Classes - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

Any class definition must include at least one method definition.

: False

A class cannot have multiple methods with the same name.

: False

Class attributes are shared across all instances of that class.

: True

Classes are objects.

: True

Each class can only be instantiated once.

: False

Every class you write must define `__init__`.

: False

Two objects of the same class with the same attributes are equal by default.

: False

`self` refers to an instance of a class.

: True

If a class implements multiple methods with the exact same signature, a call will execute the first implementation found.

: False

Certain classes cannot be instantiated.

: True

A class without any substantial functionality is commonly referred to as a 'data class'.

: True

A static method makes no references to `self`

: True

Exceptions - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

The stack uses a FIFO (First-In, First-Out) data structure.

: False

A `try` block must always be followed by an `except` block.

: False

A function that raises an exception returns `None`.

: False

Programs can be paused using a debugger.

: True

The stack uses a LIFO (Last-In, First-Out) data structure.

: True

It is possible to catch more than one type of Exception in a single `except` block.

: True

It is always better to return an error message than to raise an Exception.

: False

A debugger can automatically fix bugs.

: False

Each new function call creates a call frame on the stack.

: True

It is recommended to put as much code as possible into a single `try` block to catch as many problems as possible.

: False

A function that raises an exception does not have a return value.

: True

It is possible to create user-defined exceptions.

: True

Git - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

Git can be used to roll back changes to files in a directory.

: True

`git add` is used to add additional developers to a project.

: False

Git allows you to switch back and forth between different versions of your code.

: True

Git branches must always be merged manually.

: False

Git keeps track of changes in a file, but not of who made those changes.

: False

`git init` is used to start tracking the current working directory using Git.

: True

Every Git project has two branches.

: False

Some merges require manual intervention.

: True

Executing certain Git commands may make files in a folder appear or disappear.

: True

`git log` is used to write new log messages.

: False

Multiple files can be added to a Git project at once.

: True

`git pull` is used to remove files from a Git project.

: False

Inheritance - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

If a class `A` inherits from a class `B`, which inherits from a class `C`, then we can say that A *is a* C.

: True

An abstract class can inherit from another abstract class.

: True

A class can directly inherit from multiple classes.

: True

When overriding a method in a subclass, the overridden implementation in the superclass becomes entirely inaccessible for instances of the subclass.

: False

If a class `A` inherits from a class `B`, which inherits from a class `C`, then we can say that C *is an* A.

: False

When overriding a constructor, it is always necessary to call the super-constructor.

: False

A class can only inherit from exactly one other class.

: False

Inheritance is a way of supporting Polymorphism.

: True

When a class inherits from another class, these two classes form a type hierarchy.

: True

All classes inherit from ABC.

: False

When inheriting from multiple classes, `super()` will refer to all super classes at the same time.

: False

If a subclass `C` of an abstract class `P` does not implement all abstract methods, then `C` is also an abstract class.

: True

Testing - 2 points, 1 task

Decide for each of the following statements whether it is true or false. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

Ideally, a single unit test should test as much functionality as possible.

: False

Regression testing is a good way to prevent a previous bug from happening again.

: True

To check if two floats are equal, `assertEqual` should be used.

: False

If you use a debugger, you don't need to write unit tests.

: False

Ideally, a single unit test should test exactly one thing in isolation.

: True

The four phases of a unit test are Setup, Exercise, Teardown and Terminate.

: False

In white-box testing, the test programmer has access to the source code of the program to be tested.

: True

To test a class with 5 methods, 5 unit tests are certainly enough.

: False

Three out of the four phases of unit testing are optional.

: False

Unit tests encourage refactorings.

: True

Black-box testing tries to maximize test coverage of source code.

: False

Test-driven development refers to the practice of writing tests that automatically generate source code.

: False

Warmup - 7 points, 1 out of 3 tasks

Implement a function `to_list`, which takes an arbitrary object `thing` and an *optional* function `transform` as arguments.

The function `to_list` should attempt to call Python's built-in `list` function with `thing` as the argument and return the result. If this fails because of any exception, `to_list` should instead return the value resulting from calling `transform` with `thing` as the parameter. By default, `transform` should simply return a list containing the value passed to it as the only element.

You may assume that `to_list` will only be constructed with parameters that match the description. Make sure you correctly formulate the function signature.

Use the following template:

```
def to_list(...
    pass
```

The following example illustrates how the solution may be used:

```
print( to_list([1,2,3]) )
print( to_list((1,2,3)) )
print( to_list(1.5) )
print( to_list(True) )
print( to_list(True, lambda x: [int(x)]) )
```

The code above should produce the following output:

```
[1, 2, 3]
[1, 2, 3]
[1.5]
[True]
[1]
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env
```

```python
def to_list(thing, transform=lambda x: [x]):
    try:
        return list(thing)
    except:
        return transform(thing)

# examples
print( to_list([1,2,3]) )
print( to_list((1,2,3)) )
print( to_list(1.5) )
print( to_list(True) )
print( to_list(True, lambda x: [int(x)]) )
```

Implement a function `is_perfect_pangram`, which takes a string `sentence` and an *optional* string `alphabet` as arguments.

