Week 6: Quick Review

Sun Jun

Week 1

- Number systems
  - with base 2: 110 + 11 = 1001
  - with base 7: 65 + 32 =?
- Floating numbers, rounding errors
  - are represented in binary
  - 0.1+0.2 = ?
- Using Python to do arithmetic

Week 2

- Functions, Conditionals, While-loops, Lists, For-loops, environment diagrams
  - Exercise: write a function that separates a list of integers into two lists, one containing all positive integers from the list and the other containing all negative integers. For example, if the input is [1, -2, 3, 4, -5, 6], the output will be tuple ([1,3,4,6], [-2, -5]).

Week 3

- Nested lists, simple nested loops, tuples, Monte Carlo simulation, recursion
- Exercise: the scores of SOPH 303 is a list [s1, s2, s3, ..., s45] such that each element in the list is a list of scores for one students (i.e., for quiz 1, quiz 2, etc.). For instance, [[6,7,8,10,12], [6,3,6,8,9], [2,3,6,3,9]] is such a list assuming there are 3 students and 5 quiz. Write a function such that, given the scores, return a list of average scores of each student

Week 4

- Inputs, file read/write, try-except, string operations, dictionaries
- Exercise: See week 4, slide 68

Week 5

- OOP
- Refer to the slides for examples

State Machines

Week 6, Cohort 3

Compositional Systems | Summary

Composition is a powerful way to build complex systems.

PCAP framework to manage complexity.

	Procedures	Data
Primitives	+, *, ==, !=	numbers, booleans, strings
Combination	if, while, f(g(x))	lists, dictionaries, objects, classes
Abstraction	def	classes
Patterns	high-order procedures	super-classes, sub-classes

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)

	Procedures	Data
Primitives	+, *, ==, !=	numbers, booleans, strings
Combination	if, while, f(g(x))	lists, dictionaries, objects, classes
Abstraction	def	classes
Patterns	high-order procedures	super-classes, sub-classes

Controlling Processes

Programs that control the evolution of processes are different.

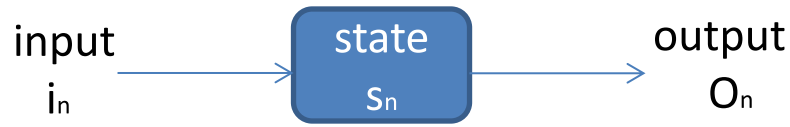
Examples:

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

We need a different kind of abstraction.

State Machines

Organizing computations that evolve with time.



