Homework 8 Instructions (The Final)

UCSD Extension CSE-41273, Summer 2021.

Please read this entire document before writing any code.

This is building upon the Point and Circle objects we have already seen. You will be turning in 2 files; their skeleton files are included in the `HW8_files.zip` file:

- `shapes.py` – This will contain the code for Point and Circle classes.
- `shapes_test.py` – This will contain the `unittest` code to test the Point and Circle classes. The starter file for this contains a number of tests already and a couple of helper functions to use.

There is a lot to this homework, but you can do it! Frankly, I think it's more tedious than difficult, but it is important to **pay attention to the details** and test what I have shown in the REPL examples for the Point and Circle classes.

Put your name at the top of each file. **Do not change the file names or make new files**. Turn in a zipped file containing only the 2 program files, as described in the *About Homework* document.

The Project

You will create a more complete 2-D Point class, along with a Circle class that uses a Point instance object to represent the center of the Circle instance. Use the classes and methods you have already created or seen in class (For example, HW4 and HW5) and modify/enhance them to fit the assignment.

Do not create new classes or subclasses. There is no inheritance in this project.

You will also be writing unit tests for the project. There is a list of tests required at the end of this document.

Everything you need to know for this has been in the lectures and handouts. Review the lectures for week 4, 5 and 7 and the handout "About Classes" from week 5. **Please also read the Notes section at the end of the instructions.**

I recommend that you start with a modified Test-Driven Development approach, since you already have some of the functionality and tests needed for this project:

1. Gather the code you have that fits this project and put it into `shapes.py`. Only use code that is needed for this project and **discard what does not apply.** (I'm looking at you, `history` attribute; you don't belong here.)
2. Write only code and tests for what you start with, and make sure the tests pass.
3. Decide what you are going to implement next, then write the test(s) for that.
4. Write the code to pass the test(s).
5. Repeat steps 3 & 4 until done.

If you want, you can start by commenting out the tests for the Circle class and write all the Point code and tests first. Once that is done, proceed to the Circle class. It doesn't really matter what order you use.

Don't forget docstrings and flake8!

You can use the REPL for simple testing (like what I show below):

```
>>> from shapes import Point, Circle
```

Mostly I show how things work in the REPL so you know how it should work.

But you are better off in my opinion with writing the test functions to test the part you are (or soon will be) writing.

Don't forget that when you change the code in the file, you need to restart the REPL and re-import Point and Circle. This is another reason why having tests is nice. Change the code & just re-run the tests.

Of course, you have to make sure your tests are correct, too. That's the fun part, hahaha. If there is something you don't understand, *please* email me.

2-D Point objects

Points are created with `x` and `y` keywords as input and default to `(0, 0)` when created with no inputs. The `x` and `y` values should be attributes of the Point instances.

```
>>> point = Point(2, 3)
>>> point.x
2
>>> point.y
3
>>> point = Point(y=5.3, x=8.75)
>>> point.x, point.y
(8.75, 5.3)
>>> point = Point()
>>> point.x, point.y
(0, 0)
```

Points should have `__str__` and `__repr__` defined so they work like this:

```
>>> point = Point(2, 3.5)
>>> point
Point(x=2, y=3.5)
>>> print(point)
Point at (2, 3.5)
>>> point = Point()
>>> point
Point(x=0, y=0)
>>> repr(point)
'Point(x=0, y=0)'
>>> point.__repr__()  # repr(object) is the same as object.__repr__()
'Point(x=0, y=0)'
>>> str(point)
'Point at (0, 0)'
>>> print(point)
Point at (0, 0)
```

Points should have a `magnitude` *property* that calculates the magnitude as if the point was a vector from `(0,0)`. Points should also have a `distance` *method* that accepts another Point object and returns the distance between the points.

```
>>> point1 = Point(2, 3)
>>> point2 = Point(5, 7)
>>> point1.magnitude
3.605551275463989
>>>> point1.distance(point2)
5.0
```

Points should be modifiable by their `x` and `y` attributes.

```
>>> point = Point()
>>> point
Point(x=0, y=0)
>>> point.x = 4.75
>>> point.y = 7.5
>>> point
Point(x=4.75, y=7.5)
```

Points should be iterable. And iterable more than once, as they are **not** iterators because iterators are *single-use* iterables. We want to be able to iterate the Point data at any time. Hint: It's in the Iteration lecture.

```
>>> point = Point(2, 3)
>>> x, y = point
>>> x, y
(2, 3)
>>> j, k = point  # If Point was an iterator, this would generate StopIteration error
>>> j, k
(2, 3)
```

