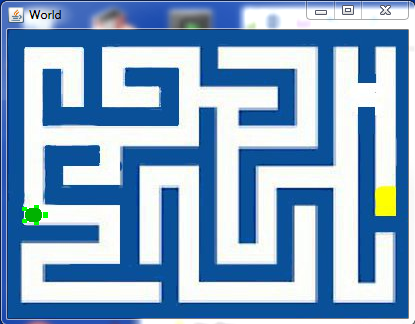
Maze

# 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 own 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 the existence of a Maze class 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. You should see the maze image displayed on the screen (without the turtle)

So far we have a Maze class and that class loads our maze.jpg image when you create an instance of it. We do not yet have the turtle world or a turtle. 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.

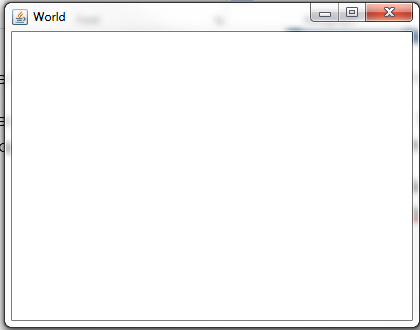
The error was:'Maze' object has no attribute 'w'

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.

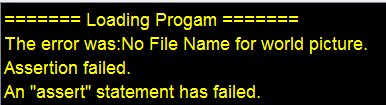


We want to see our maze appear in the world however. Write a test to check that the world has the maze as its picture

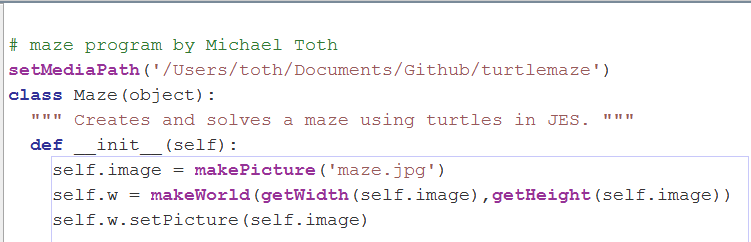
p=m.w.getPicture()

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

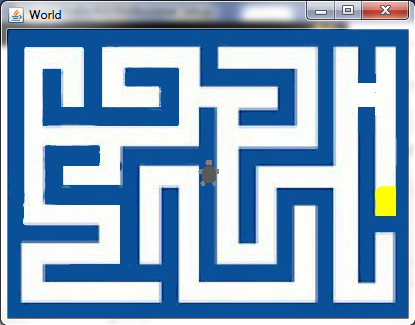
When you load the file, you should see the following error



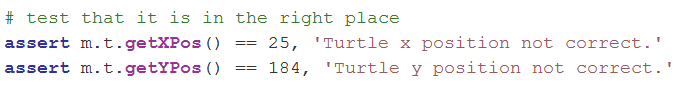
1. Not documented in the text book, it is possible to set the world background to an image. To do that, we use a method called setPicture. Now set the background to be our image using the setPicture method of our World object. Reload the file to make sure the test now passes



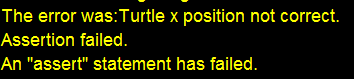
1. We need a turtle so that is the next step. Write a test to access the currently non-existent turtle to make a failing test. Then create a turtle in the \_\_init\_\_ method called t. Don’t forget to precede instance variables with the word self. The turtle will be self.t and the world will be self.w.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



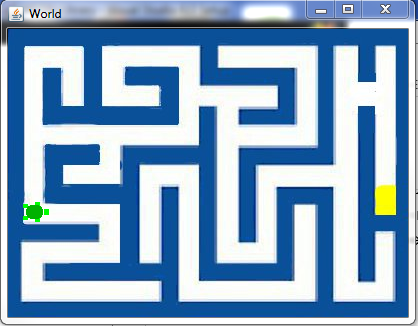
We need to move the turtle to the proper location and orientation. We want the turtle to be where the circle is and pointed to the right. Use explore(m.image) to see what the coordinates are of that spot. Write a failing test to check for the coordinates of the turtle.



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.

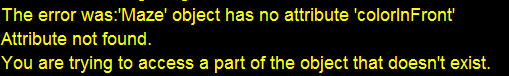


# Moving the Turtle

To solve the maze we need to move 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. For this we will create a method in Maze called colorInFront. 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.

