Systems Development for Computational Science

CS107/AC207 Fall 2022

Harvard John A. Paulson School of Engineering and Applied Sciences

F. Wermelinger
Office: Pierce 211

Homework 5

`@property` decorator, Binary search tree (BST)

Issued:	October 25th, 2022
Due:	November 8th, 2022

Note this is a 2 week exercise. *Do not procrastinate the work.*

Problem 1: Homework Submission Requirements (*10 points*)

Requirements:

1. Complete the homework on the designated branch (***5 points***)
2. Create a pull request to merge the designated homework branch into your default branch, e. g., `main` or `master` (***2 points***)
3. Merge the open pull request of the *previous homework* into your default branch **after** you have received feedback from the teaching staff and regrade requests are resolved (***3 points***).

Deliverables: your solutions to the problems below must be submitted in a directory called `submission`.

See `https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-hw`.

Problem 2: `@property` Decorator in Python (*30 points*)

Deliverables:

1. `P2.py`

Important: this problem will be fully *auto-graded*. You are expected to work with the skeleton code provided in `code/p2/P2.py`. Do not change existing code in that skeleton code. Only add your solution code in appropriate places or update values in existing variables. Otherwise you will risk that the auto-grader fails and you lose points. Read carefully the problem statement and ensure your implementation follows the specifications given.
The auto-grader works by running a set of tests on your implementation. It is therefore important that you are throwing exceptions at places in your code where values do not correspond to what is expected.

In this problem you will investigate a useful but slightly more complex *decorator* available in the Python programming language. This `@property` decorator is similar to the advanced decorator you have studied in lab 7, in the sense that it is implemented as a class and therefore allows to take advantage of more advanced Python features.

Background

This part is intended to provide you with some background. Assume you have the following Python code:

```
1  class A:
2      def __init__(self, val):
3          self.val = val
4
5      # getting value
6      def method(self):
7          return self.val # read-only
8
9      # setting value
10     def method(self, x):
11         self.val = x # write-only
```

The intention of the member function "`method`" is to either *get* or *set* the internal value (some state). The former is a *read-only* (immutable) operation and the latter will *set new state* and therefore *write* (mutable) a new value. The two member functions have the same name but a different argument list. In object oriented programming this is called *function overloading*. In your project you will do something very similar for elementary functions, but each function has a different name in this case, e. g., sin, cos, log, ln and so on.

In a programming language like C++, the two member functions defined in line 6 and 10 in the code above would be *two different* functions and the compiler could generate separate machine code for either of them. In Python, *attributes* are any values associated to an object which can be referenced by *name*.[1] Every object in Python has identity, type and value. Attributes can therefore be both, data and functions (any object really, functions are first class objects in Python as well). These attributes are stored in the special `__dict__` attribute in the object, a dictionary with keys equal to the *name of the attribute*. Because the two member functions in line 6 and line 10 above have the same name, the definition in line 10 will overwrite the earlier definition in line 6 and overloading in the sense of object oriented design is not as straightforward as it is in C++ for example. This "naive" overwriting in the case of Python happens at runtime because of its *dynamically typed* nature (functions have type too). You can test this by pasting the above code into the Python tutor (https://pythontutor.com).

Support for a small subset of three useful function overloads is provided through the `property()` class defined in the Python standard library.[2] This class can be used as a *decorator* to decorate and overload class methods for the following purpose:

1. Get the value of a data attribute in a class (*getter*)
2. Set the value of a data attribute in a class (*setter*)
3. Delete a data attribute in a class (*deleter*)

These three operations are often used to provide access to a *private data attribute* in object oriented design.[3] In Python every attribute in a class is always public. In the next part we will make use of the common convention that *private* Python attributes are indicated by a leading single underscore (`_*`).

Think of a *property* in Python as a "private" data attribute in a class for which you define access through three public member functions (part of the class *interface*) for either *getting*, *setting* or *deleting* the data property.

Note that the `@property` decorator can further be combined with the `@abstractmethod` decorator from the `abc` module in the Python standard library.[4] We will not further discuss this in the following.

