README.md

Assignment 3 - Blackjack

BACKGROUND

In this assignment, you will continue developing your ability to use conditional statements and loops to control the flow of your programs. User input and output will also be utilized.

Blackjack is a card game that is played in many casinos. It often goes by the name of 21 because the goal of the game is to be dealt cards such that the sum of the values of each card adds up to this value. You typically play at a table with multiple players. Each player is attempting to hold a set of cards that has a higher total value than those held by the dealer. Neither the player nor dealer can win while holding a group of cards with a value greater than 21.

To simplify this assignment, the only possible values that a card can have are `1` - `10`. Jack (represented by the string `J`), Queen (represented by the string `Q`), and King (represented by the string `K`) cards each have a value of `10`. Ace (represented by the string `A`) has a value of 1. Card suits will be ignored in this version of Blackjack. In addition, this implementation of Blackjack will only include a single player and the dealer.

Baby Blackjack

In order to build towards the final implementation of Blackjack, you will start by implementing a very simple version of the game called *Baby* Blackjack. For this version of the game, the player will receive two cards and the dealer will receive two cards. When the player has a higher score, the player wins. Otherwise, the player loses. An example of a game of *Baby* Blackjack is shown below:

```
Let's Play Blackjack!

Player drew 7 and 8.
Player's total is 15.

The dealer has Q and 3.
Dealer's total is 13.

YOU WIN!
```

In this game, the player received one card (labeled `'7'`) with a value of `7` and another card (labeled `'8'`) with a value of `8` for a total value of `15`. The dealer received one card (labeled `'Q'`) with a value of `10` and another card (labeled `'3'`) with a value of `3` for a total value of `13`. The player's score is higher, so the player wins the game which is indicated by the output `YOU WIN!`. An example of a game where the player loses is displayed below:

```
Let's Play Blackjack!

Player drew 8 and 3.
Player's total is 11.

The dealer has 6 and 9.
Dealer's total is 15.

YOU LOSE!
```

For *Baby* Blackjack, the program will end after outputting if the player won or lost the game.

CS1114 Blackjack

Once you have properly implemented *Baby* Blackjack, you are responsible for implementing a basic version of Blackjack which has the following rules.

- The player is dealt two cards at the beginning of each game.
- The player is prompted with the question `Hit (h) or Stay (s)?` and must respond with `s` to stop receiving cards. When the user enters `h` (the player "hit"), a new card is dealt and added to the player's total. Any other input from the user, will result in the question `Hit (h) or Stay (s)?` being repeated until a valid input is provided.
- The question prompt `Hit (h) or Stay (s)?` is continuously displayed until the user enters `s` or the total value of the cards dealt to the player exceeds 21 (in which case the player "busts" and loses).
- When the player **has not** busted, the dealer is dealt two cards just like in *Baby* Blackjack.
    - The dealer is then dealt cards until the total value of the dealer's cards is greater than `16`, at which point, no more cards will be dealt to the dealer.
    - The dealer busts if the dealer's cards' value exceeds `21`.
- The player wins in the following scenarios:
    - The dealer busts and the player has not busted.
    - The total value of the player's cards exceeds the dealers total.

The player loses in **all** other scenarios.

Below is an example of a game of CS1114 Blackjack:

```
Let's Play Blackjack!

Player drew J and 3.
Player's total is 13.

Hit (h) or Stay (s)? e

Hit (h) or Stay (s)? b

Hit (h) or Stay (s)? h

Player drew 1.
Player's total is 14.

Hit (h) or Stay (s)? h

Player drew 2.
Player's total is 16.

Hit (h) or Stay (s)? h

Player drew 3.
Player's total is 19.

Hit (h) or Stay (s)? s

The dealer has 2 and 6.
Dealer's total is 8.

Dealer drew 9.
Dealer's total is 17.

YOU WIN!

Play again (Y/N)?
```