1. Write a test for the existence of colorInFront.

As before you should see the attribute error



1. Create the method called colorInFront to fix this error.

Now we have an empty method and we want to write tests for this method called colorInFront. The first test is to see that colorInFront returns white when we are starting (because the area in front of the turtle is white).

1. Write a test to see if we get white from colorInFront so you get the following error.



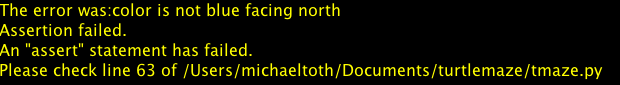
1. Add code to colorInFront so it returns the color of the spot 10 pixels in front of the turtle. Load the file to insure it returns white for the starting point of the turtle. The actual color in front of the turtle is not pure white. We will use the distance function to determine how close we are to white and if the distance is less than 150, assume it’s white.
2. We also want to see blue when we are up against a wall. Write a test so it fails when we are close to a wall. You will need to move the turtle to another location for this test.



1. Now write the code to make it pass

You may have noticed that so far we have not concerned ourselves with the direction the turtle is pointing when we check the value in front of it. We just add to the x position and look at that point. This won’t work when the turtle is facing south, west, or north.

1. Add a test for the north direction (heading = 0).



1. Now add code to make this test pass.

Keep in mind that the heading of the turtle can be either 90 or -270 for the eastward direction. For the northern direction, it is always 0. (I think)

1. Do the same kind of tests for the west and south direction.

Now we have no failures and we can detect walls in front of the turtle. It’s time to discuss the algorithm for solving the maze.

The algorithm will be recursive. Here’s a pseudo code implementation.

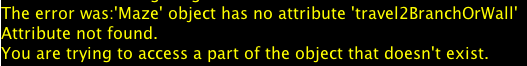
1. If the cheese is in front of the turtle, return true and play a happy sound.
2. Look in all 4 directions around the turtle.
3. If you see an empty (white) area, try to solve the maze in that direction
4. If there are no more places to move, return false
5. go to the next empty area around the turtle and try that direction
6. If it passes, return true, if there are no solutions in any of the possible directions, stop

These six steps form the basis of the solve method we will create. The part that says “solve the maze in that direction” will involve moving to the next location where a turn is possible or necessary. For instance if we hit a wall and we can turn, at that point we call solve again with the new location.

Understanding this algorithm will give us clues to our next test. We need to tell the turtle to move to the next spot where it either has to change direction like when it comes to a bend in the path, or can change direction like when there is a path to the right or left as well as straight ahead.

A method called travel2BranchOrWall will be useful. This will move the turtle forward until either it hits a wall or it sees a path to the right or the left.

1. Write a test for the existence of travel2BranchOrWall. Orient the turtle so it is facing east at the starting location. Here’s the failure.



To make this test pass with the turtle in the starting position, we know we will hit the wall before any other choices can be made.

1. Write the code to move the turtle forward until it is against a wall. You should see the turtle there at the end of the test.

We are not yet dealing with openings to the left and right. You can see if the turtle heads north from the starting location we want it to stop at the branch to the right. Use explore to find that coordinate. (I get 25,105)

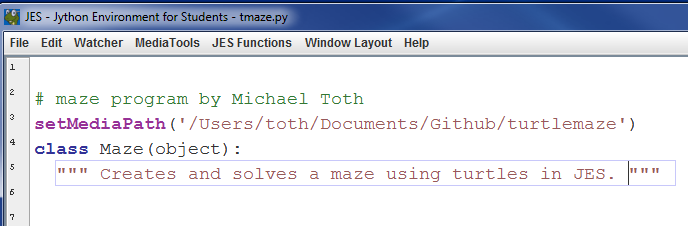
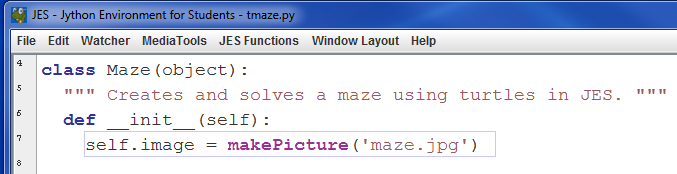
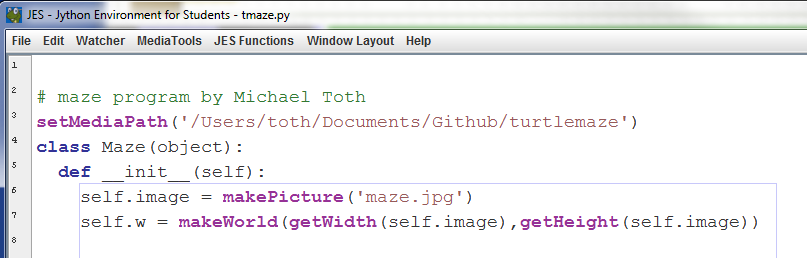
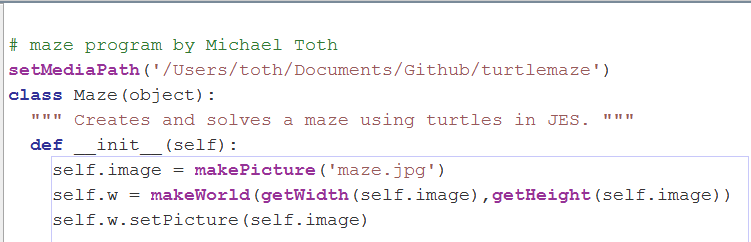


