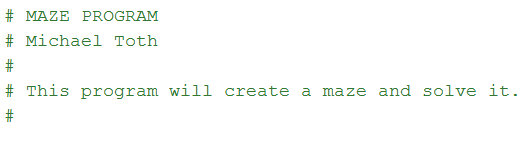
Maze 2015

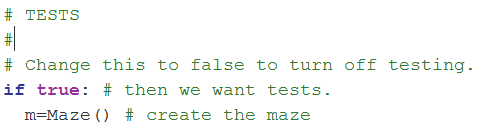
# Introduction

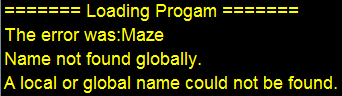
The major project for this semester is creating a maze and solving it in Python. We will use test driven development techniques. We will be using the JES framework.

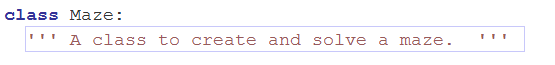
# Starting out

To begin, we want to write our first test. We will start simply with basic concepts. Normally some of these steps would not be made but we are trying to approach this from inexperience. As you get more experienced, you will learn which steps to skip. Create a file called maze.py and put your name and a description of the program as follows:

 Of course you put your own name in place of mine. Since we won’t be wanting to always run our tests and we don’t have a testing framework that operates properly in JES, we will put our own tests into the same file as our program and qualify them with an ‘if’ statement so we can turn them off if we want to. Add this to the bottom of your file.

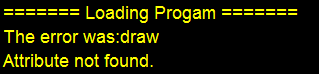
When you load this file now you should see an error. (RED CONDITION)

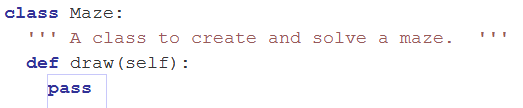
 Now we need to create the Maze class. Do this above your tests.

When you load the file now you should see no errors. (GREEN)

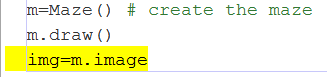
You have just take the first step in a test driven development project. You went from RED to GREEN. Now we want to add another test. Our maze must be able to be shown or displayed on the screen. A natural method to do this might be called ‘draw’. Let’s check that we have a ‘draw’ method in our class.

 When you add this to your tests, you should see the error (RED CONDITION)

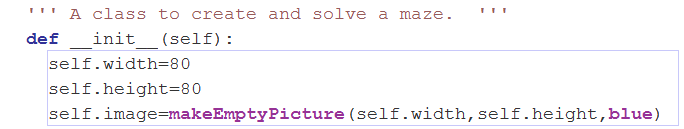
 Now add the draw method. It doesn’t need to do anything.

You should now be able to load without error. Notice that we are not actually writing ‘tests’ in the sense that we are checking for values, we are just watching exceptions occur as we add lines to our testing area. Normally we will use assertions to check for particular aspects of these objects. That will come in time as we add to our class.

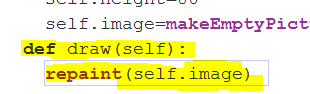
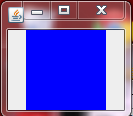
Next we want to make sure when we create a maze, we have nothing but walls except for where we are located in the maze. So we should have a totally blue square with one small white square representing our starting location. But before we can deal with that, we need to have an image to work with in the maze class. We want this image to exist from the beginning when we create our maze so in the \_\_init\_\_ method we will use makeEmptyPicture() which is supplied by JES. Write a test to see if we have an image in the class.

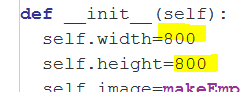
You should see the error 

since we don’t have an image created yet. Now add the \_\_init\_\_ method which will create the image and while we are at it, let’s add a width and height property to the maze.

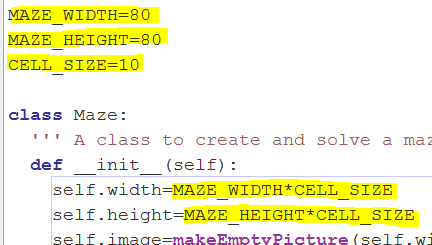
 Now you should no longer see the error.

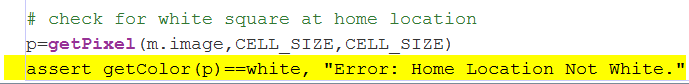
Some aspects of test driven development do not easily lend themselves to user interface or GUI behavior. For instance, if you want to verify that the image is actually displayed on the screen when you invoke draw() you need to just look rather than automate that. Right now, nothing happens when you call draw(). So now add the code to make sure you see the image when you invoke draw(). We will use repaint() rather than show() because show() creates a new window each time you use it. After a while we will be up to our elbows in windows if we don’t stop that.

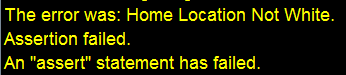
 Luckily repaint() will work properly even if the image has not yet been shown. Now when you load the file, you should see the image.  It’s a good thing we put that test in which draws the image. We can immediately see this is too small! We will never be able to work with a maze this small. Our maze will be constructed of small blocks rather than pixels so we need to adjust the size accordingly.

 Now we have a reasonably sized maze image suitable for an 80x80 grid of 10x10 squares.

We are currently in the GREEN state but now it might be useful to do a little refactoring. We have hard numbers for the size and we may want to generalize that in the future for different sized mazes and also we may want to use the dimensions later in our work, so let’s define some values for these.

