Maze Project

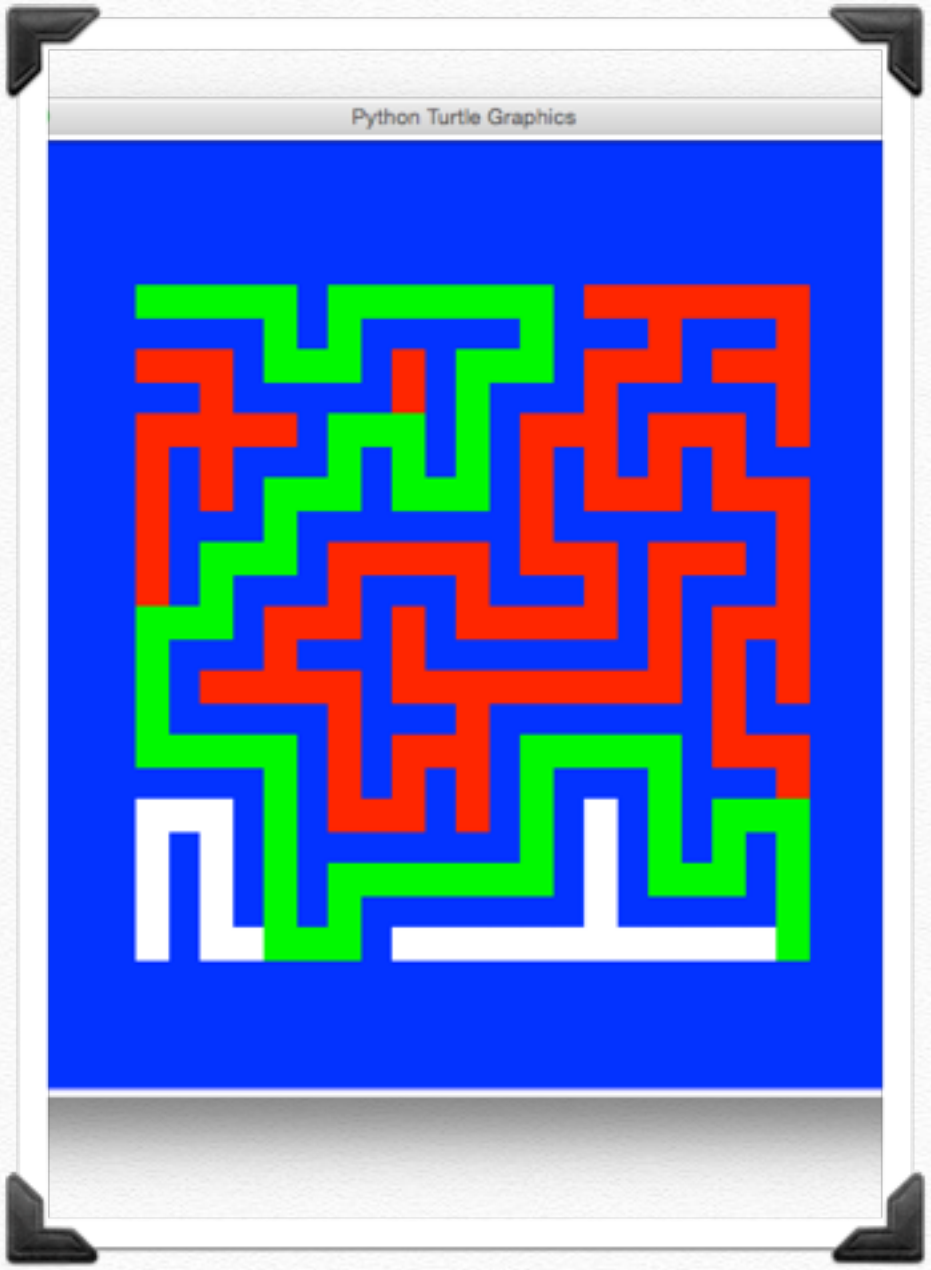
Test Driven Development

Michael Toth
Muhlenberg College

Fall 2016

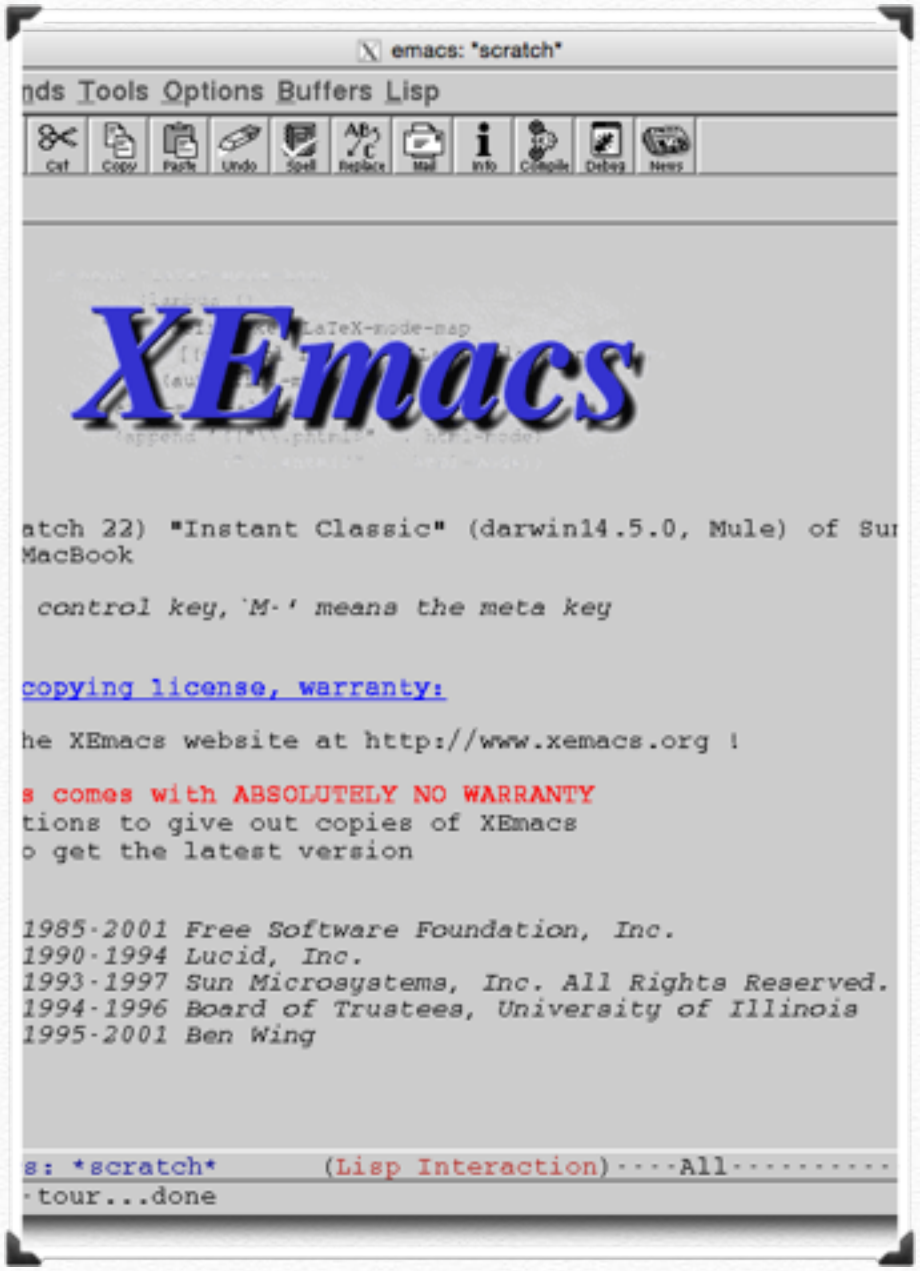
Project Overview

- Using TDD we will
  - Create a maze
  - Solve the maze we created



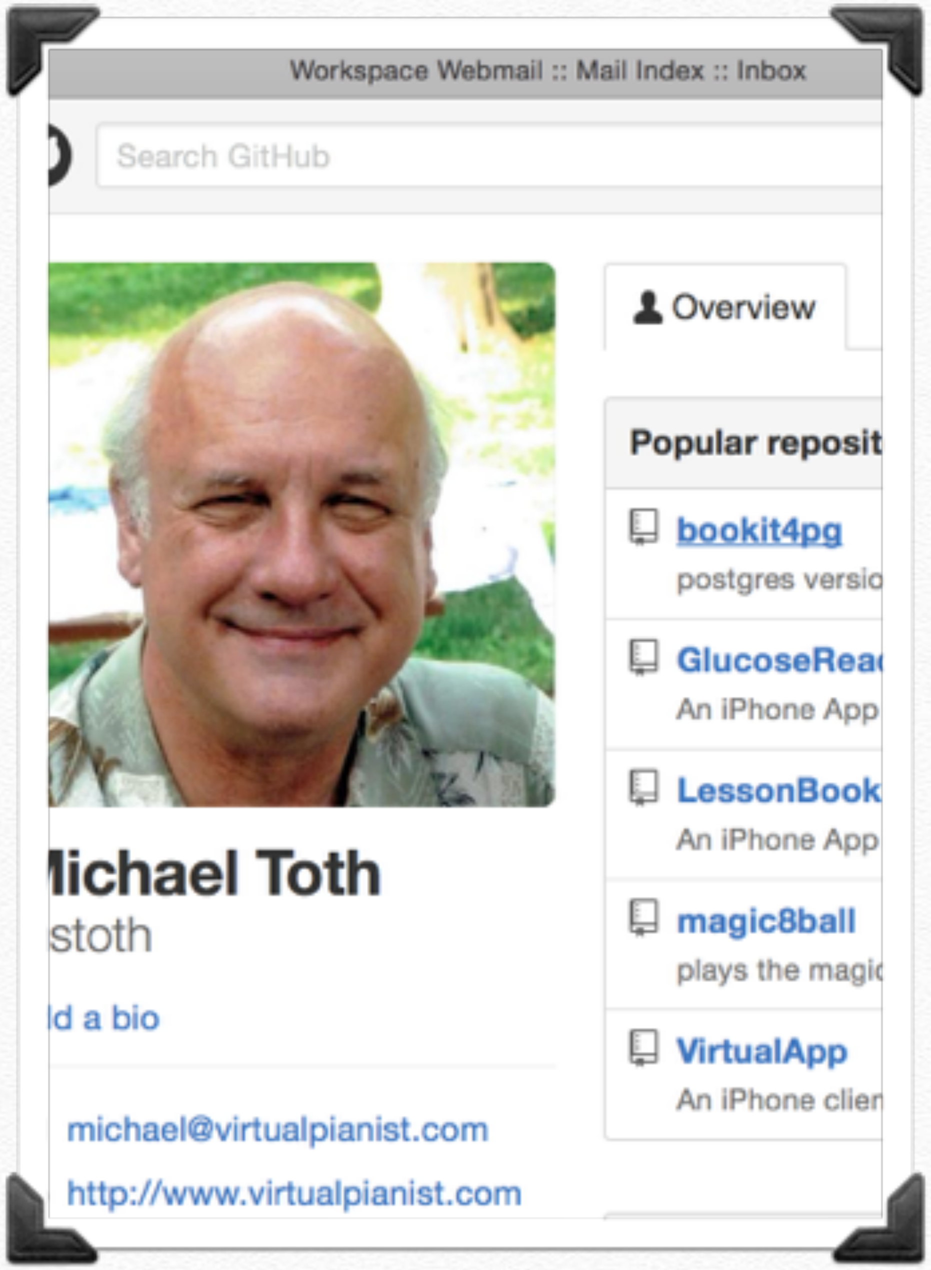
Learning Tools

- Text Editors
  - vi - http://www.tutorialspoint.com/unix/unix-vi-editor.htm
  - emacs - http://www.jesshamrick.com/2012/09/10/absolute-beginners-guide-to-emacs/
- Python
  - idle - https://sites.physics.utoronto.ca/comp-physics/manual/tutorial-part-1-first-steps-with-idle-and-python
- Shell Commands
  - Windows - http://www.computerhope.com/issues/chusedos.htm
  - Mac - http://www.macdevcenter.com/pub/a/mac/2001/12/14/terminal_one.html



