CSE 130 - Programming Assignment #6

Python

130 points

(see submission instructions below)

(click your browser's refresh button to ensure that you have the most recent version)

(Programming Assignment #6 FAQ)

Note: To download and install the latest 2.7.x version of Python on your home machines see this page. Remember that this is only to enable you to play with the assignment at home: The final version turned in must work on the ACS Linux machines. While you can use MacOS or Windows to begin working with Python, the code you turn in must be that required for the Linux environment.

Code Documentation and General Requirements

Code for all programming assignments should be **well documented**. A working program with no comments will **receive only partial credit**. Documentation entails providing documentation strings for all methods, classes, packages, etc., and comments throughout the code to explain the program logic. Comments in Python are preceded by `#` and extend to the end of the line. Documentation strings are strings in the first line of a function, method, etc., and are accessible using `help(foo)`, where foo is the name of the method, class, etc. It is understood that some of the exercises in this programming assignment require extremely little code and will not require extensive comments.

While few programming assignments pretend to mimic the "real" world, they may, nevertheless, contain some of the ambiguity that exists outside the classroom. If, for example, an assignment is amenable to differing interpretations, such that more than one algorithm may implement a correct solution to the assignment, it is incumbent upon the programmer to document not only the functionality of the algorithm (and more broadly his/her interpretation of the program requirements), but to articulate **clearly** the reasoning behind a particular choice of solution.

Assignment Overview

The objective of this assignment is to introduce you to some more advanced features of Python. This assignment will cover topics from classes and OOP to higher order functions and the decorator pattern.

The assignment is spread over four python files misc.py, vector.py, decorators.py, and test.py, the text files, and the sample output file decorators.out that you need to download. These files contain several skeleton Python functions and classes, with missing bodies or missing definitions, which currently contain the text `raise Failure("to be written")` or are present only as comments. Your task is to replace the text in those files with the the appropriate Python code for each of those expressions. An emphasis should be placed on writing concise easy to read code.

Assignment Testing and Evaluation

Your functions/programs **must** compile and/or run on a **Linux** ACS machine (e.g. `ieng6.ucsd.edu`), as this is where the verification of your solutions will occur. While you may develop your code on any system, ensure that your code runs as expected on an ACS machine prior to submission. You should test your code in the directories from which the zip files (see below) will be created, as this will approximate the environment used for grading the assignment.

Most of the points, except those for comments and style, will be **awarded automatically, by evaluating your functions against a given test suite**. The file, `test.py` contains a very small suite of tests which gives you a flavor of these tests. At any stage, by typing at the UNIX shell :

```
python < test.py | grep "130>>" > log
```

you will get a report on how your code stacks up against the simple tests.

The last (or near the bottom) line of the file `log` **must contain the word "Compiled" otherwise you get a zero for the whole assignment**. If for some problem, you cannot get the code to compile, leave it as is with the `raise ...`, with your partial solution enclosed below as a comment. **There will be no exceptions to this rule**. The second last line of the log file will contain your overall score, and the other lines will give you a readout for each test. You are encouraged to try to understand the code in `test.py`, and subsequently devise your own tests and add them to `test.py`, but you will not be graded on this.

Alternately, inside the Python shell, type (user input is in red):

```
>>> import test
.
.
.
130>> Results: ...
130>> Compiled
```

and it should print a pair of integers, reflecting your score and the max possible score on the sample tests. If instead an error message appears, your code will receive a zero.

Submission Instructions

1. Create the zip file for submission

Your solutions to this assignment will be stored in separate files under a directory called `pa6_solution/`, inside which you will place the files: `misc.py, vector.py, decorators.py` . These three files listed are the versions of the corresponding supplied files that you will have modified. There should be **no other files in the directory**.

After creating and populating the directory as described above, create a zip file called `<LastName>_<FirstName>_cse130_pa6.zip` by going into the directory `pa6_solution` and executing the UNIX shell command:

```
zip <LastName>_<FirstName>_cse130_pa6.zip *
```

2. Test the zip file to check for its validity

Once you've created the zip file with your solutions, you will use the `validate_pa6` program to see whether your zip file's structure is well-formed to be inspected by our grading system by executing the UNIX shell command:

```
validate_pa6 <LastName>_<FirstName>_cse130_pa6.zip
```

The `validate_pa6` program will output `OK` if your zip file is well-formed and your solution is compiled. Otherwise, it will output some error messages. Before going to step 3, make sure that your zip file passes `validate_pa6` program. **Otherwise you get a zero for the whole assignment.** If you have any trouble with this, refer to the instructions in step 1.

