Controlling Processes

Programs that control the evolution of processes are different.

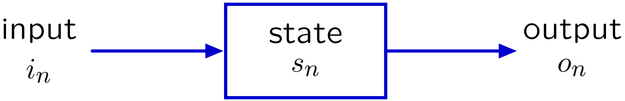
Examples:

- 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

$$\text{Inputs} = \{\text{coin}, \text{turn}, \text{none}\}$$

$$\text{Outputs} = \{\text{enter}, \text{pay}\}$$

$$\text{States} = \{\text{locked}, \text{unlocked}\}$$

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

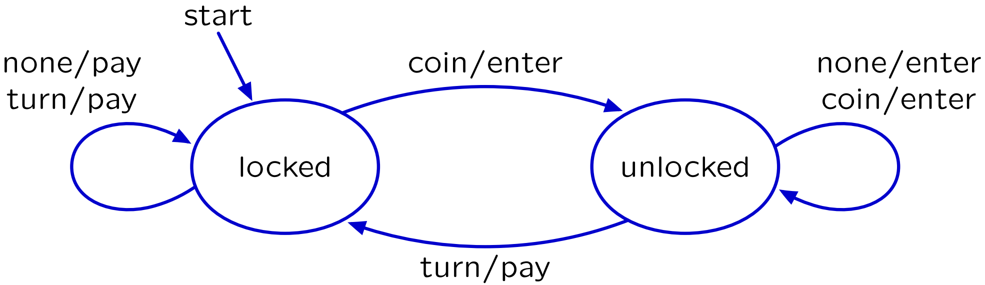
$$\text{output}(s, i) = \begin{cases} \text{enter} & \text{if nextState}(s, i) = \text{unlocked} \\ \text{pay} & \text{otherwise} \end{cases}$$

$$s_0 = \text{locked}$$

State-transition Diagram

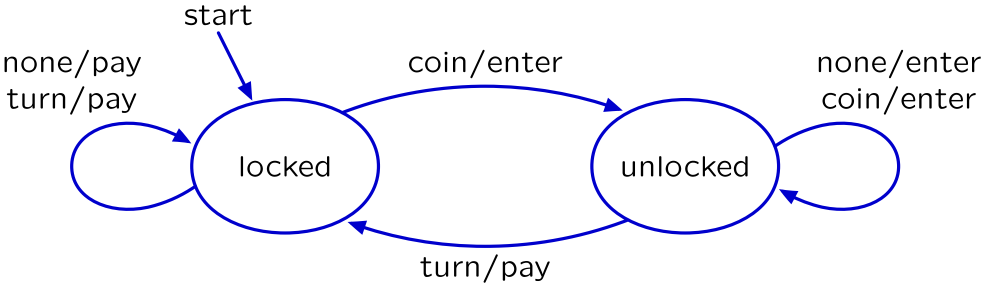
Graphical representation of process.

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



Turn Table

Transition table.



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

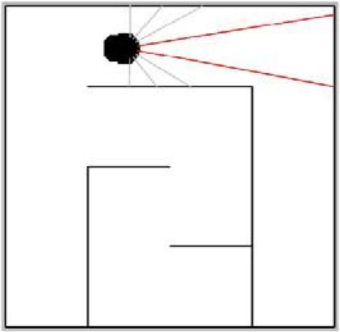
- is simple and concise
- separates system specification from looping structures over time
- is modular

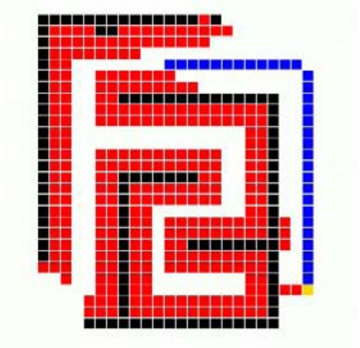
We will use this approach in controlling our robots.

Modular Design with State Machines

Break complicated problems into parts.