Below is an example of a game of CS1114 Blackjack when the player busts:

```
Let's Play Blackjack!

Player drew Q and 3.
Player's total is 13.

Hit (h) or Stay (s)? h

Player drew 1.
Player's total is 14.

Hit (h) or Stay (s)? h

Player drew 2.
Player's total is 16.

Hit (h) or Stay (s)? h

Player drew 3.
Player's total is 19.

Hit (h) or Stay (s)? h

Player drew K.
Player's total is 29.

YOU LOSE!

Play again (Y/N)?
```

Notice the prompt for the user to indicate whether or not to play again. This message will continue to be displayed until the user enters a valid input (`Y` or `N`). Below is an example of multiple attempts at entering invalid input:

```
Play again (Y/N)? y

Play again (Y/N)? n

Play again (Y/N)? b

Play again (Y/N)? N

Goodbye.
```

Notice that the `Play again (Y/N)?` prompt only stops being displayed when a valid input (`N`, in this case) is provided.

TABLE OF CONTENTS

STEP 0 - READ THE README

Before writing a single line of Python code for this assignment, you are **strongly** encouraged to read this document in its entirety. The document includes key details about the expected implementation of your program. Diving into the implementation based solely on the description of the program above is likely to result in following a path towards your program's implementation that fails to meet the requirements of this assignment.

If you would like to consume this document in an alternative format, the README has also been converted to a PDF which is available in this repository.

DECOMPOSITION

We will continue to utilize our functional decomposition strategy in the implementation of this program. For this assignment, most of the decomposition has been done for you. However, you are always welcome to further decompose your programs as you see fit.

The structure of this document breaks down the assignment into a series of steps that if followed closely will lead to an implementation that meets the requirements for the assignment. The steps also correspond to groups of tests that can be used to pursue an incremental development approach to writing your programs. Start with **Step 1**, implement the function as described for that step, and submit your (unfinished) code to Gradescope. A tool that runs automated tests against your submission will test your code to ensure that it meets the requirements of the assignment. If your submission does not pass the tests, this indicates that some requirement of the assignment has not been met. You are able to submit your assignment to Gradescope as many times as you want without any penalties. Your assignment will likely fail the non-**Step 1** tests. However, when implemented to meet the requirements in this README, the **Step 1** tests should pass. If the tests do not pass, try to alter your code to get the tests to pass before moving on to **Step 2**. Ask for help of the course staff if you are having trouble getting the tests to pass. Once the **Step 1** tests pass, proceed to Step 2 and repeat this process. This incremental approach to implementing your programs is an approach that is less likely to lead to bugs in your code and is being emphasized as a key learning goal for this course.

INSTRUCTIONS

Restrictions: No Python container datatypes (`dict`, `set`, etc) other than `list` and `tuple` are allowed in the implementation of this program

"Baby" Blackjack

Step 1 - Dealing a card

Function: `deal_card()`

```
def deal_card():
    """Evaluates to a character representing one of 13
    cards in the CARD_LABELS tuple

    :return: a single- or double-character string representing a playing card

    >>> random.seed(13)
    >>> deal_card()
    '5'
    >>> deal_card()
    '5'
    >>> deal_card()
    'J'
    """
```

The first task is to implement the `deal_card()` function. This function returns a card represented by one of the following labels: `A`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `J`, `Q`, `K`. Each time a function call is made to `deal_card()`, one of these string values will be returned randomly. Randomly selecting a card can be achieved using the `random.choice()` function from the `random` module. `random.choice()` requires one parameter argument -- a non-empty sequence. Given a sequence argument, `random.choice()` will evaluate to one of the values in the sequence. You should use the provided `CARD_LABELS` constant (a tuple containing the card labels available in CS1114 Blackjack) as the sequence in your function call to `random.choice()`.

Once `deal_card()` has been implemented, try submitting your `blackjack.py` file to Gradescope to see if your implementation passes the **Step 1** tests. Once your program passes the **Step 1** tests, proceed to **Step 2**.