 When you load the file you should still see no errors. The important part of refactoring is verifying that you have not broken any of the existing tests and you stay in the GREEN state until you write your next test. Our next test will be to see if we have the white square in our ‘home’ location. Let’s call the home location (1,1) corresponding to the upper left hand corner of the maze. We will not start at 0,0 because our boundary cells are going to be treated differently than the rest. This way we don’t have to worry about going out of bounds in our maze. Location 1,1 corresponds to one column over and one row down. Checking the color at pixel CELL\_SIZE,CELL\_SIZE should do it. Add this to your tests.

 Notice we put the message in to let us know what the problem is. Now when you load the program, you should see this.

 We knew this would fail since we created an all blue image. Let’s make this test pass by putting the white square in during initialization. Add this code to your \_\_init\_\_ method.

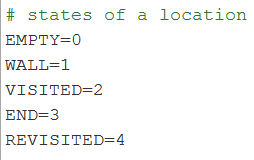


Now you should see no errors. Not much of a maze yet of course. What’s the next step? Much of this is where the art of programming comes in. You need to make a choice about what you want to look for. We know we want to first create a maze. To do this we might have the idea of ‘digging’ in the blue area and making a white path. We need a way to represent the architecture of the maze. Up until now we have only made an image. Let’s represent our maze as a matrix. Each entry in the matrix will correspond to one cell in the maze. The dimensions of the matrix will be MAZE\_WIDTH x MAZE\_HEIGHT. You can create a matrix with one line of Python. The values in each of the matrix cells corresponds to the cell being either empty, or visited, or a wall, or revisited. We will choose values to represent these different states. The value 0 will be an empty cell and the value 1 will be a wall. We will call this matrix ‘matrix’! Check to see if matrix exists and cell 1,1 is empty.

 We should now be in the RED condition.

 Maybe not the result you expected since we didn’t yet create the matrix. Let’s do that in the initialization (\_\_init\_\_) method.

 Now we are back to GREEN. Time to refactor. It’s not nice to use 1 for wall and 0 for empty because you have to remember 1 is wall and 0 is empty. Let’s define our states so we can just remember them by name. Add this above your Maze class definition.

 Now we can just use the names directory instead of values. Use this in place of the values you just entered. (Note: it’s common but not essential to use capital letters when defining constant values.)

 Verify you are in the GREEN condition.

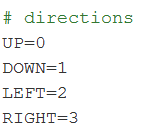
We don’t have anything yet telling us where we are in the maze. We will call it ‘location’ and it will be a duple of x and y values for the current location in the maze. Initially of course it should be (1,1).

 Notice your error now.

 You should be familiar with this one by now. Let’s add the code to the \_\_init\_\_ method.

 Now your tests should pass.

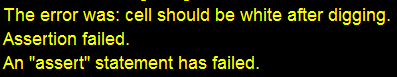
We want to be able to dig a path through the walls. Let’s make a method called dig. Dig will take an argument for a direction and will return True if it was successful. To be successful you have to dig into a wall. You don’t want to think you have successfully dug in a given direction if it is already empty or if it is at the boundary. You only want to think you successfully dug if there was a wall which is now replaced by an empty state. Now we need to determine how to represent the four directions; up, down, left, and right. Let’s say up=0, down=1, left=2, and right=3. We will do the same thing we did for the states of a cell. Let’s define the directions so we don’t have to remember them.

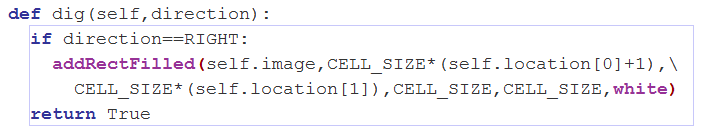
Put this code just below the states you defined earlier.

Now test for the dig method.

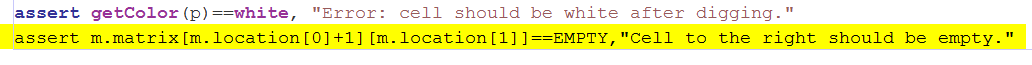
 This should give you the usual “Attribute not found” error.

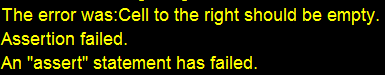
Add the dig method to your Maze class. and you should be in the GREEN condition again. But dig does nothing! That’s ok. We just first wanted to make it exist. Now the next test will be to see if it actually worked. We check for that by testing the color at 20,10.

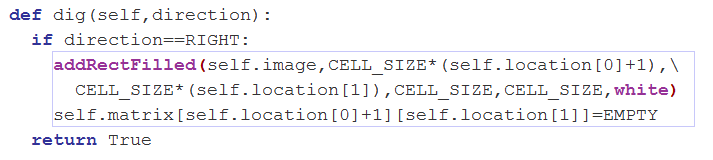
 Notice we use our constant for cell size. Your error should be . Now add the code to make it pass.

 Notice that we only worry about the RIGHT direction because we write the *minimum amount of code needed* to make the test pass. We use our location as a guide as well as our constants that we defined earlier. You should now be in the GREEN condition.

Next we verify that the matrix also reflects the state of that cell properly.