Adding two points should return a new Point object, *without modifying the original points*. Hint: It's in Operator Overloading of lecture 4, and you can also see the documentation for emulating numeric types.

```
>>> point1 = Point(2, 3)
>>> point2 = Point(4, 5)
>>> id1 = id(point1)
>>> id2 = id(point2)
>>> point3 = point1 + point2
>>> point3
Point(x=6, y=8)
>>> x, y = point3
>>> x, y
(6, 8)
>>> print(point3)
Point at (6, 8)
>>> point1  # Should be unchanged
Point(x=2, y=3)
>>> point2  # Should be unchanged
Point(x=4, y=5)
>>> id1 == id(point1)  # Should be unchanged
True
>>> id2 == id(point2)  # Should be unchanged
True
```

We should also be able to do addition of Points with the shorthand method of *augmented arithmetic*, using `+=`. Please note this is a *mutating* method; in other words, it modifies the existing point and does **not** create a new Point object. See the documentation for emulating numeric types for more information; look for the discussion of implementing *augmented arithmetic assignments*.

```
>>> point1 = Point(2, 3)
>>> point2 = Point(4, 5)
>>> id1 = id(point1)
>>> point1 += point2
>>> point1
Point(x=6, y=8)
>>> id1 == id(point1)  # Should be unchanged
True
>>> point2  # Should be unchanged
Point(x=4, y=5)
```

Multiplication of a point by a scalar should return a new Point object, without modifying the original point. Likewise, multiplication of a scalar by a point should return a new Point object, without modifying the original point. The documentation link above has more information on that, too. We should also be able to do multiplication with the shorthand method, using `*=`, the *mutating* multiply operator, that does modify the original point object.

```
>>> point1 = Point(2, 3)
>>> point3 = point1 * 3
>>> point3
Point(x=6, y=9)
>>> point2 = Point(4, 5)
>>> point4 = 2 * point2
>>> point4
Point(x=8, y=10)
>>> point1  # Should be unchanged
Point(x=2, y=3)
>>> point2  # Should be unchanged
Point(x=4, y=5)
>>> id1 = id(point1)
>>> point1 *= 4
>>> point1
Point(x=8, y=12)
>>> id1 == id(point1)  # Should be unchanged
True
```

Add a method `loc_from_tuple` that allows *updating* the x, y values of a Point instance from a tuple. This is a *mutating* method and does not create a new Point instance! Returning self is optional but allows chaining of methods. Code it so the tuple input is required; use no default parameters so the system will automatically raise an error if no input is given, as shown below. Do not try to raise the `TypeError` error yourself; it comes for free if you code the method with no defaults.

```
>>> point = Point(3, 4)
>>> p_id = id(point)
>>> point.loc_from_tuple((5, 6)) # If self is returned it will print point here.
>>> point
Point(x=5, y=6)
>>> id(point) == p_id  # Should be unchanged
True
>>> point.loc_from_tuple()
Traceback (most recent call last):
  [...]
TypeError: loc_from_tuple() missing 1 required positional argument: 'coords'
```

Add a `@classmethod` called `from_tuple` to the Point class that allows creation of Point instances from a tuple containing the x and y values. The handout "More about Classes" from week 5 has more information about class methods. As in `loc_from_tuple`, code it so the tuple input is required.

```
>>> location = 2, 3
>>> location
(2, 3)
>>> point = Point.from_tuple(location)
>>> point
Point(x=2, y=3)
>>> point = Point.from_tuple()
Traceback (most recent call last):
  [...]
TypeError: from_tuple() missing 1 required positional argument: 'coords'
```

Note that **unlike most of the methods**, I am not requiring a particular name for the input argument to `from_tuple`; I used `coords` in my version.