A "pangram" is a word or sentence that uses all letters in an alphabet at least once. A "perfect pangram" uses each letter *exactly* once.

The function `is_perfect_pangram` should check if `sentence` is a perfect pangram for the given `alphabet`. If no `alphabet` is given, the 26 letters of the English alphabet should be assumed. Character casing and any characters that are not part of the alphabet should be ignored.

You may assume that `is_perfect_pangram` will only be constructed with parameters that match the description. Make sure you correctly formulate the function signature.

Use the following template:

```python
def is_perfect_pangram(...
    pass
```

The following example illustrates how the solution may be used:

```python
print( is_perfect_pangram("Mr Jock, TV quiz PhD, bags few lynx") )
print( is_perfect_pangram("a b c", "abc") )
print( is_perfect_pangram("azbzc", "abc") )
print( is_perfect_pangram("abc", "abcd") )
print( is_perfect_pangram("abb", "abc") )
```

The code above should produce the following output:

```
True
True
True
False
False
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env

import string

def is_perfect_pangram(sentence, alphabet=string.ascii_lowercase):
    sentence = sentence.lower()
    letters = []
    for c in sentence:
        if c in alphabet:
            letters.append(c)
    return len(letters) == len(set(letters)) == len(alphabet)

# examples
print( is_perfect_pangram("Mr Jock, TV quiz PhD, bags few lynx") )
print( is_perfect_pangram("a b c", "abc") )
print( is_perfect_pangram("azbzc", "abc") )
print( is_perfect_pangram("abc", "abcd") )
print( is_perfect_pangram("abb", "abc") )
```

Implement a function `padded_dict`, which takes a list of distinct, hashable values `keys` and a list of arbitrary values `values`. It also takes an *optional* arbitrary value `padding`, which shall be `None` by default.

The function `padded_dict` should create and return a new dictionary, where the keys are taken from `keys` and the values are taken from `values` in corresponding order. If `values` is shorter than `keys`, the values for the remaining keys should be set to be `padding`. If `keys` is shorter than `values`, the superfluous values of `values` can be ignored.

You may assume that `padded_dict` will only be constructed with parameters that match the description. Make sure you correctly formulate the function signature.

Use the following template:

```
def padded_dict(...
    pass
```

The following example illustrates how the solution may be used:

```
print( padded_dict([1, "b", 3], [55, 66, 77] ) )
print( padded_dict([1, "b", 3], [55] ) )
print( padded_dict([1, "b"], [55, 66, 77] ) )
```

The code above should produce the following output:

```
{1: 55, 'b': 66, 3: 77}
{1: 55, 'b': None, 3: None}
{1: 55, 'b': 66}
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

def padded_dict(keys, values, padding=None):
    d = {}
    for i in range(len(keys)):
        try:
            val = values[i]
        except IndexError:
            val = padding
        d[keys[i]] = val
    return d

# examples
print( padded_dict([1, "b", 3], [55, 66, 77] ) )
print( padded_dict([1, "b", 3], [55] ) )
print( padded_dict([1, "b"], [55, 66, 77] ) )
```

Equality - 7 points, 1 out of 3 tasks

Implement a class `Hand`, which takes one parameter in its constructor:

- `cards`, a list of distinct integers representing card values of a simple card game

This value should be stored as instance attribute.

The `Hand` class should implement the equality operator as follows. Two instances of `Hand` are equal if the sum of card values is the same for both instances. However, in this game, the card with value `1` counts as `11` for the equality operation. Considering the first example below, the sum of cards `1` and `2` is `13` because the `1` counts as `11`. Do not forget to correctly handle cases where the value being compared to is not an instance of Currency.

You may assume that `Hand` will only be constructed with parameters that match the description.

Use the following template:

```
class Hand:
    pass
```

The following example illustrates how the solution may be used:

```
print( Hand([1, 2]) == Hand([10, 3]) )
print( Hand([2, 5, 7]) == Hand([10, 4]) )
print( Hand([2, 5, 7]) != Hand([9]) )
print( Hand([2, 5, 7]) == "Flush" )
```

The code above should produce the following output:

```
True
True
True
False
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

class Hand:
    def __init__(self, cards):
        self.__cards = cards

    def __eq__(self, other):
        if not isinstance(other, Hand):
            return NotImplemented
        this = sum(self.__cards)
        that = sum(other.__cards)
        if 1 in self.__cards:
            this += 10
        if 1 in other.__cards:
            that += 10
        return this == that

# examples
print( Hand([1, 2]) == Hand([10, 3]) )
print( Hand([2, 5, 7]) == Hand([10, 4]) )
print( Hand([2, 5, 7]) != Hand([9]) )
print( Hand([2, 5, 7]) == "Flush" )
```

Implement a class `Currency`, which takes three parameters in its constructor:

- `name`, then name of the currency
- `amount`, the amount of currency represented
- `rate`, a conversion rate representing the value of one unit of this currency relative to a gold standard

These three values should be stored as instance attributes.