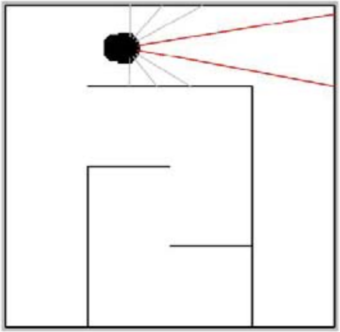
Example: consider exploration with mapping

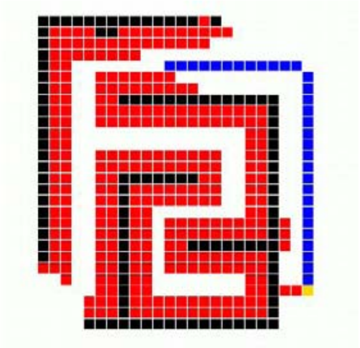




Modular Design with State Machines

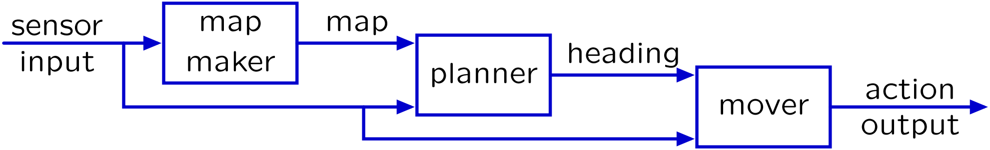
Break complicated problems into parts.





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']
```

Accumulator

```
class Accumulator(SM):
    startState = 0

    def getNextValues(self, state, inp):
        return (state + inp, state + inp)
```

Check Yourself

```
>>> 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)**

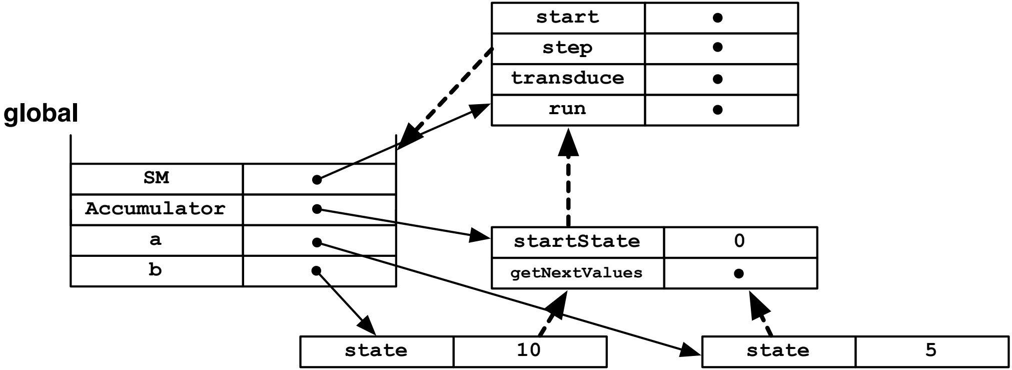
3: **15 (18, 18) (23, 23)**

4: **15 (21, 21) (21, 21)**

5: none of the above

Classes and Instances for Accumulator

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



Check Yourself

```
>>> 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? 2

1: **5 (18, 18) (23, 23)**

2: **5 (21, 21) (21, 21)**

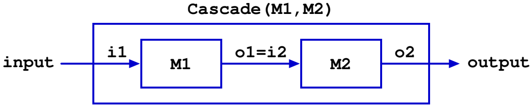
3: **15 (18, 18) (23, 23)**

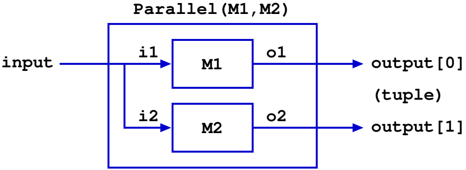
4: **15 (21, 21) (21, 21)**

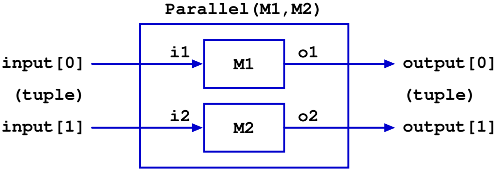
5: none of the above

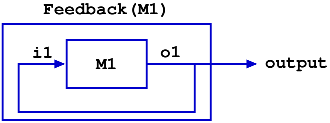
State Machine Combinators

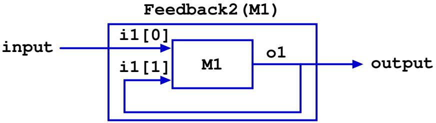
State machines can be **combined** for more complicated tasks.











Check Yourself

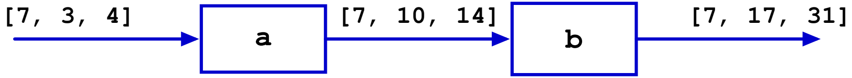
```
>>> a = Accumulator()
>>> b = Accumulator()
>>> c = Cascade(a,b)
>>> print c.transduce([7,3,4])
```

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

Check Yourself

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



Check Yourself

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

What will be printed? 3

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