Circle Objects

Modify the Circle class to use a Point object as the center. Use `center` and `radius` keywords, with the `center` as the first parameter.

```
>>> point1 = Point(2, 3)
>>> circle1 = Circle(center=point1, radius = 2)
>>> circle1.center
Point(x=2, y=3)
>>> print(circle1.center)
Point at (2, 3)
>>> circle1.radius
2
>>> point2 = Point(y=5, x=8)
>>> circle2 = Circle(point2, 3)
>>> circle2
Circle(center=Point(8, 5), radius=3)
>>> circle2.center
Point(x=8, y=5)
>>> circle2.radius
3
>>> point3 = Point(4, 5)
>>> circle2.center = point3
>>> circle2
Circle(center=Point(4, 5), radius=3)
>>> circle2.radius = 3.75
>>> circle2
Circle(center=Point(4, 5), radius=3.75)
```

Circle objects have `__str__` and `__repr__` defined such that:

```
>>> point1 = Point(2.75, 3)
>>> circle = Circle(center=point1, radius = 1.5)
>>> circle
Circle(center=Point(2.75, 3), radius=1.5)
>>> print(circle)
Circle with center at (2.75, 3) and radius 1.5
```

Defaults for a circle with no inputs are: center at a *default instance of a Point object*, and radius of 1.

```
>>> circle = Circle()
>>> circle
Circle(center=Point(0, 0), radius=1)
>>> print(circle)
Circle with center at (0, 0) and radius 1
```

Circle objects should have radius, diameter, and area properties that work appropriately. Circle objects cannot have a negative radius; this should raise a `ValueError`. Review Homework 5 if needed.

```
>>> circle = Circle(Point(1, 2), 2)
>>> circle.radius
2
>>> circle.diameter
4
>>> circle.area
12.566370614359172
>>> circle.radius = -1
[Traceback...]
ValueError: The radius cannot be negative!
>>> circle.diameter = -1
[Traceback...]
ValueError: The radius cannot be negative!
```

Your Circle class must define the creation order so it works like this *when used without keywords*:

```
>>> circle = Circle(Point(1, 2), 2)
>>> circle
Circle(center=Point(1, 2), radius=2)
>>> circle2 = Circle(Point(1, 2))
>>> circle2
Circle(center=Point(1, 2), radius=1)
>>> circle = Circle(1.5, Point(2, 3))
[This should raise an error that the center must be a Point; see below]
```

Verify the center of the circle instance is a Point object and raise a `TypeError` if it isn't. **Hint:** Use `isinstance()`.

```
>>> circle = Circle(center=(0, 0), radius=1.5)
[Traceback...]
TypeError: The center must be a Point!
>>> circle = Circle(center=Point(3, 4), radius=2)
>>> circle.center = (3, 4)
[Traceback...]
TypeError: The center must be a Point!
```

Allow Circle instances to be added together, which returns a *new Circle object*, and does not modify any of the original Circle instances. The new center is located at the location of `circle1.center + circle2.center`, and the radius is `circle1.radius + circle2.radius`.

```
>>> circle1 = Circle(radius=2.5, center=Point(1, 1))
>>> circle2 = Circle(center=Point(2, 3), radius=1)
>>> circle1
Circle(center=Point(1, 1), radius=2.5)
>>> circle2
Circle(center=Point(2, 3), radius=1)
>>> id1, id2 = id(circle1), id(circle2)
>>> circle3 = circle1 + circle2
>>> circle3
Circle(center=Point(3, 4), radius=3.5)
>>> circle1  # Should be unchanged
Circle(center=Point(1, 1), radius=2.5)
>>> circle2  # Should be unchanged
Circle(center=Point(2, 3), radius=1)
>>> id1 == id(circle1)  # Should be unchanged
True
>>> id2 == id(circle2)  # Should be unchanged
True
```

We should also be able to do addition of Circles with the shorthand method, using `+=`, the *mutating* addition.

```
>>> circle1 = Circle(radius=2.5, center=Point(1, 1))
>>> circle2 = Circle(center=Point(2, 3), radius=1)
>>> id1 = id(circle1)
>>> circle1 += circle2
>>> circle2  # Should be unchanged
Circle(center=Point(2, 3), radius=1)
>>> circle1
Circle(center=Point(3, 4), radius=3.5)
>>> id1 == id(circle1)  # Should be unchanged
True
```

Add a method `center_from_tuple()` that allows the center of a Circle instance to be *modified* with a tuple as input instead of a Point. Note: This is a *mutating* method and does not create a new Circle instance! Code it so the tuple input is required, but do not allow modification of the radius. If you code the `center_from_tuple()` method correctly, you get the TypeError for free from Python. As it is a mutating method, it should return the self object, to allow chaining.

```
>>> circle = Circle(Point(2, 3), 2)
>>> circle
Circle(center=Point(2, 3), radius=2)
>>> id1 = id(circle)
>>> new_center = 4, 5
>>> circle.center_from_tuple(center=new_center)
Circle(center=Point(4, 5), radius=2)
>>> circle.center_from_tuple((3, 7))
Circle(center=Point(3, 7), radius=2)
>>> id1 == id(circle)
True
>>> circle.center_from_tuple()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: center_from_tuple() missing 1 required positional argument: 'center'
```