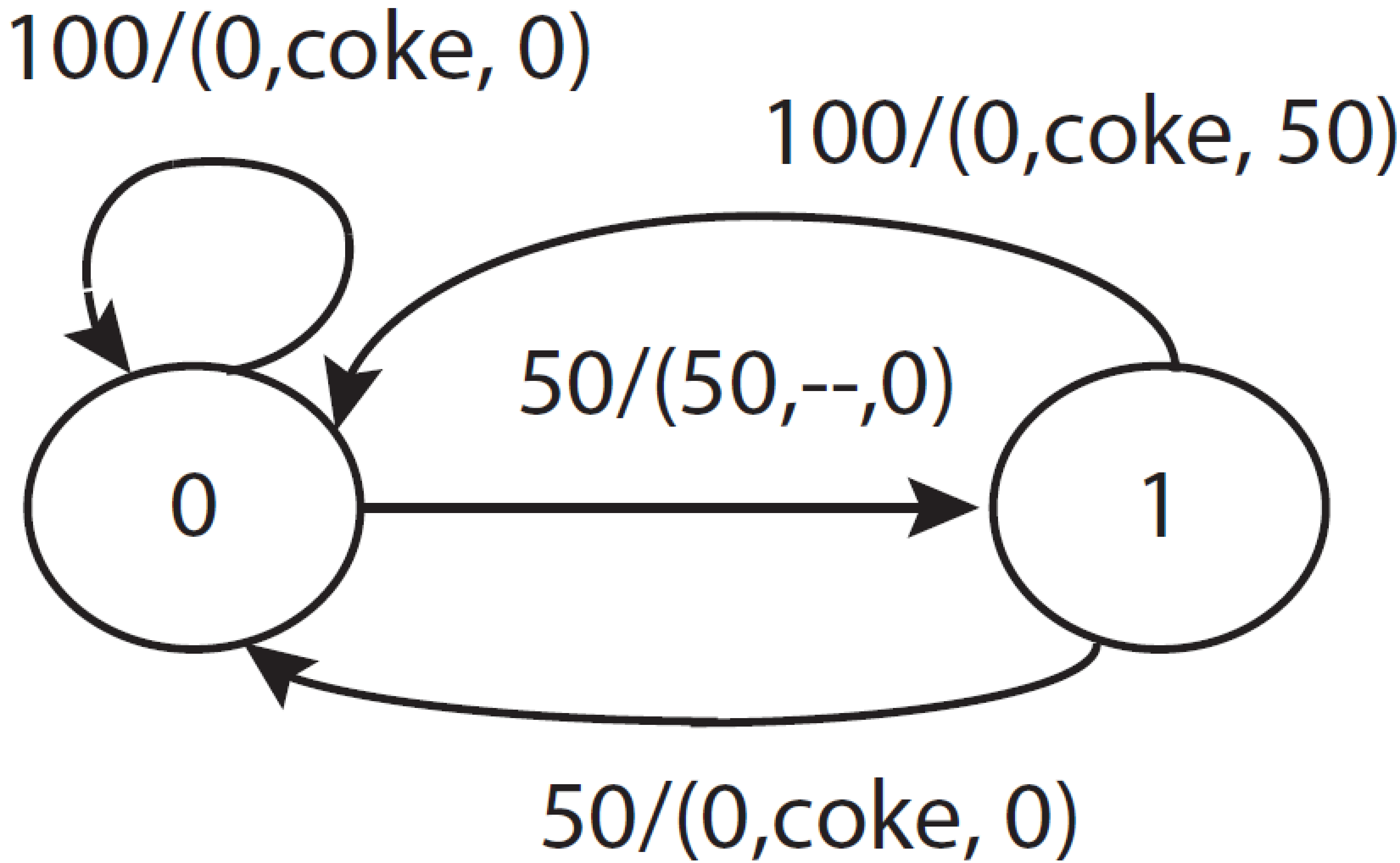
On the n-th **step**, the system

- 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



State Machines

Example: Turnstile

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

nextState(s, i) = unlocked if i = coin
nextState(s, i) = locked if i = turn
nextState(s, i) = s otherwise

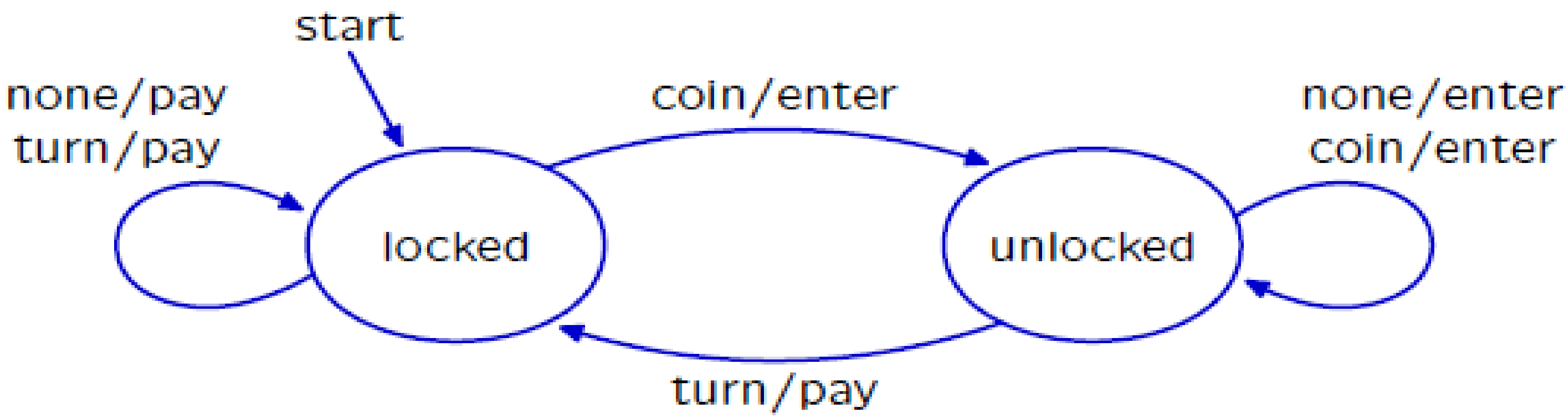
output(s,i) = enter if nextState(s,i) = unlocked
output(s,i) = pay otherwise