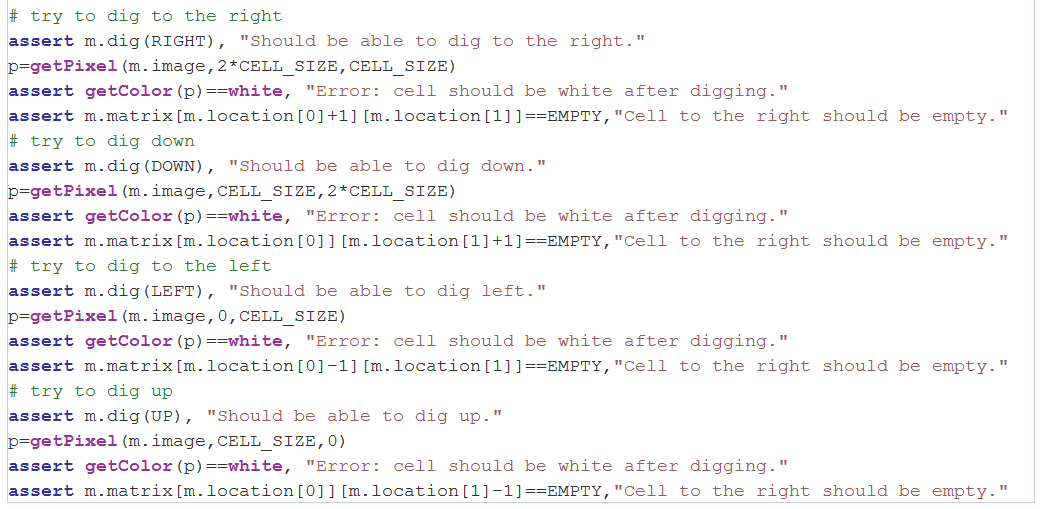
 This puts you back in the RED state.

 Make this pass by adding the code to the dig method.

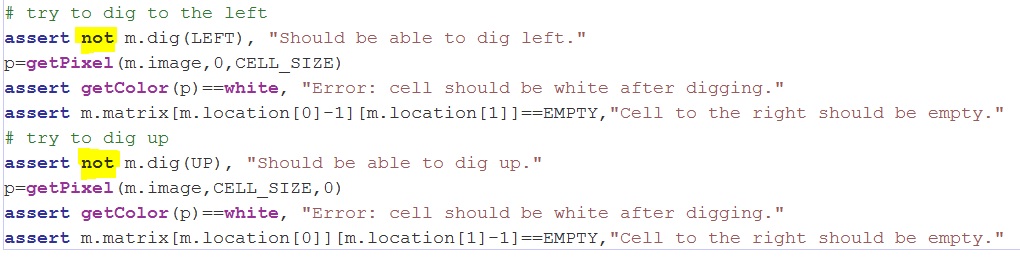
 and you are back to GREEN.

The other 3 directions are done in the same way. I will not include that in this tutorial. You can do it yourself. Add them now so you can continue with the tutorial because there are some differences we will deal with in the ‘left’ and ‘up’ directions.

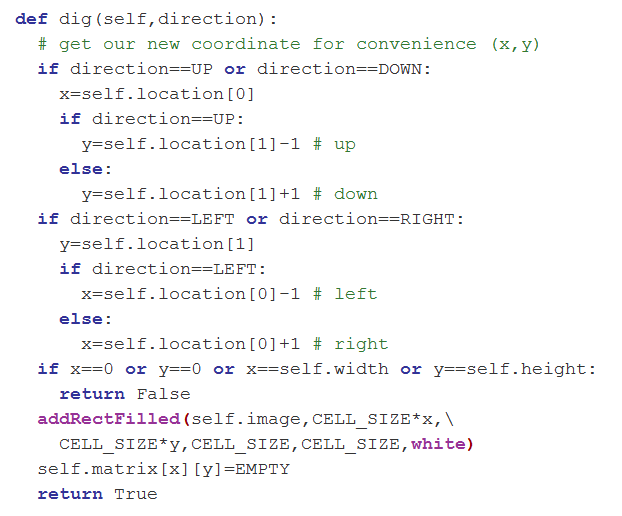
This is what your tests should look like.

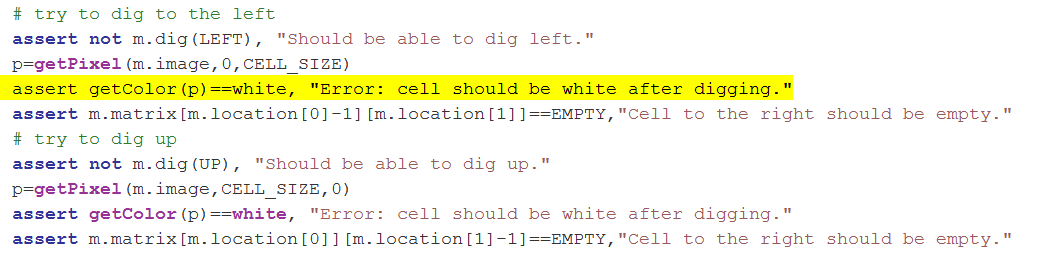


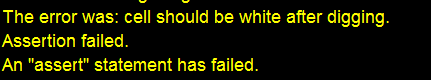
This is just the start for our tests for digging. Notice that from our starting location, (1,1), we should not be able to dig left or up. This would dig into the boundary which we want to keep out-of-bounds and always a wall. So we want to change these two tests for those directions.

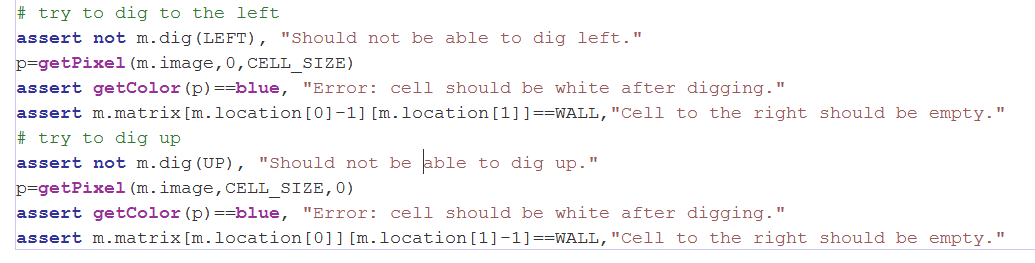


These should fail now. We need to put a check so we return False if we hit the boundary. Let’s create the coordinates we want to dig to for convenience so we can check this. After refactoring, the method should look like this.

