Lab Report

By: Jahmil Ally (501045419)

Observations and Analysis

Q1:

I converted my past lab 2 into Python to implement this lab to test for branch and statement coverage. Using “coverage run --branch -m unittest discover -s test” and “coverage html” I was able to get the results.

```
PS C:\Users\jazza\Documents\Coding Projects\COE891Labs\lab6> coverage run --branch -m unittest discover -s test
.........................................
----------------------------------------------------------------------
Ran 44 tests in 0.007s

OK
```

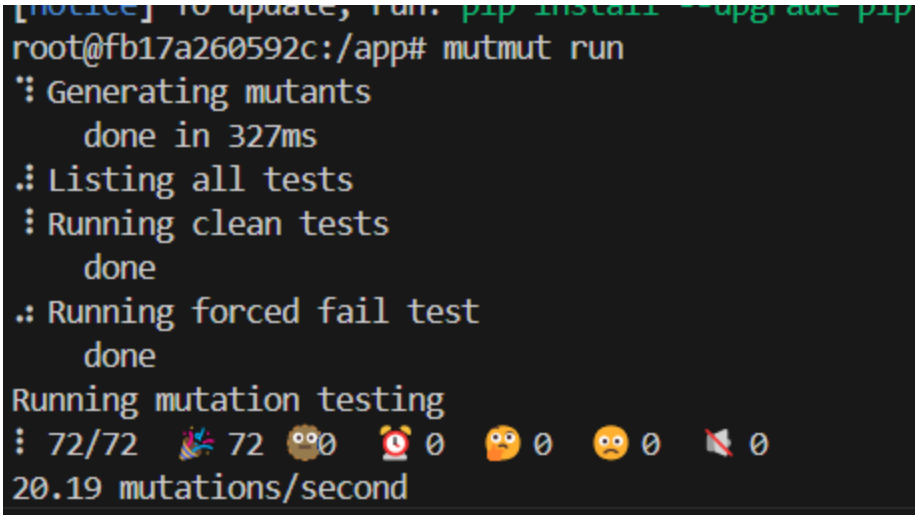
File	statements	missing	excluded	branches	partial	coverage
main\IMoney.py	26	8	0	0	0	69%
test\test_Money.py	113	1	0	2	1	98%
test\test_MoneyBag.py	124	1	0	2	1	98%
main\Money.py	38	0	0	6	0	100%
main\MoneyBag.py	73	0	0	30	0	100%
Total	374	10	0	40	2	97%

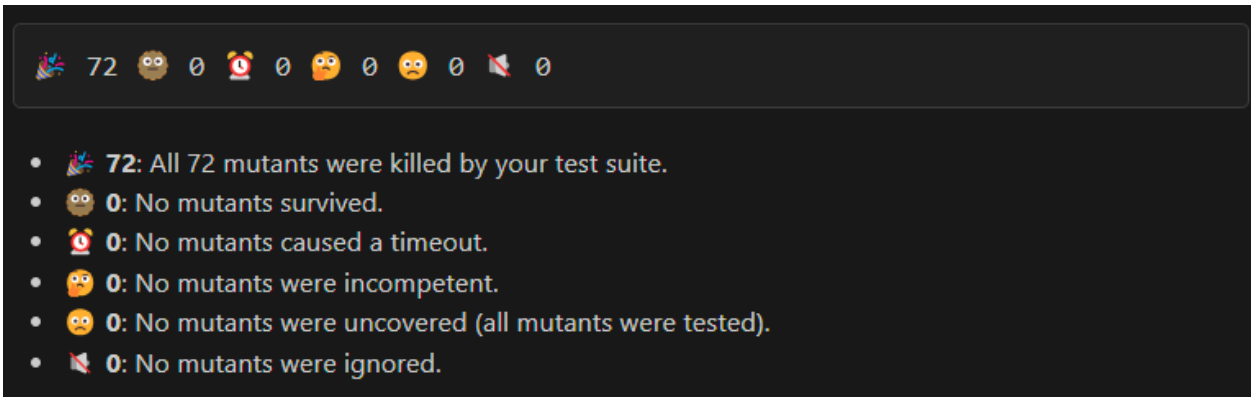
The 8 missing statements for execution from IMoney.py is from the 8 functions that have a pass statement which never get executed as it is an interface. The 2 other missing lines belong to the mains for each respective test.