The `Currency` class should implement the equality operator as follows. Two instances of Currency are equal if multiplying the amount with the rate results in the same value for both instances. Considering the first example below, 15 EUR are equal to 10 GBP because `15 * 0.5` equals `10 * 0.75`. Do not forget to correctly handle cases where the value being compared to is not an instance of Currency.

You may assume that `Currency` will only be constructed with parameters that match the description.

Use the following template:

```
class Currency:
    pass
```

The following example illustrates how the solution may be used:

```
print( Currency("EUR", 15, 0.5) == Currency("GBP", 10, 0.75) )
print( Currency("EUR", 15, 0.5) != Currency("GBP", 15, 0.75) )
print( Currency("GBP", 1, 0.75) == Currency("GBP", 10, 0.75) )
print( Currency("CHF", 1.50, 0.78) == "Enough to buy a coffee" )
```

The code above should produce the following output:

```
True
True
False
False
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

class Currency:
    def __init__(self, name, amount, rate):
        self.__name = name
        self.__amount = amount
        self.__rate = rate

    def __eq__(self, other):
        if not isinstance(other, Currency):
            return NotImplemented
        this = self.__amount * self.__rate
        that = other.__amount * other.__rate
        return this == that

# examples
print( Currency("EUR", 15, 0.5) == Currency("GBP", 10, 0.75) )
print( Currency("EUR", 15, 0.5) != Currency("GBP", 15, 0.75) )
print( Currency("GBP", 1, 0.75) == Currency("GBP", 10, 0.75) )
print( Currency("CHF", 1.50, 0.78) == "Enough to buy a coffee" )
```

Implement a class `Order`, which takes two parameters in its constructor:

- `items`, a list of strings
- `cost`, a number

These two values should be stored as instance attributes.

The `Order` class should implement the equality operator as follows. Two instances of `Order` are equal if and only if the list of items contains the same strings and if the cost is the same. Mind that the same item may appear multiple times in an order. Do not forget to correctly handle cases where the value being compared to is not an instance of `Order`.

You may assume that `Order` will only be constructed with parameters that match the description.

Use the following template:

```
class Order:
    pass
```

The following example illustrates how the solution may be used:

```
print( Order(["Beer", "Wine", "Beer"], 14.50) == Order(["Wine", "Beer", "Beer"], 14.50) )
print( Order(["Beer", "Wine", "Beer"], 14.50) != Order(["Wine", "Beer"], 14.50) )
print( Order(["Beer", "Pretzels"], 14.50) == "Healthy meal" )
```

The code above should produce the following output:

```
True
True
False
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

class Order:
    def __init__(self, items, cost):
        self.__items = items
        self.__cost = cost

    def __eq__(self, other):
        if not isinstance(other, Order):
            return NotImplemented
        return sorted(self.__items) == sorted(other.__items) and self.__cost == other.__cost

# examples
print( Order(["Beer", "Wine", "Beer"], 14.50) == Order(["Wine", "Beer", "Beer"], 14.50) )
print( Order(["Beer", "Wine", "Beer"], 14.50) != Order(["Wine", "Beer"], 14.50) )
print( Order(["Beer", "Pretzels"], 14.50) == "Healthy meal" )
```

State - 11 points, 1 out of 2 tasks

Implement a class `Market` that keeps track of how much vendors are paying for having a stall at a market during a day.

A market rents out stalls on an hourly basis. The market and the vendors agree on an hourly fee when the vendor joins the market. The vendor then pays the fee for each hour they stay on the market until they leave.

The `Market` class should support adding new vendors with a given hourly fee. It should have a method `progress_hour` that runs the clock one hour forward (the clock starts at zero). It should keep track internally of how many vendors there are and what fees they are paying. A method `stats` should return a dictionary that lists:

- the number of vendors
- the current hour

- the hourly profit that can be made given the currently present vendors and
- the total profit made so far

Important: The examples contain essential information, such as how the methods should be called and used and what the stats dictionary should look like. Make sure your implementation works with the provided examples.