3. Submit the zip file via the `turnin_pa6` program

Once you've created the zip file with your solutions, you will use the `turnin` program to submit this file for grading by going into the directory `pa6_solution/` and executing the UNIX shell command:

```
turnin_pa6 <LastName>_<FirstName>_cse130_pa6.zip
```

The `turnin_pa6` program will provide you with a confirmation of the submission process; make sure that the size of the file indicated by `turnin_pa6` matches the size of your zip file. Note that you may submit multiple times, but your latest submission overwrites previous submissions, and will be the ONLY one we grade. If you submit before the assignment deadline, and again afterwards, we will count it as if you only submitted after the deadline.

Hints

May of the questions in this assignment use more advanced features of Python. You may find the following links useful.

Useful Links

- Python Library Reference
- Python Language Reference

Defining Classes

All classes used in this assignment should be new-style Python classes which have either `object` as a base class or only other new-style classes as base classes.

```
class foo(object):
    pass

class bar(foo):
    pass
```

Quirks / Features of Python

- `None` doesn't print a line when it is the return value
- `None`, `0`, and `[]` all evaluate to false when used as a predicate.
- Many objects can be iterated over with `for` loops, not just lists (you'll be defining many such objects).

Problem #0: Documentation

None of the functions in this assignment are documented (except the Failure exception). You are expected to document all modules (.py files), all classes, and all public functions with doc strings. Doc strings should describe the behavior of the function/class/module. For example:

> "The function prod takes two integers and returns their product".

If the implementation is not straightforward and obvious, there should be comments. For example:

```
# prod is implemented using a FFT to get O(n log n) time
```

Once documented you should get the following behavior at the Python prompt:

```
>>> import misc
>>> import decorators
>>> help(misc)
Screen full of documentation with all your doc strings
>>> help(decorators)
Screen full of documentation with all your doc strings
```

Problem #1: Vector class (`vector.py`)

The Vector class that you will implement will be a fixed length vector which implements a variety of operations.

(a) 10 points

Add a constructor to the `Vector` class. The constructor should take a single argument. If this argument is either an int or a long or an instance of a class derived from one of these, then consider this argument to be the length of the `Vector`. In this case, construct a `Vector` of the specified length with each element is initialized to 0.0. If the length is negative, raise a `ValueError` with an appropriate message. If the argument is not considered to be the length, then if the argument is a sequence (such as a list), then initialize with vector with the length and values of the given sequence. If the argument is not used as the length of the vector and if it is not a sequence, then raise a `TypeError` with an appropriate message.

Next implement the `__repr__` method to return a string of python code which could be used to initialize the `Vector`. This string of code should consist of the name of the class followed by an open parenthesis followed by the contents of the vector represented as a list followed by a close parenthisis. Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from vector import *
>>> Vector(3)
Vector([0.0, 0.0, 0.0])
```

```
>>> Vector(3L)
Vector([0.0, 0.0, 0.0])
>>> Vector([4.5, "foo", 0])
Vector([4.5, 'foo', 0])
>>> Vector(0)
Vector([])
>>> Vector(-4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "vector.py", line 14, in __init__
    raise ValueError("Vector length cannot be negative")
ValueError: Vector length cannot be negative
```

(b) 10 points

Implement the functions `__len__` and `__iter__` in `Vector`. The function `__len__` should return the length of the Vector. The function `__iter__` should return an object that can iterate over the elements of the `Vector`. This is most easily done using `yield()`. See here for more information.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from vector import *
>>> [x*2 for x in Vector([3,3.25,"foo"])]
[6, 6.5, 'foofoo']
>>> len(Vector(23))
23
```

(c) 10 points

Implement the + and += operators for `Vector`. The other argument to + and the second argument to += can be any sequence of the same length as the `vector`. All of these should implement component-wise addition. See here for more information on how to implement these operators. Other arithmetic operators are implemented in a similar way, however it is not required for this assignment.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from vector import *
>>> Vector([6,8,2])+Vector([4,-3,2])
Vector([10, 5, 4])
>>> Vector([6,8,2])+[4,-3,2]
Vector([10, 5, 4])
>>> (6,8,2)+Vector([4,-3,2])
Vector([10, 5, 4])
>>> v=Vector(["f","b"])
>>> v+=("oo","oo")
>>> v
Vector(['foo', 'boo'])
```

(d) 10 points

Add the method `dot` which takes either a `Vector` or a sequence and returns the dot product of the argument with current `Vector` instance. The dot product is defined as the sum of the component-wise products. The behavior of this function if any elements are not numeric is undefined.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from vector import *
>>> Vector([6,8,2]).dot(Vector([4,-3,2]))
4
>>> Vector([6,8,2]).dot([4,-3,2])
4
```