In order to test mutation testing in python I had to open my project in docker using “docker run -it --rm -v "C:\Users\jazza\Documents\Coding Projects\COE891Labs\lab6:/app" python:3.9 bash”

I installed my necessary requirements “pip install -r requirements.txt” and then ran the command to test. This uses various methods of mutation such as:

- Integer literals are changed by adding 1. So 0 becomes 1, 6 becomes 7, etc.
- < is changed to >=
- break is changed to continue and vice versa
- The mutations would be as subtle as possible. See mutmut __init__.py for the full list.


root@fb17a260592c:/app# mutmut run
Generating mutants
done in 327ms
Listing all tests
Running clean tests
done
Running forced fail test
done
Running mutation testing
72/72 🎉 72 🙁 0 ⏰ 0 🤔 0 😕 0 🔇 0
20.19 mutations/second


🎉 72 🙁 0 ⏰ 0 🤔 0 😕 0 🔇 0
🎉 72: All 72 mutants were killed by your test suite.
🙁 0: No mutants survived.
⏰ 0: No mutants caused a timeout.
🤔 0: No mutants were incompetent.
😕 0: No mutants were uncovered (all mutants were tested).
🔇 0: No mutants were ignored.

Appendix:

Q1:

IMoney.py

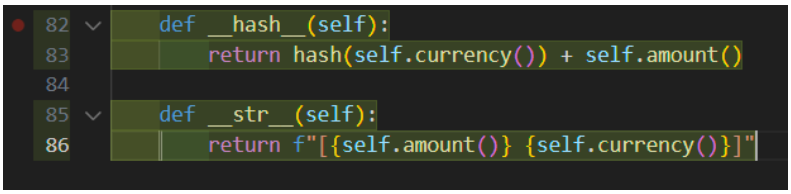
main > IMoney.py > IMoney

```python
from abc import ABC, abstractmethod

class IMoney(ABC):
    """
    The common interface for simple Monies and MoneyBags
    """

    @abstractmethod
    def add(self, m):
        """
        Adds a money to this money.
        """
        pass

    @abstractmethod
    def add_money(self, m):
        """
        Adds a simple Money to this money. This is a helper method for
        implementing double dispatch.
        """
        pass

    @abstractmethod
    def add_money_bag(self, s):
        """
        Adds a MoneyBag to this money. This is a helper method for
        implementing double dispatch.
        """
        pass

    @abstractmethod
    def is_zero(self):
        """
        Tests whether this money is zero.
        """
        pass

    @abstractmethod
    def multiply(self, factor):
        """
        Multiplies a money by the given factor.
        """
        pass

    @abstractmethod
    def negate(self):
        """
        Negates this money.
        """
        pass

    @abstractmethod
    def subtract(self, m):
        """
        Subtracts a money from this money.
        """
        pass

    @abstractmethod
    def append_to(self, m):
        """
        Append this to a MoneyBag m.
        """
        pass
```