Use the following template:

```
class Market:
    pass
```

The following example illustrates how the solution may be used:

```
m = Market()
print(m.stats())
m.add_vendor(30)
m.add_vendor(5)
m.add_vendor(5)
print(m.stats())
m.progress_hour()
m.progress_hour()
m.progress_hour()
print(m.stats())
try:
    m.remove_vendor(11)
except:
    print("No vendor with that fee exists")
m.remove_vendor(5)
print(m.stats())
m.progress_hour()
print(m.stats())
```

The code above should produce the following output:

```
{'number of vendors': 0, 'hour': 0, 'hourly profit': 0, 'total profit': 0}
{'number of vendors': 3, 'hour': 0, 'hourly profit': 40, 'total profit': 0}
{'number of vendors': 3, 'hour': 3, 'hourly profit': 40, 'total profit': 120}
No vendor with that fee exists
{'number of vendors': 2, 'hour': 3, 'hourly profit': 35, 'total profit': 120}
{'number of vendors': 2, 'hour': 4, 'hourly profit': 35, 'total profit': 155}
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

import random

class Market:
    def __init__(self):
        self.__vendors = []
        self.__hour = 0
        self.__profits = 0

    def stats(self):
        return {
            "number of vendors": len(self.__vendors),
            "hour": self.__hour,
            "hourly profit": sum(self.__vendors),
            "total profit": self.__profits
        }

    def add_vendor(self, fee):
        self.__vendors.append(fee)

    def remove_vendor(self, fee):
        if fee not in self.__vendors:
            raise Warning
        self.__vendors.remove(fee)

    def progress_hour(self):
        self.__profits += sum(self.__vendors)
        self.__hour += 1

# examples
m = Market()
print(m.stats())
m.add_vendor(30)
m.add_vendor(5)
m.add_vendor(5)
print(m.stats())
m.progress_hour()
m.progress_hour()
m.progress_hour()
```

```
print(m.stats())
try:
    m.remove_vendor(11)
except:
    print("No vendor with that fee exists")
m.remove_vendor(5)
print(m.stats())
m.progress_hour()
print(m.stats())
```

Implement a class `Coinflips` which can be used to play multiple rounds of flipping a coin and guessing which side it will land on. The player wins if an *absolute* majority of his guesses were correct.

`Coinflips` should implement a method `play` which takes the player's guess as a string parameter, either "heads" or "tails". A `Warning` should be raised if some other string is passed. `play` then randomly selects the result of the coin flip for the current round of the game. `play` should return a string `"heads"` or `"tails"` depending on the random result. A method `winner` can be called at any time to determine, whether the player's guesses were correct a majority of the time. `winner` should return `True` if the player wins, `False` otherwise. The player can continue playing for more rounds even after calling `winner`. Finally, the string representation of a `Coinflips` instance should be the history of "heads" and "tails" that occurred in the game.

Important: The examples contain essential information, such as how the methods should be called and used and what the string representation of `Coinflips` should look like. Make sure your implementation works with the provided examples.

Use the following template:

```
class Coinflips:
    pass
```

The following example illustrates how the solution may be used:

```
t = Coinflips()
try:
    t.play("arms")
except Warning:
    print("invalid choice")
# Your play results may be different from this example due to randomness
print(t.play("heads"))
print(t.play("tails"))
print(t.play("heads"))
print(t)
print(t.winner())
```

The code above should produce the following output:

```
invalid choice
heads
heads
tails
heads, heads, tails
False
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```
#!/usr/bin/env

import random

class Coinflips:
    def __init__(self):
        self.__results = []
        self.__choices = []

    def play(self, choice):
        if choice not in ["heads", "tails"]:
            raise Warning
        result = random.choice(["heads", "tails"])
        self.__results.append(result)
        self.__choices.append(choice)
        return result

    def winner(self):
        wins = sum([1 for h, c in zip(self.__results, self.__choices) if h == c])
        losses = len(self.__results) - wins
        return wins > losses
```

```python
    def __str__(self):
        return ", ".join(self.__results)

# examples
t = Coinflips()
try:
    t.play("arms")
except Warning:
    print("invalid choice")
# Your play results may be different from this example due to randomness
print(t.play("heads"))
print(t.play("tails"))
print(t.play("heads"))
print(t)
print(t.winner())
```

Testing - 8 points, 1 out of 4 tasks

You are given a correct implementation of a function `min_max_mean`, which takes a list of numbers as a parameter `values` and returns a tuple with 3 values representing the minimum, maximum and average value of `values`.

Write a test suite that can identify faulty implementations exhibiting the following problems:

- The minimum value is incorrect
- The maximum value is incorrect
- The mean value is incorrect

Implement **exactly 3** tests. For an implementation exhibiting exactly one of the isolated faults listed above, exactly **one** test must **fail** while the other **two** must **pass**.

You may want to introduce bugs into `min_max_mean` to check whether your test suite works correctly.

Submit your entire Test class as given in the template and do not change its name. Do not submit the `min_max_mean` function.

