Mod 2 Homework: OOP & TDD

This assignment introduces object-oriented programming (OOP) and test-driven development (TDD). Both techniques generally require a good amount of coding experience to fully appreciate the benefits of, but know that they're both useful and lucrative skills to have.

Part 1 - OOP with Animals

This first part is a quick problem to get comfortable with OOP structuring. It is autograded, only worth 10% of the assignment, and should take you ~20 minutes to complete.

Create classes to represent the following hierarchy using inheritance in a module called `Animals.py`.

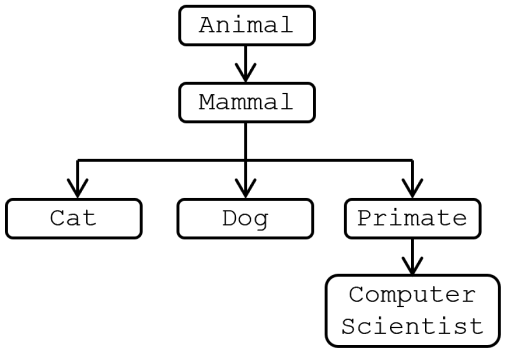

Figure 1: Taxonomy to implement

Implement the following functionality:

a) Every class should be initialized with a name - call this instance variable `name`. Because all objects share this behavior, you can put it in the `Animal` constructor `__init__`.

b) Every class *except* `ComputerScientist` should define its own `speak()` method, which returns a string giving the name and an appropriate sound for that class. We only test the actual return string for `Cat.speak`, but we do test that the other objects all implement thier *own* `speak` method.

```
>>> c1 = Cat('Babs')
>>> c1.speak()
'Babs says Meow!'
```

c) Do not implement `speak` in the `ComputerScientist` subclass - it should default to whatever method you define for `Primate`.

d) The `Animal` superclass should define a method called `reply()`, which just calls the relevant `speak`:

```
>>> c1 = Cat('Babs')
>>> c1.speak() # should call Cat.speak()
'Babs says Meow!'
>>> c1.reply() # should call Animal.reply
'Babs says Meow!'
```

Part 2 - OOP & TDD with Cards

This is where the bulk of your work will be for this assignment. Anticipate spending at least a few hours here if you are new to OOP and TDD, and consider breaking it up over a few days if your schedule allows.

Card games are a classic application of OOP. They let us use composition (decks of cards *contain* several card objects) and inheritance (a hand of cards can be treated as a specialized deck). The diagram below shows the inheritance model and the specific instance variables and bound methods we'll implement here.

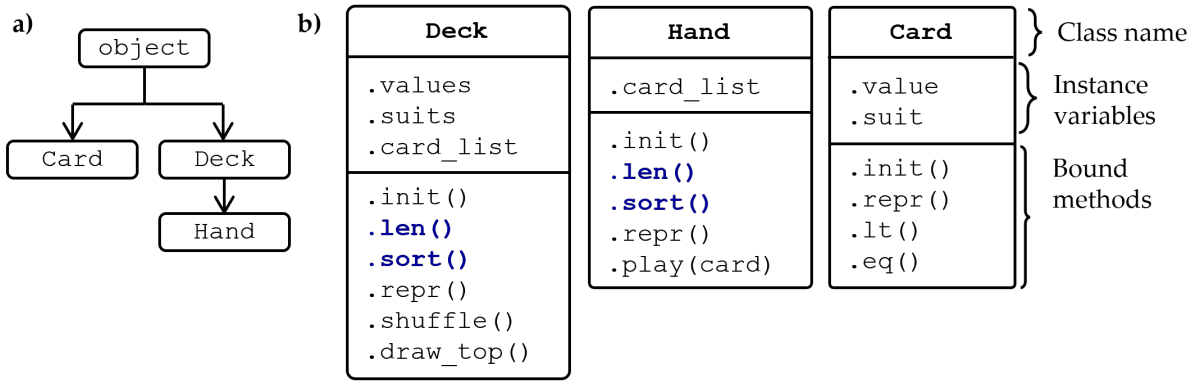

Figure 2: (a) Class diagrams and (b) attributes for each class. Inherited attributes are in blue.

Use TDD to create the classes above in a file called `Cards.py`, writing unittests as you go in a file called `TestCards.py`.

