When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download the **q3helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **point.py** and **private.py** modules online by Thursday, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on Friday morning.

---

1. (20 pts) Complete the class **Point**, which stores and manipulates **(x,y,z)** coordinates (all **int**). As specified below, write the required methods, including those needed for overloading operators. Exceptions messages should include the class and method names, and identify the error (including the value of all relevant arguments). Hint see the **type_as_str** function in the **goody**.py module.

   1. The class is initialized with three **int** coordinates (the **x** coordinate first, the **y** coordinate second, the **z** coordinate third). If any parameter is not an **int**, raise an **AssertionError** with an appropriate string describing the problem/values.
   2. Write methods that return (a) the standard **repr** function of a **Point**, and (b) a **str** function of a **Point**: **str** for **Point(1,2,3)** returns **'(x=1,y=2,z=3)'**.
   3. Write a method that interprets coordinate **(0,0,0)** as **False** and any other coordinate as **True**.
   4. Overload the **+** operator to allow adding two **Point** objects, producing a new **Point** object as result: its **x** coordinate is the sum of the **x** coordinates of the operand **Point** objects, and its **y** coordinate is the sum of the **y** coordinates of the operand **Point** objects, and its **z** coordinate is the sum of the **z** coordinates of the operand **Point** objects. If the right operand is not a **Point**, raise a **TypeError** exception with an appropriate string describing the problem/values.
   5. Overload the ***** operator to allow multiplying an **int** by a **Point** or a **Point** by an **int**, producing a result **Point**: its **x** coordinate is the product of the **int** and the **Point**'s x coordinate, and its **y** coordinate is the product of the **int** and the **Point**'s **y** coordinate, and its **z** coordinate is the product of the **int** and the **Point**'s **z** coordinate. If the other operand is not an **int**, raise a **TypeError** exception with an appropriate string describing the problem/value.
   6. Overload the **<** operator to allow comparing two **Point** objects. The left **Point** is less than the right one if the distance from **Point(0,0,0)** to the left **Point** is less than the distance from **Point(0,0,0)** to the right one. Also allow the right operand to be an **int** or **float**: in this case, return whether the distance from **Point(0,0,0)** to the left **Point** is less than the right operand. If the right operand is any other type, raise a **TypeError** exception with an appropriate string describing the problem/values.
   7. Write the **__getitem__** method to allow the class objects to be indexed by either a **str** with value **'x'** or **'y'** or **'z'**or an **int** with value **0** or **1** or **2**: an index of **'x'** or **0** returns the **x** coordinate; an index of **'y'** or **1** returns the **y** coordinate; an index of **'z'** or **2** returns the **z** coordinate. If the index is not one of these types or values, raise an **IndexError** exception with an appropriate string describing the problem/values.
   8. Write the **__call__** method to allow an object from this class to be callable with three **int** arguments: update the **x** coordinate of the object to be the first argument, and the **y** coordinate of the object to be the second argument, and the **z** coordinate of the object to be the third argument. Return **None**. If any parameter is not an **int**, raise an **AssertionError** with an appropriate string describing the problem/values.

   The **q3helper** project folder contains a **bsc1.txt** file (examine it) to use for batch-self-checking your class, via the **driver.py** script. These are rigorous but not exhaustive tests. Incrementally write and test your class.

2. (5 pts) Complete the class `C`, which should implement **semi-private** variables (functioning much like `__attributes`: attributes prefaced by double-underscore in Python) which were discussed in lecture (from the **class review** lecture nodes). Here, any attributes defined in the `__init__` method will be considered semi-private, usable only in methods in that class (except there is one loophole). Remember that `o.__dict__` stores object `o`'s namespace; inside one of `C`'s methods refer to by `self.__dict__`. Use this information in your solution. Submit your code with all methods as they appear in the download, except for `__setattr__` and `__getattr__`.

Complete class `C` by writing the `__setattr__` and `__getattr__` methods. Assume below `C` that `o = C()`

1. The `__init__` method can define any number of attributes. I can test your code with an `__init__` that has a different body than the one included in the class (which sets two attributes `a` and `b`). As you will see below, `__setattr__` will store these attributes in a special form (as the attributes `private_a` and `private_b`). Then, `__getattr__` and `__setattr_` will automatically use these private attributes only when inside of methods defined inside class `C`. Other non-private attributes can be added/used outside of the methods define inside class `C`.
2. Write the `__setattr__` method to work as follows
   (a) raise the `NameError` exception whenever it is called with a name that explicitly starts with `private_`: e.g., writing `self.private_a = ...` or `o.private_a = ...` anywhere raises this exception.
   (b) otherwise, when a `self` attribute is set in the `__init__` method, set the name prefixed by `private_` to its value: e.g., for the `__init__` supplied, the name `private_a` will be set to `1`, and the name `private_b` will be set to `2`: `o.__dict__` will be `{'private_a': 1, 'private_b': 2}`
   (c) otherwise, if the name prefaced by `private_` that is already an attribute in the object
      i. if the attribute is being set in a method defined inside the class `C`, then set the name prefixed by `private_` to its value: it is OK to set a privately defined attribute in any of `C`'s methods.
      ii. otherwise raise the `NameError` exception: it is not OK to set a privately defined attribute outside of `C`'s defined methods.
   (d) otherwise, just set just the name (with no prefix) to its value.
3. Write the `__getattr__` method (remember, it is called if we try to access an attribute name that is not already in the object's namespace: if we define `self.x` in `__init__` it will be store in the object's namespace as `private_x`; so if we access `self.x` or `o.x` Python won't find `x` in the object's namespace, so it will call `__getattr__` to try find it binding) to work as follows.
   (a) if the name prefaced by `private_` is already an attribute in the object
      i. if the attribute is being gotten in a method defined inside the class `C`, return the value associated with the name prefixed by `private_`: it is OK to get a privately defined attribute in any of `C`'s methods.
      ii. otherwise raise the `NameError` exception: eit is not OK to get a privately defined attribute outside of C's defined methods.

Note: similarly to double-underscore, if we write `o.private_a` Python will find this attribute in the namespace, and therefore will never call `__getattr__` so this is a loophole in this mechanism: attributes cannot be 100% private for use inside the methods in a class. To fix this loophole we need to know about inheritance and the `__getattribute__` method (which we will cover later in the quarter).

How can we determine whether `__setattr__`/`__getattr__` is called from `__init__` or a method inside class `C` or somewhere outside of class `C` methods? It requires a little Python magic, which I'll explain how to use. In these methods I have written as a first line `calling = inspect.stack()[1]`: with this line, Python finds the function/method call `1` back on the **call stack**, storing into `calling` information about the function/method that called `__setattr__`/`__getattr__`; `calling` is a named-tuple, with important fields `function` and `frame`.

1. `calling.function` is a string naming the function/method: it could be `'_init_'` or `'bump'` (methods defined in class `C`) or `'<module>'` or `'f'` (some statement/function defined in a module).

2. For any function/method name defined in class **C**, then **C.__dict__[calling.function].__code__** is the code object of **C**'s method with that name.
3. **calling.frame.f_code** is a code object for the **calling** function/method

So, to determine whether **__setattr__**/**__getattr__** is called from a method defined in class **C**, we check whether the code object for that method/function is the same code object for the method/function name defined in **C**. I have written the static **in_C** method to perform this comparison. It is called like **C.in_C(calling)**. Use code based on this to figure a way to determine whether **__setattr__** was called from **C**'s **_init_** method.

The **q3helper** project folder contains a **bsc2.txt** file (examine it) to use for batch-self-checking your class, via the **driver.py** script. These are rigorous but not exhaustive tests.