Feedback on Quiz 1

Avoid redundant tests

Some codes for __add__ like this:

```
def __add__(self,other):
    if self.currency == other.currency:
        new_value = self.value + other.value
        return Money(new_value, self.currency)
    elif self.currency != other.currency:
        raise ValueError(
       "cannot add money with different currencies")
```

Extra "`elif`" test is redundant.

In Java or C# this code would not compile because method is returning something (Money) in "if" block, but nothing is returned in the missing "else" case.

Misuse of Scope

Python lets you do this, but its bad code.

```
def __add__(self,other):
    if self.currency != other.currency:
        raise ValueError(
       "cannot add money with different currencies")
    else:
        sum = self.value + other.value
    return Money(sum, self.currency)
```

"scope" is the region of code where a variable is defined and visible.

In other languages, "sum" would not be defined outside the "else" block.

Avoid putting long code in "if ... else"

Long code inside "if" or "else" block is harder to read.

a) handle the error case(s) first -- return or raise exception

b) then handle the valid case, which is usually longer

```
def __add__(self,other):
    # validate the parameters first
    if self.currency != other.currency:
        raise ValueError(
       "cannot add money with different currencies")
    # now, What the method is supposed to do...
    sum = self.value + other.value
    return Money(sum, self.currency)
```

Rule #1: Code should be easy to read.

Don't read instructions

When your code is finished there should not be any TODO comments!

```
Java:    //TODO or // TODO
Python: #TODO  or # TODO
```

Many money.py contain:

```
#TODO write an __add__() method.  And remove this
  "TODO" comment.
def __add__(self,other):
    if self.currency != other.currency:
        raise ValueError(
       "cannot add money with different currencies")
    sum = self.value + other.value
    return Money(sum, self.currency)
```

Don't read instructions

3. Write 3 unit test methods in test_money.py for the __add__ method, invoked using the + operator.

-- Many students put all tests in **one** method.

This Doesn't Work

```
def test_add_different_currency(self):
    a = Money(10, "Baht")
    b = Money( 5, "Ringit")
    self.assertRaises( ValueError, a+b )
```

Reason:

`a+b` is evaluated before the **`assertRaises`** is called.

So the ValueError is not caught.

Just like `print(2+3)`.

2+3 is evaluated first, then print(5) is called.

What the Python Docs say:

`assertRaises(`*exception*, *callable*, **args*, ***kwds***`)`**

Test that an *exception* is raised when *callable* is called with any positional or keyword arguments.

Callable:

something that you can call. :-)

Such as a function.

Just write the function name -- no ( ).

Define a function to call

```
def test_add_different_currency(self):
    def adder():
        a = Money(10, "Baht")
        b = Money( 5, "Ringit")
        sum = a+b  # should raise exception
    self.assertRaises( ValueError, adder)
```

callable

assertRaises is called with `adder` (function) as parameter.

assertRaises will invoke `adder`() itself.

So, assertRaises can catch exception raised by adder

Simpler: use __add__ as callable

```
def test_add_different_currency(self):
    a = Money(10, "Baht")
    b = Money( 5, "Ringit")
    self.assertRaises( ValueError,
                       a.__add__, b)
```

callable ***args for callable***

assertRaises will invoke `a.__add__( b )`

Much simpler: with assertRaises

```
def test_add_different_currency(self):
    with self.assertRaises(ValueError):
        a = Money(10, "Baht")
        b = Money( 5, "Ringit")
        sum = a+b  # should raise exception
```

Function as parameter

A function can be a parameter to another function. Sometimes called "higher order functions".

```
def eval_and_print(fun, x):
    print( fun(x) )

# you can assign function to a variable
f = math.sqrt
# you can pass function as an argument
eval_and_print(f, 25)
```

Ref: Programming 2 Lecture 6, "*Functional Programming*".

Quality Deduction (-1 point)

Must use Python naming convention.

Must use basic Python coding convention.

```
class Test_money(unittest.TestCase):
    # use of attribs is bad but no penalty
    a = Money(10, "Baht")
    b = Money(10, "Ringit")
    def testConstructor(self):
        self.assertEqual(10, a.value)
    def test_equal(self):
        self.assertFalse(a==b)
```

Quality Corrections

Class name is **TitleCase**, method names are **snake_case**.

Blank line between class & attributes & between methods.

```
class TestMoney(unittest.TestCase):

    # better- unittest calls setUp before each
    def setUp(self):                     # test
        self.a = Money(10, "Baht")
        self.b = Money(10, "Ringit")

    def test_constructor(self):
        self.assertEqual(10, self.a.value)

    def test_equal(self):
        self.assertFalse(self.a == self.b)
```

Reality Check

The unit testing assignment (Fraction) is *excellent* prep for Quiz 1. Why the big difference in scores?

Student ID	Fraction (20 pts)	Quiz 1 (30 pts)
xxxxxxxx03	19	**0**
xxxxxxxx81	18.5	**0**
xxxxxxxx54	18.5	**0**
xxxxxxxx20	18	**5**
xxxxxxxx78	17.5	**0**
xxxxxxxx57	17	**0**
xxxxxxxx01	16.5	**0**
xxxxxxxx31	14.5	**4**
xxxxxxxx21	8.5	**0**

Learning is Individual Effort

Help your friends learn by ...

- *<u>not</u> giving them the solution*
- *<u>not</u> sharing code*
- *refer them to TAs for help*

TAs and instructor are happy to help.

This is an opportunity to learn -- don't waste it.

Don't risk your friends' grade

- *if I detect copying, both the "source" and copier get F*

- *no second chance*