Lesson 9

Admin Matters

This is Week 9 – Programming quiz is on Session 3

- A programming quiz also occurs in **Week 10**

2D Project

- Anyone here retaking this course?

Recall Lesson 8

Classes

```
class A(object):
     def __init__(self,i = 0):
          self.i = i


     def up(self):
          self.i += 1

     #the rest of the class is not shown

a = A()
b = A(2)
```

What **instance methods** are invoked when the following codes are executed?

```
     (1)   A(1)
     (2)   a(1)
     (3)   a == b
     (4)  print( a )
```

Inheritance

```
class Circle(object):
    def __init__(self,radius = 0):
        self.radius = radius


    def get_area(self):
        return math.pi*self.radius**2


    def __str__(self):

        return 'Circle: radius = ' + str(self.radius)


class Dot(Circle):
    def __init__(self,radius = 0, colour = 'green'):
        Circle.__init__(radius)
        self.colour = colour
```

Questions to consider.

1. You would like your `Dot` class to have a method to return its area. In the Dot class, do you need to write a `get_area` method?

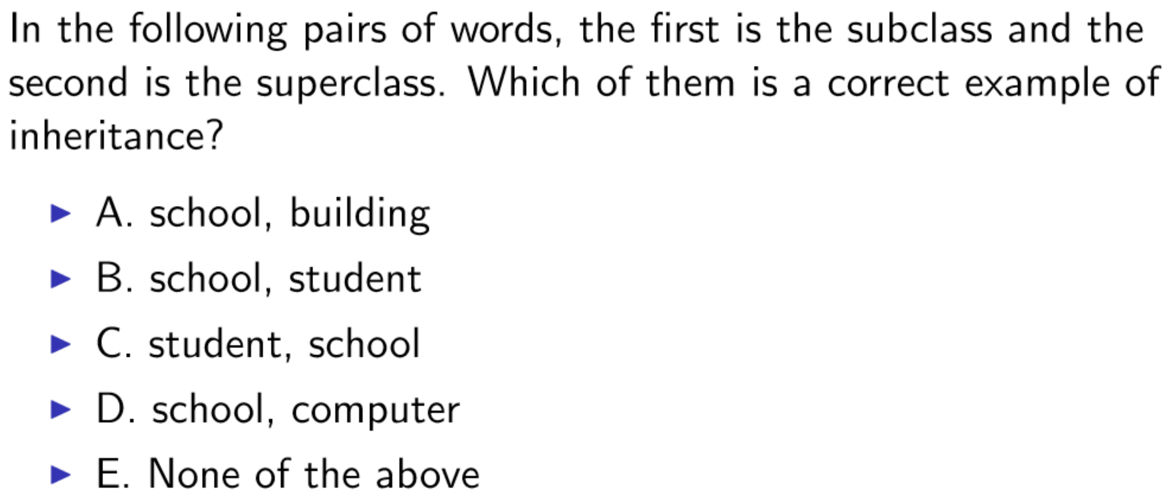
2. Does the `Dot` class have a `radius` instance attribute?

3. Do you need to override the `__str__` method?

4. Consider the following code. What is the output?

   ```
   a = Dot()
   print( isinstance(a, Circle) )
   ```

Inheritance (2) (Question from Daniel Zingaro)



In the following pairs of words, the first is the subclass and the second is the superclass. Which of them is a correct example of inheritance?

- A. school, building
- B. school, student
- C. student, school
- D. school, computer
- E. None of the above

State Machines

What are they

A way of thinking about problems where some "memory" is needed

Many applications e.g. parsing HTML, spellchecker, software, hardware

The next state depends on the previous state.

Turnstile

Draw the **State Transition Diagram** for a coin-operated turnstile which allows the user to move the turnstile when a token is deposited.

When the turnstile is **locked**:

- Pushing it causes it to remain **locked**
  and displays the message, "token please"

- Inserting a token makes it **unlocked**
  and displays the message, "please enter"

When the turnstile is **unlocked**:

- Pushing it causes it to be **locked**
  and displays the message, "token please"

- If nothing happens, it remains **unlocked**
  and displays the message, "please enter"

