ETGG1803	Lab1: **VectorN basics**	Points: **55**
Assigned: **1/9/2017**	Due: **before 11:59pm on Wednesday 1/18/2017**	

Objectives:

- Python OOP refresher (plus an application of some "advanced" OOP features)
- Lay the framework for our vector python module.

Tasks:

1. Create a new class, **VectorN**, in a new module named **vector.py.**
    a. (**5 points**) General class structure and well-written docstrings (and comments on any tricky sections)
    b. Attributes:
        i. **mData**: a list of scalars.
        ii. **mDim**: the dimension of the vector. We can calculate this by using len(self.mData), but this (hopefully) will make the code a bit more readable.
    c. Methods:
        i. (**10 points**) **constructor** (__init__):
            1. Include a variable-length argument-list.
            2. Convert each element in that list to a float and add it to mData.
        ii. (**4 points**) **__str__**: Return a string of the form "<Vector3: 0.1, 2.8, -3.9>" or "<Vector2: -0.3, 1.9>". Don't worry about how many decimals places you show, but do add the commas and other formatting.

these 3 methods make our VectorN class behave as if it were a sequence.

        iii. (**1 point**) **__len__**: Return the dimension of the vector (i.e. the # of things in mData). This will be called when a VectorN object is passed to the len function.
        iv. (**5 points**) **__getitem__** and **__setitem__**: Should get / set, respectively, an individual value in mData. Make sure to convert to float in __setitem__.
        v. (**3 points**) **copy**:
            1. Creates a *new* VectorN object and copies all values from self's data to it.
            2. Make sure you don't make a "shallow" copy (ask about this if you're not sure).
        vi. (**4 points**) **__eq__**: Returns true if all the elements of the right-hand-side vector are the same (and there are the same number of elements), otherwise returns False.
        vii. (**3 points**) **i**: A property (using decorators) that returns a tuple of the elements of the vector converted to int's (useful in pygame). You might find the tuple and list functions handy here (ask if you'd like an example). Make sure this method doesn't alter mData's contents.
2. (**5 points**) Create (still in vector.py) another class called **Vector2** which uses inheritance to derive from **VectorN**.
    a. You shouldn't need a constructor (just use the one that was inherited)
    b. (**4 points**) Have getter and setter property decorators for an "x" and "y" property (which get / set the first and second value in the internal list)
    c. (**3 points**) Have two more properties: "radians" and "degrees" which calculate and return the radians (or degrees) of this Cartesian vector in polar space (hint: you should've seen this in 1801; bigger hint: it'll use a math function that starts with "a" and ends with "2").
    d. (**2 points**) Make sure your VectorN copy method appropriately handles the __class__ variable for Vector2's and Vector3's.
3. (**3 points**) Create a **Vector3** class (similar to Vector2) that also derives from **VectorN**.
    a. Include an "x", "y", and "z" property getter and setter
    b. Note: I'm asking you to derived from VectorN rather than Vector2 because I *don't* want to inherit the radian and degree properties (which only make sense in 2d)
4. (**3 points**) Write a function (not a method) called polar_to_vector2 that takes a radian value and a hypotenuse distance and returns a Vector2. Hint: You'll use the math library again (a function that starts with "s" and a function that starts with "c").
5. Note: I'm going to grade this with an automated "**unit-testing**" program. Make absolutely sure your attribute / method names are ***exactly*** as that described or the unit-tester will mark it as failed. The points for each method are generally all-or-none.
6. Submit your python program on blackboard on or before the due date.

This is a test program you can use to *help* debug / test your program – make sure you read the lab specifications carefully, though – this isn't testing everything…. The "correct" output is written in a comment.

```
# in vector.py
class VectorN(object):
      # Your code here

if __name__ == "__main__":
      # Note: By adding this if statement, we'll only execute the following code
      # if running this module directly (F5 in Idle, or the play button in
      # pyscripter).  But...if we import this module from somewhere else (like our
      # raytracer), it won't execute this code.  Neat trick, huh?
      import pygame

      v = VectorN(0, 0, 0, 0, 0)
      print(v)                        # <Vector5: 0.0, 0.0, 0.0, 0.0, 0.0>
      w = VectorN(1.2, "7", 5)
      # q = VectorN(pygame.Surface((10,10)))    # Should raise an exception
      # q = VectorN(1.2, "abc", 5)              # Should raise an exception
      print(w)                        # <Vector3: 1.2, 7.0, 5.0>
      z = w.copy()
      z[0] = 9.9
      z[-1] = "6"
      # z["abc"] = 9.9                            # Should raise an exception
      print(z)                        # <Vector3: 9.9, 7.0, 6.0>
      print(w)                        # <Vector3: 1.2, 7.0, 5.0>
      print(z == w)                   # False    [same as print(z.__eq__(w))]
      print(z == VectorN(9.9, "7", 6))    # True
      print(z == 5)                   # False
      print(z[0])                     # 9.9
      print(len(v))                   # 5
      print(w.i)                      # (1, 7, 5)

      w = Vector2(5, "3")
      k = Vector2(-1.4, 2)
      print(w.x)                      # 5.0
      print(w.y)                      # 3.0
      w.x = 6
      w.y = 4
      print(w)                        # <Vector2: 6.0, 4.0>
      print(w.radians)                # 0.5880026035475675
      print(w.degrees)                # 33.690067525979785

      q = Vector3(9, 0, -2)
      q.z += 5
      print(q)                        # <Vector3: 9.0, 0.0, 3.0>

      print(polar_to_vector2(1.459, 10.0))
                              # <Vector2: 1.1156359273295111, 9.937572967161127>
```