S_0 = locked

State-transition Diagram

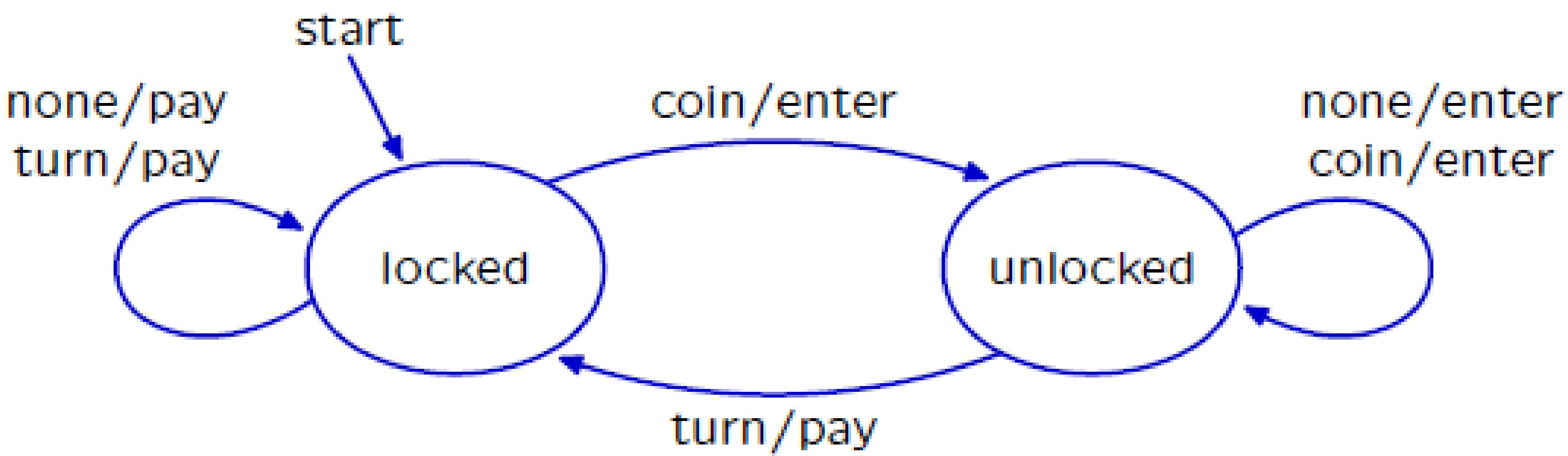
Graphical representation of process.

- Nodes represent states
- Arcs represent transitions: label is input / output



Turn Table

Transition table.



time	0	1	2	3	4	5	6
state	locked	locked	unlocked	unlocked	locked	locked	unlocked
input	none	coin	none	turn	turn	coin	coin
output	pay	enter	enter	pay	pay	enter	enter