Learning Tools

- github
  - create account
  - add repository

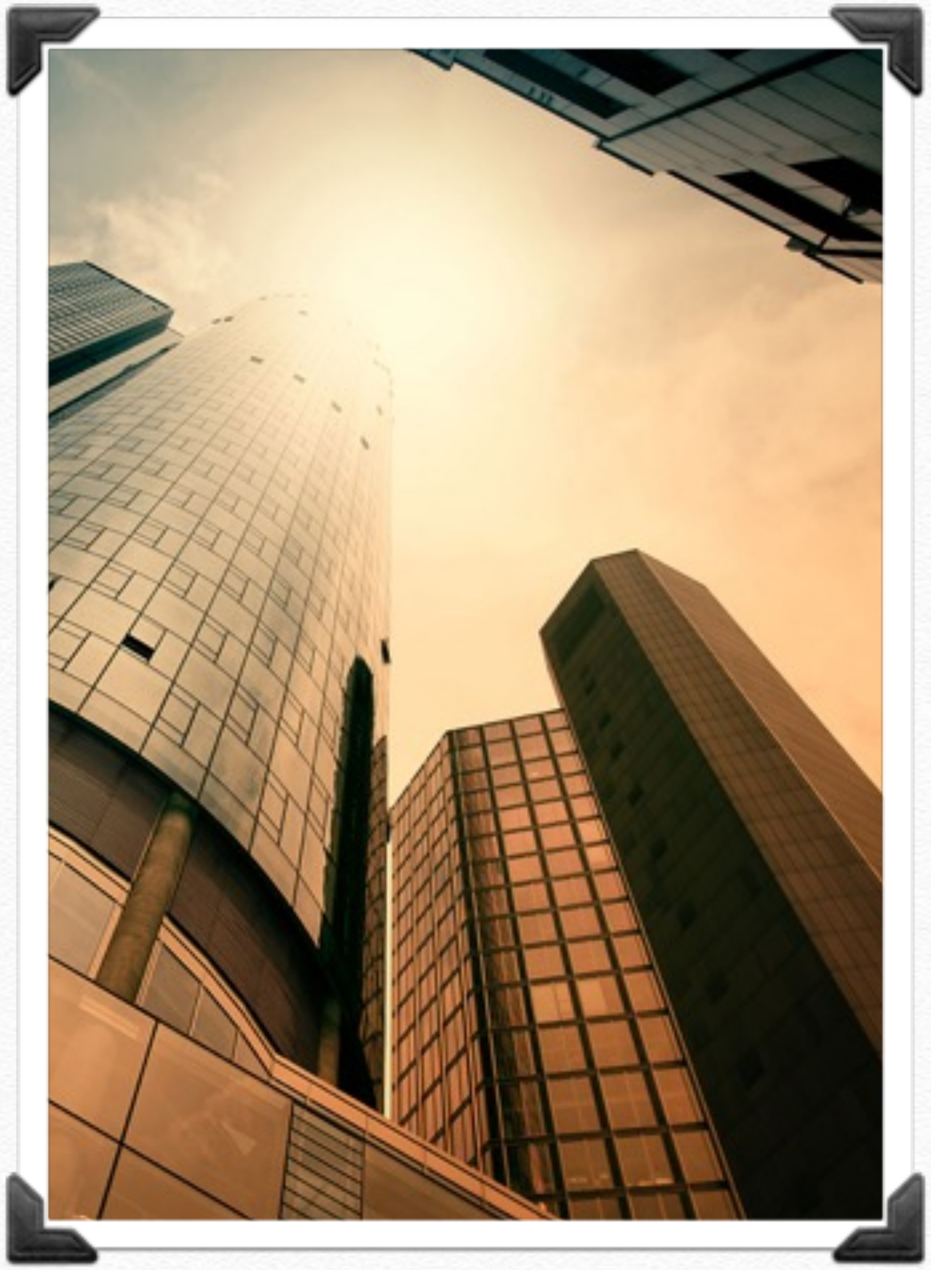


On Windows you will need to use Chrome

- Selenium and Firefox are not currently working togther well
- Install Chrome
- Install chromedriver
  - chromedriver.storage.googleapis.com/index.html?path=2.23/
  - you need to have chromedriver.exe in your path

Computers

- CPU Central Processing Unit
- ROM Read Only Memory
- RAM Random Access Memory
- How it works.



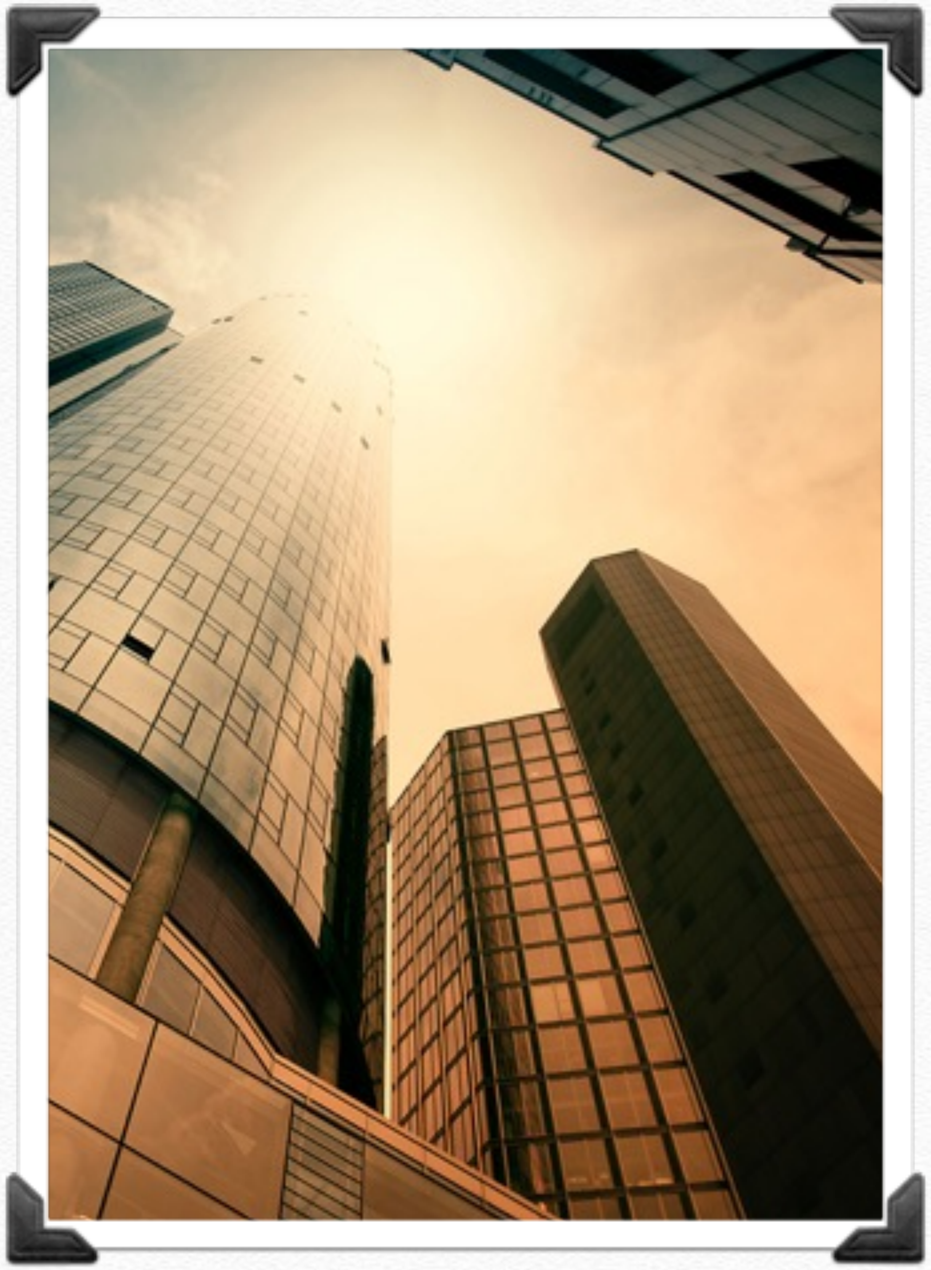
Making a Web Page

- Text Editor (syntax highlighting)
  - emacs has a template
- GUI
  - DreamWeaver
    - expensive
    - powerful
  - KompoZer
    - Free and works well. No fancy bells and whistles.
  - Coda
    - Need to know html pretty well. Not much power.

Helpful Links

- http://www.w3.org/TR/html5/
- https://html.spec.whatwg.org/multipage/
- https://www.webplatform.org
- https://developer.mozilla.org/en-US/

You will need to edit HTML



Some Basic Python

http://www.tutorialspoint.com/python/python_online_test.htm

- Identifiers
- Reserved Words
- Lines and Indentation
- Multi-line statements
- Quotations
- Comments

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.
Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and, exec, not, assert, finally, or, break, for, pass, class, from, print, continue, global. raise, def, if, return, del, import, try, elif, in, while, else, is, with, except, lambda, yield

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example −

```
if True:
    print "True"
else:
  print "False"
```

However, the following block generates an error −

```
if True:
    print "Answer"
    print "True"
else:
    print "Answer"
  print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks −

Multi-Line Statements

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

```
total = item_one + \
        item_two + \
        item_three
```

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python

# First comment
print "Hello, Python!" # second comment
```

This produces the following result −

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression −

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows −

```
# This is a comment.
# This is a comment, too.
# This is a comment, too.
# I said that already.
```

More Python Information

- Lists
- Tuples
- Dictionaries
- Decision Making
- Loop Control
- Classes

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type. The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example −

```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

This produce the following result −

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example −

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple           # Prints complete list
print tuple[0]        # Prints first element of the list
print tuple[1:3]      # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

This produce the following result –

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.
Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example −

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result −

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Decision Making - If Statement

Following is the general form of a typical decision making structure found in most of the programming languages −

Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.
Python programming language provides following types of decision making statements. Click the following links to check their detail.

An **if statement** consists of a boolean expression followed by one or more statements.

An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.

You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

Single Statement Suites

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.
Here is an example of a **one-line if** clause −

```
#!/usr/bin/python

var = 100

if ( var  == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

When the above code is executed, it produces the following result −

```
Value of expression is 100
Good bye!
```

Python programming language provides following types of loops to handle looping requirements.

while loop

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

for loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

nested loops

You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements.

break statement

Terminates the loop statement and transfers execution to the statement immediately following the loop.

continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

pass statement

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.
Let us go through the loop control statements briefly −

Creating Classes

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows −

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

Example

Following is the example of a simple Python class –

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all instances of a this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

- Useful Resources
    - https://www.python.org
    - https://wiki.python.org/moin/WebProgramming
    - http://starship.python.net
    - http://www.jython.org
    - http://www.pythonware.com
    - http://archaeopteryx.com
    - http://www.vim.org/scripts/script.php?script_id=790
    - http://pyxml.sourceforge.net/topics/
    - http://www.greenteapress.com/thinkpython/html/index.html

- Getting used to a text editor (write down new commands as you learn them)
- Look at a cheat sheet to help see if there is a way to do something you want to do like deleting a word or finding a word
- write down new ones that you learn so you can refer to those quickly. You will probably need them again.
    - moving by word
        - emacs: META-f
        - vi: w
    - go down a page
        - emacs: CTRL-v
        - vi: CTRL-f
    - delete the word at the cursor
        - emacs: META-d
        - vi dw

- https://github.com/seleniumbase/SeleniumBase/blob/master/help_docs/method_summary.md
- Learn what methods are available in selenium by looking here.

