EPIC Lab 1: Python Blackjack

Introduction:

Welcome to the first EPIC lab! In this lab, we will be creating a text-based Blackjack game, where even the cards are text. This lab is simpler than the next two EPIC labs, as most of you are just starting to program for the first time, so this will be a good lab to ease you into things. We will still be covering concepts that you have not yet learned in class, so I (the EPIC IAI) will be giving a few short examples of these concepts at the beginning of the lab to help you out. We will be seeing classes, conditional statements, and while loops for the first time (based off the 1D04 curriculum). Games like Blackjack offer excellent programming exercises when you are first starting out, as they offer opportunities to implement plenty of decisions in your code (the aforementioned conditional statements). If you've programmed before, this lab will likely not be that difficult for you, so feel free to help your fellow classmates if you finish early.

The format for the lab is meant to be easy to follow. You will be starting with an incomplete python file, called blackjack.py. In this file, there is some code already laid out for you (starting from scratch would take longer than the 2-hour lab allows). Each of the steps in the lab is numbered, and this number corresponds with a comment in the file, showing you where each step takes place in the code. For example, if you are working on step #5, there will be a line in the file that starts with #5 and all the code for step 5 will take place in this part of the file (hashtags denote comments in Python, lines of code that Python will not read). The steps are ordered from the top to the bottom of the file, so step #1 is near the top of the file and the final step is near the bottom of the file. If you get lost in the file, scroll back to the top and continue down the page until you find where you left off. Lastly, if you have any questions, do not hesitate to ask the EPIC IAI (me) for help! Now, on to the lab.

(Note: While I will be doing examples, and explaining some of the lab in order to help facilitate things, you are more than welcome to ignore that and start the lab immediately if you feel like it. Just be considerate of your fellow group members and make sure they are okay with it too. I understand that not everyone needs a recap of the concepts covered in this lab.)

Step 1: In this step, we go over the functions we've imported and the Card class we will be using for much of the lab. At first, we can see that we've imported two functions, each from a different library:

```
from random import sample
from time import sleep
```

If you haven't seen this notation before, we're simply saying that we only want one specific function from a given library. This saves us the trouble of importing the entire random library when all we want is the sample function from this library, for example. The sample function requires two arguments and is called like so: `sample(population, k)`. In this example, population is a list of 1 or more elements, and k is an integer designating how many samples we would like to take from the population. This function implements sampling without replacement, which means when we take a sample from the population, the item sampled is permanently removed from the population. We will be using this function to shuffle our deck of cards; by taking 52 samples from a deck of cards, we randomly reorder the cards and create a new deck. The sleep function only takes one argument: `sleep(seconds)`. It pauses the Python script for the designated number of seconds. We use it to slow down the program and make it easier for the player to process what is going on, as otherwise, Python runs all the game steps near instantly. Next, we look at the Card class, and then on to the coding!

```
# Possible values for value: 2,3,4,5,6,7,8,9,T,J,Q,K,A
# Possible values for suit: "C" = clubs, "D" = diamonds, "H" = hearts, "S" = spades
class Card:
    def __init__(self,value,suit):
        if value == 'T':
            self.value = "10"
        else:
            self.value = str(value)
        suit_dict = {'C': 1, 'D': 2, 'H': 3, 'S': 4}
        self.suit = '♣♦♥♠'[suit_dict[str(suit)]-1] # 1,2,3,4 = ♣♦♥♠

    def card_front(self):
        return[
                ' ┌─────────┐ ',
                f'| {self.value:<2}      |',
                '|         |',
                f'|    {self.suit}    |',
                '|         |',
                f'|     {self.value:>2} |',
                ' └─────────┘ '
                ]
    def card_back(self):
        """creates back of card as a list of strings, used mainly for dealer"""
        return[
                ' ┌─────────┐ ',
                ' │▒▒▒▒▒▒▒▒▒│ ',
                ' │▒▒▒▒▒▒▒▒▒│ ',
                ' │▒▒▒▒▒▒▒▒▒│ ',
                ' │▒▒▒▒▒▒▒▒▒│ ',
                ' │▒▒▒▒▒▒▒▒▒│ ',
                ' └─────────┘ '
                ]
    def get_value(self):
        """Returns the card's numerical value"""
        if self.value in ['T', 'J', 'Q', 'K']:
            card_value = 10
        elif self.value == "A":
            card_value = [1,11]
        else:
            card_value = int(self.value)
        return card value
```