Money.py

main > Money.py > Money

```python
from main.IMoney import IMoney
from main.MoneyBag import MoneyBag  # Assuming MoneyBag is implemented elsewhere

class Money(IMoney):
    """
    A simple Money implementation.
    """

    def __init__(self, amount, currency):
        """
        Constructs a money object with the given amount and currency.
        """
        self._amount = amount
        self._currency = currency

    def add(self, m):
        """
        Adds a money to this money. Forwards the request to the add_money helper.
        """
        return m.add_money(self)

    def add_money(self, m):
        """
        Adds a simple Money to this money.
        """
        if m.currency() == self.currency():
            return Money(self.amount() + m.amount(), self.currency())
        return MoneyBag.create(self, m)

    def add_money_bag(self, s):
        """
        Adds a MoneyBag to this money.
        """
        return s.add_money(self)

    def amount(self):
        """
        Returns the amount of this money.
        """
        return self._amount

    def currency(self):
        """
        Returns the currency of this money.
        """
        return self._currency

    def is_zero(self):
        """
        Checks if this money is zero.
        """
        return self.amount() == 0

    def multiply(self, factor):
        """
        Multiplies this money by a given factor.
        """
        return Money(self.amount() * factor, self.currency())

    def negate(self):
        """
        Negates this money.
        """
        return Money(-self.amount(), self.currency())

    def subtract(self, m):
        """
        Subtracts a money from this money.
        """
        return self.add(m.negate())

    def append_to(self, m):
        m.append_money(self)

    def __eq__(self, other):
        if self.is_zero() and isinstance(other, IMoney):
            return other.is_zero()
        if isinstance(other, Money):
            return self.currency() == other.currency() and self.amount() == other.amount()
        return False
```

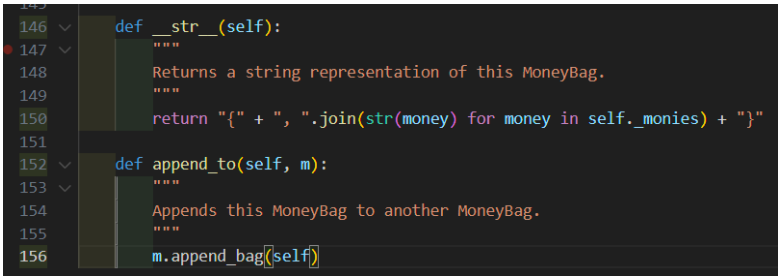

82 def __hash__(self):
83 return hash(self.currency()) + self.amount()
84
85 def __str__(self):
86 return f"[{self.amount()} {self.currency()}]"

MoneyBag.py

main > MoneyBag.py > MoneyBag

```python
from main.IMoney import IMoney

class MoneyBag(IMoney):
    """
    A MoneyBag defers exchange rate conversions. For example, adding
    12 Swiss Francs to 14 US Dollars is represented as a bag
    containing the two Monies 12 CHF and 14 USD. Adding another
    10 Swiss francs gives a bag with 22 CHF and 14 USD. Due to
    the deferred exchange rate conversion, we can later value a
    MoneyBag with different exchange rates.
    """

    def __init__(self):
        """
        Constructs an empty MoneyBag.
        """
        self._monies = []

    @staticmethod
    def create(m1, m2):
        """
        Creates a MoneyBag containing two IMoney objects.
        """
        result = MoneyBag()
        m1.append_to(result)
        m2.append_to(result)
        return result.simplify()

    def add(self, m):
        """
        Adds a money to this MoneyBag.
        """
        return m.add_money_bag(self)

    def add_money(self, m):
        """
        Adds a Money to this MoneyBag.
        """
        return MoneyBag.create(m, self)

    def add_money_bag(self, s):
        """
        Adds another MoneyBag to this MoneyBag.
        """
        return MoneyBag.create(s, self)

    def append_bag(self, a_bag):
        """
        Appends all monies from another MoneyBag to this MoneyBag.
        """
        for money in a_bag._monies:
            self.append_money(money)

    def append_money(self, a_money):
        """
        Appends a Money to this MoneyBag.
        """
        if a_money.is_zero():
            return
        old = self.find_money(a_money.currency())
        if old is None:
            self._monies.append(a_money)
        else:
            self._monies.remove(old)
            sum_money = old.add(a_money)
            if not sum_money.is_zero():
                self._monies.append(sum_money)
```

```
    def __eq__(self, other):
        """
        Checks equality between this MoneyBag and another object.
        """
        if self.is_zero() and isinstance(other, IMoney):
            return other.is_zero()

        if isinstance(other, MoneyBag):
            if len(other._monies) != len(self._monies):
                return False
            for money in self._monies:
                if not other.contains(money):
                    return False
            return True
        return False

    def find_money(self, currency):
        """
        Finds a Money in this MoneyBag by its currency.
        """
        for money in self._monies:
            if money.currency() == currency:
                return money
        return None

    def contains(self, m):
        """
        Checks if this MoneyBag contains a specific Money.
        """
        found = self.find_money(m.currency())
        return found is not None and found.amount() == m.amount()

    def __hash__(self):
        """
        Returns the hash code for this MoneyBag.
        """
        return sum(hash(money) for money in self._monies)

    def is_zero(self):
        """
        Checks if this MoneyBag is empty or contains only zero values.
        """
        return len(self._monies) == 0

    def multiply(self, factor):
        """
        Multiplies all monies in this MoneyBag by a given factor.
        """
        result = MoneyBag()
        if factor != 0:
            for money in self._monies:
                result.append_money(money.multiply(factor))
        return result

    def negate(self):
        """
        Negates all monies in this MoneyBag.
        """
        result = MoneyBag()
        for money in self._monies:
            result.append_money(money.negate())
        return result

    def simplify(self):
        """
        Simplifies this MoneyBag. If it contains only one Money, return it.
        """
        if len(self._monies) == 1:
            return self._monies[0]
        return self

    def subtract(self, m):
        """
        Subtracts a money from this MoneyBag.
        """
        return self.add(m.negate())
```



```
    def __str__(self):
        """
        Returns a string representation of this MoneyBag.
        """
        return "{" + ", ".join(str(money) for money in self._monies) + "}"

    def append_to(self, m):
        """
        Appends this MoneyBag to another MoneyBag.
        """
        m.append_bag(self)
```