Maze Week 1

- Introduction to unittest module
- Introduction to git
- Introduction to Test Driven Development
- Introduction to Python
- Implement the first test

The unittest module

- It's a framework for our tests
- Each test begins with "Test"
- There is a set up method for repeated operations when starting a test.
- You need to add a check to run unittest.main()
- We create a class called TestMaze and inherit from unittest.TestCase
- When unittest.main() is called, each method in our TestMaze class will be run.
- https://cgoldberg.github.io/python-unittest-tutorial/

Git

- We use git to get the code and the different versions
- The two commands below will clone from the repository and set the version for the first slide (Red 1)
- **git clone https://github.com/mstoth/clusterfall2016**
- **git checkout 50f86d**
- Versions are seen with "git log"

Git

- use git checkout <hash> where the hash is the first 6 characters of the long hash in the log display.
- Alternate between red and green conditions
- Use the following command to get back to the complete project and then checkout the next step
- **git checkout master**
- Learn git for your own use. Other commands like add, commit, tag.
- Get a github account.

The Python Class

- Our class is called Maze()
- It will have many methods
- m=Maze() instantiates a new maze called m
- m is an object of type Maze
- m.create() will draw a new maze on the screen
- m.solve() will solve maze just created.
- Maze.py is the file which defines the class

Red 1

```
Traceback (most recent call last):
  File "/Users/michaeltoth/Documents/cluster/clusterfall2016/
MazeTests.py", line 1, in <module>
    from Maze import *
ImportError: No module named ‘Maze'
```

This is the message (with a different path for you of course) when you type

```
python MazeTests.py
```

Here’s the test

Red 1

- The error is due to not having a maze class available.
- *python MazeTests.py* produces the error

Maze Test

```
from Maze import *
import turtle
import unittest
```

Modules are imported here. Read about modules and specifically the turtle and unites modules

```
class testMaze(unittest.TestCase):
```

We inherit the methods of TestCase here

```
    def setUp(self):
```

This is done before every test

```
        # this checks for a Maze class
        self.m=Maze()
```

The error occurs when we try to use the class

Green 1

- We create class called Maze with the keyword ‘class’
- There is a special method called __init__(). This gets called when a class is instantiated.
- All methods must start at least with 1 argument called ‘self’
- pass is a no-operation command in Python
- This code makes our test pass with the minimal amount of coding.
- The check for __name__ is missing.
- if __name__==“__main__”: unittest.main() needed to run from idle. (add it to the end of the MazeTests.py file)

Green 1

```
class Maze():
    def __init__(self):
        pass
```

Week 2

- Tests 2-4
- More about the turtle module and methods
- Python type() function
- Python class structure
- Python assert statement
- Setting background color of screen

Red 2

- We test for the screen by checking the type
- The type is turtle._Screen

Red 2

```
    assert type(self.m.s) == turtle._Screen
AttributeError: Maze instance has no attribute 's'
```

Green 2

- We have a class called Maze in Maze.py
- We have defined a Screen to be self.s
- if ‘m’ is a Maze object, m.s will be the Screen object
- typing ‘type(m.s)’ will tell us the type

Green 2

```
import turtle
class Maze():
    def __init__(self):
        self.s = turtle.Screen()
```

Red 3

```
        assert self.m.s.window_width == 420

  File "/Users/michaeltoth/Documents/rsmazeidle/idle/github/
clusterfall2016/MazeTests.py", line 15, in testScreenExists
    assert self.m.s.window_width == 420
AssertionError
```

Red 3

- We want to be sure that the screen size is 420
- We set a default size to be 420 so we can change it if we want.
- window_width is a property of a Screen
- window_height is another property
- You can always ask the screen for its size this way
- you can also set the screen to a given size by setting these properties
- precede with ‘self.’ to make something a property

Green 3

```
import turtle
class Maze():
    def __init__(self,size=420):
        self.s = turtle.Screen()
        self.size=size
        self.s.window_width = self.size
        self.s.window_height = self.size
```

Red 4

- We want to have a blue background for the screen
- bgcolor is the property to set the background color on the screen
- many colors are understood with strings.

Red 4

```
assert self.m.s.bgcolor() == ‘blue'
```

```
  File "MazeTests.py", line 16, in testScreenExists
    assert self.m.s.bgcolor() == 'blue'
AssertionError
```

Green 4

- Add a penup() call to make sure we don't draw while we are moving the turtle around.

Green 4

```
self.t.color('blue')
self.t.penup()
```

Week 3

- Tests 5-7
- The Turtle Class
- penup and pendown
- list comprehension
- Python len() function
- constants

Red 5

- We need a turtle
- We test for it with the type() function
- The class is turtle.Turtle

Red 5

```
assert type(self.m.t) == turtle.Turtle
```

Green 5

- We create a turtle called self.t
- We use the turtle module method called Turtle()
- penup prevents lines being drawn while moving. (pendown is the other method)

Green 5

```
self.t = turtle.Turtle()
self.t.penup()
```

Red 6

- We will represent our maze with a matrix. A matrix of integers with the height and width of our maze size divided by our path width.
- len() is a general Python function which gives the length of the argument.
- The value only gives us the number of rows
- Since we know both number of rows and columns are the same, we only need one.
- We use list comprehension to make the matrix

Red 6

```
assert len(self.m.matrix)==21

AttributeError: Maze instance has no attribute 'matrix'
```

Green 6

- Lists are in brackets.
- [1] is a list of length 1 and the value is the integer 1
- List comprehension uses brackets
- include the for statement inline
- [1 for i in range(3)] = [1, 1, 1]
- Use double nesting to make a matrix

Green 6

```
self.matrix = [[1 for i in range(21)] \
                for j in range(21)]
```

Red 7

- It’s a good idea to make a reset function. It will put the turtle in the upper left hand corner and clear the maze so it’s filled with the number 1 except for the upper left hand corner which will be 0
- the number 1 will signify a wall. 0 will signify empty.
- We are using the variable SIZE all in caps to signify it is a constant value. It isn’t really in Python, but we treat it like one.

Red 7

```
def testReset(self):
    self.m.reset()
    assert self.m.matrix[0][0]==0
    assert self.m.t.pos()==(-(SIZE/2-10),SIZE/2-10)
```

Green 7

- We use the goto method in the turtle module.
- Then we manually set m[0][0] to 0
- The rest is just moving code from the __init__ method to the reset method.