The Problem

Suppose you work on a small open source Python library for video games and you have written a simple class to model animals

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species
```

[1]https://docs.python.org/3/glossary.html#term-attribute

[2]https://docs.python.org/3/library/functions.html#property

[3]Deletion is special to Python and related to the `del` statement. In C++ memory management (e.g. deletion of objects) is not exposed to the public.

[4]https://docs.python.org/3/library/abc.html

```
    def __repr__(self):
        cls = type(self)
        return f"{cls.__name__}('{self.name}', '{self.species}')"
```

You have published this code as version 1 on the Python package index and there are clients who use your code in their game application where animals can be morphed into different species with code that may look like this

```
def morph(animal, into):
    animal.species = into
    return animal
```

One day you wake up after a dreamy night and you realize that your `Animal` class should be designed for animals in an H. C. Andersen universe only,[5] and your clients should not be able to morph animals into a Cthulhu[6] which is not part of the universe you dreamed about.

One way you could implement this restriction is to define a `set_species` method to *set* the species attribute

```
class Animal:
    # class attribute for species that exist in universe
    _species_universe = [
        'cat', 'dog', 'duck', 'elf', 'goblin', 'horse', 'human', 'mermaid',
        'nightingale', 'pig', 'swan', 'wolf'
    ]

    @staticmethod
    def _valid_species(s):
        if s not in Animal._species_universe:
            raise ValueError(f'Species `{s}` does not exist in universe')

    def __init__(self, name, species):
        self._valid_species(species)
        self.name = name
        self.species = species

    def __repr__(self):
        cls = type(self)
        return f"{cls.__name__}('{self.name}', '{self.species}')"

    def set_species(self, species):
```

[5] https://en.wikipedia.org/wiki/Hans_Christian_Andersen
[6] https://en.wikipedia.org/wiki/Cthulhu

```
        self._valid_species(species)
        self.species = species
```

After studying your code a second time, you realize two issues:

1. You have forgotten to update your documentation about this change and maybe your clients will not read it anyway after you publish the new version of your library. Because attributes are public in Python, your clients will likely not change their code in the `morph(animal, into)` function they are using in their games and therefore *silently bypass* (without possibly realizing) the restriction you want to enforce in your new version.
2. You *really do not want* your clients to change any code in their applications unless absolutely necessary. Backwards portability would therefore be preserved and you may publish your new library as version 1.1 instead of a major version change. *In this way your new release will not introduce a new interface requirement which may break existing code.* In your current solution, clients are required to change lines like

   ```
   animal.species = 'cat'
   ```

   to

   ```
   animal.set_species('cat')
   ```

   in their application code.

a) ***10 points***

Implement the new restriction using a `@property` decorator in your `Animal` class.[7] Please start with the skeleton code provided in `code/p2/P2.py`. In this part you implement the `@property` *getter* version that allows your clients to obtain the value of the private `_species` data attribute of an `Animal` instance by calling a public method. After this change you should be able to do

```
>>> animal = Animal('Snoopy', 'dog')
>>> animal.species
'dog'
```

Moreover, try what happens when you execute the following statement

```
>>> animal.species = 'cat'
```

You do not need to report your observation.

b) ***20 points***

You should have observed an error when you try to *set* a new species value above. Your clients currently use your old version where statements like

[7]See `https://docs.python.org/3/library/functions.html#property`. You may use either of the two forms documented at the link.

```
>>> animal.species = 'cat'
```

are supposed to work. Add another public method in your `Animal` class that acts as a `@property` *setter*. Make sure that your implementation raises a `ValueError` if somebody tries to set a species that does not exist in your universe. Here is where you enforce your new restriction.

Finally, you should implement a third public method that acts as a `@property` *deleter*. This public method will be called when

```
>>> del animal.species
```

is executed. The `del` statement removes the binding of a name.[8] After deleting the `species` property, the statement