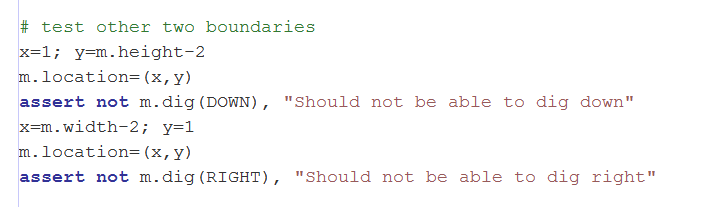
 This refactoring is a bigger step than usual. Take time to understand how we simplified this method by pre-computing the new coordinates x and y. We still have some failures in our tests however.



 We fixed the image color but not the matrix cell state. Fix that now.

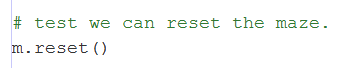
 Now you should be in the GREEN state.

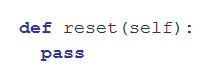
We need to test the other two boundaries to be complete. Add those tests.

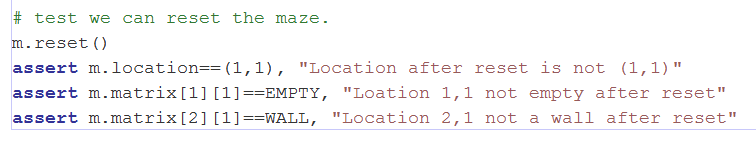
 You should still be in the GREEN state. If not, check your code to see what went wrong.

Now that we can dig, we want to be able to move into the space we dug.

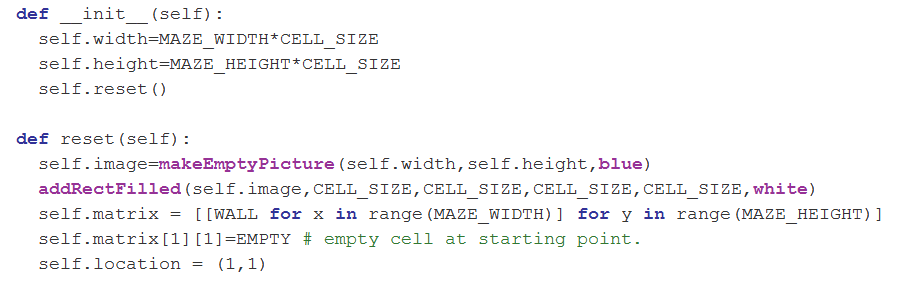
We are starting to make several changes to the image and to the matrix. For further testing it is useful to create a reset method to make things go back to the initial state.

This of course gives us our old “Attribute not found” friend. Create the method.

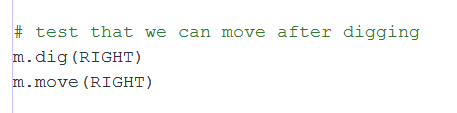
 Now write better tests for reset.

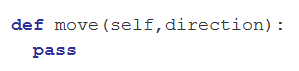


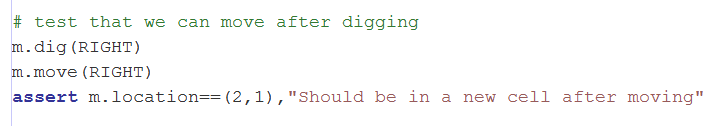
Notice they don’t pass since we don’t do anything for reset yet. Also notice that what we want to do is already done in \_\_init\_\_ so let’s just move that code to reset() and call reset() from \_\_init\_\_

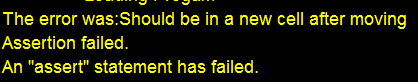
We combined the coding with the refactoring in this step to save time.

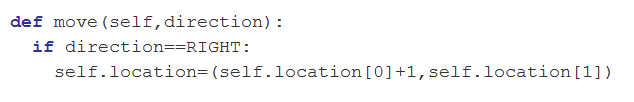
You should now be in the GREEN condition. Now we want to test for moving after we dig.

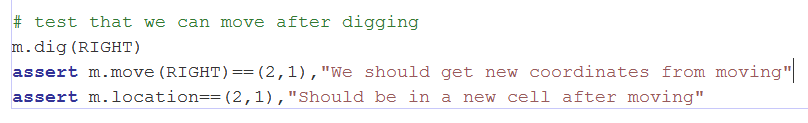
Add this test and say hello to your old “Attribute not found” friend.

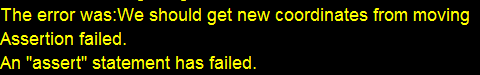
Add the move method. to make the tests pass.

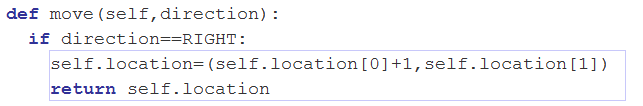
Modify the tests so we can verify we actually did move! 

Check your tests.  Now add to the move method to make the test pass.

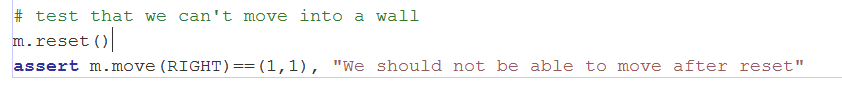
Now your test should pass. We need to verify that we have moved or not however. Right now we don’t return anything from move(). Let’s change that. Test that when we have a successful move, we get the new coordinate returned. If we haven’t moved, we get the same one we started with returned.

 But we don’t

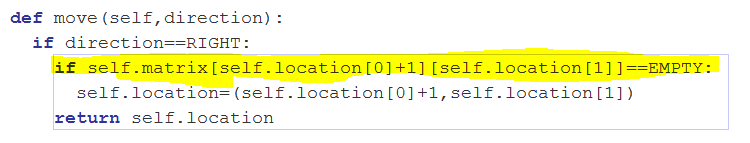
 Fix this.