test_Money.py

test > test_Money.py > ...

```python
import unittest
import sys
import os

# Add the parent directory to the Python path
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from main.Money import Money
from main.MoneyBag import MoneyBag


class TestMoney(unittest.TestCase):
    def test_add(self):
        ten_euro = Money(10, "EURO")
        self.assertEqual(10, ten_euro.amount())
        self.assertEqual("EURO", ten_euro.currency())

    def test_negative_amount(self):
        negative_money = Money(-5, "USD")
        self.assertEqual(-5, negative_money.amount())

    def test_zero_amount(self):
        zero_money = Money(0, "USD")
        self.assertEqual(0, zero_money.amount())
        self.assertTrue(zero_money.is_zero())

    def test_different_currencies_not_equal(self):
        usd = Money(10, "USD")
        eur = Money(10, "EUR")
        self.assertNotEqual(usd, eur)

    def test_hash_code_consistency(self):
        money1 = Money(10, "USD")
        money2 = Money(10, "USD")
        self.assertEqual(hash(money1), hash(money2))

    def test_equals_with_imoney(self):
        zero_money = Money(0, "USD")
        mock_imoney = Money(0, "USD")
        self.assertEqual(zero_money, mock_imoney)

    def test_equals_with_non_imoney_object(self):
        zero_money = Money(0, "USD")
        non_imoney_object = object()
        self.assertNotEqual(zero_money, non_imoney_object)

    def test_equals_with_imoney_when_zero(self):
        zero_money = Money(0, "USD")
        another_zero_money = Money(0, "USD")
        self.assertEqual(zero_money, another_zero_money)

    def test_add_money(self):
        ten_euro = Money(10, "EURO")
        five_euro = Money(5, "EURO")
        result = ten_euro.add_money(five_euro)
        self.assertEqual(Money(15, "EURO"), result)

        ten_usd = Money(10, "USD")
        result = ten_euro.add_money(ten_usd)
        expected = MoneyBag.create(ten_euro, ten_usd)
        self.assertEqual(expected, result)

    def test_add_money_bag(self):
        ten_euro = Money(10, "EURO")
        bag = MoneyBag.create(Money(5, "USD"), Money(15, "EURO"))
        result = ten_euro.add_money_bag(bag)
        expected = MoneyBag.create(Money(25, "EURO"), Money(5, "USD"))
        self.assertEqual(expected, result)

    def test_amount(self):
        ten_euro = Money(10, "EURO")
        self.assertEqual(10, ten_euro.amount())

    def test_currency(self):
        ten_euro = Money(10, "EURO")
        self.assertEqual("EURO", ten_euro.currency())

    def test_equals(self):
        ten_euro = Money(10, "EURO")
        self.assertEqual(ten_euro, ten_euro)