```
>>> animal.species
```

will result in an `AttributeError`.

You have now successfully implemented your new species restriction in your library in such a way that your clients will not need to undertake any changes in their code, once you publish the new version of your software. The Python `@property` decorator is a powerful approach to enable basic set and get interfaces for data attributes in an object oriented design manner.

[8] https://docs.python.org/3/reference/simple_stmts.html#the-del-statement

Problem 3: Binary Search Tree (*60 points*)

Deliverables:

1. P3.py

Important: this problem will be fully *auto-graded*. You are expected to work with the skeleton code provided in code/p3/P3.py. Do not change existing code in that skeleton code. Only add your solution code in appropriate places or update values in existing variables. Otherwise you will risk that the auto-grader fails and you lose points. Read carefully the problem statement and ensure your implementation follows the specifications given.
The auto-grader works by running a set of tests on your implementation. It is therefore important that you are throwing exceptions at places in your code where values do not correspond to what is expected.

A binary search tree (BST) is a binary tree with the following properties:

1. The left subtree of a node A contains only nodes with keys that *compare less* than the key in A.
2. The right subtree of a node A contains only nodes with keys that *compare greater* than the key in A.
3. The left and right subtrees are binary search tree themselves.

The skeleton code in code/p3/P3.py contains a partially implemented binary search tree. The class Node models a node in the binary search tree. A node contains a key-value pair and has references to its left and right subtrees. Note that the key parameter must support comparison operations < and >. Any type that supports these operations can be used as a key. The value parameter val can be *any* type and represents data associated with the node. The BinarySearchTree class models the binary search tree. It maintains a reference to the root node of the tree and contains methods that manipulate the tree. Each node in the tree, including the root node, are of type Node.

A possible application for a BST could be a lookup table that *maps* a key to a certain value. Both, key and value could be of the same type. If the key is a string, for example, comparison is based on a lexicographic order.[9] For example, consider the mapping of Greek to Roman gods and goddesses. Simple client code could look like this

```
>>> greek_to_roman = BinarySearchTree()
>>> greek_to_roman.put(key='Athena', val='Minerva')
>>> greek_to_roman.put(key='Eros', val='Cupid')
>>> greek_to_roman.put(key='Aphrodite', val='Venus')
>>> greek_to_roman.get(key='Eros')
'Cupid'
```

[9] https://en.wikipedia.org/wiki/Lexicographical_order

The public interface for the BST data structure is provided by the put and get methods. To insert a new node in the tree, the put method is called. If a node with a specific key exists, then its value is updated with the value specified in the put call. The get method allows to retrieve the value of a node identified by the given key. If the key does not exist, the get method must raise a KeyError exception.

You can print a binary search tree to inspect its structure. For the example above

```
>>> print(greek_to_roman)
Node(key=Athena, val=Minerva)
left  -> Node(key=Aphrodite, val=Venus)
         left  -> None
         right -> None
right -> Node(key=Eros, val=Cupid)
         left  -> None
         right -> None
```

Hint: *Assume that keys are unique (one-to-one mapping) and of the same type.*

a) ***40 points***

In this task you are implementing node insertion. The public put interface is already implemented in the skeleton code code/p3/P3.py. It delegates the call to the private _put method which you are asked to implement in this task. The _put(self, node, key, val) method inserts a new node into the binary search tree or updates the value val for an existing node with matching key. It further computes the number of nodes in the subtree (including node itself) with root node given by parameter node. See the size attribute in the Node class and the static _subtree_size method in BinarySearchTree. Inserting a new node will create a new subtree and therefore a new root node for the subtree. The _put method must return a reference to the root node of the subtrees (either existing nodes or inserted nodes). If key corresponds to an existing node in a subtree, then the _put method updates the value in the existing node with the parameter val.

Your implementation must work for any type of keys which support comparison operators < and >. This can be strings, integers, floating point numbers or any other types that implement the __lt__ and __gt__ special methods. A few further considerations

- The algorithm is *recursive*.
- What should be returned if node is None?
- Which of the two subtrees should you follow if node is not None?
- Do not forget to update the size attribute in each node. The size attribute in the Node class is the number of nodes in the subtree including the root of the subtree.

b) ***20 points***

In this task you are implementing value retrieval from a node with a given key. The public get interface is already implemented in the skeleton code code/p3/P3.py. It delegates the call to the private _get method which you are asked to implement in this task. The `_get(self, node, key)` returns the value `val` of a node identified by `key` when searching in the subtree with the root corresponding to the argument `node`. If no node with `key` exists, the method must raise a `KeyError` exception.