 Now in GREEN state.

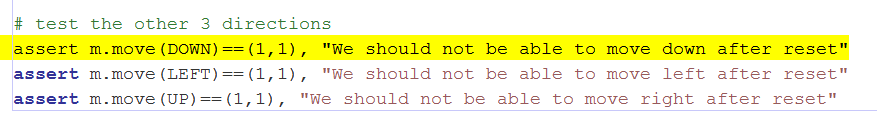
We can’t move into a wall so verify we get the same location if we try to move after a reset.

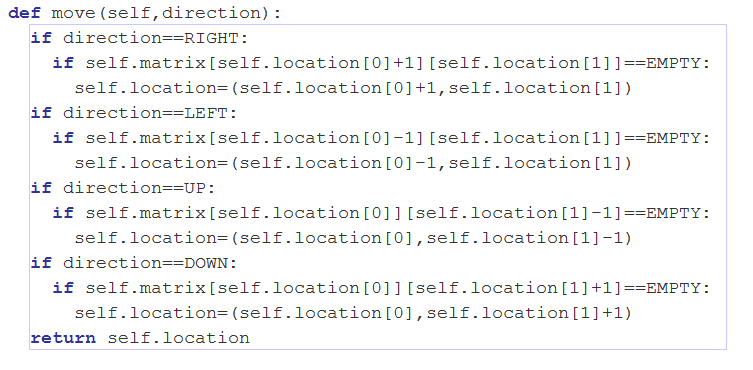
This will fail.

Fix it.

Now in GREEN state.

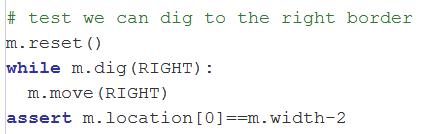
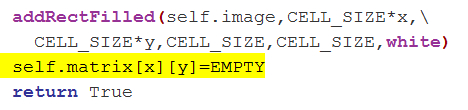
Like before, we need to deal with the other 3 directions. Here are the tests.

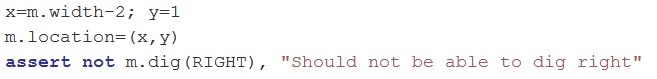
Fixing this gives us the following code.

Now in GREEN state.

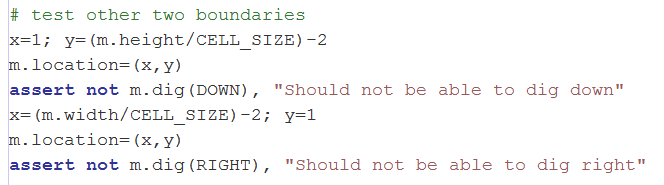
Now we can dig and move. We are getting close to making a path for our maze. We want the path to be somewhat random so it’s not a trivial maze. We will use the random module for that purpose. The path starts at (1,1) and ends at either the right-hand border or the lower border.

Let’s start by trying to dig to the right-hand border.

Our test is because dig() should return False when we hit the border, we should be adjacent to the border when we stop. Instead we get the error  which occurs in our dig routine.  so x is getting to 80 when the largest value it can have is 79. Why is that?

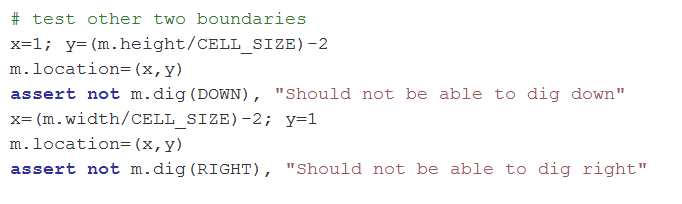
When our x location is 78, and we call dig, we should return False because we don’t want to dig into column number 79. We have a test for this though. Remember this test? 

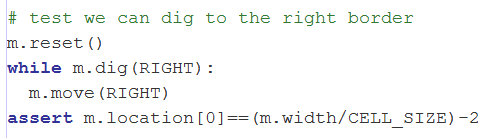
This passes because we set x to be m.width-2 but our thinking was not correct. m.width is really 800! Fix this test and the test above it.

Now these tests should fail. We fix the code as follows.

Change the dig method to use the following

and change our tests to be as follows.

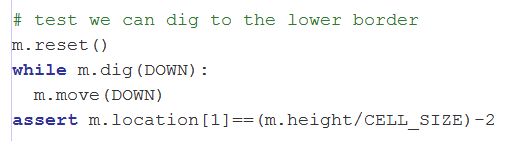
and



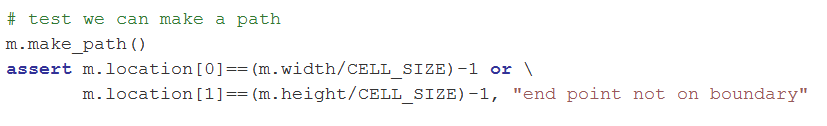
Now your tests should pass. GREEN!

Problems like this get caught early when using test driven development.