Use the following template:

```python
def min_max_mean(values):
    from statistics import mean
    return min(values), max(values), mean(values)

from unittest import TestCase

class TestSuite(TestCase):
    pass # implement here
```

No examples are provided for this task

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env
# points: 2

def min_max_mean(values):
    from statistics import mean
    return min(values), max(values), mean(values)

from unittest import TestCase

class TestSuite(TestCase):
    def test_min(self):
        actual = min_max_mean([1,2,3])
        expected = 1
        self.assertEqual(expected, actual[0])

    def test_max(self):
        actual = min_max_mean([1,2,3])
        expected = 3
        self.assertEqual(expected, actual[1])

    def test_mean(self):
        actual = min_max_mean([1,2,3])
        expected = 2
        self.assertEqual(expected, actual[2])
```

You are given a correct implementation of a function `count_booleans`, which takes a list of booleans as a parameter `booleans` and returns a tuple with 3 values: the number of `True`s, the number of `False`s and the total number of values in `booleans`.

Write a test suite that can identify faulty implementations exhibiting the following problems:

- The number of `True`s is incorrect
- The number of `False`s is incorrect
- The total number of elements is incorrect

Implement **exactly 3** tests. For an implementation exhibiting exactly one of the isolated faults listed above, exactly **one** test must **fail** while the other **two** must **pass**.

You may want to introduce bugs into `count_booleans` to check whether your test suite works correctly.

Submit your entire Test class as given in the template and do not change its name. Do not submit the `count_booleans` function.

Use the following template:

```python
def count_booleans(booleans):
    return booleans.count(True), booleans.count(False), len(booleans)

from unittest import TestCase

class TestSuite(TestCase):
    pass # implement here
```

No examples are provided for this task

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env
# points: 2

def count_booleans(booleans):
    return booleans.count(True), booleans.count(False), len(booleans)

from unittest import TestCase

class TestSuite(TestCase):
    def test_true(self):
        actual = count_booleans([True, False, True, False, False])
        expected = 2
        self.assertEqual(expected, actual[0])

    def test_false(self):
        actual = count_booleans([True, False, True, False, False])
        expected = 3
        self.assertEqual(expected, actual[1])

    def test_total(self):
        actual = count_booleans([True, False, True, False, False])
        expected = 5
        self.assertEqual(expected, actual[2])
```

You are given a correct implementation of a function `unique_sorted`, which takes a list as a parameter `values` and returns a list containing all unique elements from `values` in descending order.

Write a test suite that can identify faulty implementations exhibiting the following problems:

- An implementation that doesn't work correctly when given an empty list, but works correctly otherwise
- An implementation that sorts the result, but does not remove duplicate values
- An implementation that does remove duplicates, but does not sort the result

Implement **exactly 3** tests. For an implementation exhibiting exactly one of the isolated faults listed above, exactly **one** test must **fail** while the other **two** must **pass**.

You may want to introduce bugs into `unique_sorted` to check whether your test suite works correctly.

Submit your entire Test class as given in the template and do not change its name. Do not submit the `unique_sorted` function.

Use the following template:

```python
def unique_sorted(values):
    return list(sorted(set(values), reverse=True))

from unittest import TestCase

class TestSuite(TestCase):
    pass # implement here
```

No examples are provided for this task

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env
# points: 2

def unique_sorted(values):
    return list(sorted(set(values), reverse=True))

from unittest import TestCase

class TestSuite(TestCase):
    def test_empty(self):
        actual = unique_sorted([])
        expected = []
        self.assertEqual(expected, actual)

    def test_result_is_sorted(self):
        actual = unique_sorted([3,2,2,1])
        self.assertEqual(sorted(actual, reverse=True),actual)

    def test_results_are_unique(self):
        actual = unique_sorted([1,1,2,3,3])
        self.assertTrue(len(actual) == 3)
```

You are given a correct implementation of a function `upper_lower_capitalize`, which takes a string as a parameter `string` and returns a tuple with 3 values: `string` in upper case, `string` in lower case, and `string` with only the first letter in upper case.

Write a test suite that can identify faulty implementations exhibiting the following problems:

- The upper case string is wrong
- The lower case string is wrong
- The capitalized string is wrong

Implement **exactly 3** tests. For an implementation exhibiting exactly one of the isolated faults listed above, exactly **one** test must **fail** while the other **two** must **pass**.

You may want to introduce bugs into `upper_lower_capitalize` to check whether your test suite works correctly.

Submit your entire Test class as given in the template and do not change its name. Do not submit the `upper_lower_capitalize` function.

Use the following template:

```python
def upper_lower_capitalize(string):
    return string.upper(), string.lower(), string.capitalize()

from unittest import TestCase

class TestSuite(TestCase):
    pass # implement here
```

No examples are provided for this task

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env
# points: 2

def upper_lower_capitalize(string):
    return string.upper(), string.lower(), string.capitalize()
```

```python
from unittest import TestCase

class TestSuite(TestCase):
    def test_upper(self):
        actual = upper_lower_capitalize("heLLo")
        expected = "HELLO"
        self.assertEqual(expected, actual[0])

    def test_lower(self):
        actual = upper_lower_capitalize("heLLo")
        expected = "hello"
        self.assertEqual(expected, actual[1])

    def test_capitalize(self):
        actual = upper_lower_capitalize("heLLo")
        expected = "Hello"
        self.assertEqual(expected, actual[2])
```

Encode - 10 points, 1 out of 2 tasks

Implement two functions `encode` and `decode` that work correctly with the following example calls. First, try to understand what these functions do. Note that `decode` implements the reverse operations from `encode` and that each function returns three values, which represent the intermediate and final states reached within the respective functions. The parameters of each function are the input string to be encoded and the two modification parameters that influence how the intermediate transformations are performed. Your solution should also work if these modification parameters take on any other kind of value of the same type as given in the examples, although the second string passed to encode or decode will never be empty.

