Homework 7 Instructions (The Final)

UCSD Extension CSE-41273, Summer 2022.

Please read this entire document before writing any code.

This is building upon the Point and Circle objects we have already seen, and adding more unit tests for our classes that are in the `shapes.py` file. You will be turning in 2 files; their skeleton files are included in the `HW7_files.zip` file:

- `shapes.py` – This will contain the code for Point and Circle classes.
- `shapes_test.py` – This file contains your `unittest` code to test the Point and Circle classes. Some tests are there; you have to write the others. **You will add code to this file to complete the tests.**

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.

There is a lot to this homework, but you can do it! 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.

Even if you think you know what you are doing, be sure to read the Important Notes at the end of this document.

I'm warning you now, I'm going to be strict on grading, so pay attention to details. For best results, any messages or fixed strings should be copied from this document, instead of typing it in, which results in typos.

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 of any kind in this project.

You will also be writing some 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. There is a bit of new-ish stuff, but it should be explained - if you have **any** questions, or something you don't understand, please email me! **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` list 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.

The tests you have to write currently raise a NonImplementedError, which you should remove when you implement that test.

You can test individual tests and multiple tests using the `-k` option:

```
$ python shapes_test.py -k test_p01_create_point_no_data
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

This feature also does some pattern matching, so for example, if you want to run only the Point tests, or if you want to run all the tests for `loc_from_tuple` or `radius`:

```
$ python shapes_test.py -k test_p
.......................
----------------------------------------------------------------------
Ran 24 tests in 0.002s

OK
$ python shapes_test.py -k loc_from_tuple
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

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 can see how each feature 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.

For operator overloading: Remember that `__add__`, `__mul__` and `__rmul__` are NOT mutating methods. That means that they create and return new objects. The methods `__iadd__` and `__imul__` are mutating methods, so they should modify the `self` object and return `self`.

2-D Point objects

- (tests P-01, P-02, P-03) Points are created with `x` and `y` keywords as input and default to `(0, 0)` when created with no inputs. The `x` and `y` attributes will need to be properties (see next paragraph). Points should be modifiable by their `x` and `y` attributes.

```
>>> point = Point(2, -3)  # positional arguments to the Point class
```

```
>>> point.x
2
>>> point.y
-3
>>> point = Point(y=5.3, x=8.75)  # keyword arguments to the Point class
>>> point.x, point.y
(8.75, 5.3)
>>> point = Point()  # Default Point instance
>>> point.x, point.y
(0, 0)
>>> point.x = 4.75  # Modify the values
>>> point.y = 7.5
>>> point.x, point.y
(4.75, 7.5)
```

- (P-04) Put in checking to verify that the input `x` and `y` are numbers; they are allowed to be either `int` or `float` type. Use `isinstance` the way we did in HW4. Also, if you have Python version 3.10+, you can use `int | float` instead of `(int, float)` for the second parameter of `isinstance` (I forgot to add that to the HW4 answer file). This error checking should only be in the setter methods, like the `radius` check in HW5.

NOTE: You will need to make `x` and `y` into properties, similar to what we did for the `radius` from HW5, so you can check them if they are modified.

```
>>> point = Point('a')
Traceback (most recent call last):
  [...]
TypeError: Invalid coordinate values for Point
>>> point = Point(x=5.25, y='a')
Traceback (most recent call last):
  [...]
TypeError: Invalid coordinate values for Point
>>> point = Point((2, 3))
Traceback (most recent call last):
  [...]
TypeError: Invalid coordinate values for Point
>>> point = Point(x=5.25, y=8)
>>> point.y = 'a'
Traceback (most recent call last):
  [...]
TypeError: Invalid coordinate values for Point
>>> point = Point(x=5.25, y=8)
>>> point.x = (2, 3)
Traceback (most recent call last):
  [...]
TypeError: Invalid coordinate values for Point
```

Note: The `[...]` represents the traceback from Python. I have removed it for brevity and also because yours will probably be different from mine.

- (P-10, P-11) Points should have `__str__` and `__repr__` defined so they work exactly like this:

```
>>> point = Point(2, 3.5)
>>> point  # This uses the __repr__ version
Point(x=2, y=3.5)
>>> print(point)  # This uses the __str__ version
Point at (2, 3.5)
>>> point = Point()
>>> point
Point(x=0, y=0)
>>> point.__repr__()  # repr(object) is the same as object.__repr__()
'Point(x=0, y=0)'
>>> repr(point)  # But always use repr()! Note it returns a string
'Point(x=0, y=0)'
```

```
>>> str(point)
'Point at (0, 0)'
>>> print(point)
Point at (0, 0)
```

- (P-07, P-08) Points should have a `magnitude` *property* that calculates the magnitude as if the point was a vector from `(0,0)`.
- (P-09) 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)  # I used a Pythagorean triple to get an "integer" answer
5.0
```

- (P-05, P-06) 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 at any time. Hint: Look 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)
>>> next(point)  # not allowed because it is not an iterator
Traceback (most recent call last):
  [...]
TypeError: 'Point' object is not an iterator
```

NOTE: You will get this error for free from Python if you code things correctly.