Step 2 - Getting card values

Function: `get_card_value(card_label)`

```
def get_card_value(card_label):
    """Evaluates to the integer value associated with
    the card label (a single- or double-character string)

    :param card_label: a single- or double-character string representing a card
    :return: an int representing the card's value

    >>> card_label = 'A'
    >>> get_card_value(card_label)
    1
    >>> card_label = 'K'
    >>> get_card_value(card_label)
    10
    >>> card_label = '5'
    >>> get_card_value(card_label)
    5
    """
```

With the ability to deal a card, as implemented in the `deal_card()` function, **Step2** will focus on defining the `get_card_value()` function which evaluates to the integer value associated to each card based on the card's label. The integer value of each card label is displayed in the table below:

label	value
`'A'`	`1`
`'2'`	`2`
`'3'`	`3`
`'4'`	`4`
`'5'`	`5`
`'6'`	`6`
`'7'`	`7`
`'8'`	`8`
`'9'`	`9`
`'10'`	`10`
`'J'`	`10`
`'Q'`	`10`

label	value
`'K'`	`10`

Given a `card` referencing one of the 13 labels, the `get_card_value()` function will return the corresponding `value` from the table above. The constant variables `FACE_CARD_VALUE` and `ACE_VALUE` should be used in this function to avoid magic numbers in your program.

Once `get_card_value()` has been implemented, try submitting your `blackjack.py` file to Gradescope to see if your implementation passes the **Step 2** tests. Once your program passes the **Step 2** tests, proceed to **Step 3**.

Step 3 - Deal cards to player (*Baby* Blackjack)

Function: `deal_cards_to_player()`

```
def deal_cards_to_player():
    """Deals cards to the player and returns the card
    total

    :return: the total value of the cards dealt
    """
```

With the ability to deal a card and evaluate the card's value, the initial `deal_cards_to_player()` function can be defined. The function should display the two cards dealt to the player. The total value of the two cards should also be output.

```
Player drew J and 3.
Player's total is 13.

```

Notice that an additional blank line is output after displaying the player's total. In addition, this function should **return** the total value (as an `int`) of the cards dealt to the player. This function will make use of function calls to `deal_card()` and `get_card_value()` within the function body.

Once the initial version of `deal_cards_to_player()` has been implemented, try submitting your `blackjack.py` file to Gradescope to see if your implementation passes the **Step 3** tests. Once your program passes the **Step 3** tests, proceed to **Step 4**.

Step 4 - Deal cards to dealer (*Baby* Blackjack)

Function: `deal_cards_to_dealer()`

```
def deal_cards_to_dealer():
    """Deals cards to the dealer and returns the card
    total

    :return: the total value of the cards dealt
    """
```

The `deal_cards_to_dealer()` function is very similar to the `deal_cards_to_player()` function. The `deal_cards_to_dealer()` function should display the two cards dealt to the dealer. The total value of the two cards should also be output. The major difference between the functions is in the wording of the output that is displayed.

```
The dealer has 6 and 9.
Dealer's total is 15.

```

Again, notice that an additional blank line is output after displaying the total. This function should **return** the total value (as an `int`) of the cards dealt to the dealer. This function will also make use of function calls to `deal_card()` and `get_card_value()` in the function definition.

Once the initial version of `deal_cards_to_dealer()` has been implemented, try submitting your `blackjack.py` file to Gradescope to see if your implementation passes the **Step 4** tests. Once your program passes the **Step 4** tests, proceed to **Step 5**.