```

```
81          (variable) another_ten_euro: Money
82          another_ten_euro = Money(10, "EURO")
83          self.assertEqual(ten_euro, another_ten_euro)
84
85          five_euro = Money(5, "EURO")
86          self.assertNotEqual(ten_euro, five_euro)
87
88          ten_usd = Money(10, "USD")
89          self.assertNotEqual(ten_euro, ten_usd)
90
91          self.assertNotEqual(ten_euro, None)
92          self.assertNotEqual(ten_euro, "string")
93
94      def test_hash_code(self):
95          ten_euro = Money(10, "EURO")
96          another_ten_euro = Money(10, "EURO")
97          self.assertEqual(hash(ten_euro), hash(another_ten_euro))
98
99      def test_is_zero(self):
100         zero_euro = Money(0, "EURO")
101         self.assertTrue(zero_euro.is_zero())
102
103         ten_euro = Money(10, "EURO")
104         self.assertFalse(ten_euro.is_zero())
105
106     def test_multiply(self):
107         ten_euro = Money(10, "EURO")
108         result = ten_euro.multiply(2)
109         self.assertEqual(Money(20, "EURO"), result)
110
111         result = ten_euro.multiply(0)
112         self.assertEqual(Money(0, "EURO"), result)
113
114         result = ten_euro.multiply(-2)
115         self.assertEqual(Money(-20, "EURO"), result)
116
117     def test_negate(self):
118         ten_euro = Money(10, "EURO")
119         result = ten_euro.negate()
120         self.assertEqual(Money(-10, "EURO"), result)
121
122         zero_euro = Money(0, "EURO")
123         result = zero_euro.negate()
124         self.assertEqual(Money(0, "EURO"), result)
125
126         negative_euro = Money(-10, "EURO")
127         result = negative_euro.negate()
128         self.assertEqual(Money(10, "EURO"), result)
129
130     def test_subtract(self):
131         ten_euro = Money(10, "EURO")
132         five_euro = Money(5, "EURO")
133         result = ten_euro.subtract(five_euro)
134         self.assertEqual(Money(5, "EURO"), result)
135
136         zero_euro = Money(0, "EURO")
137         result = ten_euro.subtract(zero_euro)
138         self.assertEqual(Money(10, "EURO"), result)
139
140         negative_euro = Money(-5, "EURO")
141         result = ten_euro.subtract(negative_euro)
142         self.assertEqual(Money(15, "EURO"), result)
143
144     def test_to_string(self):
145         ten_euro = Money(10, "EURO")
146         self.assertEqual("[10 EURO]", str(ten_euro))
147
148
149 if __name__ == "__main__":
150     unittest.main()
```