- (P-12) Adding two points should return a **new** Point object, *without modifying the original points*. Hint: Look 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
```

- (P-13) 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)
```

- (P-14) Add error checking so that an attempt to add a Point instance with anything that is not a Point will cause an error. Again, use `isinstance` for testing, but **for operator overrides** (and FYI, also comparison operators, which we are not implementing here), you should just return `NotImplemented`, rather than raise your own `NotImplementedError`. Python manages the error messages if you do it this way (so easy!).

```
>>> point1 = Point(x=5.25, y=8)
>>> point2 = point1 + 'a'
Traceback (most recent call last):
  [...]
TypeError: unsupported operand type(s) for +: 'Point' and 'str'
```

If the non-Point instance object is the left operand, you will get different messages, because they will come from the type of the left operand.

```
>>> point2 = 'a' + point1
Traceback (most recent call last):
  [...]
TypeError: can only concatenate str (not "Point") to str
>>> point2 = (2, 3) + point1
Traceback (most recent call last):
  [...]
TypeError: can only concatenate tuple (not "Point") to tuple
```

- (P-21, P-22) Multiplication of a point by a number (either an `int` or `float`) should return a new Point object, without modifying the original point.
- Likewise, multiplication of a number by a point should return a new Point object, without modifying the original point. The documentation for emulating numeric types linked above has more information on that, too, as it requires another dunder method.
- (P-23) We should also be able to do multiplication with the shorthand method, using `*=`, the *mutating* multiply operator, that **does** modify the original point object, instead of creating a new Point instance.

```
>>> 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
```

- (P-24) Add error checking so that an attempt to multiply a Point instance with anything that is not a number will cause an error. Again, as in addition, use `isinstance` for testing, and return `NotImplemented`, so Python manages the messages. You will get different messages, depending upon the type of the invalid operand.

```
>>> point1 = Point(x=5.25, y=8)
>>> point2 = Point(3, -12)
>>> point3 = point1 * point2
Traceback (most recent call last):
  [...]
TypeError: unsupported operand type(s) for *: 'Point' and 'Point'
>>> point3 = point1 * (1, 2)
Traceback (most recent call last):
  [...]
TypeError: can't multiply sequence by non-int of type 'Point'
>>> point3 = 'a' * point2
Traceback (most recent call last):
  [...]
TypeError: can't multiply sequence by non-int of type 'Point'
>>> point2 *= point1
Traceback (most recent call last):
  [...]
TypeError: unsupported operand type(s) for *=: 'Point' and 'Point'
```

- (P-18, P-19) 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! Code it so the tuple input argument is named `coords`, and 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 this `TypeError` error yourself; it comes for free if you code the method with no defaults. As it is a mutating method, it should return the `self` object, to allow chaining.

```
>>> point = Point(3, 4)
>>> p_id = id(point)
>>> point.loc_from_tuple(coords=(5, 6))
Point(x=5, y=6)
>>> 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'
```

Note that the message may show `TypeError: Point.loc_from_tuple() ...`, depending on your version of Python.

- (P-20) Add error checking so that if the input is anything other than a tuple, a `TypeError` is raised.

```
>>> point = Point(x=5.25, y=8)
>>> point.loc_from_tuple(3)
Traceback (most recent call last):
  [...]
TypeError: Input to loc_from_tuple should be a tuple
>>> point.loc_from_tuple('a')
Traceback (most recent call last):
  [...]
TypeError: Input to loc_from_tuple should be a tuple
```

- (P-15, P-16) 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 a tuple input named `coords` is required, though it can also be a positional argument.

```
>>> location = 2, 3
>>> location
(2, 3)
>>> point = Point.from_tuple(coords=location)  # keyword argument
>>> point
Point(x=2, y=3)
>>> point = Point.from_tuple(coords=(-3, 4.5))
>>> point
Point(x=-3, y=4.5)
>>> point = Point.from_tuple((4, -5))  # Positional argument
>>> point
Point(x=4, y=-5)
>>> point = Point.from_tuple()  # missing argument
Traceback (most recent call last):
  [...]
TypeError: from_tuple() missing 1 required positional argument: 'coords'
```

Note that the message may show `TypeError: Point.loc_from_tuple() ...`, depending on your version of Python.