Step 5 - Determine game outcome

Function: `determine_outcome(player_total, dealer_total)`

```
def determine_outcome(player_total, dealer_total):
    """Determines the outcome of the game based on the value of
    the cards received by the player and dealer. Outputs a
    message indicating whether the player wins or loses.

    :param player_total: total value of cards drawn by player
    :param dealer_total: total value of cards drawn by dealer
    :return: None
    """
```

With the player's and dealer's respective card totals calculated, the outcome of the game can now be determined. This outcome message is output by a function call to the `determine_outcome()` function. When the player's cards have a total greater than the dealer's total, display the message:

```
YOU WIN!

```

Otherwise, display the message

```
YOU LOSE!

```

Be sure to include a blank line after the outcome message.

Once the initial version of `determine_outcome()` has been implemented, try submitting your `blackjack.py` file to Gradescope to see if your implementation passes the **Step 5** tests. Once your program passes the **Step 5** tests, proceed to **Step 6**.

Step 6 - Running "Baby" Blackjack

Function: `play_blackjack()`

```
def play_blackjack():
    """Allows user to play Blackjack by making function calls for
    dealing cards to the player and the dealer as well as
    determining a game's outcome

    :return: None
    """
```

Now, use function calls to `deal_cards_to_player()`, `deal_cards_to_dealer()`, and `determine_outcome()` in the function definition of `play_blackjack()` to execute one game of *Baby* Blackjack. One game of *Baby* Blackjack includes:

1. displaying the welcome message (`Let's Play Blackjack!`) **followed by a blank line**
2. displaying the cards for the player and dealer and the total value of the cards
3. displaying a message indicating the game outcome when the player wins (`YOU WIN!`) or loses (`YOU LOSE!`)
4. displaying the message `Goodbye.` before the program completes

Satisfying the requirements of **Step 6**, signifies that you have successfully implemented *Baby* Blackjack. Submitting your code should result in all tests for **Step 1** through **Step 6** passing. Now, you should move on to the full program: CS1114 Blackjack.

CS1114 Blackjack

CS1114 Blackjack introduces the possibility that more than two cards may be dealt to the player or dealer.

Step 7 - Allow dealer to receive additional cards

Function to update: `deal_cards_to_dealer()`

Update the `deal_cards_to_dealer()` function to allow the dealer to receive more than 2 cards. The process for dealing cards to the dealer from *Baby* Blackjack can be used to deal the initial two cards. The dealer's total is once again displayed after the first two cards are dealt.

You must update the function so that the dealer continues to be dealt cards with each new card's value displayed along with the updated total. The dealer should stop receiving cards only when the dealer's total **exceeds** 16 (the `DEALER_THRESHOLD` constant variable references this value). In addition, the dealer's total should be returned by the function when the function call completes.

```
The dealer has 2 and 6.
Dealer's total is 8.

Dealer drew 3.
Dealer's total is 11.

Dealer drew 6.
Dealer's total is 17.
```

The output above shows an example of the dealer receiving two additional cards with the dealer's turn only ending when the total value of the dealer's cards exceeds `DEALER_THRESHOLD`. Note the blank line output after the final total message is displayed.

Once this version of `deal_cards_to_dealer()` has been implemented, try submitting your `blackjack.py` file to Gradescope to see if your implementation passes the **Step 7** tests. Once your program passes the **Step 7** tests, proceed to **Step 8**.

Step 8 - Allow player to receive additional cards

Function to update: `deal_cards_to_player()`

The function for dealing cards to the player should be updated for the implementation of CS1114 Blackjack. The changes to `deal_cards_to_player()` have some similarities to those of `deal_cards_to_dealer()`. The difference is that when dealing cards to the player, it is necessary to get user input regarding whether to have another card dealt (*hit*) or stop receiving cards (*stay*). Therefore, dealing two initial cards to the player still occurs. The player's total is still displayed after the two cards are dealt. And the player's total is returned by the function. You must update the function so that the player is prompted (`Hit (h) or Stay (s)?`) after the initial two cards are dealt. As long as the input to this prompt is `h`, the player receives a card, the card label is output, and the updated total is displayed.

```
Hit (h) or Stay (s)? h

Player drew 2.
Player's total is 16.
```