TestCards.py

Some cool OOP effects you'll encounter:

- Composition - Decks are comprised of (a list of) `Card` objects
- Inheritance - Some `Hand` are directly inherited from `Deck` and do not need to be coded at all

We explicitly define the expected input/ouput of each method (the *interface*) below, but most of them do what you'd intuit. Before you start implementing code, review these guidelines:

- Use TDD. Write a unittest first, then implement functionality.
  - This includes any exceptions you should raise. See the unittest basic example for an illustration of how to test that an error is raised (link)
  - Your final `TestCards.py` class should include 3 classes - one for each class you are trying to test:

```
from Cards import Card, Deck, Hand
import unittest

class TestCard(unittest.TestCase):
   "Test cases specific to the Card class"
   def test_init(self):
      "Add a docstring here"

   # other tests
```

```
class TestDeck(unittest.TestCase):
    # your tests here

class TestHand(unittest.TestCase):
    # your tests here

unittest.main() # Runs all tests above
```

- Structure your code based on OOP principles. `Hand` should inherit from `Deck`, and should not overload any methods unless it needs to.
- Every method (including your unittests) should have a docstring.
- When writing tests, create your own examples. Do not use any of the examples shown below.

Card

- `init` - initialize a new card based on the specified parameters:
  - `value` - the value of a card (i.e. the `3` in `3 of hearts`)
  - `suit` - the suit of a card (i.e. the `hearts` in `3 of hearts`)
- `repr` - return something like `'Card(3 of hearts)'`
- `lt` - Implement as a magic method (`__lt__`) so it can be called with the standard operator (`<`). Sort by suit first, then value (suits are sorted alphabetically, so `clubs` < `diamonds`)

Examples

```
>>> c1 = Card(3, 'hearts')
>>> repr(c1)
'Card(3 of hearts)'
>>> c2 = card(3, 'spades')
>>> c1 < c2
True
>>> c3 = Card(4, 'hearts')
>>> c3 < c2
True
```

Deck

- `init` - initialize a deck based on the collection of values and suits passed in - make one card for each value/suit combination.
  - `values` - collection of values stored in deck. This should be a parameter with a default of the numbers 1 through 13
  - `suits` - collection of suits stored in deck. This should be a parameter with a default of `('clubs', 'diamonds', 'hearts', 'spades')`
  - `card_list` - list of `Card` objects, containing all cards in the deck
- `len` - the number of cards in the deck. Use the **magic method** `__len__`.
- `sort` - sorts the cards in the deck

- `repr` - returns a string representation of the deck. **Magic method.**
- `shuffle` - randomize the order of the deck. Import the `random` module and use `random.shuffle` here.
- `draw_top` - remove and return the top card of the deck.
  - Treat the last item in `card_list` as the "top" of the deck.
  - Raise a `RuntimeError` if someone tries to draw from an empty deck

Examples

```
>>> d1 = Deck() # use default values and suits
>>> len(d1)
52
>>> d2 = Deck([2, 1], ['triangles', 'dots'])
>>> repr(d2)
'Deck: [Card(1 of dots), Card(2 of dots), Card(1 of triangles), Card(2 of triangles)]'
>>> d2.shuffle()
>>> repr(d2)
'Deck: [Card(2 of triangles), Card(1 of dots), Card(2 of dots), Card(1 of triangles)]'
>>> d2.draw_top()
Card(1 of triangles)
>>> d2.draw_top()
Card(2 of dots)
>>> d2.draw_top()
Card(1 of dots)
>>> d2.draw_top()
Card(2 of triangles)
>>> d2.draw_top()
Traceback (most recent call last):
...
RuntimeError: Cannot draw from empty deck
```

Hand

- `init` - create a `Hand` with a passed in collection of cards
- `repr` - returns a string representation of the Hand. **Magic method**
- `play(card)`
  - removes and returns `card` from hand
  - raises a runtime error if `card` is not in hand

Examples

```
>>> h_clubs = Hand([Card(value, 'clubs') for value in range(5, 0, -1)])
>>> repr(h_clubs)
'Hand: [Card(5 of clubs), Card(4 of clubs), Card(3 of clubs), Card(2 of clubs), ...'
>>> h_clubs.sort() # inherited from Deck
>>> repr(h_clubs)
'Hand: [Card(1 of clubs), Card(2 of clubs), Card(3 of clubs), Card(4 of clubs), ...'
>>> len(h_clubs)
```

```
5
>>> h_clubs.play(Card(1, 'clubs'))
`Card(1 of clubs)`
>>> h_clubs.play(Card(1, 'clubs'))
Traceback (most recent call last):
...
RuntimeError: Attempt to play Card(1 of clubs) that is not in Hand: [Card(5 of clubs),...
```

Submitting

At a minimum, submit the following files:

- `Animals.py`
- `Cards.py`
- `TestCards.py`

Students must submit individually by the due date (typically Tuesday at 11:59 pm EST) to receive credit.