Green 7

```
def reset(self):
    self.s = turtle.Screen()
    self.s.window_width = self.size
    self.s.window_height = self.size
    self.s.bgcolor('blue')
    self.t = turtle.Turtle()
    self.t.penup()
    self.matrix = [[1 for i in range(21)] for j in range(21)]
    self.t.goto(-(self.size/2-10),self.size/2-10)
    self.matrix[0][0]=0
```

Week 4

- Tests 8-11
- Python tuples
- Indexing matrices
- Refactoring
- comparisons using ==

Red 8

- We need to know the value of the matrix at the position of the turtle.
- As the turtle digs into the wall to make a maze, the value of the matrix at that location will change from 1 to 0
- The method will be called getMatrixValueAt()
- It will have one argument; an (x,y) tuple for the position of the turtle
- v=getMatrixValueAt((0,0)) will give the value at the center of the matrix. (-SIZE/2,SIZE/2) is the upper left hand corner
- Usually we will use the pos() method of our turtle to get that position.

Red 8

```
def testGetMatrixValueAt(self):
    self.m.reset()
    xpos = -(self.m.size/2-10)
    ypos = self.m.size/2-10
    assert self.m.getMatrixValueAt((xpos,ypos))==0
```

Green 8

- We need to convert a position (x,y) to an index [x][y] to index our matrix.
- the variable pos is a tuple.
- it is the position we want to map to our matrix
- matrix[x][y] will be the value we want
- matrix[0][0] is the upper left hand corner
- (-self.size/2,self.size/2) is the upper left hand corner

Green 8

- We want -self.size/2 to become 0 for the x index
- adjust by adding self.size/2
- Need to divide by the path width to adjust for the proper range.
- We want self.size/2 to become 0 for the y index
- adjust by subtracting self.size/2
- Divide by the path width to adjust for the proper range.

Green 8

```
def getMatrixValueAt(self,pos):
    x=int(pos[0]+self.size/2)/20
    y=(self.size/2 - pos[1])/self.pathWidth
    if x < 0 or y < 0 or x > self.size/20-1 or y > self.size/20-1:
        return -1
    v=self.matrix[x][y]
    return v
```

Refactor

- We start using EMPTY for 0
- We will also add NORTH SOUTH EAST and WEST shortly

Refactor

```
import turtle
EMPTY=0

class Maze():
    def __init__(self,size=420,pathWidth=20):
        self.size=size
        self.pathWidth=pathWidth
        self.reset()

    def reset(self):
        self.s = turtle.Screen()
        self.s.window_width = self.size
        self.s.window_height = self.size
        self.s.bgcolor('blue')
        self.t = turtle.Turtle()
        self.t.penup()
        self.matrix = [[1 for i in range(self.size/self.pathWidth)] for j in range(21)]
        self.t.goto(-(self.size/2-self.pathWidth/2),self.size/2-self.pathWidth/2)
        self.matrix[0][0]=EMPTY

    def getMatrixValueAt(self,pos):
        x = int(pos[0]+self.size/2)/self.pathWidth
        y=(self.size/2 - pos[1])/self.pathWidth
        if x < 0 or y < 0 or x > self.size/self.pathWidth-1 or y > self.size/self.pathWidth-1:
            return -1
        else:
            return self.matrix[x][y]
```

Red 9

- We need to know the direction between two positions.
- Typically we will need to know the direction to a particular position from the current turtle's position
- We want a method called direction()
- direction will have 2 arguments; position 1, and position 2
- d=direction(p1,p2) is how you call it.
- p1 and p2 are tuples representing locations on the Screen
- d=direction(self.t.pos(),p2) will give the direction from the turtle to position p2
- We pick 4 arbitrary numbers for NORTH, SOUTH, EAST, and WEST

Red 9

```
EAST=2
SOUTH=1
WEST=3
NORTH=0

def testDirection(self):
    self.m.reset()
    self.m.t.goto(0,0)
    assert self.m.direction((0,0),(10,0))==EAST
    assert self.m.direction((0,0),(-10,0))==WEST
    assert self.m.direction((0,0),(0,10))==NORTH
    assert self.m.direction((0,0),(0,-10))==SOUTH
```

Green 9

- Just for convenience we put the 4 values for the 2 position tuples into 4 variables called p1x, p1y, p2x, p2y
- Saves us from typing pos1[0] and makes the code a little cleaner to read
- First just check if the two positions are the same. If so, return 0.
- 0 is not one of NORTH, SOUTH, EAST, or WEST so we know there is no direction between the two positions
- Then break the problem down into two parts, whether it is NORTH or SOUTH and whether it is EAST or WEST
- then split those two problems into two to make the decision.
- We use if, else, and elif to do this kind of decision making

Green 9

```
def direction(self,pos1,pos2):
    p1x=int(pos1[0]); p1y=int(pos1[1]); p2x=pos2[0]; p2y=pos2[1]
    if p1x==p2x and p1y==p2y:
        return 0
    """ returns the direction from position 1 to position 2 """
    if p1x==p2x: # x position the same, either NORTH or SOUTH
        if p2y>p1y: # NORTH
            return NORTH
        else:
            return SOUTH
    else:
        if p2x>p1x: # EAST
            return EAST
        else:
            return WEST
```

Red 10

- We also need to set the matrix value at a given turtle position.
- setMatrixValueAt will be our method name
- There will be 2 arguments; position and value
- The position will be a tuple, the value will be an integer
- It will return True or False depending on the success of the attempt
- After a reset, the value at (0,0) should be -1 when we set it to that using setMatrixValueAt((0,0),-1)
- We tested getMatrixValueAt so we can use that to test setMatrixValueAt

Red 10

```
def testSetMatrixValueAt(self):
    self.m.reset()
    self.m.setMatrixValueAt((0,0),-1)
    assert self.m.getMatrixValueAt((0,0))==-1
```

Green 10

- We use try to handle problems where x and y are out of bounds.
- We add VISITED, FAILED, and GOAL and make sure we stamp the correct color for the value.
- VISITED = green, FAILED = red, GOAL = yellow, WALL = blue, and EMPTY = white

Green 10

```
def setMatrixValueAt(self,pos,value):
    x=int(pos[0]+self.size/2)/self.pathWidth
    y=(self.size/self.pathWidth)-int((pos[1]+self.size/2)/self.pathWidth)-1
    try:
        self.matrix[x][y]=value
    except:
        return False
    spos = self.t.pos()
    self.t.goto(pos)
    if value == WALL:
        self.t.color('blue')
        self.t.stamp()
    elif value == VISITED:
        self.t.color('green')
        self.t.stamp()
    elif value == FAILED:
        self.t.color('red')
        self.t.stamp()
    elif value == GOAL:
        self.t.color('yellow')
        self.t.stamp()
    else:
        self.t.color('white')
        self.t.stamp()
    self.t.goto(spos)
    return True
```

Red 11

- The main operation to make a maze is called dig()
- dig takes one argument, a direction
- dig returns the position of the turtle after the attempt

Red 11

```
def testDig(self):
    print "testDig"
    self.m.reset()
    spos = self.m.t.pos()
    self.m.dig(EAST)
    assert self.m.t.pos()==(spos[0]+self.m.pathWidth,spos[1])
    spos=self.m.t.pos()
    self.m.dig(SOUTH)
    assert self.m.t.pos()==(spos[0],spos[1]-self.m.pathWidth)
    spos=self.m.t.pos()
    self.m.dig(WEST)
    assert self.m.t.pos()==(spos[0]-self.m.pathWidth,spos[1])
    self.m.t.goto(0,0)
    self.m.dig(NORTH)
    assert self.m.t.pos()==(0,self.m.pathWidth)
```

Green 11

- To dig, we go one space in the specified direction
- If it’s a wall there, make it empty. Otherwise, go back and return the original position of the turtle.