test_MoneyBag.py

```python
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

import unittest
from main.Money import Money
from main.MoneyBag import MoneyBag
from main.IMoney import IMoney


class MoneyBagTest(unittest.TestCase):
    def setUp(self):
        """
        Set up test fixtures.
        """
        self.f12CHF = Money(12, "CHF")
        self.f14CHF = Money(14, "CHF")
        self.f7USD = Money(7, "USD")
        self.f21USD = Money(21, "USD")

        self.fMB1 = MoneyBag.create(self.f12CHF, self.f7USD)
        self.fMB2 = MoneyBag.create(self.f14CHF, self.f21USD)

    def tearDown(self):
        """
        Tear down test fixtures.
        """
        str(self.fMB1)  # Ensures toString is called for testing purposes.

    def test_bag_multiply(self):
        """
        Test multiplying a MoneyBag.
        """
        expected = MoneyBag.create(Money(24, "CHF"), Money(14, "USD"))
        self.assertEqual(expected, self.fMB1.multiply(2))
        self.assertEqual(self.fMB1, self.fMB1.multiply(1))
        self.assertTrue(self.fMB1.multiply(0).is_zero())

    def test_bag_negate(self):
        """
        Test negating a MoneyBag.
        """
        expected = MoneyBag.create(Money(-12, "CHF"), Money(-7, "USD"))
        self.assertEqual(expected, self.fMB1.negate())

    def test_bag_simple_add(self):
        """
        Test adding a Money to a MoneyBag.
        """
        expected = MoneyBag.create(Money(26, "CHF"), Money(7, "USD"))
        self.assertEqual(expected, self.fMB1.add(self.f14CHF))

    def test_bag_subtract(self):
        """
        Test subtracting a MoneyBag from another MoneyBag.
        """
        expected = MoneyBag.create(Money(-2, "CHF"), Money(-14, "USD"))
        self.assertEqual(expected, self.fMB1.subtract(self.fMB2))

    def test_bag_sum_add(self):
        """
        Test adding two MoneyBags.
        """
        expected = MoneyBag.create(Money(26, "CHF"), Money(28, "USD"))
        self.assertEqual(expected, self.fMB1.add(self.fMB2))

    def test_is_zero(self):
        """
        Test if a MoneyBag is zero.
        """
        self.assertTrue(self.fMB1.subtract(self.fMB1).is_zero())
        self.assertTrue(MoneyBag.create(Money(0, "CHF"), Money(0, "USD")).is_zero())

    def test_mixed_simple_add(self):
        """
        Test adding two different currencies.
        """
        expected = MoneyBag.create(self.f12CHF, self.f7USD)
        self.assertEqual(expected, self.f12CHF.add(self.f7USD))
```

```python
    def test_bag_not_equals(self):
        """
        Test inequality of MoneyBags.
        """
        bag = MoneyBag.create(self.f12CHF, self.f7USD)
        self.assertNotEqual(bag, Money(12, "DEM").add(self.f7USD))

    def test_money_bag_equals(self):
        """
        Test equality of MoneyBags.
        """
        self.assertNotEqual(self.fMB1, None)
        self.assertEqual(self.fMB1, self.fMB1)
        equal = MoneyBag.create(Money(12, "CHF"), Money(7, "USD"))
        self.assertEqual(self.fMB1, equal)
        self.assertNotEqual(self.fMB1, self.f12CHF)
        self.assertNotEqual(self.f12CHF, self.fMB1)
        self.assertNotEqual(self.fMB1, self.fMB2)

    def test_money_bag_hash(self):
        """
        Test hash codes of MoneyBags.
        """
        equal = MoneyBag.create(Money(12, "CHF"), Money(7, "USD"))
        self.assertEqual(hash(self.fMB1), hash(equal))

    def test_money_equals(self):
        """
        Test equality of Money objects.
        """
        self.assertNotEqual(self.f12CHF, None)
        equal_money = Money(12, "CHF")
        self.assertEqual(self.f12CHF, self.f12CHF)
        self.assertEqual(self.f12CHF, equal_money)
        self.assertEqual(hash(self.f12CHF), hash(equal_money))
        self.assertNotEqual(self.f12CHF, self.f14CHF)

    def test_money_hash(self):
        """
        Test hash codes of Money objects.
        """
        self.assertNotEqual(self.f12CHF, None)
        equal = Money(12, "CHF")
        self.assertEqual(hash(self.f12CHF), hash(equal))

    def test_simplify(self):
        """
        Test simplifying a MoneyBag.
        """
        money = MoneyBag.create(Money(26, "CHF"), Money(28, "CHF"))
        self.assertEqual(Money(54, "CHF"), money)

    def test_normalize2(self):
        """
        Test normalization of a MoneyBag.