1. Write a test to check that the turtle stops at the branch when heading north from the starting location. It should fail.



1. Write the code to make this pass. This is a little trickier. With the turtle facing north at the starting location, we have a path to the right already but we don’t want to stop for that path because that path is going to be the first one we try. We don’t want to go into places we already visited. So we need to move forward enough so we don’t see that passage to the right before we start checking for that. If we move forward first until we don’t have that branch in our sights, then we can start checking for paths to the right and left. It might be convenient to have a function that gives us a description of all four directions at once and we can use that to make decisions.
2. Write a test for a function called surroundings that will return a list of 5 items. Each item can have the following possible values; ‘empty’,’wall’,’visited’,’end’.

# Assignment Solutions

1. 
2. 
3. 
4. 
5. penUp(self.t)

moveTo(self.t,25,184)

self.t.setHeading(90)

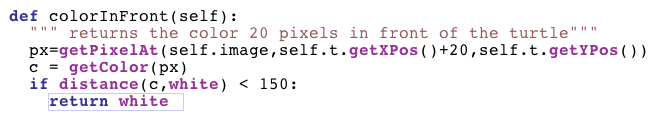
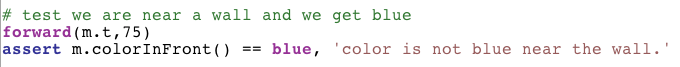
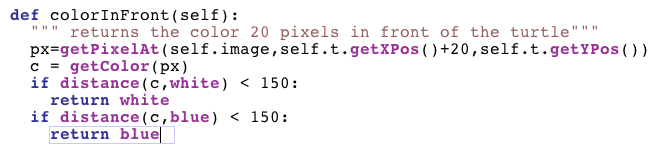
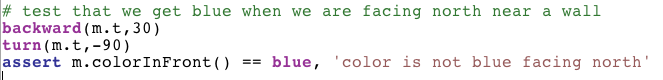
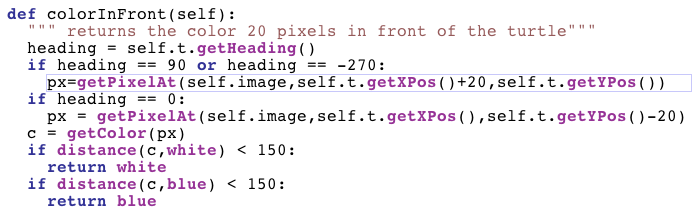
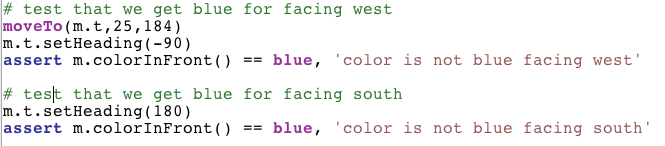
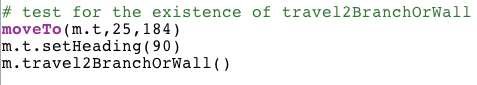
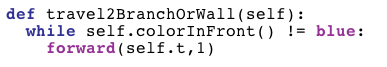
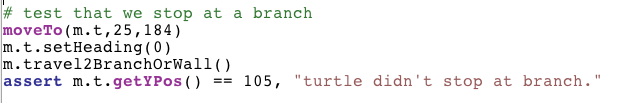
penDown(self.t)

1. penUp(self.t)

moveTo(self.t,25,184)

self.t.setHeading(90)

penDown(self.t)

1. 
2. 
3. 
4. 
5. 
6. 
7. 
8. 
9. 
10. 
11. 
12. 

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"