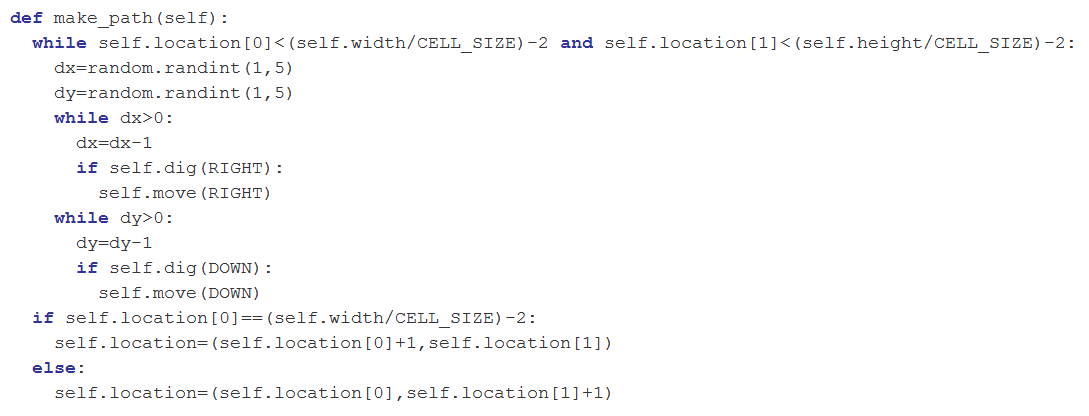
Just for completeness, let’s do the same thing going down to the lower border.

 It should pass right away.

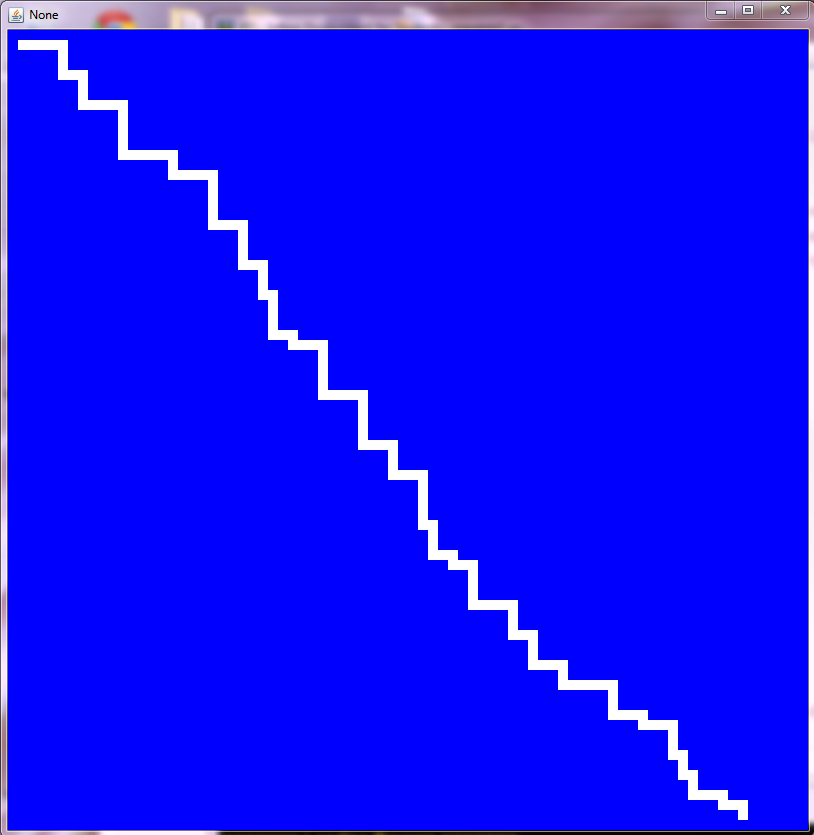
Now lets make a method called create\_path() which zig-zags randomly to the right and down until it hits a border. First the test of course!

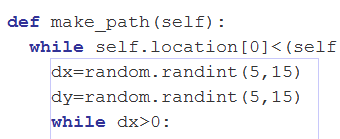
 RED CONDITION

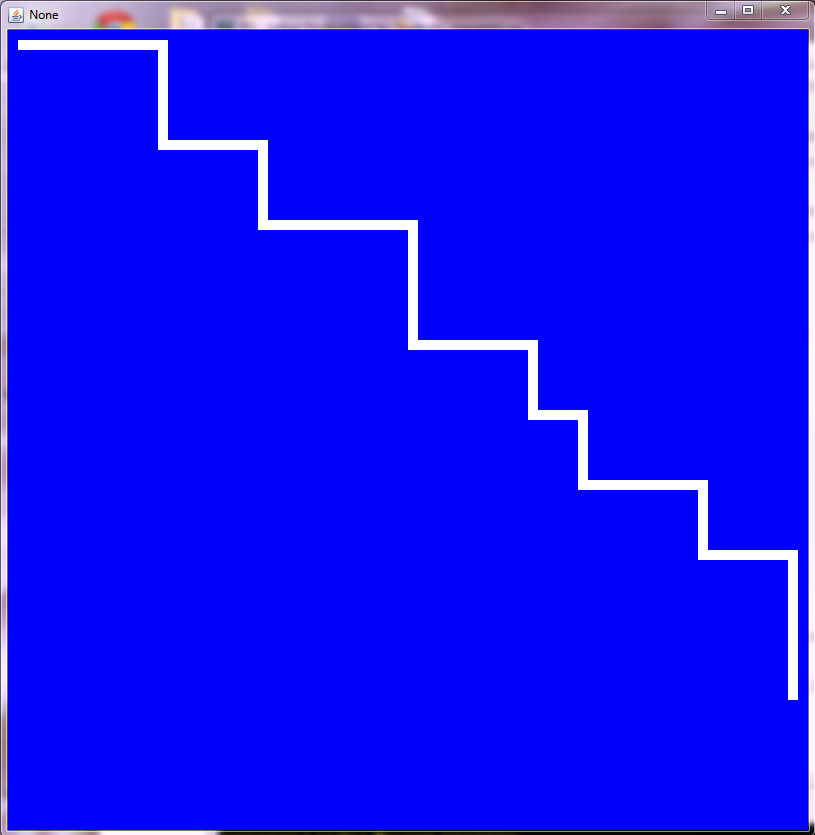
Now the code. (Don’t forget to import random at the top of the file)

GREEN CONDITION!