State Machines

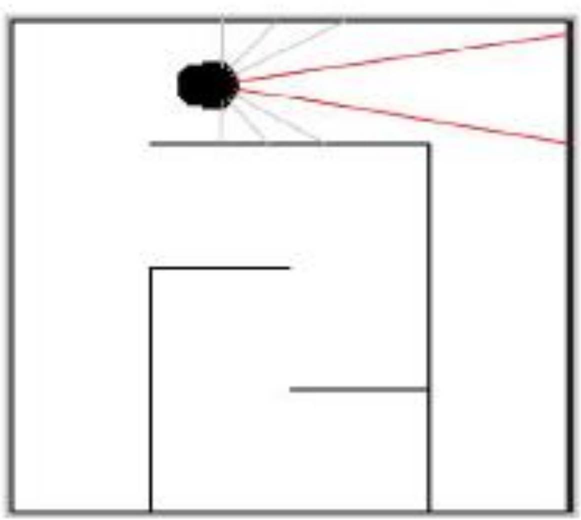
The state machine representation for controlling processes

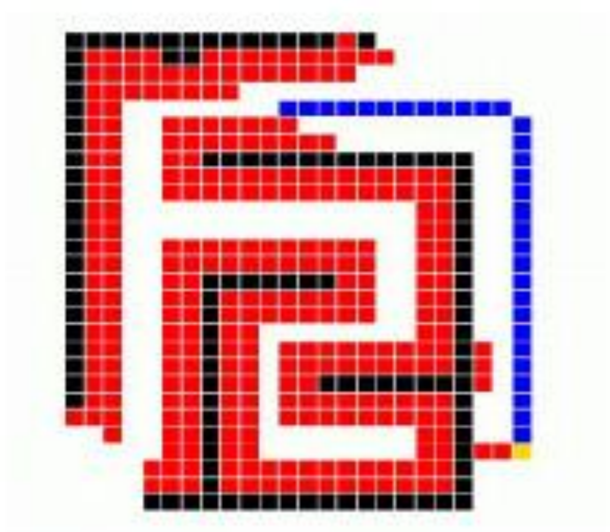
- is simple and concise
- separates system specification from looping structures over time
- is modular

We will use this approach in controlling our robots.

Modular Design with State Machines

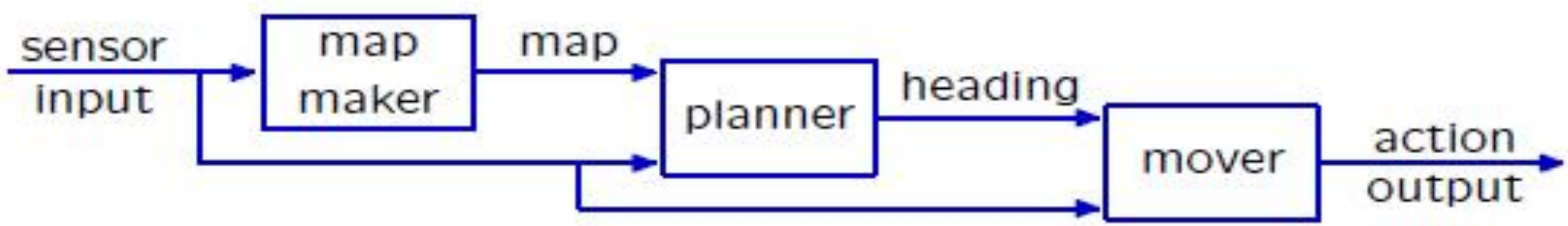
Break complicated problems into parts.





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

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’)
        elif state == ’locked’:
            return (’locked’, ’pay’)
        else:
            return (’unlocked’, ’enter’)
```

Turn, Turn, Turn

A particular turnstyle **ts** is represented by an instance.

```
testInput = [None, ’coin’, None, ’turn’, ’turn’, ’coin’, ’coin’]
ts = Turnstile()
ts.transduce (testInput)
Start state: locked
In: None        Out: pay        Next State: locked
In: coin        Out: enter      Next State: unlocked
In: None        Out: enter      Next State: unlocked
In: turn        Out: pay        Next State: locked
In: turn        Out: pay        Next State: locked
In: coin        Out: enter      Next State: unlocked
In: coin        Out: enter      Next State: unlocked
[’pay’, ’enter’, ’enter’, ’pay’, ’pay’, ’enter’, ’enter’]
```