Add a `@classmethod` called `from_tuple()` to the Circle class that allows Circle objects to be created with a tuple instead of a Point for the center's location. The resultant instance should have a Point as the center just like any other Circle object. Code it so the tuple input is required, but allow the radius to default to 1, so it is optional. If you code the `from_tuple()` class method correctly, you get the TypeError for free from Python.

```
>>> center_point = 3, 4
>>> circle = Circle.from_tuple(center=center_point)
>>> circle
Circle(center=Point(3, 4), radius=1)
>>> circle = Circle.from_tuple(center=center_point, radius=3)
>>> circle
Circle(center=Point(3, 4), radius=3)
>>> circle = Circle.from_tuple(center_point, 2)
>>> circle
Circle(center=Point(3, 4), radius=2)
>>> circle = Circle.from_tuple()
Traceback (most recent call last):
  [...]
TypeError: from_tuple() missing 1 required positional argument: 'center'
```

Some other things that you can test to make sure that things are working:

```
>>> circle = Circle(Point(2, 3), 4)
>>> center = circle.center
>>> center
Point(x=2, y=3)
>>> center.x = 5.5
>>> center.y = 6.5
>>> circle
Circle(center=Point(5.5, 6.5), radius=4)
>>> circle = Circle(Point(3.5, 2), 0)
>>> circle
Circle(center=Point(3.5, 2), radius=0)
```

Unit Tests

The `shapes_test.py` file will contain the unit tests (some have been provided, others are skeletons for you to implement) to test the Point and Circle classes. Review the Testing lecture from Week 7 if you need help, especially how to test the errors raised in the Circle class.

DO NOT CHANGE ANY OF THE EXISTING TESTS THAT COME WITH THE FILE! If you cannot get your code to pass one of my tests, then your code is wrong. If you are not convinced of that, please email me about it.

When you are testing the data of a Point or Circle instance (for example, see P-1, P-2, P-3, C-1, and C-2, etc.), **always use the point_data and circle_data helper functions**. Do not do things like compare repr(instance) with a string - that should only happen when testing that the `__repr__` method works. The main reason for this is that you are dependent upon `__repr__` working when you do that - this is not good for testing, because if your `__repr__` is broken, then all the tests that use it will be broken when I test your tests against my good Point and Circle code. It has happened multiple times, so learn from the failures of others.

Be sure to do "sanity" checks on your tests to make sure they are testing what they are supposed to test. Make sure you test all the cases that apply to each test from the examples shown from the REPL above. For example, the testing of Point addition (`+`) should verify that it returns a new Point instance **and** the original Points have not been modified. And the testing of Point `+=` should make sure that it is a mutating method, so the id of the modified Point should be the same after the operation. I will be running your `shapes.py` against my `shapes_test.py` and vice-versa, along with manually inspecting your code.

When you are finished and run the test program, it should run 46 tests.

To run the tests:

```
$ python shapes_test.py
..............................................
----------------------------------------------------------------------
Ran 46 tests in 0.004s

OK
```

If you have any questions about what these tests are supposed to test, please ask! The tests marked **Done** are included with shapes_test.py.

Tests for Points:
P-1. Create a Point with no arguments **-Done**
P-2. Create a Point with values **-Done**
P-3. Verify modification of x, y **-Done**
P-4. Verify Point is iterable **-Done**
P-4a. Verify point is not an iterator **-Done**
P-5. Verify Point magnitude property **-Done**
P-6. Verify magnitude property changes after Point changes **-Done**
P-7. Verify distance between two Point objects **-Done**
P-8. Verify Point addition **-Done**
P-8a. Verify Point `+=` mutating addition **-Done**
P-9. Verify Point str result **-Done**
P-10. Verify Point repr result **-Done**

You write:
P-11. Create a Point using `from_tuple`
P-12. Verify error when using `from_tuple` with no arguments
P-13. Verify modification of x, y using `loc_from_tuple` with values
P-14. Verify error when using `loc_from_tuple` with no arguments
P-15. Verify Point multiplied with scalar
P-16. Verify scalar multiplied with Point
P-17. Verify Point `*=` mutating multiply with scalar