If you look at the result, you should see something like this.

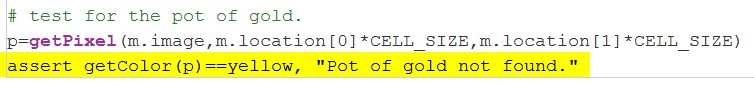
 Certainly not much of a maze yet but we are making progress. Seems to me the path lengths should be a bit longer. Let’s increase the random limit to 15 rather than 5 and make the minimum length 5.

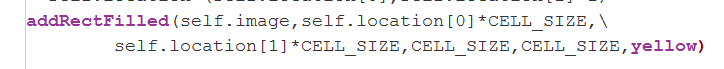
With this change, our make\_path method gives us something like this.

 This is looking better.

Now we want a pot of gold at the end of our maze. Let’s put a yellow square at the end.

Testing 1,2,3…

Remember after make\_path is called, one of our coordinates is placed on the border. This is the final location. It should be yellow. It’s not. Make it so! Put this at the end of make\_path()

GREEN!

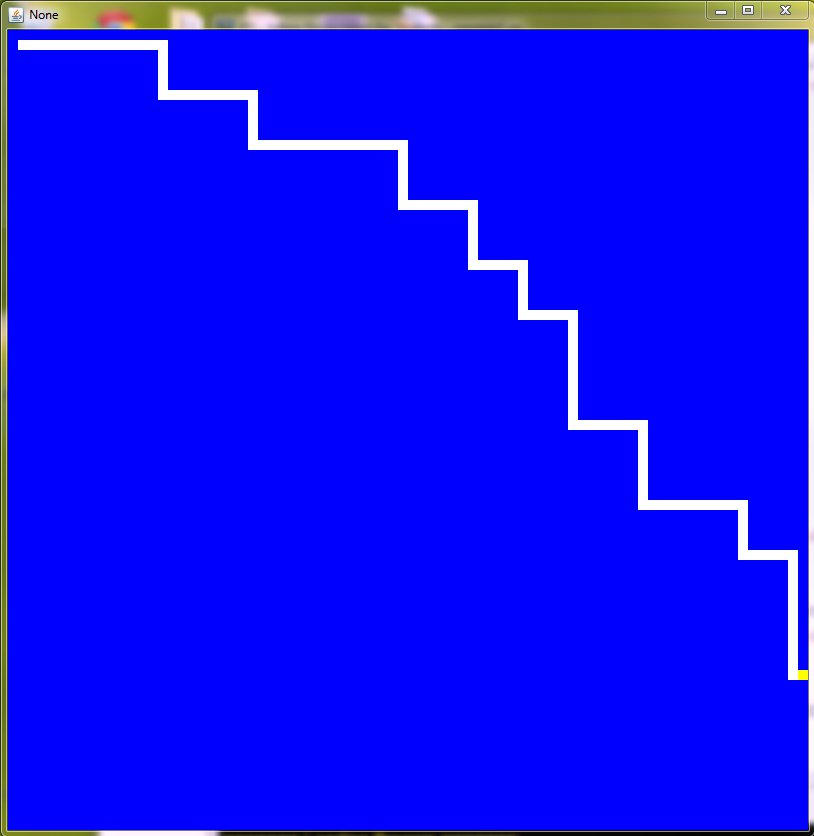
Not only do we have the color to check, but also the matrix value should be END for that location.

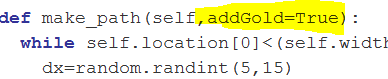
 RED!

Fix this by adding this line at the end of make\_path().

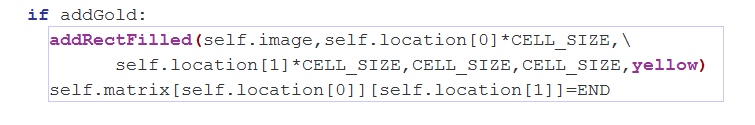
 GREEN!

Here’s how it looks.

 Notice the pot of gold.

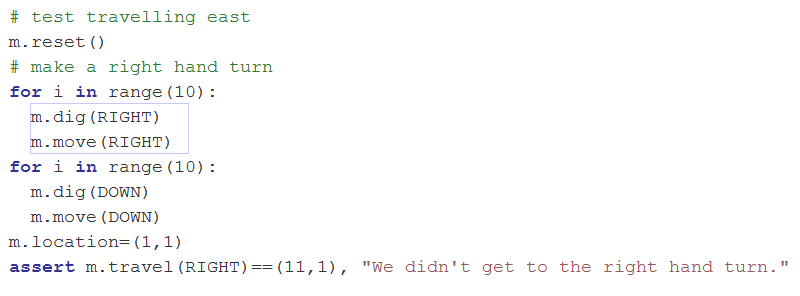
The last thing to do is fill in the rest of the maze with false paths. Since we only want one pot of gold. Let’s add an argument to make\_path so that can be turned off. Change make\_path. 

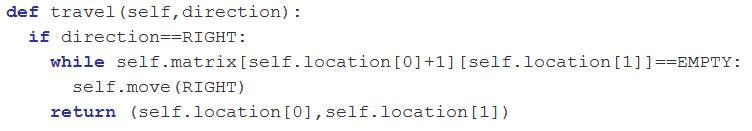
Now when we don’t pass in that argument it is assumed to be True. When we fill in the rest of the maze, we will set it to be False. For it to be useful, we need to use it inside the method. Qualify the last two lines of make\_path with the if statement as follows.

Everything should still work as before. GREEN!