Use the following template:

```python
# Your implementation of the necessary class(es)
class
```

The following example illustrates how the solution may be used:

```python
print(encode("hello", "xyz", -2) ==
     (['h', 'x', 'e', 'y', 'l', 'z', 'l', 'x', 'o', 'y'], [102, 118, 99, 119, 106, 120, 106, 118, 109, 119], 'fvcwjxjvmw'))
print(decode("fvcwjxjvmw", "xyz", -2) ==
     ([102, 118, 99, 119, 106, 120, 106, 118, 109, 119], ['h', 'x', 'e', 'y', 'l', 'z', 'l', 'x', 'o', 'y'], 'hello'))
```

The code above should produce the following output:

```
True
True
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env

def encode(string, salt, offset):
    # add salt
    salted = []
    salt_index = 0
    for char in string:
        salted.append(char)
        salted.append(salt[salt_index])
        salt_index += 1
        if salt_index == len(salt):
            salt_index = 0
    # convert to ints
    ints = []
    for char in salted:
        ints.append(ord(char))
    # apply offset
    shifted = []
    for i in ints:
        shifted.append(i + offset)
    # back to string
    res = ""
    for i in shifted:
        res += chr(i)
    # return steps
    return salted, shifted, res

def decode(string, salt, offset):
    # to shifted ints
    shifted = []
    for char in string:
        shifted.append(ord(char))
    # un-apply offset
    ints = []
    for i in shifted:
        ints.append(i - offset)
    # back to characters
    salted = []
    for i in ints:
        salted.append(chr(i))
    # unsalt
    res = ""
    for i, char in enumerate(salted):
        if i % 2 == 1:
            continue
        res += char
    # return steps
    return shifted, salted, res

# examples
print(encode("hello", "xyz", -2) ==
     (['h', 'x', 'e', 'y', 'l', 'z', 'l', 'x', 'o', 'y'], [102, 118, 99, 119, 106, 120, 106, 118, 109, 119], 'fvcwjxjvmw'))
print(decode("fvcwjxjvmw", "xyz", -2) ==
     ([102, 118, 99, 119, 106, 120, 106, 118, 109, 119], ['h', 'x', 'e', 'y', 'l', 'z', 'l', 'x', 'o', 'y'], 'hello'))
```

Implement two functions encode and decode that work correctly with the following example calls. First, try to understand what these functions do. Note that decode implements the reverse operations from encode and that each function returns three values, which represent the intermediate and final states reached within the respective functions. The parameters of each function are the input string to be encoded and the two modification parameters that influence how the intermediate transformations are performed. Your solution should also work if these modification parameters take on any other kind of value of the same type as given in the examples, although the third parameter will always be a non-empty list of positive numbers.

Use the following template:

```python
# Your implementation of the necessary class(es)
class
```

The following example illustrates how the solution may be used:

```python
print(encode("hello", 3, [65,99,120]) ==
     ([107, 104, 111, 111, 114], [107, 65, 104, 99, 111, 120, 111, 65, 114, 99], 'kAhcoxoArc'))
print(decode("kAhcoxoArc", 3, [65,99,120]) ==
     ([107, 65, 104, 99, 111, 120, 111, 65, 114, 99], [107, 104, 111, 111, 114], 'hello'))
```

The code above should produce the following output:

```
True
True
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env

def encode(string, offset, salt):
    # convert to ints
    ints = []
    for char in string:
        ints.append(ord(char))
    # apply offset
    shifted = []
    for i in ints:
        shifted.append(i + offset)
    # add salt
    salted = []
    salt_index = 0
    for number in shifted:
        salted.append(number)
        salted.append(salt[salt_index])
        salt_index += 1
        if salt_index == len(salt):
            salt_index = 0
    # back to string
    res = ""
    for i in salted:
        res += chr(i)
    # return steps
    return shifted, salted, res

def decode(string, offset, salt):
    # to shifted ints
    shifted = []
    for char in string:
        shifted.append(ord(char))
    # unsalt
    unsalted = []
    for i, char in enumerate(shifted):
        if i % 2 == 1:
            continue
        unsalted.append(char)
    # un-apply offset
    ints = []
    for i in unsalted:
        ints.append(int(i - offset))
    # back to characters
    res = []
    for i in ints:
        res.append(chr(i))
    # return steps
    return shifted, unsalted, "".join(res)

# examples
print(encode("hello", 3, [65,99,120]) ==
     ([107, 104, 111, 111, 114], [107, 65, 104, 99, 111, 120, 111, 65, 114, 99], 'kAhcoxoArc'))
print(decode("kAhcoxoArc", 3, [65,99,120]) ==
     ([107, 65, 104, 99, 111, 120, 111, 65, 114, 99], [107, 104, 111, 111, 114], 'hello'))
```

