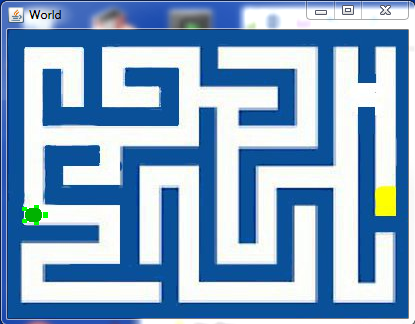
Maze

# Introduction

We will be constructing a Python program to solve a maze. The maze is shown below:



The turtle will look for the cheese (the yellow spot on the right).

The maze will be represented by a class called Maze. The class will contain the maze image as a JES picture. The image of the maze is called maze.jpg. As you follow this tutorial, do each numbered assignment. We will be using TDD to create our program. Since JES does not support the Python testing frameworks like unittest and doctest, we will be making a set of our owndddddddddddddd tests at the bottom of our program file. Separate the program from the tests with a comment line

# tests

The first task is to create our Maze class. Write a test for creating a maze as follows:

m = Maze()

Loading in the file now should produce the following error

======= Loading Progam =======

The error was:Maze

Name not found globally.

A local or global name could not be found. You need to define the function or variable before you try to use it in any way.

This is our failing test

1. Create a class called Maze to make the test pass. Include a doc string under the class definition. Load the file to insure that the test you made now passes.

Now we want to associate our maze image, maze.jpg, with the maze class. Write a failing test to look at the maze image which we will just call ‘image’ inside the maze object.

# test for image

show(m.image)

You should see the failure of this test when you load the file. The failure should look like this:

The error was:'Maze' object has no attribute 'image'

Attribute not found.

You are trying to access a part of the object that doesn't exist.

1. Load the image, maze.jpg, into self.image in the \_\_init\_\_ method. Load the file to insure your test now passes.

>>> m=Maze()

>>> show(m.image)

You should see the maze image displayed on the screen (without the turtle)

Not documented in the text book, it is possible to set the world background to this image. To do that, we use a method called setPicture. First write a test to access the world inside of the maze. To do this just try to access m.w as follows.

world = m.w

You should see the following error when you try to read in the file.

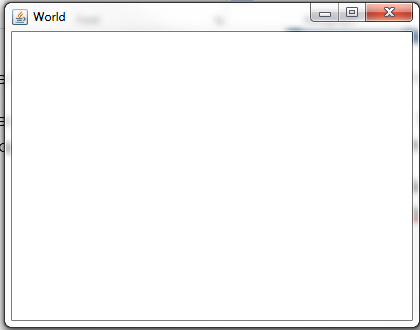
The error was:'Maze' object has no attribute 'w'

Attribute not found.

You are trying to access a part of the object that doesn't exist.

1. Create a world called w in the \_\_init\_\_ method. Use the width and height of the image to set the dimensions of the world. Load your file to insure the test for the world existence passes.

You should see a blank image appear. That is the image of our world. It will immediately be covered up by our test for the image so move the image of the maze aside to see the blank world.

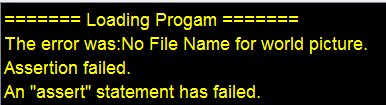


We want to see our maze appear in the world however. Write a test to check that the world has the maze as its picture

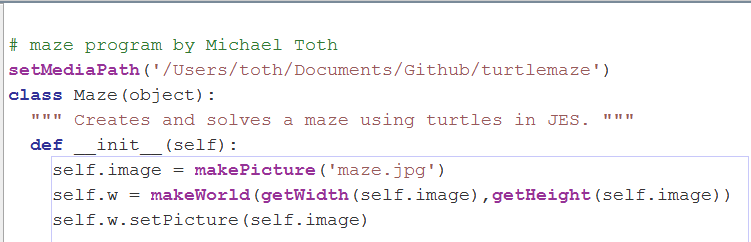
p=m.w.getPicture()

assert p.getFileName() != ‘None’, ‘No file name for world picture.’

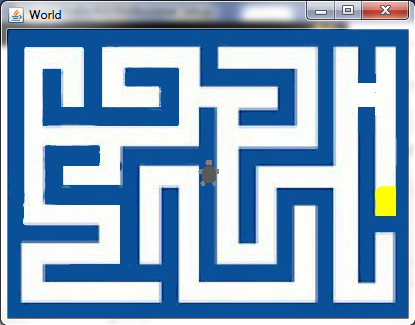
When you load the file, you should see the following error



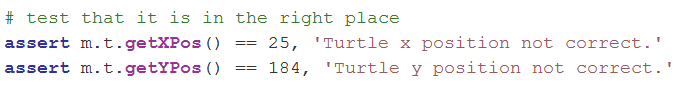
1. Now set the background to be our image using the setPicture method of our World object. Reload the file to make sure the test now passes



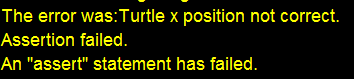
1. We need a turtle so that is the next step. Write a test to access the currently non-existent turtle to make a failing test. Create a turtle in the \_\_init\_\_ method called t. Don’t forget to precede instance variables with the word self. The turtle will be self.t and the world will be self.w. Now when you create a maze you will see the turtle but it’s on a wall in the center of the world view.



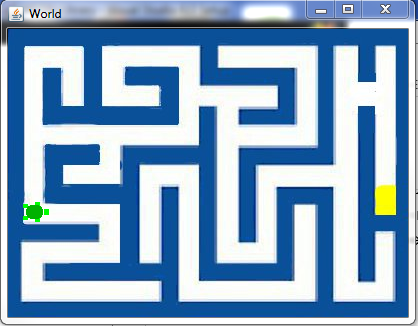
We need to move the turtle to the proper location and orientation. We want the turtle to be where the circle is and pointed to the right. Use explore(m.image) to see what the coordinates are of that spot. Write a failing test to check for the coordinates of the turtle.