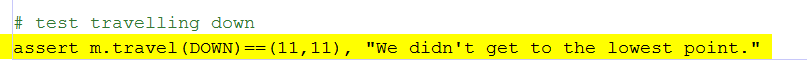
How do we fill in the rest of the maze? We can try first by making random paths from every turn in the original path. For this we need to identify the turns. One way to approach it is to have a method called travel() which will go as far as we can in a given direction. Then when we stop, we know we are at a turn.

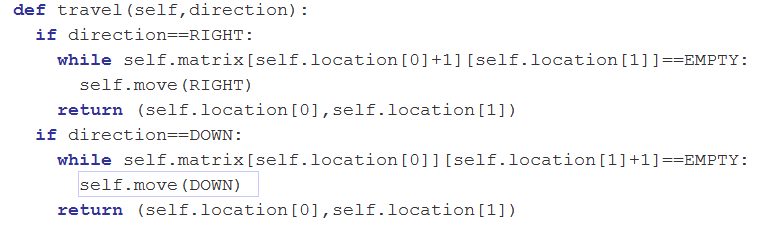
Let’s test for this by making a simple right hand turn and see if we get there.

I’m skipping the error that the method doesn’t exist and starting with the method logic.

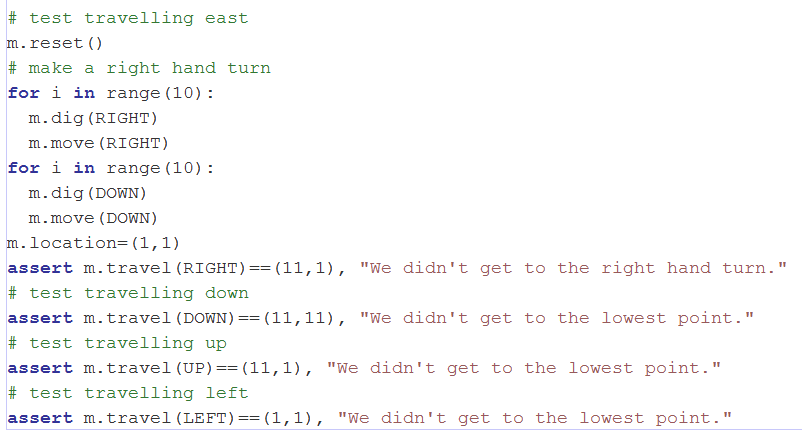
 GREEN!

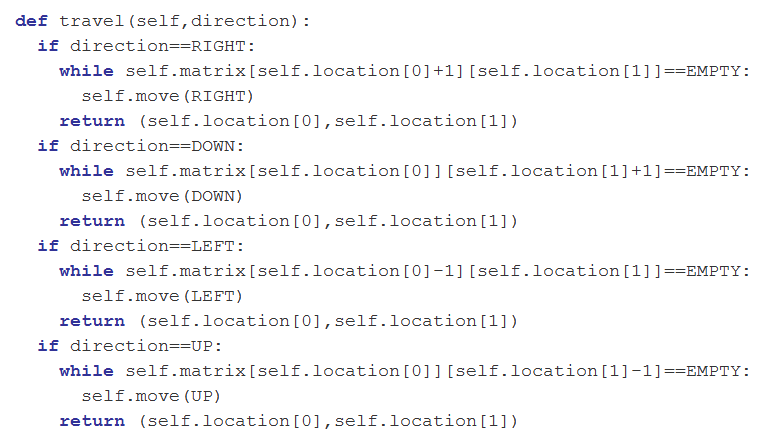
Now let’s just check the downward travel to assure ourselves that works. (A little extra work but not much)

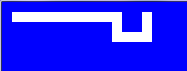
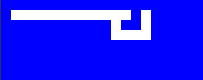
RED!

 GREEN!

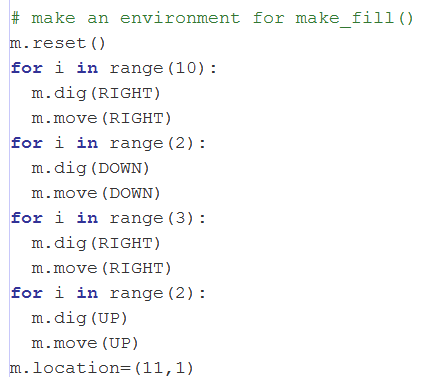
Complete the other two directions.

RED!

GREEN!!

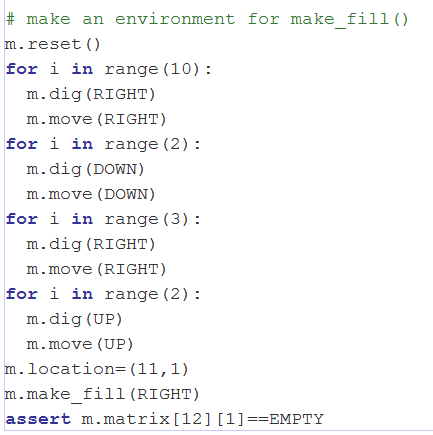
Now we have a way of getting to the turns. We want to start digging from each turn and stop before hitting the border or another empty square. The paths should be random in all four directions. No need to worry about which border we are heading for. Let’s call the method make\_fill() and give it an initial direction. How can we check this function? One way is to make an artificial environment where we can predict the result. Consider the following image.  when we fill in at the first turn, the only choice is this. 

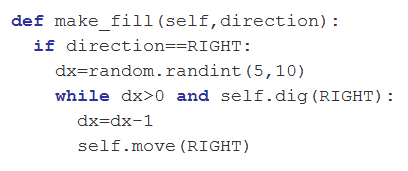
So let’s set up a test for that. Here’s the preparation.

This produces the first image above and places the location at the first turn.

Now after calling make\_fill(RIGHT) we should get the second image.

The test just adds two more lines.

Of course there is no make\_fill yet. So fix that.

GREEN!