Tests for Circles:
C-0. Make sure Circle centers are different objects for default **-Done**
(This test makes sure that you are creating the default circle correctly.)
C-1. Create Circle with no arguments **-Done**
C-2. Create Circle with center Point but no radius **-Done**
C-3. Create Circle with radius but no center Point **-Done**
C-4. Create Circle with center Point and radius **-Done**
C-5. Create Circle without keywords **-Done**
C-5A. Verify moving center Point of Circle works **-Done**
C-6. Verify area property **-Done**

You write:
C-7. Verify radius attribute change works
C-8. Verify area changes correctly when radius changes
C-9. Verify center attribute change works
C-10. Verify error if center is not a Point on creation
C-11. Verify error if changing center to something not a Point
C-12. Verify diameter property works
C-13. Verify diameter changes works
C-14. Verify error when creating a Circle with radius < 0
C-15. Verify error when changing radius < 0
C-16. Verify error when changing diameter < 0
C-17. Verify Circle addition
C-17a. Verify Circle `+=` mutating addition
C-18. Verify Circle str result
C-19. Verify Circle repr result
C-20. Create Circle using `from_tuple` instead of a Point
C-21. Verify error using `Circle.from_tuple` with no arguments
C-22. Create Circle using `from_tuple` with only tuple
C-23. Verify Circle modification with `center_from_tuple` method
C-24. Verify error using `center_from_tuple` with no arguments

Grading: The 26 unit tests that you write are 4 points each (104). Checking Circle center being a Point, each `from_tuple` classmethod, `loc_from_tuple`, and `center_from_tuple` are 8 points each (32). The remaining Point and Circle methods make up the rest for 200 points total.

Important Notes

- Whenever you are testing the *data* of a Point or Circle instance, **always** use the `point_data` and `circle_data` functions as in the tests that are already in the file. Do not compare using `__str__` or `__repr__` unless you are specifically testing their implementation. If your `__str__` or `__repr__` are incorrect, your tests might be wrong.

- When you are implementing **str** and **repr**, please take note of wording and spacing, etc. My tests require that they produce the strings with the *exact* spacing, wording and parentheses as shown in these instructions.
- Only create properties when you are sure you need them; and you will need some, but not all attributes need to be properties.
- **You do not need to import any modules other than the ones given.** If you *really* think you need another import, please email me and tell me what and why so I can explain why not.
- There should only be **one** place in the code that raises the "ValueError: The radius cannot be negative!" error. Likewise, there should only be **one** place that raises the "TypeError: The center must be a Point!" error. If you find you need it more than once, look at the answer file for HW5, and think about how the properties work.
- As an example of what/how to test, C-9 is "Verify center change works". Basically, for this, you create a Circle, then change the `circle.center` to a different point and make sure the circle is changed correctly.
- Tests P-15 and P-16 should verify that the original points have *not* been modified. Likewise, C-17 should do the same for the Circle objects.
- **What is the difference between the tests C-20 and C-22?** The `@classmethod` `from_tuple` for the Circle class should require the tuple input (don't give it a default), but allow the radius to be optional, with the same default as a normal Circle. Test C-20 should use `from_tuple` to create the Circle with a radius value and verify that it works; also test with and without named keyword arguments. C-22 should use `from_tuple` to create the Circle without an input radius value and verify that it is set to the required default value.
- When I say "no arguments" in the test description, I mean *no arguments*, not arguments of `None`.

Note that when you create a Circle instance, it's attributes are objects and can be treated just like any other object. For example, we can do something like this:

```
>>> from shapes import Point, Circle
>>> circle = Circle()
>>> point_x = circle.center.x
>>> point_y = circle.center.y
>>> print(f'My circle center is at ({point_x}, {point_y})')
My circle center is at (0, 0)
```

If you are having trouble with the test C-0, try this:

```
>>> from shapes import Point, Circle
>>> circle1 = Circle()
>>> circle2 = Circle()
>>> circle1
Circle(center=Point(0, 0), radius=1)
>>> circle2
Circle(center=Point(0, 0), radius=1)
>>> circle1.center.x = 2
>>> circle1
Circle(center=Point(2, 0), radius=1)
>>> circle2
Circle(center=Point(0, 0), radius=1)
```

If that is not the answer you get for `circle2`, please see my blog post Variables and References in Python, in particular, the last paragraph of the "Dire Warnings" section, and the linked "Bad Kangaroo" exercise and explanation from the book *Think Python*.

My email is dianechen.ucsdext@gmail.com. Please do not hesitate to email me if you have questions.