Reverse - 12 points, 1 out of 2 tasks

Read the example calls and provided output below. Then implement the necessary class(es) such that they will produce the same output given the example calls.

Use the following template:

```python
# Your implementation of the necessary class(es)
class
```

The following example illustrates how the solution may be used:

```python
passengers = Airplane.parse_manifest(("Montgomery,Rich,1;Tim,Merchant,2;Sally,Sale,2;Peter,Poor,3"))
a = Airplane("A388", "G-XLEK")
a.board(passengers)
print(a.get_passengers())
p = a.get_passengers(2)
print(type(p[1]))
print(p[1])
```

The code above should produce the following output:

```
[Montgomery Rich, 1st class, Tim Merchant, business class, Sally Sale, business class, Peter Poor, economy class]
<class '__main__.Passenger'>
Sally Sale, business class
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env

class Passenger:
    def __init__(self, fname, lname, ticket_class):
        self.fname = fname
        self.lname = lname
        self.ticket_class = ticket_class

    def __repr__(self):
        class_str = "1st"
        if self.ticket_class == 2: class_str = "business"
        if self.ticket_class == 3: class_str = "economy"
        return f"{self.fname} {self.lname}, {class_str} class"

class Airplane:
    def __init__(self, model, registration):
        self.model = model
        self.registration = registration
        self.classes = [[], [], []]
    @staticmethod
    def parse_manifest(manifest):
        return [Passenger(f, l, int(t)) for (f, l, t) in [m.split(",") for m in manifest.split(";")]]

    def board(self, passengers):
        for p in passengers:
            self.classes[p.ticket_class-1].append(p)

    def get_passengers(self, ticket_class=None):
        if ticket_class == None:
            res = []
            for c in self.classes:
                res.extend(c)
            return res
        return self.classes[ticket_class-1]

# examples
passengers = Airplane.parse_manifest(("Montgomery,Rich,1;Tim,Merchant,2;Sally,Sale,2;Peter,Poor,3"))
a = Airplane("A388", "G-XLEK")
a.board(passengers)
print(a.get_passengers())
p = a.get_passengers(2)
print(type(p[1]))
print(p[1])
```

Read the example calls and provided output below. Then implement the necessary class(es) such that they will produce the same output given the example calls.

Use the following template:

```python
# Your implementation of the necessary class(es)
class
```

The following example illustrates how the solution may be used:

```python
riders = Train.parse_reservations(("Montgomery,Rich,GA;Tim,Merchant,GA;Sally,Sale,D;Peter,Poor,M"))
a = Train("IC17", "Columbus")
a.board(riders)
print(a.get_riders())
p = a.get_riders("GA")
print(type(p[1]))
print(p[1])
```

The code above should produce the following output:

```
[Montgomery Rich, general ticket, Tim Merchant, general ticket, Sally Sale, day ticket, Peter Poor, monthly ticket]
<class '__main__.Rider'>
Tim Merchant, general ticket
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env

class Rider:
    def __init__(self, fname, lname, ticket_class):
        self.fname = fname
        self.lname = lname
        self.ticket_class = ticket_class

    def __repr__(self):
        class_str = "general"
        if self.ticket_class == "D": class_str = "day"
        if self.ticket_class == "M": class_str = "monthly"
        return f"{self.fname} {self.lname}, {class_str} ticket"

class Train:
    def __init__(self, model, registration):
        self.model = model
        self.registration = registration
        self.riders = []

    @staticmethod
    def parse_reservations(reservations):
        return [Rider(f, l, t) for (f, l, t) in [m.split(",") for m in reservations.split(";")]]

    def board(self, p):
        self.riders.extend(p)

    def get_riders(self, ticket_class=None):
        if ticket_class == None:
            return self.riders
        return [p for p in self.riders if p.ticket_class == ticket_class]

# examples
riders = Train.parse_reservations(("Montgomery,Rich,GA;Tim,Merchant,GA;Sally,Sale,D;Peter,Poor,M"))
a = Train("IC17", "Columbus")
a.board(riders)
print(a.get_riders())
p = a.get_riders("GA")
print(type(p[1]))
print(p[1])
```

Inheritance - 25 points, 1 out of 96 tasks

Important: read the problem description, implementation notes and provided examples carefully for a comprehensive specification. All three parts are relevant for understanding the requirements! Also, make sure you do not run out of time to submit your solution!