This may be the first time you've seen a Class before. A class is basically a blueprint or a template for creating an object, which is the backbone for object oriented programming (OOP). Python is an example of an OOP language, and this style of programming can open novel and clever ways of solving programming problems. An object is basically a collection of functions and variables, and an object gets its functions and variables from the class definition. In this class, we can see there are four functions: `__init__(self, value, suit)`, `card_front(self)`, `card_back(self)`, and `get_value(self)`. We will discuss these quickly, and then finish off with an example of how to use class functions on a given object. The init function is known as the constructor, and it defines what values a given object needs before it can be created. All classes must have a constructor method. The variable self is a bookkeeping detail, and only used for classes. We won't worry about this much right now, as you won't be covering classes in detail for many more weeks, and it is not that important to this lab. The other two variables, value and suit, are used for creating a card. They must be strings, and we can see in a comment above the class what the acceptable values are for these variables. Card_front(self) will return a list of strings representing the text-based image of the card, while card_back(self) will represent the card as if it was upside down (used for the dealer). Lastly, the get_value(self) function will return the numerical value of the card as determined by the rules of blackjack.

To finish up this part, we quickly create a card object and show off how to call class functions on the card object. In the code below, we create a card object, check the value and suit of the card, and then print out the card to the shell.

```
>>> myCard = Card("3", "H")
>>> myCard
<__main__.Card object at 0x03036FF0>
>>> myCard.suit
'♥'
>>> myCard.value
'3'
>>> myCard.get_value()
3
>>> for line in myCard.card_front():
        print(line)
```

```
┌─────────┐
│ 3       │
│         │
│    ♥    │
│         │
│       3 │
└─────────┘
```

We can see that myCard is storing the object we just created, so we use the variable myCard whenever we want to refer to this specific object. We created the card by specifying the value and suit, in this case a 3 of hearts. We can check the suit and value by using the variables designated in the constructor, self.suit and self.value. Note that these return the values as strings, so if we want a numerical value instead of a string, we use myCard.get_value(). This function is known as an accessor, since it is accessing some information about the object for us (it's numerical value). Also note that when calling class functions, we do not include self as one of the parameters in the function call (self is called the first formal parameter and it is assumed to always be present, so we don't have to type it out each time). Lastly, myCard.card_front() returns a list of strings, and we can have a for loop iterate over any list. As a result, we go through each string in the returned list and print it one line at a time, resulting in the simple ASCII card shown.

If this seems like a lot of information at once, it can be, especially if you're new to programming, so don't be afraid to ask questions. Also, don't worry too much if this section is confusing, as you don't need to understand classes well to implement them in the steps below.

Step 2: Here's where we get in to some actual coding. We're filling in some blanks to make our first helper function that creates and shuffles a deck for us to use later.

```
def create_deck():
    """ Generates a full deck of 52 cards using the Card class. The deck is
    returned as a list of Card objects, and is shuffled."""
    # Step 2:
    suits = ["C", "D", "H", "S"]
    values = ["2","3","4","5","6","7","8","9","T","J","Q","K","A"]
    deck = []
    for suit in _____:
        for value in ______:
            # add the new card to the deck here using append
    deck =  # shuffle the deck we just created
    return deck
```

Here we have a few predefined lists to store the possible card suits and values, as well as an empty list where we will be storing our deck of card objects. We have a nested for loop set up that needs a few variables filled in before it will work. The nested for loop should create 52 card objects with all possible suits and values, and at the end of each loop, we should add the new card object to the deck list using the append method. The append method looks like list_variable.append(item), and it adds the item in the brackets to the specified list. Lastly, we will need to use the sample function (explained in step 1) to shuffle the deck now that we've made one. Once you think you've got it working, we now have a function to make and shuffle a deck for us! That will make things easier when we get to the main game loop.

Step 3: Now we need a function to deal both the player and the dealer a hand at the beginning of a game. For this function, the input is the deck of cards (which is a list of card objects) that we want to deal from.

```
# Start of Step 3 ------------------------------------------------------------
def start_of_game_deal(deck):
    """Used at the start of a game, deals the player and the dealer two
    cards each"""
    dealer_hand = []
    player_hand = []
    # implement a for loop to deal two cards to both the player and the dealer

    return player_hand, dealer_hand
# End of step 3 --------------------------------------------------------------
```

All we have to do here is implement a for loop to deal cards to the player and dealer, two each. Generally, in blackjack, the dealing goes player->dealer->player->dealer, so that each player gets one card at a time. Still, if your function outputs a list of two card objects for both the dealer and the player, this function will work. For a hint, there is a list function called pop(), which can be used to remove the last element of a list permanently and return it to be used somewhere else. A quick example of the pop function in action:

```
>>> myList = [2,3,4,5]
>>> myList.pop()
5
>>> myList
[2, 3, 4]
>>>
```

We can see the final element, the 5, was removed from myList, and if we wanted to, we could have assigned the removed element to a variable, like x = myList.pop(). Using this method, it should be straightforward to implement a for loop that deals both the player and the dealer two cards from the deck.

