Homework 5 Instructions

UCSD Extension CSE-41273, Summer 2022.

Included with this instruction file is the file `HW5.py`, containing the skeleton code for the homework.

Please read everything and follow all the directions carefully!

Put your name in the appropriate comment on line 2 of the program file `HW5.py`. **See Part 3 for directions to line 3 of the program file `HW5.py`.** Turn in a zipped file as described in the *About Homework* document.

There are 3 parts to the homework: a `Circle` class, a `BankAccount` class, and a function `my_counter`.

Note: Docstrings are required for any new methods you create on either the Circle class or the BankAccount class!

Another Note: If you somehow think you need `__getattr__` and `__setattr__`, please email me and explain why you think you need them. Then we can talk about it. 95% of homework answers using them are wrong.

Part 1. Circle class. 70 points total

Modify the Circle class given in 3 ways:
1 (a) Implement `__str__` and `__repr__` methods *exactly* as shown in the examples below.
1 (b) Add an attribute variable `history` to the *instance*. **Note that this is NOT a property, nor is it a method.** It is an attribute just like `radius`, however, it is **not an input to the instance** when it is created. Initialize `self.history` in the `__init__` method. The `history` attribute is a *list* to contain radius values that have belonged to the circle, where the last item in the list would be the same as the *current* value of the radius. In other words, I want you to keep a history of the changes to the radius of the circle object. **Each time the radius changes, add the new value to the list.**

Because you have to do things automatically *whenever the radius is modified* (namely, add to the history list), you will need to make the `radius` a *property* and also add a `radius` "setter" property method. This is similar to the `diameter` properties, with one major difference. You will need to store the *actual radius value* somewhere as an attribute that is **not** named `radius`, since the `radius` will now be a property method. I suggest using an attribute variable named `_radius` for this.

Note that once you add the radius `@property` and `@radius.setter`, they apply **everywhere**, including **all** methods of the class, which is why you need a *separate* attribute variable containing the actual radius value.

If you have the radius `@property` method just return `self.radius`, you cause recursion, because what is `self.radius`? It is the `@property` for radius, so it just keeps calling itself! When the code says `self.radius = some_new_radius`, then the `@radius.setter` is called with the `some_new_radius` value as the second argument, and if you try to set `self.radius` in the radius setter method, then again, you will have a recursion problem.

The two radius methods (property and setter) for dealing with the radius should be the ONLY methods to read/modify `self._radius`. It should appear nowhere else. Other methods such as `__init__` should **not** reference `self._radius`. See FAQ at the end of this document for more explanation if you don't want to take my word for it.

The `self._radius` and `self.history` are both *attribute variables*, NOT properties. The **only** code to use/modify `self._radius` are the two radius property methods. Oh, wait, Did I say that already? Yes, I did, because **it's important!**

The last value of the `self.history` list should always be the same value as `self._radius`, as that is the current actual radius value. **Do not use `self.history` in place of `self._radius`.** They should be completely independent of each other.

```
>>> from HW5 import Circle
>>> circle = Circle()
>>> circle
Circle(radius=1)
>>> repr(circle)
'Circle(radius=1)'
>>> print(circle)
Circle whose radius is 1
>>> str(circle)
'Circle whose radius is 1'
>>> circle.history
[1]
>>> circle.radius = 2
>>> circle.diameter = 3
>>> circle.radius
1.5
>>> circle.history
[1, 2, 1.5]
>>> circle
Circle(radius=1.5)
>>> print(circle)
Circle whose radius is 1.5
>>> circle2 = Circle(radius=2)
>>> circle2.history
[2]
>>> circle2.radius = 2
>>> circle2.history
[2, 2]
```

To verify that the `history` attribute is an *instance* attribute, you can do this test also:

```
>>> from HW5 import Circle
>>> circle = Circle()
>>> circle.radius = 2
>>> Circle.history
Traceback (most recent call last):
  [... traceback information]
AttributeError: type object 'Circle' has no attribute 'history'
```

This `AttributeError` is NOT one you put in, but is from Python, which you *should* get, to show that there is not a class-level attribute variable also named `history`, the same name as the instance-level attribute variable. **If you do not see the `AttributeError`, then your code is wrong.**

We did not have this issue with the area or diameter properties, because those property methods only read or modify the `self.radius` and do not have attribute variables associated with them. Note that `area` and `diameter` properties must continue to work! **Do not make any modifications to these property methods.**

1 (c)

Make your life easier and implement (a) and (b) so that the code works correctly before implementing (c). Once you have completed parts (a) and (b), modify the Circle class so that it will raise a `ValueError` if the `radius` or `diameter` is set to less than zero.