Problem description

You've been tasked with implementing shipping logistics for a cookie factory.

Each cookie produced is given a name and a price. It should be possible to find out the name and price of a cookie.

The factory ships cookies in different kinds of containers, namely wrappers and boxes, although maybe there will be additional kinds of containers in the future. Every container has contents, for which it should be possible to determine the total price rounded to two decimal points. Furthermore, it should be possible to determine the number of cookies within any kind of container.

A wrapper is used to hold between 3 and 5 cookies. Boxes are used to ship multiple wrappers. A box can contain at most 200 cookies, no matter how many wrappers have been used to hold them.

For quality control, each box shall have a unique retrievable ID, starting at 1 for the first box and incrementing by 1 for each additional box produced.

Additional implementation instructions:

- Fill in the missing class implementations in the provided template. Do **not** add any other top-level definitions.
- When attempting to create a `Wrapper` or `Box` with an unsupported number of `Cookies`, a `Warning` should be raised.
- Make sure to minimize code duplication.
- The provided examples contain valuable information on how the solution should work. Your solution *must* be compatible with the provided examples.
- Do **not** include any example calls in your submission. Submit only top-level class definitions and any imports you might need.
- Some points will be given for each feature that works correctly, some points will be given for optimally designing the implementation. Recommendation: prioritize implementing a solution that works correctly with the provided examples. Optimize the design afterwards.

Use the following template:

```python
from abc import ABC, abstractmethod

class Cookie:
    pass

class Container:
    pass

class Wrapper:
    pass

class Box:
    pass
```

The following example illustrates how the solution may be used:

```python
def make_cookies(n):
    return [Cookie("Chocolate", 0.50) if x % 2 == 0 else Cookie("Vanilla", 0.60) for x in range(n)]

cookies = make_cookies(4)
print(cookies[0].get_name())
print(cookies[0].get_price())

w1 = Wrapper(cookies)
print([c.get_name() for c in w1.get_contents()])
```

```python
w2 = Wrapper(make_cookies(4))
b = Box([w1, w2])
print(f"\nCookies in box: {b.get_number_of_cookies()}")
print(f"  Price of box: {b.get_price()}")
print(f"     ID of box: {b.get_id()}\n")

more_wrappers = [Wrapper(make_cookies(4)) for x in range(52)] # 208 cookies
try:
    overfull = Box(more_wrappers)
except Warning:
    print("Too many cookies for one box\n")
```

The code above should produce the following output:

```
Chocolate
0.5
['Chocolate', 'Vanilla', 'Chocolate', 'Vanilla']

Cookies in box: 8
  Price of box: 4.4
     ID of box: 1

Too many cookies for one box
```

Paste your solution into the following field. Utilize the template above and make absolutely sure to avoid syntax errors! Submit everything that is necessary to import your solution - not more, not less!

Sample solution

```python
#!/usr/bin/env

from abc import ABC, abstractmethod

class Cookie:
    def __init__(self, name, price):
        self.__name = name
        self.__price = price

    def get_name(self):
        return self.__name

    def get_price(self):
        return self.__price

class Container(ABC):
    def __init__(self, contents):
        self.__contents = contents

    def get_price(self):
        return round(sum([w.get_price() for w in self.__contents]), 2)

    def get_contents(self):
        return self.__contents[:]

    @abstractmethod
    def get_number_of_cookies(self):
        pass

class Wrapper(Container):
    def __init__(self, cookies):
        super().__init__(cookies)
        if len(cookies) < 3 or len(cookies) > 5:
            raise Warning

    def get_number_of_cookies(self):
        return len(self.get_contents())

class Box(Container):
    seq = 1
    def __init__(self, wrappers):
        super().__init__(wrappers)
        if self.get_number_of_cookies() > 200:
            raise Warning
        self.__id = Box.seq
        Box.seq += 1

    def get_id(self):
        return self.__id

    def get_number_of_cookies(self):
        return sum([w.get_number_of_cookies() for w in self.get_contents()])

# examples
def make_cookies(n):
    return [Cookie("Chocolate", 0.50) if x % 2 == 0 else Cookie("Vanilla", 0.60) for x in range(n)]
```

```python
cookies = make_cookies(4)
print(cookies[0].get_name())
print(cookies[0].get_price())

w1 = Wrapper(cookies)
print([c.get_name() for c in w1.get_contents()])

w2 = Wrapper(make_cookies(4))
b = Box([w1, w2])
print(f"\nCookies in box: {b.get_number_of_cookies()}")
print(f"  Price of box: {b.get_price()}")
print(f"     ID of box: {b.get_id()}\n")

more_wrappers = [Wrapper(make_cookies(4)) for x in range(52)] # 208 cookies
try:
    overfull = Box(more_wrappers)
except Warning:
    print("Too many cookies for one box\n")
```