Green 11

```
def dig(self,direction):
    oldpos=self.t.pos()
    if direction == EAST:
        self.t.goto(oldpos[0]+self.pathWidth,oldpos[1])
    if direction == SOUTH:
        self.t.goto(oldpos[0],oldpos[1]-self.pathWidth)
    if direction == WEST:
        self.t.goto(oldpos[0]-self.pathWidth,oldpos[1])
    if direction == NORTH:
        self.t.goto(oldpos[0],oldpos[1]+self.pathWidth)
    if self.getMatrixValueAt(self.t.pos())==WALL:
        self.setMatrixValueAt(self.t.pos(),EMPTY)
    else:
        self.t.goto(oldpos[0],oldpos[1])
    return self.t.pos()
```

Week 5

- Tests 12-14
- Mapping from turtle position to matrix location

Red 12

- It's possible to dig into a wall and accidentally break through to an existing path.
- We will need to be able to tell if we will be too close to an existing empty space after a dig.
- We create a method called tooClose() which will return True if we are too close to an existing path to dig, and False if we are not too close and it will be ok to dig.
- After a reset, we should not be able to go NORTH or WEST but we should be able to go EAST and SOUTH

Red 12

```
def testTooClose(self):
    self.m.reset()
    assert self.m.tooClose(NORTH)==True
    assert self.m.tooClose(EAST)==False
    assert self.m.tooClose(SOUTH)==False
    assert self.m.tooClose(WEST)==True
```

Green 12

- We use try to handle errors in the index range. If there are errors from trying to index the matrix, we are too close.
- If no error, then make sure the 3 cells in question are all walls.

Green 12

```
def tooClose(self,direction):
    spos = self.t.pos()
    x=int(spos[0]+self.size/2)/self.pathWidth
    y=(self.size/self.pathWidth)-int((spos[1]+self.size/2)/self.pathWidth)-1

    if direction == EAST:
        if x==self.size/self.pathWidth-1:
            return True
        try:
            if self.matrix[x+1][y-1] == WALL and self.matrix[x+1][y+1]==WALL and \
               self.matrix[x+1][y] == WALL:
                return False
        except:
            return True
        return True
    if direction == SOUTH:
        print x,y
        if y==self.size/self.pathWidth-1:
            return True
        try:
            if self.matrix[x+1][y+1] == WALL and self.matrix[x-1][y+1]==WALL and \
               self.matrix[x][y+1] == WALL:
                return False
        except:
            return True
        return True
```

tooClose continued

```
if direction == WEST:
    if x==0:
        return True
    try:
        if self.matrix[x-1][y-1] == WALL and self.matrix[x-1][y+1]==WALL and \
           self.matrix[x-1][y] == WALL:
            return False
    except:
        return True
    return True
if direction == NORTH:
    if y==0:
        return True
    try:
        if self.matrix[x][y+1] == WALL and self.matrix[x-1][y+1]==WALL and \
           self.matrix[x+1][y+1] == WALL :
            return False
    except:
        return True
    return True
```

Red 13

- These tests check that we can't dig and break through or dig beyond the limits of the matrix.

Red 13

```
def testDig(self):
    self.m.reset()
    spos = self.m.t.pos()
    self.m.dig(EAST)
    assert self.m.t.pos()==(spos[0]+self.m.pathWidth,spos[1])
    spos=self.m.t.pos()
    self.m.dig(SOUTH)
    assert self.m.t.pos()==(spos[0],spos[1]-self.m.pathWidth)
    spos=self.m.t.pos()
    self.m.dig(WEST)
    assert self.m.t.pos()==(spos[0],spos[1])
    self.m.t.goto(0,0)
    self.m.dig(NORTH)
    assert self.m.t.pos()==(0,self.m.pathWidth)
```

Red 13

```
        # make sure we can't dig west from a reset
        self.m.reset()
        spos = self.m.t.pos()
        self.m.dig(WEST)
        assert self.m.t.pos()==spos
        # make sure we can't dig north from a reset
        self.m.reset()
        self.m.dig(NORTH)
        assert self.m.t.pos()==spos
        # make sure we can't dig east from the right hand corner
        self.m.reset()
        self.m.t.goto(self.m.size/2-self.m.pathWidth/2,self.m.size/2-self.m.pathWidth/2)
        spos=self.m.t.pos()
        self.m.dig(EAST)
        assert self.m.t.pos()==spos
        # make sure we can't dig south from the lower right hand corner
        self.m.reset()
        self.m.t.goto((self.m.size/2-self.m.pathWidth/2,-(self.m.size/2-self.m.pathWidth/
2)))
        spos=self.m.t.pos()
        self.m.dig(SOUTH)
        assert self.m.t.pos()==spos
```

Red 13

```
# make sure we can't dig east
# if it would break through to an existing space
self.m.reset()
self.m.setMatrixValueAt((-(self.m.size/2-5*self.m.pathWidth/2),self.m.size/2-self.m.pathWidth/2),0)
spos=self.m.t.pos()
self.m.dig(EAST)
assert self.m.t.pos()==spos
# make sure we can't dig south
# if it would break through to an existing space
self.m.reset()
self.m.setMatrixValueAt((-(self.m.size/2-self.m.pathWidth/2),self.m.size/2-5*self.m.pathWidth/2),0)
spos=self.m.t.pos()
self.m.dig(SOUTH)
assert self.m.t.pos()==spos
# make sure we can't dig west
# if it would break through to an existing space
self.m.reset()
self.m.setMatrixValueAt((-(self.m.size/2-5*self.m.pathWidth/2),self.m.size/2-self.m.pathWidth/2),0)
self.m.t.goto(-(self.m.size/2-5*self.m.pathWidth/2),self.m.size/2-self.m.pathWidth/2)
spos=self.m.t.pos()
self.m.dig(WEST)
assert self.m.t.pos()==spos
# make sure we can't dig north
# if it would break through to an existing space
self.m.reset()
self.m.setMatrixValueAt((-(self.m.size/2-self.m.pathWidth/2),self.m.size/2-5*self.m.pathWidth/2),0)
self.m.t.goto(-(self.m.size/2-self.m.pathWidth/2),self.m.size/2-5*self.m.pathWidth/2)
spos=self.m.t.pos()
self.m.dig(NORTH)
assert self.m.t.pos()==spos
```

Red 13

```
        # make sure we can't dig west
        # if it would break through to an existing space
        self.m.reset()
        self.m.setMatrixValueAt((-(self.m.size/2-5*self.m.pathWidth/2),self.m.size/2-
self.m.pathWidth/2),0)
        self.m.t.goto(-(self.m.size/2-5*self.m.pathWidth/2),self.m.size/2-
self.m.pathWidth/2)
        spos=self.m.t.pos()
        self.m.dig(WEST)
        assert self.m.t.pos()==spos
        # make sure we can't dig north
        # if it would break through to an existing space
        self.m.reset()
        self.m.setMatrixValueAt((-(self.m.size/2-self.m.pathWidth/2),self.m.size/
2-5*self.m.pathWidth/2),0)
        self.m.t.goto(-(self.m.size/2-self.m.pathWidth/2),self.m.size/
2-5*self.m.pathWidth/2)
        spos=self.m.t.pos()
        self.m.dig(NORTH)
        assert self.m.t.pos()==spos
```

