</td>
<td>turn/pay</td>
<td>turn/pay</td>
<td>coin/enter</td>
<td>coin/enter</td>
</tr>
</tbody>
</table>
Transition/Output Table

Transition Table

<table>
<thead>
<tr>
<th></th>
<th>none</th>
<th>turn</th>
<th>coin</th>
</tr>
</thead>
<tbody>
<tr>
<td>locked</td>
<td>locked</td>
<td>locked</td>
<td>unlocked</td>
</tr>
<tr>
<td>unlocked</td>
<td>unlocked</td>
<td>locked</td>
<td>unlocked</td>
</tr>
</tbody>
</table>

Output Table

<table>
<thead>
<tr>
<th></th>
<th>none</th>
<th>turn</th>
<th>coin</th>
</tr>
</thead>
<tbody>
<tr>
<td>locked</td>
<td>pay</td>
<td>pay</td>
<td>enter</td>
</tr>
<tr>
<td>unlocked</td>
<td>enter</td>
<td>pay</td>
<td>enter</td>
</tr>
</tbody>
</table>
Problem Wk.8.1.3: WK8 CS, Qs3, State Machines
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.
Modular Design with State Machines

Break complicated problems into parts.

Map: black and red parts.
Plan: blue path, with **heading** determined by first line segment.
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
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**.

```python
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')
        elif state == 'locked':
            return ('locked', 'pay')
        else:
            return ('unlocked', 'enter')
Turn, Turn, Turn

A particular turnstyle \texttt{ts} is represented by an instance.

testInput = [None, 'coin', None, 'turn', 'turn', 'coin', 'coin']
ts = Turnstile()
\texttt{ts.transduce} (testInput)

Start state: ______

In: ______  Out: ______  Next State: ______
In: ______  Out: ______  Next State: ______
In: ______  Out: ______  Next State: ______
In: ______  Out: ______  Next State: ______
In: ______  Out: ______  Next State: ______
In: ______  Out: ______  Next State: ______
In: ______  Out: ______  Next State: ______
In: ______  Out: ______  Next State: ______
In: ______  Out: ______  Next State: ______
A particular turnstyle \textbf{ts} is represented by an instance.

ts = Turnstile()
ts.transduce (testInput, verbose=True)
Start state: locked

<table>
<thead>
<tr>
<th>In:</th>
<th>Out:</th>
<th>Next State:</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>pay</td>
<td>locked</td>
</tr>
<tr>
<td>coin</td>
<td>enter</td>
<td>unlocked</td>
</tr>
<tr>
<td>None</td>
<td>enter</td>
<td>unlocked</td>
</tr>
<tr>
<td>turn</td>
<td>pay</td>
<td>locked</td>
</tr>
<tr>
<td>turn</td>
<td>pay</td>
<td>locked</td>
</tr>
<tr>
<td>coin</td>
<td>enter</td>
<td>unlocked</td>
</tr>
<tr>
<td>coin</td>
<td>enter</td>
<td>unlocked</td>
</tr>
</tbody>
</table>

class Accumulator (SM):
    startState = 0

    def getNextValues (self, state, inp):
        return (state + inp, state + inp)
Cohort Question 1

```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)
```
Classes and Instances for Accumulator

a = Accumulator()
a.start()
a.step(7)
b = Accumulator()
b.start()
b.step(10)
a.step(-2)
Accumulator

How to define an accumulator such that the initial value is not known beforehand?

class Accumulator (SM):
    startState = 0

    def getNextValues (self, state, inp):
        return (state + inp, state + inp)
What does the following State Machine do?

```python
import libdw.sm as sm
class MySM(sm.SM):
    def __init__(self, v0):
        self.startState = v0
    def getNextValues(self, state, inp):
        return (inp, state)
```
Cohort Question

Problem Wk.8.1.5: WK8 CS, Qs5, Double Delay SM
State machines can be combined for more complicated tasks.

Cascade

\[ \text{Cascade}(m_1, m_2) \]
class Cascade(SM):
    def __init__(self, sm1, sm2):
        self.startState = (sm1.startState, sm2.startState)
        self.sm1 = sm1
        self.sm2 = sm2

    def getNextValues(self, state, inp):
        (newstate1, output1) = self.sm1.getNextValues(state[0], inp)
        (newstate2, output2) = self.sm2.getNextValues(state[1], output1)
        return ((newstate1, newstate2), output2)
Cohort Question 2

Problem Wk.8.1.2: WK8 CS, Qs2, Accumulator 2
State Machine Combinators

sm.Parallel

Parallel(m₁, m₂)

m₁

m₂

i

- o₁

- o₂
class Parallel (SM):
    def __init__(self, sm1, sm2):
        self.m1 = sm1
        self.m2 = sm2
        self.startState = (sm1.startState, sm2.startState)

def getNextValues(self, state, inp):
    (s1, s2) = state
    (newS1, o1) = self.m1.getNextValues(s1, inp)
    (newS2, o2) = self.m2.getNextValues(s2, inp)
    return ((newS1, newS2), (o1, o2))
State Machine Combinators

sm.Parallel2

Parallel2(m₁, m₂)
State Machine Combinators

sm.Feedback
class Feedback (SM):
    def __init__(self, sm):
        self.m = sm
        self.startState = self.m.startState

    def getNextValues(self, state, inp):
        (ignore, o) = self.m.getNextValues(state, 'undefined')
        (newS, ignore) = self.m.getNextValues(state, o)
        return (newS, o)
This Week

**Readings:** Chapter 4 of Digital World Notes *(mandatory!)*

**Cohort Exercises & Homework:** Practice with simple state machines & OOP (note the due dates & times)

**Cohort Session 2 & 3:** Controlling robots with state machines