Write the **state transition function in a table form**

Language Acceptor

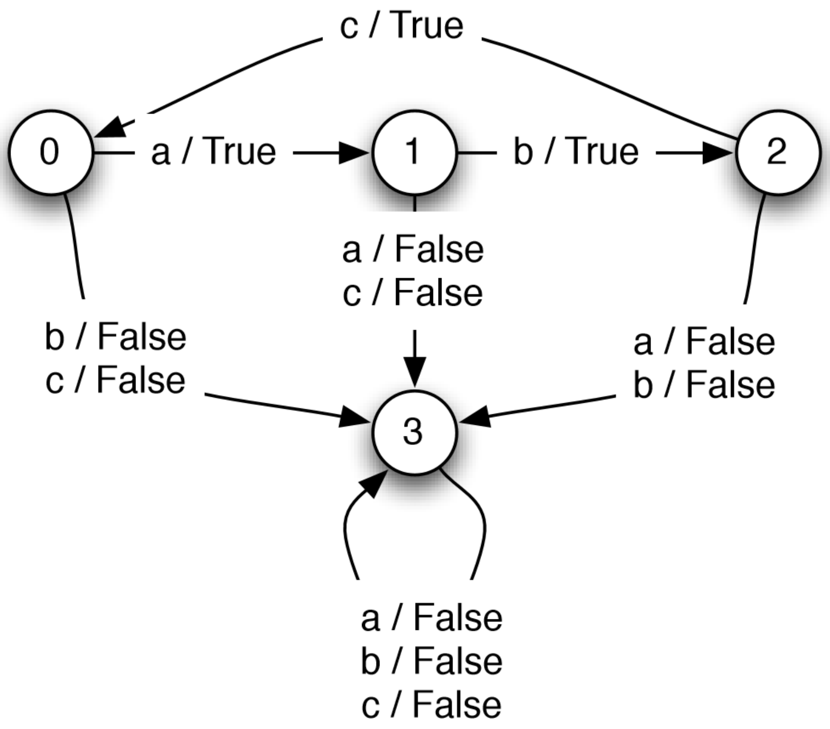


Figure 4.1: State transition diagram for language acceptor.

Q1. The current state is 0. An input *b* is given.

What is the next state?

A. State 0
B. State 1
C. State 2
D. State 3

Q2a. The current state is 1. An input *b* is given.

What is the next state?

A. State 0
B. State 1
C. State 2
D. State 3

Q2b. The current state is 1.

An input *b* is given.

What is printed on the screen?

True / False

Q3. I am currently at state 2.

The input that got me to this state

must have been *b*.

Yes / No.

Q4. I am currently at state 3.

Regardless of the input,

I will never switch to another state.

True / False

Q5. Draw the state transition function in a table form.

Programming the State Machine

Documentation here: https://tinyurl.com/dwsmclass

Import the **sm** module from libdw

```
from libdw import sm
```

What concerns us is the **SM** class,

which is a generic class with state machine methods.

Our state machines class will inherit this class.

```
class MyStateMachine(sm.SM):
```

If you used

`import libdw`

What would you fill in below?

`class MyStateMachine( ?? ):`

A) `sm.SM`
B) `sm`
C) `SM`
D) `libdw.sm.SM`

If you used

`import libdw.sm as LW`

What would you fill in below?

`class MyStateMachine( ?? ):`

A) `sm.SM`
B) `sm`
C) `LW`
D) `LW.SM`

Using the SM class

In the **SM** class, few things you have to do in your custom class

```
class MyStateMachine(sm.SM):
    #define your starting state.
    start_state =

    #this is your transition function.
    #You have to override it with your own definition
    def get_next_values(self, state, inp):

        #it must return two outputs
        return next_state, output
```

In **`getNextValues`**, do not update **`self.state`** – this is managed by **`step()`**

Then you can use the following methods inherited from SM

```
a = MyStateMachine()
a.start()           #set a.state = a.start_state
print(a.state)      #print to see the state
a.step( inp )       #call getNextValues with input inp
print(a.state)      #print to see the state
```

This can be a chore, so this is your next alternative

```
a = MyStateMachine()
list_of_inputs = [1,2,3]
#the second argument below is optional
a.transduce( list_of_inputs, verbose = True)
```

Homework Problem 1

Please read it!

```
>> inputstr = 'def f(x): # comment\n   return 1'
>>> m = CommentsSM()
>>> m.transduce(string)
[None, None, None, None, None,
None, None, None, None, None,
'#', ' ', 'c', 'o', 'm', 'm', 'e', 'n', 't',
None, None, None, None, None,
None, None, None, None, None, None, None ]
```

Questions to consider

What input changes the output?

A) #
B) \n
C) Both A & B
D) Not enough information

How many states are there?

A) 1
B) 2
C) 3
D) 4

Next, define your states by giving them meaningful names
and try to draw the state transition diagram