(e) 10 points

Implement the `__getitem__` and `__setitem__` methods to allow element level access to the `Vector`. Indexing should be 0 based (as in C). If the index is negative, it should translate to the length of the `Vector` plus the index. Thus, index -1 is the last element. If the index is out of range, your implementation should raise an `IndexError` with an appropriate message. This behavior should be identical to that of a list. These methods should preserve the length of the `Vector`.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from vector import *
>>> v=Vector(7)
>>> v[4]
0.0
>>> v[4]="foo"
>>> v[4]
'foo'
>>> v
Vector([0.0, 0.0, 0.0, 0.0, 'foo', 0.0, 0.0])
```

(f) 10 points

Extend your implementation of `__getitem__` and `__setitem__` methods to allow slice level access to the `Vector`. These methods should preserve the length of the `Vector`. If an assignment to a slice would change the length of the vector, raise a `ValueError` exception. The semantics otherwise should mimic those of `list`.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from vector import *
>>> v=Vector(7)
>>> v[2:4]=[4,4]
>>> v
Vector([0.0, 0.0, 4, 4, 0.0, 0.0, 0.0])
>>> v[6:2:-3]=[-1,-2]
>>> v
Vector([0.0, 0.0, 4, -2, 0.0, 0.0, -1])
```

(g) 10 points

Implement comparison functions for `Vectors`. Two vectors should be considered equal if each element in the first `Vector` is equal to the respective element in the second `Vector`. A `Vector`, a, should be considered greater than a `Vector`, b, if the largest element of a is greater than the largest element of b. If the largest elements of both are equal, then compare the second-largest elements, and so forth. If every pair compared in this fashion is equal, then a should not be considered greater than b, but a should be considered greater than or equal to b. Note that if a is greater than b, then a is also greater than or equal to b. If a is greater than b, then b is less than a. This is a nonstandard method for comparing vectors, and for a pair of vectors v and w, v>=w does not imply that v>w or v==w. When a `Vector` is compared to

something that isn't a `Vector`, they should never be equal. You can assume that a Vector will never be compared with something that is not a Vector for any comparison operators other than "==", "!=". (i.e. you don't need to handle non-vectors when you implement <, >, <=, >=). You can also assume that vectors will not be compared with a Vector of a different length.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from vector import *
>>> a=Vector([1,3,5])
>>> b=Vector([5,1,3])
>>> c=Vector([4,5,4])
>>> a<b
False
>>> a>b
False
>>> a>=b
True
>>> a>c
False
>>> a<c
True
>>> a>=c
False
>>> a<=c
True
>>> a==c
False
>>> a==a
True
>>> a!=c
True
>>> a!=[1,3,5]
True
```

Problem #2: Decorators (`decorators.py`)

Here are some links on `*args` and `**args` tutorial, python manual. In the file `decorators.py` there is an example decorator, `profiled` and stub code for decorators that you will be writing. At the bottom of the file are many examples of decorated functions. The expected output for these functions is available here: decorators.out.

(a) 30 points

Complete the definition for the decorator `traced`. When the decorated function is called, the decorator should print out an ASCII art tree of the recursive calls and their return values. The format of the tree should be as follows:

1. Print a pipe symbol followed by a space ("`| `") for every level of nested function calls.
2. Print a comma then a minus sign then a space ("`,- `") next.
3. Print the name of the function being traced followed by an open parenthesis followed by the `repr()` of all of the arguments. Arguments should be seperated by a comma followed by a space ("`, `"). After the normal arguments, print all the keyword arguments in the form keyword then equals sign then `repr()` of the value of the keyword argument. The keyword arguments should also be

seperated by a comma followed by a space. Keyword arguments should be printed in the order returnd by `dict.items()`.
4. Next increase the nesting level and call the function itself.
5. At the original nesting level, print a pipe symbol followed by a space ("| ") for every level of nested function calls.
6. Print a backquote then a minus sign then a space("`- ").
7. Finally, print the `repr()` of the return value.

The return value of the function should be return to the caller after all printing is complete. If an exception occurs in the funciton, the nesting level must be adjusted to the appropriate level where the exception is caught. See `change` for an example.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from decorators import *
>>> @traced
>>> def foo(a,b):
...     if a==0: return b
...     return foo(b=a-1,a=b-1)
...
>>> foo(4,5)

,- foo(4, 5)
| ,- foo(a=4, b=3)
| | ,- foo(a=2, b=3)
| | | ,- foo(a=2, b=1)
| | | | ,- foo(a=0, b=1)
| | | | `- 1
| | | `- 1
| | `- 1
| `- 1
`- 1
1
```

(b) 30 points

Complete the definition of the `memoized` decorator. When the decorated function is called, the decorator should check to see if the function has already been called with the given arguments. If so, the decorator should return the value the the function returned when it was last called with the given arguments. If the function last threw an exception when called with the given arguments, the same exception should be thrown again. If the function has not been called with the given arguments, then call it and record the return value or exception. Then return the return value or raise the thrown exception.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from decorators import *
>>> from time import sleep
>>> @memoized
>>> def foo(a):
...     sleep(a)
...     return a
...
>>> foo(5)
# 5 second pause
5
>>> foo(5)
```

```
    # practically instantaneous
    5
```