I would hope this is obvious, but everything from (a) and (b) should still work! And, remembering the DRY principle of *Don't Repeat Yourself*, the error should be raised from *only one* location in the code. If an error is raised, it should **not** affect the existing Circle instance. In other words, do not modify the Circle instance until you know the new value is correct.

```
>>> from HW5 import Circle
>>> circle2 = Circle(-2)
Traceback (most recent call last):
  [... traceback information]
ValueError: Radius cannot be negative!
>>>
>>> circle = Circle(radius=2)
>>> circle
Circle(radius=2)
>>> circle.history
[2]
>>> circle.radius = -1
  [... traceback information]
    raise ValueError("Radius cannot be negative!")
ValueError: Radius cannot be negative!
>>> circle
Circle(radius=2)
>>> circle.history
[2]
>>>
>>> circle = Circle()
>>> circle
Circle(radius=1)
>>> circle.history
[1]
>>> circle.diameter = -2
Traceback (most recent call last):
  [... traceback information]
ValueError: Radius cannot be negative!
>>> circle
Circle(radius=1)
>>> circle.history
[1]
```

Note: I use "[... traceback information]" as a *placeholder* since the actual traceback result may be different depending upon your computer and line number of where the error is raised. I have added blank lines to make it all easier to read.

Part 2. BankAccount class. 20 points total

Here is our old friend BankAccount in a simplified version.

2 (a). (6 points)

Implement `__str__` and `__repr__` **exactly** as shown below. Please note that we want decimal output of dollars and cents. You have already done this in the previous homework assignment.

2 (b). (5 points)

Implement *truthiness* for our BankAccount objects, such that an instance of the class BankAccount is *truthy* if the balance is greater than zero and *falsey* if the balance is less than or equal to zero.

```
>>> from HW5 import BankAccount
>>> account1 = BankAccount(100.5)
>>> account2 = BankAccount()
>>> account1
BankAccount(balance=100.50)
>>> print(account1)
Account with balance of $100.50
>>> account2
BankAccount(balance=0.00)
>>> bool(account1)
True
>>> bool(account2)
False
>>> if account1:
...     print("account1 has a positive balance.")
... else:
```

```
...     print("account1 has no money!")
...
account1 has a positive balance.
>>> account1.withdraw(200)
>>> bool(account1)
False
```

2 (c). (9 points) Implement comparisons for our BankAccount objects, such that instances can be compared based on their balance.

You should be able to do this with:

- 2 well-chosen comparison methods, and `@functools.total_ordering`. See documentation here.
- 3 well-chosen comparison methods
- All 6 comparison methods if you want code completeness.

Don't worry about checking for valid inputs; you don't need to implement an `is_valid_operand` method; you can assume you will get good input. In Real Life, you would want to do such checking, but that is not the purpose of this exercise. (making unnecessary work for yourself will not help your grade).

```
>>> from HW5 import BankAccount
>>> account1 = BankAccount(100.50)
>>> account2 = BankAccount()
>>> account3 = BankAccount(100.50)
>>> account1 == account2
False
>>> account1 == account3
True
>>> account1 != account3
False
>>> account1 != account2
True
>>> account1 < account2
False
>>> account1 > account2
True
>>> account1 <= account2
False
>>> account1 >= account2
True
>>> account1 <= account3
True
>>> account1 >= account3
True
```

Part 3, my_counter. 10 points total

Another crazy exercise to see if you can follow directions! In this exercise you will need to put a 4 digit number in the comment on line 3 of the program file `HW5.py`. **Do not leave it as `0000`.** This number will be used inside the function as the variable `my_number`.

The function `my_counter` has one input, `any_integer`. Take the input of `any_integer`, multiply it by `8888`, then multiply that by `my_number`, which is the 4 digit number from line 3 of the program file. Take that result to the third power (`**3`). Using the Counter from the Collections module, determine the frequencies of the digits that make up the number. Then **return** the string with the digit that is the most common of the digits of the result as follows:

```
'The most common digit is x, occurring y times'
```

It is not likely that 2 digits will tie for the most common digit, but just use the first one you get from `Counter`.

Remember that *strings are iterables*.

