Lab 10

# Class Basics

3

|  |
| --- |
| point(1, 2) point(4, 6)  1 4 2 6  At first, the points and the class call are printed  The coordinates of the points are then printed as the class is referred to |

4

|  |
| --- |
| 5.0  5.0  0.0  The distance between the two points is printed as the method inside the class for distance is called |

5

|  |
| --- |
| There is a traceback/error |

6

|  |
| --- |
| point(3, 2) point(4, 6) |

7

|  |
| --- |
| point(3, 2) point(6, 6) point(6, 6) True  It first prints point a which was modified in the last section, then prints point b which was just modified by addind a.x to 3. Since a = b and they are dependent on each other, it prints the same thing for both b and c, and True to show that they are the same |

8

|  |
| --- |
| point(3, 2) point(6, 6) point(6, 6) False  It prints the exact same thing as before, however, b and c are not dependent on each other as c is simply just copied from b. This is shown when the is statement is used demonstrating that they are false |

9

|  |
| --- |
| False True True False  False True  The two points are compared using the method eq in the class, printing out true or false whenever it is used |

10

|  |
| --- |
| True True False  False True True  This prints out that a is a point and an object but not an integer as it was not run through the point class  It also prints out that once it is run through the point class, it is no longer a point, but it is still an object as it is returned as integers.  This tells me that the structure of the python hierarchy that integer and the class is derived from object |

# Modifying the class

1

|  |
| --- |
| This doesn’t work as you cant add both points together since they are both objects |

2

|  |
| --- |
| **def** \_\_add\_\_(self, other):  '''addition of 2 points'''  addx = self.x + other.x  addy = self.y + other.y  **return point(**addx, addy) |

3

|  |
| --- |
| point(1, 2) point(3, 4) point(4, 5)  It prints out the points. I don’t think it would work if another name is given as python probably uses it based on convention. It would work however if it was implicitly called |

4

|  |
| --- |
| No global distance() function yet.  OK |

5

|  |
| --- |
| OK |

# Basics of inheritance

1

|  |
| --- |
| a = point3d(1,2,3)  print(a)  Traceback (most recent call last):  File "<input>", line 1, in <module>  File "/Users/gordonng/Documents/Coding/Python/Cegep/Labs/lab 10/point.py", line 21, in \_\_repr\_\_  return "point({}, {})".format(self.x, self.y)  AttributeError: 'point3d' object has no attribute 'x' |

4

|  |
| --- |
| a = vector([1, 2, 3, 4])  print(a.product())  24  b = list([1, 2, 3, 4])  print(b.product())  Traceback (most recent call last):  File "<input>", line 1, in <module>  AttributeError: 'list' object has no attribute 'product'  It works at first because it calls method vector while there is no method for list, causing a traceback |

5

|  |
| --- |
| print(a + b)  [4, 4, 4]  print(a.product())  6  print(a.cross(b))  [-4, 8, -4]  It prints the calculations of the two vectors based on the method used |

6

|  |
| --- |
| **def** \_\_mul\_\_(self, other):  '''Compute the dot product of two vectors.'''  **if** len(self) != 3 **or** len(other) != 3:  **return None   return** (self[0]\*other[0]) + (self[1]\*other[1]) + (self[2]\*other[2])  OK |

|  |
| --- |
|  |