Step 4: In this step, we need to make a function that calculates the numerical value of a given player's hand. Remember that a hand is just a list of card objects, like a deck but smaller. In this case, the function takes in the hand that we want to check the value of as input. First, we initialize two variables to store the total value of the hand and the number of aces. In the beginning part of this function, we want a for loop that can sum up all the card values in the hand that ARE NOT Aces. This means we will have to implement for loop with a conditional statement that cycles through all the cards in the hand, and only adds the current card's value to the total if it is not an Ace. This is because we will handle the aces in the next step, as Ace's can have a value of 1 or 11 depending on the cards in your hand.

This brings up our first conditional check in the lab, which we will resolve using an if statement. If we look at the card class, we can see that Ace cards are the only card to not return a single numerical value when we use card.get_value(), but instead returns a list of two values, [1,11]. We can use this to check if a card is an Ace, because we can check the type of a value in Python. We know if the type of the returned value is a list, it must be an Ace, so we will not add that cards value to hand_value now. An example of checking a variable's type using an if statement is shown below:

```
>>> myString = "This is a string."
>>> type(myString)
<class 'str'>
>>> if type(myString) == str:
        print("This variable holds a string.")


This variable holds a string.
>>>
```

The == operator is checking for equality. If the item on the left of the operator is the same as the item on the right, the condition evaluates to True and we execute the code indented below the if statement. If the condition evaluates to False, we skip the code indented below the if statement, and continue in our script. Using this information, you should be able to implement the first part of this function by filling in the blanks, as shown below.

```
# Start of Step 4 ------------------------------------------------------------------
def check_hand_value(hand):
    """Checks the numerical value of the cards in the given hand. A hand is a
    list of card objects"""
    hand_value = 0
    num_aces = 0
    # Sum up all non-Ace card values first (HINT: use card.get_value() here)
    for card in ____:
        card_value = _____# get the card's value here
        if type(____) == ____: # Only Ace cards return a list for their value
            # set the ace card's value to 0 and add 1 to the number of aces
        hand_value += _____ # add the value of the card to the hand
```

In the next part of this function, we have to check multiple conditions that are mutually exclusive (i.e if one is true, the other are false by default). We will be doing this with the if-elif-else statements. We've already seen the if statement, so moving on, we have an elif statement. Short for else-if, this statement will only evaluate itself if all previous conditional statements on the same level of indentation (that is, statements lined up vertically with the elif) have evaluated to False. The else statement, on the other hand, does not have a condition to check, and only runs if all other previous conditionals are False. The else statement is a way of saying "if everything else fails, do this no matter

what". Keeping that in mind, here is a simple Python script showing what these statements do:

```
>>> myList = ["This is a string.", 7, True]
>>> for item in myList:
        if type(item) == str:
                print("This item is a string.")
        elif type(item) == int:
                print("This item is an integer.")
        else:
                print("This item is something else.")


This item is a string.
This item is an integer.
This item is something else.
```

Using the above example, you'll want to implement similar checks for the code below:

```
# Now check if the value of an Ace should be 1 or 11 in the given hand
if ______:           # First check if we have 0 aces, in which case we c
    return hand_value
elif ________ > 21: # Check if counting the first ace as 11 and all oth
    hand_value += _____ # Add the value of the aces to the hand, in th
    return hand_value
else:                # The else runs if the first ace has a value of 11
    hand_value += _____ # Add the value of the aces to the hand, the f
    return hand_value
```

There are hints next to each conditional for what you should be checking and what you should be adding to the value of the hand. Once you've completed this section, we're almost done implementing all the functions we'll be using in our final game loop.