To test it against these examples, use `my_number` of `2222`. **Remember that when you turn in your code,** `my_number` **should be the 4-digit number from line 3 of the file! Do not use** `2222`

```
>>> my_counter(333)
'The most common digit is 8, occurring 6 times'
>>> my_counter(5555)
'The most common digit is 0, occurring 7 times'
>>> my_counter(1234)
'The most common digit is 4, occurring 7 times'
>>> my_counter(555)
'The most common digit is 0, occurring 7 times'
>>> my_counter(1)
'The most common digit is 7, occurring 6 times'
>>> my_counter(11)
'The most common digit is 2, occurring 6 times'
>>> my_counter(111)
'The most common digit is 3, occurring 4 times'
>>> my_counter(1111)
'The most common digit is 8, occurring 5 times'
```

FAQ for Part 1, Circle class

(From previous student questions)

Student Question. The document says: “I suggest using an attribute named `_radius` for this. ... The two radius methods (property and setter) for dealing with the radius should be the only methods to read/modify this attribute..”

Student asks: "I am a little confused about the reason why `_radius` should be used here? I've searched for it and I only find that it's a private variable and can't be accessed outside the class. Please explain more."

Long-winded Answer. That information about "private" variables is incorrect (see below). In order to implement the homework, you need to make the `radius` a property with the `@property` decorator and make a `@radius.setter` method, because *you have to do special things when the radius is modified.* Therefore you need to store the *actual radius value* somewhere else, for example in another attribute variable that I suggest be named `_radius`. It could be called `actual_radius_value` or something else, but there is a strong convention for naming this kind of variable with a *single* initial underscore character.

The two property methods for getting and setting the radius should be the only ones to read/modify this attribute variable that contains the actual radius value.

Note that Python does NOT have truly private variables, attributes or methods in the way that other languages like C++ and Java do, and an attribute named `_radius` *can* be accessed outside the class if a user of the class knows about it. *It is often said that Python is a programming language for consenting adults.* This means that the users of our code (namely, other programmers) have enough rope to hang themselves if they want to be bad programmers. The underscore preceding the attribute `_radius` is a *convention* that we use to tell users of the class (if they are looking at the code instead of the interface definition or documentation) that `_radius` is considered an "internal" attribute and a user of the class has no business reading or modifying it. If someone uses the `_radius` attribute, then they cannot complain if things don't work right if an update to the code makes things break, since it would be perfectly reasonable for the developer of our Circle class to refactor the code so that the `_radius` no longer exists or behaves differently. This is why I suggested it be called `_radius`, but you don't have to call it that. **It's just that you can't call it** `radius` **because that is now a property of the class for interfacing with users of the class, and if you call it** `radius`**, you will get Python recursion errors.**

NOTE: In your search travels (because I know you will use Google instead of thinking, haha), you may find examples of making "private" variables that are prefixed with *two underscores* instead of one, which causes Python to make a weird-looking replacement name. This is called *"name-mangling"* and its true purpose is for preventing potential name clashes or ambiguity in an inheritance hierarchy; it is *not* for making variables "private" (since there is no such thing in Python). Please resist the temptation to use name-mangling as it is completely unnecessary here and makes debugging difficult, should you need to do any debugging. **In addition, if I see name-mangling, I know that you don't understand what you are doing and are just copying something from the internet, so I will take points off.**

So a *user* of our Circle class should only use the `radius`, `area`, `diameter`, and `history` attributes.

As I said (sorry to keep repeating myself, but it doesn't seem to stick for some students) in order to implement the homework, you will need to implement the radius as property methods for both getter (`@property`) and setter (`@radius.setter`), *because you have to do special things when someone wants to change the radius value.*

More Notes

You do not need any try/except blocks anywhere. When you raise an error (such as the `ValueError`), the error should propagate up to the *calling* program. Any code that **uses** the Circle class should be responsible and have a `try/except` block around creating the Circle instance, or when changing the radius or diameter, *if there is some way that it might be possible to have a negative radius.* For example, perhaps there is some calculation (from some unknown data) of the radius for creating Circle instances, where the radius might end up being negative due to some bogus data. You would want to have a `try/except` block to capture and handle those errors, in case there is something else you want to do to deal with the improper data calculation. *The exception capturing does not belong in the class methods, because we don't care if the error is captured or not.* So the `try/except` blocks might be necessary for code that **uses** the Circle class. We are not writing any of that (except me, for my test code that I use to grade the homework). In any case, you should never use a `try/except` to catch errors *unless you know how to deal with the error*. In all other cases, the error should propagate up to the calling code.

So, if you *really* think you need a `try/except` block in this homework, please email your code to me with an explanation of why you *think* you need it. Then I can explain to you why you don't need it. :-)

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

