This document walks through the design of the Calculator class that is used as an example of a class that we want to perform unit testing on.

Class description

This class was quickly written, and therefore, may not be the most optimal way of generating such a class. The intention of this class is to accept single-line, simple, mathematical expressions to compute their values using the order of operations. We restrict ourselves to not using external libraries since these libraries have often already been tested and have conditionals which may identify errors outside of our unit test. Within the body of the class definition, the 'call' method is defined. This method will be used in subsequent methods to call other functions within the class by their name.

```
def call(self, name, v1, v2):
    f = getattr(self, name)
    return f(v1, v2)
```

The important part of this function is the use of the default getattr() method to retrieve the callable function by name from within the body of, 'self', the class call() is defined in.

The next function is to scan the user-defined input for significant and relevant characters for calculating expressions.

```
def scan(self):
    ...
```

After logging some of the characters and their indices, we determine the number of quantites that the expression holds. This is done by quantifying the number of pairs of parentheses that the expression has and then the range of indicies each quantity is held at in the user-defined input string is returned.

```
def quantities(self):
    ...
```

The calculate method first calls the quantities() method to determine the index ranges for values and operators in and outside of quantities. In a production level environment, it would be best to split such a function into smaller functions. The actions are gathered into a list by matching a given operator to its associated named key which is held in a previously defined dict instance attribute. The process of recognizing an operator, calling the named function represented by this operator, and returning the answer for a given quantity is repeated for all quantities in the expression. We have not completed the code to also compute expressions held outside of those indicated as quantities. This will be identified by a simple example of unit testing.

```
  def calculate(self):
      ...
```

To write a class for unit testing our basic calculator, we can use the 'unittest' module that is packaged with Python. The 'TestCase' class will serve as the base class for our test class. There are a number of ways to configure your unit testing module. Therefore, writing a series of classes to test each class of your programs, as we do in this example, may not be best suited for all development environments.

```
  class TestCalculator(unittest.TestCase):
      ...
```

Once we inherit from 'unittest.TestCase', method with the test_ prefix will update TestCase attributes associated with the unittest module. The methods belonging to 'unittest.TestCase' are accessed by first initializing the base class with the super() method. A series of test strings are also defined for identification of the type of string being examined by the 'assertEqual' method using a dictionary.

Running this code, we will see that our error occurs after testing for "Single Quantity Addition".

One would need to ensure that, as they write their Test() class for unit testing their other classes / modules, that their tests are sufficiently written to capture (un)expected behavior. For writing more sophisticated tests, the documentation has lists and examples to work from which can be found at https://docs.python.org/3/library/unittest.html.