Green 13

- We use tooClose() to make sure we don't break through

Green 13

```
def dig(self,direction):
    oldpos=self.t.pos()
    if direction == EAST:
        self.t.goto(oldpos[0]+self.pathWidth,oldpos[1])
        tooClose = self.tooClose(EAST)
    if direction == SOUTH:
        self.t.goto(oldpos[0],oldpos[1]-self.pathWidth)
        tooClose = self.tooClose(SOUTH)
    if direction == WEST:
        self.t.goto(oldpos[0]-self.pathWidth,oldpos[1])
        tooClose = self.tooClose(WEST)
    if direction == NORTH:
        self.t.goto(oldpos[0],oldpos[1]+self.pathWidth)
        tooClose = self.tooClose(NORTH)
    spos = self.t.pos()

    if self.getMatrixValueAt(spos)==WALL and not tooClose:
        self.setMatrixValueAt(self.t.pos(),EMPTY)
    else:
        self.t.goto(oldpos[0],oldpos[1])
    return self.t.pos()
```

Green 14

- We want to check the return values of dig()
- These tests passed so we skipped Red 14

Green 14
(Tests pass)

```
def testReturnValuesOfDig(self):
    self.m.reset()
    self.m.t.goto((-110,110))
    assert self.m.dig(EAST)==(-90,110)
    self.m.reset()
    self.m.t.goto((-110,110))
    assert self.m.dig(SOUTH)==(-110,90)
    self.m.reset()
    self.m.t.goto((-110,110))
    assert self.m.dig(NORTH)==(-110,130)
    self.m.reset()
    self.m.t.goto((-110,110))
    assert self.m.dig(WEST)==(-130,110)
```

Week 6

- Tests 15-Create
- Python Lists

Red 15

- If the corner cells (cells after a knight's move from the turtle position) are empty, we don't want to dig.

Red 15

```
def testDigRefusesIfCornersAreEmpty(self):
    self.m.reset()
    self.m.t.goto(0,0)
    self.m.setMatrixValueAt((2*self.m.pathWidth,self.m.pathWidth),0)
    self.m.dig(EAST)
    assert self.m.t.pos()==(0,0)
    self.m.reset()
    self.m.t.goto(0,0)
    self.m.setMatrixValueAt((2*self.m.pathWidth,-self.m.pathWidth),0)
    self.m.dig(EAST)
    assert self.m.t.pos()==(0,0)
    self.m.reset()
    self.m.t.goto(0,0)
    self.m.setMatrixValueAt((self.m.pathWidth,-2*self.m.pathWidth),0)
    self.m.dig(SOUTH)
    assert self.m.t.pos()==(0,0)
    self.m.reset()
    self.m.t.goto(0,0)
    self.m.setMatrixValueAt((-self.m.pathWidth,-2*self.m.pathWidth),0)
    self.m.dig(SOUTH)
```

Red 15

```
assert self.m.t.pos()==(0,0)
self.m.reset()
self.m.t.goto(0,0)
self.m.setMatrixValueAt((-2*self.m.pathWidth,self.m.pathWidth),0)
self.m.dig(WEST)
assert self.m.t.pos()==(0,0)
self.m.reset()
self.m.t.goto(0,0)
self.m.setMatrixValueAt((-2*self.m.pathWidth,-self.m.pathWidth),0)
self.m.dig(WEST)
assert self.m.t.pos()==(0,0)
self.m.reset()
self.m.t.goto(0,0)
self.m.setMatrixValueAt((self.m.pathWidth,2*self.m.pathWidth),0)
self.m.dig(NORTH)
assert self.m.t.pos()==(0,0)
self.m.reset()
self.m.t.goto(0,0)
self.m.setMatrixValueAt((-self.m.pathWidth,2*self.m.pathWidth),0)
self.m.dig(NORTH)
assert self.m.t.pos()==(0,0)
```

```
def tooClose(self,direction):
    spos = self.t.pos()
    x=int(spos[0]+self.size/2)/self.pathWidth
    y=(self.size/self.pathWidth)-int((spos[1]+self.size/2)/self.pathWidth)-1

    if direction == EAST:
        if x==self.size/self.pathWidth-1:
            return True
        try:
            if self.matrix[x+1][y-1] == WALL and self.matrix[x+1][y+1]==WALL and \
               self.matrix[x+1][y] == WALL and \
               self.matrix[x+2][y-1] == WALL and self.matrix[x+2][y+1]==WALL and \
               self.matrix[x+2][y] == WALL:
                return False
        except:
            return True
        return True
    if direction == SOUTH:
        if y==self.size/self.pathWidth-1:
            return True
        try:
            if self.matrix[x+1][y+1] == WALL and self.matrix[x-1][y+1]==WALL and \
               self.matrix[x][y+1] == WALL and \
               self.matrix[x+1][y+2] == WALL and self.matrix[x-1][y+2]==WALL and \
               self.matrix[x][y+2] == WALL:
                return False
        except:
            return True
        return True
```

Green 15

```
if direction == WEST:
     if x==0:
         return True
     try:
         if self.matrix[x-1][y-1] == WALL and self.matrix[x-1][y+1]==WALL and \
            self.matrix[x-1][y] == WALL and \
            self.matrix[x-2][y-1] == WALL and self.matrix[x-2][y+1]==WALL and \
            self.matrix[x-2][y] == WALL:
             return False
     except:
         return True
     return True
 if direction == NORTH:
     if y==0:
         return True
     try:
         if self.matrix[x][y-1] == WALL and self.matrix[x-1][y-1]==WALL and \
            self.matrix[x+1][y-1] == WALL and \
            self.matrix[x][y-2] == WALL and self.matrix[x-1][y-2]==WALL and \
            self.matrix[x+1][y-2] == WALL:
             return False
     except:
         return True
     return True
```

Green 15

```
def dig(self,direction):
    oldpos=self.t.pos()
    if direction == EAST:
        self.t.goto(oldpos[0]+self.pathWidth,oldpos[1])
        toooClose = self.tooClose(EAST)
    if direction == SOUTH:
        self.t.goto(oldpos[0],oldpos[1]-self.pathWidth)
        toooClose = self.tooClose(SOUTH)
    if direction == WEST:
        self.t.goto(oldpos[0]-self.pathWidth,oldpos[1])
        toooClose = self.tooClose(WEST)
    if direction == NORTH:
        self.t.goto(oldpos[0],oldpos[1]+self.pathWidth)
        toooClose = self.tooClose(NORTH)
    spos = self.t.pos()

    if self.getMatrixValueAt(spos)==WALL and not toooClose:
        self.setMatrixValueAt(self.t.pos(),EMPTY)
    else:
        self.t.goto(oldpos[0],oldpos[1])
    return self.t.pos()
```

Red 16

- It would be nice to know what the state is in the four neighbors around the turtle
- We create a function called neighbors
- It returns a list of 4 values [NORTH,SOUTH,EAST,WEST]
- we can then check with that list what are the possible choices we have to dig.
- After a reset, we should get [-1,1,1,-1]

Red 16

```
def testNeighbors(self):
    print 'testNeighbors'
    self.m.reset()
    va=[]
    n=self.m.neighbors()
    for nn in n:
        va.append(nn[1])
    assert va == [-1,1,1,-1]
```

Green 16

- For each direction, check the limits to see if we can access the matrix
- If we can't, insert a INVALID or -1 into the list
- Otherwise insert the value of the matrix cell at that location

Green 16

```
    def neighbors(self):
        p=self.t.position()
        r=[]
        # North
        if p[1]+2*self.pathWidth>(self.size/2-self.pathWidth/2):
            r.append([(p[0],p[1]+2*self.pathWidth),-1])
        else:

r.append([(p[0],p[1]+2*self.pathWidth),self.getMatrixValueAt((p[0],p[1]+2*self.pathWidth))])
        # South
        if p[1]-2*self.pathWidth<-(self.size/2-self.pathWidth/2):
            r.append([(p[0],p[1]-2*self.pathWidth),-1])
        else:

r.append([(p[0],p[1]-2*self.pathWidth),self.getMatrixValueAt((p[0],p[1]-2*self.pathWidth))])
        # East
        if p[0]+2*self.pathWidth>(self.size/2-self.pathWidth/2):
            r.append([(p[0]+2*self.pathWidth,p[1]),-1])
        else:

r.append([(p[0]+2*self.pathWidth,p[1]),self.getMatrixValueAt((p[0]+2*self.pathWidth,p[1]))])
        # West
        if p[0]-2*self.pathWidth<-(self.size/2-self.pathWidth/2):
            r.append([(p[0]-2*self.pathWidth,p[1]),-1])
        else:

r.append([(p[0]-2*self.pathWidth,p[1]),self.getMatrixValueAt((p[0]-2*self.pathWidth,p[1]))])
        return r
```

