1. (20 pts) Complete the class **Fraction**, which stores and manipulates fractions stored as pairs of numbers: a numerator and a denominator. As specified below, write the required methods, including those needed for overloading operators. Exception messages should include the class and method names, and identify the error (including the bindings of all relevant names).

1. **Fraction** class objects are initialized with one or two **int** values (a mandatory numerator first, an optional denominator second, which if absent should default to **1**). Ultimately, the values stored in the two self/instance names (you **must** call them **num** and **denom**, because I used those names in the **__call__** method defined in this class) should satisfy the following properties: (a) the denominator must be positive: note $\frac{n}{-d} = \frac{-n}{d}$; (b) if the numerator is **0**, the denominator must be **1**; (c) the numerator and denominator must have no common factors: divide each by their Greatest Common Divisor (see the **gcd** method written in the module). For example **Fraction(4,-6)** results in a numerator of **-2** and a positive denominator of **3**.

   Raise an **AssertionError** exception if either argument is not an **int** or if the denominator is **0**.
2. Write a **__repr__** method for **Fraction**: **repr** for **Fraction(4,-6)** returns **'Fraction(-2,3)'**; write a **__str__** method for **Fraction**: **str** for **Fraction(4,-6)** returns **'-2/3'**. See the **__call__** method for another kind of returned string.
3. Write a **__bool__** method that interprets any **Fraction** equal to **0** as **False** and not equal to **0** as **True**; recall from the requirements of **__init__**, if a **fraction** is **0**, its numerator will be **0** and its denominator **1**.
4. Write a **__getitem__** method that returns the numerator for an argument of **0** or any string that is non-empty and a prefix of **'numerator'**; returns the denominator for an argument of **1** or any string that is non-empty and a prefix of **'denominator'**. For example, if **x = Fraction(1,2)**, then **x[0]**, and **x['num']** return **1**; hint: see **str.find** function or prefix processing. Raise a **TypeError** in other cases: eg. **x['']** or **x[2]** or **x['foo']**.

Methods specified in 5-6 must return a newly constructed **Fraction** object. No methods should mutate fractions.

5. Write the **__pos__**, **__neg__**, **__abs__**, methods so the unary operator **+** returns a copy of that fraction, **-** returns the negation of the fraction, and the function **abs** returns the absolute value of the fraction
6. Overload the **+**, **-**, *****, **/** (named **__truediv__**), and ****** operators to allow addition, subtraction, multiplication, (true) division and raising a fraction to a power. In the case of all but ******, allow the right operand to be a **Fraction** or **int**; for ****** it must be an **int** (positive or negative: where $x^{-2} = \frac{1}{x^2}$).

   Hints: (a) My methods were all 2-5 lines long. (b) They always called the **_validate_arithmetic** method (pre-written in the class) first, to determine whether or not to raise an exception and if so, its text. (c) We can convert any **int x** into an equivalent **Fraction** by **Fraction(x)** or **Faction(x,1)**: so, to perform arithmetic on a **Fraction** and an **int**, operate on the **Fraction** and a **Fraction** equivalent to the **int**. (d) Simplify the **__r...__** methods by commutivity (and in the case of binary -, using the unary -/**__neg__**).
7. Overload the **==**, **<**, and **>** operators to allow comparing two **Fraction** objects. Two fractions are equal if they have the same numerator and denominator. Note that $\frac{a}{b} < \frac{c}{d}$ when $ad < bc$.

   Hints: Also, allow comparing a **Fraction** to an **int**. See the Hints in part 6 (but for these operators, use the **_validate_relational** method instead).
8. Write the **__setattr__** method to prohibit a change to the **num** and **denom** self/instance variables once they are initialized in **__init__**. Hint: if a name is not already in **__dict__** allow it to be added; but if a name is already in **__dict__** raise a **NameError** exception.

The **q3helper** project folder contains a **bscq31F20.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 notes). 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_** 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.

I can test your **setattr__** and **__getattr__** methods in any class **C** including ones that define more methods, or change the bodies of the other methods shown in the download; I can also test it with other functions defined outside the class.

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 `bscq32F20.txt` file (examine it) to use for batch-self-checking your class, via the `driver.py` script. These are rigorous but not exhaustive tests.