- (P-17) Add error checking so that if the input is anything other than a tuple, a `TypeError` is raised.

```
>>> point = Point.from_tuple('a')
Traceback (most recent call last):
  [...]
TypeError: Input to Point.from_tuple should be a tuple
```

Circle Objects

- (C-01, C-02, C-03, C-04, C-05, C-08) Modify the Circle class to use a Point instance object as the center. Use `center` and `radius` keywords in `__init__`, with the `center` as the first parameter. If the Circle is created with an *existing* Point instance, then if that Point instance moves, the circle should move, too.

```
>>> 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)
```

(C-11, C-12) 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
```

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]
```

Default values for a circle with one or both arguments missing should be: 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 = Circle(radius=3.5)
>>> circle
Circle(center=Point(0, 0), radius=3.5)
>>> circle = Circle(Point(4, -7))
>>> circle
Circle(center=Point(4, -7), radius=1)
```

NOTE: In `Circle.__init__`, do not use `Point()` or `Point(0,0)` as the default value for the `center` argument. Use `None` for the default and handle it inside the `__init__` method. The TLDR answer for why you need to do that is that `Point()` and `Point(0,0)` are mutable objects, and you will not have a happy result if you use them for the default argument. If you want to understand why, see my comments about **init** arguments and test C-02 at the end of the instructions and the link there.

- (C-05, C-06, C-07, C-09, C-10) `Circle` objects should have `radius`, `diameter`, and `area` properties that work appropriately, as we have seen before. There is no need to modify the `diameter` and `area` properties. You will need to add another property as well for another attribute.
- (C-15, C-16, C-17) `Circle` objects cannot have a negative radius; this should raise a `ValueError` (see below). Review Homework 5 if needed. This error should be raised from only **one** location in the code.

```
>>> 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!
```

- (C-13, C-14) Verify the center of the circle instance is a Point instance and raise a `TypeError` if it isn't. **Hint:** Use `isinstance()`. This error should be raised from only **one** location in the code.

Note: You might think that you should allow the `center` to be *changed* to `None` to make a new default center Point instance, because `None` is what we use for the default for input. But that makes no sense from a user's point of view, because the user does not know that we use `None` as a default argument to make the center a default Point instance. If a user wanted to change the `center` to be a default Point instance, they can just assign it to `Point()`.

Note also: You will need to make the `center` a property in a similar manner to the `radius`.

```
>>> 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!
>>> circle = Circle(center=Point(3, 4), radius=2)
>>> circle.center = None
[Traceback...]
TypeError: The center must be a Point!
```

- (C-18) 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
```

- (C-19) 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
```

- (C-20) Add verification of the operands so that any attempt to add a Circle with something other than another Circle will generate a `TypeError`. This should work for regular addition and also `+=`. As with Point, use `isinstance` and return `NotImplemented` if the operand is invalid.

```
>>> point = Point(12, -7)
>>> circle1 = Circle(point, 5)
>>> circle2 = Circle(Point(5.25, 14), 3.55)
>>> circle3 = circle1 + (3, 4)
[Traceback...]
TypeError: unsupported operand type(s) for +: 'Circle' and 'tuple'
>>> circle3 = (2, 3) + circle1
[Traceback...]
TypeError: can only concatenate tuple (not "Circle") to tuple
>>> circle2 += (9, 3)
[Traceback...]
TypeError: unsupported operand type(s) for +=: 'Circle' and 'tuple'
>>> circle2 += 'a'
[Traceback...]
TypeError: unsupported operand type(s) for +=: 'Circle' and 'str'
```

- (C-25, C-26) 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 `coords` 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(coords=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...]
TypeError: center_from_tuple() missing 1 required positional argument: 'coords'
```

Note that the message may show `TypeError: Circle.center_from_tuple() ...`, depending on your version of Python.

- (C-27) Add verification that the input `coords` is a tuple, and raise a `TypeError` if it is not.

```
>>> circle = Circle(center=Point(12, -7), radius=5)
>>> circle.center_from_tuple('a')
[Traceback...]
TypeError: Input to center_from_tuple should be a tuple
```

- (C-21, C-22, C-23) 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. As in `center_from_tuple`, code it so a tuple input named `coords` is required, though it can also be a positional argument. 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(coords=center_point)
>>> circle
```

```
Circle(center=Point(3, 4), radius=1)
>>> circle = Circle.from_tuple(coords=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: 'coords'
```

Note that the message may show `TypeError: Circle.from_tuple() ...`, depending on your version of Python.

- (C-24) Add verification that the input to `Circle.from_tuple` is a tuple.