Implement Create

- Now we are at a point where we can write create()
- It will be a recursive function
- It calls itself starting from the reset point.
- Dig in any of the possible 4 directions randomly chosen
- from this new point, repeat by calling create
- whenever all directions are considered, return to the previous instance of create()

Implement Create (look at the Maze.py file for other changes)

```
def create(self):
    spos=self.t.pos()
    n=self.neighbors()
    while len(n)>0:
        self.t.goto(spos[0],spos[1])
        nchoice=random.choice(n)
        n.remove(nchoice)
        if nchoice[1]==WALL:
            d=self.direction(self.t.pos(),nchoice[0])
            if not self.dig(d)==self.dig(d):
                self.create()
```

Week 7

- Start on solving the maze
- travel() method

Ready to start working on solving the maze

- We now have the create() function working. Time to work on solving it.
- We need to keep track of whether or not we visited a cell.
- We aren't digging anymore
- We are travelling along a path already dug
- We want to mark the places we have been as VISITED
- When we travel back over a VISITED cell, we want to mark it FAILED

Red 18

```
    def testLeaveVisited(self):
        self.m.reset()
        [self.m.dig(EAST) for i in range(3)]
        self.m.t.goto(-self.m.size/2+self.m.pathWidth/
2,self.m.size/2-self.m.pathWidth/2)
        [self.m.travel(EAST) for i in range(3)]
        assert self.m.matrix[0][0]==VISITED
        assert self.m.matrix[1][0]==VISITED
        assert self.m.matrix[2][0]==VISITED
        assert self.m.t.pos()==(-self.m.size/
2+7*self.m.pathWidth/2,self.m.size/2-self.m.pathWidth/2)
        [self.m.travel(WEST) for i in range(3)]
        assert self.m.matrix[2][0]==FAILED
```

Green 18

```
    def travel(self,direction):
        if direction == EAST:
            if self.getMatrixValueAt((self.t.pos()
[0]+self.pathWidth,self.t.pos()[1]))==WALL or \
                self.getMatrixValueAt((self.t.pos()
[0]+self.pathWidth,self.t.pos()[1]))==INVALID:
                 return self.t.pos()
            self.t.goto(self.t.pos()[0]+20,self.t.pos()[1])
```

Green 18

```
        if direction == WEST:
            if self.getMatrixValueAt((self.t.pos()[0]-
self.pathWidth,self.t.pos()[1]))==WALL or \
               self.getMatrixValueAt((self.t.pos()[0]-
self.pathWidth,self.t.pos()[1]))==INVALID:
                return self.t.pos()
            self.t.goto(self.t.pos()[0]-20,self.t.pos()
[1])
```

Green 18

```
        if direction == NORTH:
            if self.getMatrixValueAt((self.t.pos()
[0],self.t.pos()[1]+self.pathWidth))==WALL or \
               self.getMatrixValueAt((self.t.pos()
[0],self.t.pos()[1]+self.pathWidth))==INVALID:
                return self.t.pos()
            self.t.goto(self.t.pos()[0],self.t.pos()[1]+20)
```

Green 18

```
        if direction == SOUTH:
            if self.getMatrixValueAt((self.t.pos()
[0],self.t.pos()[1]-self.pathWidth))==WALL or \
               self.getMatrixValueAt((self.t.pos()
[0],self.t.pos()[1]-self.pathWidth))==INVALID:
                return self.t.pos()
            self.t.goto(self.t.pos()[0],self.t.pos()[1]-
self.pathWidth)
```

Green 18

```
if self.getMatrixValueAt(self.t.pos())==EMPTY:
    self.setMatrixValueAt(self.t.pos(),VISITED)
    self.t.color('green')
    self.t.stamp()
else:
    self.setMatrixValueAt(self.t.pos(),FAILED)
    self.t.color('red')
    self.t.stamp()
return self.t.pos()
```

Red 19

- We need to travel to a branch or a wall
- We make a method called travel2BranchOrWall
- There is one argument, the direction to travel

Red 19

```
def testTravel2BranchOrWall(self):
    self.m.reset()
    [self.m.dig(EAST) for i in range(10)]
    self.m.t.goto(-self.m.size/2+self.m.pathWidth/
2,self.m.size/2-self.m.pathWidth/2)
    self.m.travel2BranchOrWall(EAST)
    assert self.m.matrix[0][0]==VISITED
    assert self.m.matrix[9][0]==VISITED
```

Week 8

- Tests 19-Solve

Green 19

- We create a method called 'emptyNeighbors' which tells us how many neighbors are EMPTY
- We create a method called 'immediateNeighbors' which returns a list of the neighbors immediately next to the turtle position.

Green 19

```
    def travel2BranchOrWall(self,direction):

        if self.immediateNeighbors()[direction][1]==EMPTY:
            oldpos = self.t.pos()
            if oldpos == self.travel(direction):
                return self.t.pos()
            while self.immediateNeighbors()[direction]
[1]==EMPTY and \
                  self.emptyNeighbors()==1:
                self.travel(direction)
            self.setMatrixValueAt(self.t.pos(),VISITED)
            if self.immediateNeighbors()[direction]
[1]==GOAL:
                self.t.goto(self.immediateNeighbors()
[direction][0])
        return self.t.pos()
```

Green 20

- We also need to travel up to a branch as well as a wall.

Green 20

```
    def testTravel2BranchOrWallWithTurn(self):
        self.m.reset()
        [self.m.dig(EAST) for i in range(10)]
        spos=self.m.t.pos()
        [self.m.dig(SOUTH) for i in range(10)]
        self.m.t.goto(-self.m.size/2+self.m.pathWidth/
2,self.m.size/2-self.m.pathWidth/2)
        self.m.travel2BranchOrWall(EAST)
        assert self.m.t.pos()==spos,"got
"+str(self.m.t.pos())
```

Solve

- The solve algorithm can now be implemented.
- We include a method called backtrack for cases where all directions fail from a given position.

Backtrack

```
def backtrack(self,pos):
    self.setMatrixValueAt(self.t.pos(),FAILED)
    if self.t.pos()[0]>pos[0]:
        while self.t.pos()[0]>pos[0]:
            self.travel(WEST)
    elif self.t.pos()[0]<pos[0]:
        while self.t.pos()[0]<pos[0]:
            self.travel(EAST)
    elif self.t.pos()[1]>pos[1]:
        while self.t.pos()[1]>pos[1]:
            self.travel(SOUTH)
    elif self.t.pos()[1]<pos[1]:
        while self.t.pos()[1]<pos[1]:
            self.travel(NORTH)
    self.setMatrixValueAt(self.t.pos(),VISITED)
```

Solve

```
def solve(self):
    if self.getMatrixValueAt(self.t.pos())==GOAL:
        return True
    else:
        savedpos=self.t.pos()
        for d in [EAST,NORTH,WEST,SOUTH]:
            if self.travel2BranchOrWall(d) != savedpos:
                if self.solve():
                    return True
                else:
                    self.backtrack(savedpos)
```