        """
        expected = Money(7, "USD")
        self.assertEqual(expected, self.fMB1.subtract(self.f12CHF))

    def test_normalize3(self):
        """
        Test normalization of a MoneyBag with multiple currencies.
        """
        ms1 = MoneyBag.create(Money(12, "CHF"), Money(3, "USD"))
        expected = Money(4, "USD")
        self.assertEqual(expected, self.fMB1.subtract(ms1))

    def test_normalize4(self):
        """
        Test normalization of a MoneyBag with subtraction.
        """
        ms1 = MoneyBag.create(Money(12, "CHF"), Money(3, "USD"))
        expected = Money(-3, "USD")
        self.assertEqual(expected, self.f12CHF.subtract(ms1))
```

```python
    def test_print(self):
        """
        Test string representation of Money.
        """
        self.assertEqual("[12 CHF]", str(self.f12CHF))

    def test_simple_add(self):
        """
        Test simple addition of Money objects.
        """
        expected = Money(26, "CHF")
        self.assertEqual(expected, self.f12CHF.add(self.f14CHF))

    def test_simple_bag_add(self):
        """
        Test adding a Money to a MoneyBag.
        """
        expected = MoneyBag.create(Money(26, "CHF"), Money(7, "USD"))
        self.assertEqual(expected, self.f14CHF.add(self.fMB1))

    def test_simple_multiply(self):
        """
        Test multiplying a Money object.
        """
        expected = Money(28, "CHF")
        self.assertEqual(expected, self.f14CHF.multiply(2))

    def test_simple_negate(self):
        """
        Test negating a Money object.
        """
        expected = Money(-14, "CHF")
        self.assertEqual(expected, self.f14CHF.negate())

    def test_simple_subtract(self):
        """
        Test subtracting Money objects.
        """
        expected = Money(2, "CHF")
        self.assertEqual(expected, self.f14CHF.subtract(self.f12CHF))

    def test_money_bag_equals_edge_cases(self):
        empty_bag = MoneyBag()

        # Case 1: Comparing an empty MoneyBag to itself
        self.assertTrue(empty_bag == empty_bag)

        # Case 2: Comparing an empty MoneyBag to another empty MoneyBag
        another_empty_bag = MoneyBag()
        self.assertTrue(empty_bag == another_empty_bag)

        # Case 3: Comparing an empty MoneyBag to a non-empty MoneyBag
        self.assertFalse(empty_bag == self.fMB1)

        # Case 4: Comparing MoneyBag with a Money object (should return False)
        self.assertFalse(self.fMB1 == self.f12CHF)

        # Case 5: Comparing MoneyBag to a None object
        self.assertFalse(self.fMB1 == None)

        # Case 6: Comparing MoneyBag with different sizes
        small_bag = MoneyBag.create(Money(5, "USD"), Money(10, "CHF"))
        large_bag = small_bag.add(Money(20, "EUR"))
        self.assertFalse(small_bag == large_bag)
```

```
220
221     def test_money_bag_equals_edge_cases_2(self):
222         empty_bag = MoneyBag()
223         non_imoney_object = object()  # Completely unrelated object
224
225         # Case 1: Comparing an empty MoneyBag to itself
226         self.assertTrue(empty_bag == empty_bag)
227
228         # Case 2: Comparing an empty MoneyBag to a non-IMoney object
229         self.assertFalse(empty_bag == non_imoney_object)
230
231         # Case 3: Comparing an empty MoneyBag to a non-empty MoneyBag
232         self.assertFalse(empty_bag == self.fMB1)
233
234         # Case 4: Comparing MoneyBag with a Money object (should return False)
235         self.assertFalse(self.fMB1 == self.f12CHF)
236
237         # Case 5: Comparing MoneyBag to a None object
238         self.assertFalse(self.fMB1 == None)
239
240         # Case 6: Comparing MoneyBag with different sizes
241         small_bag = MoneyBag.create(Money(5, "USD"), Money(10, "CHF"))
242         large_bag = small_bag.add(Money(20, "EUR"))
243         self.assertFalse(small_bag == large_bag)
244
245     def test_equals_with_non_imoney_object(self):
246         money_bag = MoneyBag.create(Money(10, "USD"), Money(5, "EUR"))
247         non_imoney_object = object()  # Completely unrelated object
248
249         self.assertFalse(
250             money_bag == non_imoney_object,
251             "MoneyBag should not be equal to a non-IMoney object",
252         )
253
254
255 if __name__ == "__main__":
256     unittest.main()
```