```
>>> circle = Circle.from_tuple(3)
Traceback (most recent call last):
  [...]
TypeError: Input to Circle.from_tuple should be a tuple
```

- 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)` or `str(instance)` with a string - that should only happen when testing that the `__repr__` or `__str__` method works. The main reason for this is that when you do that, you are dependent upon `__repr__` (or `__str__`)working - 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, and the comments in the skeleton file for the tests. For example, the testing of Point addition (`+`) should verify that it returns a new Point instance **and** that 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. I will also be running your `shapes_test.py` file against a "bad" version of `shapes.py` to make sure your tests work correctly.

When you are finished and run the test program, it should run 51 tests.

To run the tests:

```
$ python shapes_test.py
...................................................
----------------------------------------------------------------------
Ran 51 tests in 0.010s

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-01. Create a Point with no arguments. **-Done**
P-02. Create a Point with values. **-Done**
P-03. Verify modification of x, y works. **-Done**
P-04. Verify error raised if x or y is not a number. **-Done**
P-04. Verify Point is iterable. **-Done**
P-06. Verify point is not an iterator. **-Done**
P-07. Verify Point magnitude property. **-Done**
P-08. Verify magnitude property changes after Point changed. **-Done**
P-09. Verify distance between two Point objects. **-Done**
P-10. Verify Point str result. **-Done**

P-11. Verify Point repr result. **-Done**
P-12. Verify Point addition. **-Done**
P-13. Verify Point += mutating addition. **-Done**
P-14. Verify error if bad arguments to addition. **-Done**
P-15. Create a Point using `Point.from_tuple`. **-Done**
P-16. Verify error when using `Point.from_tuple` with no arguments. **-Done**
P-17. Verify error raised when `Point.from_tuple` gets bad input. **-Done**

You write:
P-18. Verify mod of x, y using `loc_from_tuple`.
P-19. Verify error when using `loc_from_tuple` with no arguments.
P-20. Verify error when `loc_from_tuple` gets bad input.
P-21. Verify Point multiplied with number.
P-22. Verify number multiplied with Point.
P-23. Verify Point *= mutating multiply with number.
P-24. Verify error multiplying bad input.

Tests for Circles:
C-01. Create default Circle with no arguments. **-Done**
C-02. Make sure Circle centers are different objects for default. **-Done**
C-03. Create Circle with Point and radius. **-Done**
C-04. Verify moving center Point of Circle works. **-Done**
C-05. Verify radius attribute change works. **-Done**
C-06. Verify area property. **-Done**
C-07. Verify area changes correctly when radius changes. **-Done**
C-08. Verify center attribute change works. **-Done**
C-09. Verify diameter property works. **-Done**
C-10. Verify diameter changes causes radius change. **-Done**

You write:
C-11. Verify Circle str result.
C-12. Verify Circle repr result.
C-13. Verify error if center is not a Point on creation.
C-14. Verify error if changing center to something not a Point.
C-15. Verify error on creation with radius < 0.
C-16. Verify error when radius changes to be < 0.
C-17. Verify error when diameter changes to be < 0.
C-18. Verify Circle addition.
C-19. Verify Circle += mutating addition.
C-20. Verify error if addition given bad input.
C-21. Test circle creation using `Circle.from_tuple`.
C-22. Verify error using `Circle.from_tuple` with no arguments.
C-23. Test circle creation using `Circle.from_tuple` with only tuple.
C-24. Verify error raised when `Circle.from_tuple` gets bad input.
C-25. Test circle modify with `center_from_tuple` method.
C-26. Verify error using `center_from_tuple` with no arguments.
C-27. Verify error using `center_from_tuple` with bad arguments.

Grading: The 24 unit tests that you write are 4 points each (96). Checking Circle center being a Point, each `from_tuple` classmethod, `loc_from_tuple`, and `center_from_tuple` are 8 points each (32). Each verification of input is 8 points (52). The remaining code make up the rest for 200 points total.

Important Notes

- Use tests P-01, P-02, and P-03 as a guideline for making the other Point tests. Use tests C-01, C-02, and C-03 as a guideline for the Circle tests you need to write. Use test P-06, P-14, etc., as guidelines for the tests that need to verify that an error is raised.
- When you are testing whether an error is raised, for example if you have `circle = Circle(radius=-1)`, you will get flake8 warnings that "circle" is not used. You can use the temporary variable of `_` and just have `_ = Circle(radius=-1)`, since it should never return if things work correctly, because the error is raised from the `Circle` class creation. See here for more info.
- 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. Why? 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 commas, and parentheses as shown in these instructions. For best results, copy from the instructions and paste into your code.
- **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.
- **What is the difference between the tests C-21 and C-23?** 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-21 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-23 should use `from_tuple` to create the Circle without an input radius value and verify that it is set to the required default value, and also test with and without named keyword arguments.
- 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-02, 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.