The error should appear



1. Once you find the coordinates, move the mouse there and point it to the right (heading = 90). You can use the setHeading method of the Turtle object to change orientation. Make sure you use penUp and penDown to keep from drawing when you move the turtle. Now when you create an instance of Maze, you should see the turtle in the correct starting position and your test should pass.

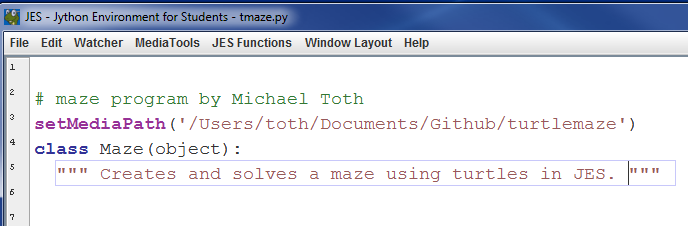
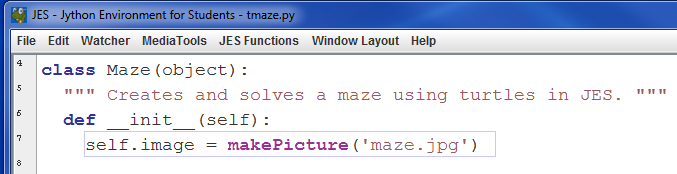
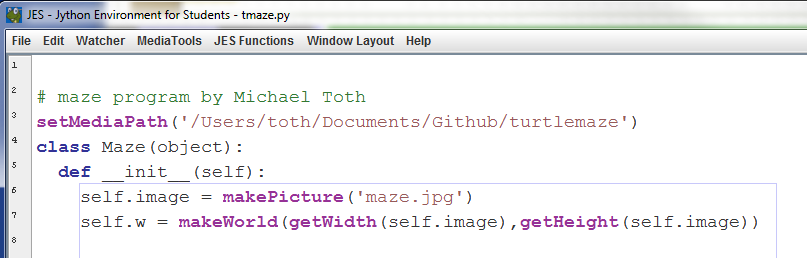


# Moving the Turtle

To move a turtle there is a method called forward. In solving the maze we want to be able to know when the turtle can and can not move. The turtle should not move through walls. We will detect a wall by looking in the image at a spot just before the turtle. We use getPixelAt and getColor to determine if we are hitting a wall. When we call getPixelAt we want to give it coordinates in front of the turtle. There are two methods for the Turtle object which give us the coordinates of the turtle; getXPos and getYPos. Write a test for the existence of colorInFront.

1. Create a method in the Maze which returns the color of the spot 20 pixels in front of the turtle. Write a test to insure it returns white for the starting point of the turtle.
2. It should return pure colors even though the colors of the image are not pure. (when it detects a wall, it should return blue, not the approximate blue color it actually is). Call the method colorInFront. Test it by calling it when the turtle is in the starting location and also when it is near a wall. You will need to check the orientation (heading) of the turtle to be sure you are looking at the spot in front of the turtle. You will also need to check if the turtle is less than 20 pixels to the edge of the image. If it is you don’t want to be trying to get the color of that non-existent pixel! If the turtle is facing an edge and is closer than 20 pixels to the edge, you should return blue assuming it is a wall. The turtle can be pointing in 4 different directions; north, east, south, and west. The heading for each of these respectively is 0, 90, 180, and 270. It can also be 0, -90, -180, and -270 if you are turning the turtle left. When you check for the orientation, be sure to look for both 90 and -270 when checking for the eastwardly direction. Do the same with the other orientations. Use the distance function to determine how close the actual color of the pixel in front of the turtle is to blue. If the distance is below 150, it’s probably blue. If the distance is less than 150, just return blue. Don’t worry about other colors yet, just blue and white.

# Assignment Solutions

1. 
2. 
3. 
4. def \_\_init\_\_(self):

self.image = makePicture('maze.jpg')

self.w = makeWorld(getWidth(self.image),getHeight(self.image))

self.w.setPicture(self.image)

self.t = makeTurtle(self.w)

penUp(self.t)

moveTo(self.t,26,182)

self.t.setHeading(90)

penDown(self.t)

1. penUp(self.t)

moveTo(self.t,26,182)

self.t.setHeading(90)

penDown(self.t)

self.w = makeWorld(getWidth(self.image),getHeight(self.image))

self.w.setPicture(self.image)

self.t = makeTurtle(self.w)

penUp(self.t)

moveTo(self.t,26,182)

self.t.setHeading(90)

penDown(self.t)

def colorInFront(self):

if self.t.getHeading() == 0:

if getYPos(self.t) < 20:

return blue

else:

xpos,ypos = self.t.getXPos(),self.t.getYPos()-20

if self.t.getHeading() == 90 or self.t.getHeading() == -270:

if getXPos(self.t) + 20 >= getWidth(self.image):

return blue

else:

xpos,ypos = self.t.getXPos()+20,self.t.getYPos()

if self.t.getHeading == 180 or self.t.getHeading() == -180:

if getYPos(self.t) >= getHeight(self.image):

return blue

else:

xpos,ypos = self.t.getXPos(),self.t.getYPos()+20

if self.t.getHeading == -90 or self.t.getHeading == 270:

if getXPos(self.t) < 20:

return blue

else:

xpos,ypos = self.t.getXPos()-20,self.t.getYPos()

p = getPixelAt(self.image,xpos,ypos)

c = getColor(p)

if distance(c,blue) < 150:

return blue

if distance(c,white) < 150:

return white

raise "Unknown color"