Step 5: In this step, we'll be working on two similar functions instead of one single function. The two functions in this case take in a player's hand as input, and prints either all the cards face up (used mainly for the player and to see the dealer's final hand) or it prints only the first card face up and the rest face down (used for the dealer so you can't see all its cards). The two options are very similar once they've been coded, so completing the first function should make the second function easier to implement. The comments are cut off in the photo, but make sure to read them for useful hints on completing this portion of the lab.

```
# Start of Step 5 ----------------------------------------
def show_hand_unhidden_cards(hand):
    player_hand_graphics = []
    # Implement a for loop to create a list of lists of str
    # HINT: A hand is a list of card objects, and you can u
    # for loop goes here

    # Each card has seven lines to print
    for i in range(7):
        card_str = ""            # initialize an empty strin
        for j in range(___):     # The second loop should ru
            card_str += player_hand_graphics[_][_] + " " # 
        print(card_str)          # prints all of the cards i

def show_hand_hidden_cards(hand):
    dealer_hand_graphics = [hand[0].card_front()] # The fir
    for i in range(_, ____):                      # This lo
        dealer_hand_graphics.append(____)         # Add the
    # Again, each card has 7 lines to print. Implement the
```

This function isn't doing anything we haven't seen before, outside of the multiple indexes used for player_hand_graphics. Since we are creating a list of lists in the first for loop, we need to use more than one index to access the values we want. The Python script below illustrates this concept:

```
>>> list_of_lists = [[1,2,3], ["four", "five", "six"], [7.0, 8.0, 9.0]]
>>> list_of_lists[0]
[1, 2, 3]
>>> list_of_lists[1]
['four', 'five', 'six']
>>> list_of_lists[0][1]
2
```

The first index specifies which list inside of list_of_lists we want, but this just returns another list. If we add a second index, we get the value stored in the second index from the list designated by the first index (i.e [0][1] gives the second value from the list at index 0).

After implementing these two functions, we have one step left before entering the main game loop, and then testing our game!

Step 6: We're almost done before the last step, where we complete the main game loop and (hopefully) have a working game! This step covers the hit function, which adds a card to a given hand, and the dealer_turn function, which simulates the rudimentary dealer AI for a given hand.

```
def hit(deck, hand):
    """Allows a player to hit and draw one more card"""
    _____# Add a card from the deck to the hand here. HINT: remember pop()
    return hand

def dealer_turn(deck, hand):
    dealer_stands_on = 17
    print("Now it's the dealer's turn! The dealer stands on %s." % dealer_stands_on )
    print("Dealer's hand:")
    _____ # On this line, show the dealer's unhidden hand using the function from ste
    while _____ < _____: # Keep looping for as long as the value of the hand is less
        print("The dealer draws a card.")
        ______    # add a card to the hand using the hit function above
        ______    # Show the unhidden hand of the dealer
                  # Pause to allow the player to process what the dealer draws. HINT:
    if ____ > ____: # Check if the dealer's hand has a value over 21, which means the
        print("The dealer busts!")
        return 0
    else:         # Here, the dealer hit between 17 and 21 and now we can resolve the
        print("The dealer is finished drawing cards.")
        print("Dealer's final hand:")
        _____     # Show the unhidden hand of the dealer
        sleep(_) # Pause for readability
        return _____ # return the final value of the hand
```

This section introduces our first While loop, which is different from a for loop. A for loop iterates over a set number of items, so we always know how many times a given for loop will run. A while loop, on the other hand, runs until the given condition is False. This opens a lot of neat behavior for our programs, but it also means we cannot tell at a glance how long a given while loop will run. In fact, it is even possible to implement an infinite loop that will run forever if you aren't careful. If this happens, hit Ctrl + C on your keyboard to trigger Python's built-in KeyboardInterrupt exception and halt the program. The code below shows off a simple while loop:

```
>>> x = 0
>>> while x < 10:
        print(x)
        x += 1


0
1
2
3
4
5
6
7
8
9
```

Once you've filled in all the blanks, you are done the lab! The main game loop has already been implemented; you do not need to change any code in the next step. It is an optional step where you look at the main game loop to see how it works.

Step 7: Here we implement the final game loop using all the functions we just finished defining. The game loop will take the form of a while loop, specifically an interactive loop. We will give the player some initial amount of chips to play with, and the game will end when either the user runs out of chips or decides to quit the game. The game loop might look busy, but most of the code here is just print statements. Go ahead and try running your code once you are sure everything is complete! The automated Python error messages often show you exactly what line is causing the problem, as well as a brief description of the error.

And that's it! The next lab will be a bit more engineering-related, where we will be going over some simple image processing to create an ASCII-art generator! Before and after pictures below (the second is a text file):