No additional cards should be dealt when the user inputs `s`

```
Hit (h) or Stay (s)? h

Player drew 3.
```

```
Player's total is 19.

Hit (h) or Stay (s)? s

The dealer has 2 and 6.
Dealer's total is 8.
```

or the player's total exceeds 20 (use constant variable `BLACKJACK` to avoid using magic numbers).

```
Hit (h) or Stay (s)? h

Player drew 5.
Player's total is 21.

The dealer has 4 and Q.
Dealer's total is 14.
```

Any other input should result in the re-display of the prompt for a hit or stay decision without a card being dealt.

```
Hit (h) or Stay (s)? e

Hit (h) or Stay (s)? b
```

Take note of the required blank lines being displayed after getting user input.

Once this version of `deal_cards_to_player()` has been implemented, try submitting your `blackjack.py` file to Gradescope to see if your implementation passes the **Step 8** tests. Once your program passes the **Step 8** tests, proceed to **Step 9**.

Step 9 - Update possible outcomes

Functions to update: `play_blackjack()` and `determine_outcome(player_total, dealer_total)`

Use the two updated functions that you implemented `deal_cards_to_player()` and `deal_cards_to_dealer()` to execute one game of CS1114 Blackjack in the function definition of `play_blackjack()`. One game of CS1114 Blackjack should include displaying the welcome message (`Let's Play Blackjack!`), dealing cards to the player and dealer, indicating whether the player wins (`YOU WIN!`) or loses (`YOU LOSE!`), and outputting `Goodbye.` at the completion of the program.

Be careful about what it means for the player to win.

- When the player's total is 21 (`BLACKJACK`), the dealer still receives cards **and** the player loses when the dealer holds cards totaling 21.
- When the player's total is over 21 (`BLACKJACK`), the player loses.
- Furthermore, the dealer should not be dealt any cards when the player busts (receives a card total greater than 21).
- When the dealer's total is over 21, the player wins (as long as the the player's total is 21 or under).

You may need to adjust the definition of your `determine_outcome()` function to enforce these rules.

Once these versions of `play_blackjack()` and `determine_outcome()` have been implemented, try submitting your `blackjack.py` file to Gradescope to see if your implementation passes the **Step 9** tests. Once your program passes the **Step 9** tests, proceed to **Step 10**.

Step 10 - Update program flow

Function to update: `play_blackjack()`

The full CS1114 Blackjack program allows the user to play multiple games in one program execution. After a game is complete (indicated by the display of a `YOU WIN!` or `YOU LOSE!` message), the user should be allowed to play again. The user will see a prompt

```
YOU LOSE!

Play again (Y/N)?
```

A new game (beginning with dealing cards to the player) of CS1114 Blackjack should be executed when the player inputs `Y`.

```
YOU LOSE!

Play again (Y/N)? Y

Player drew J and 5.
Player's total is 15.

Hit (h) or Stay (s)? h

Player drew 3.
Player's total is 18.

Hit (h) or Stay (s)? s

The dealer has 2 and 6.
Dealer's total is 8.

Dealer drew 9.
Dealer's total is 17.

YOU WIN!

Play again (Y/N)?
```

The program should end when the user enters `N`. Any other input should result in the prompt being displayed again until the user provides valid input (`Y` or `N`).

```
Play again (Y/N)? y

Play again (Y/N)? n

Play again (Y/N)? b

Play again (Y/N)? N

Goodbye.
```

After every game of CS 11114 Blackjack, the user should be prompted for input regarding their choice of playing again until the player indicates that the player no longer wants to play (user inputs `N`).

Take note that each prompt to play again is followed by a blank line after the user inputs a value.

At this point, submitting your `blackjack.py` file to Gradescope should result in passing all tests run by the Autograder. If not, address the failed tests. Failed tests may indicate that a change made to your program inadvertently affected previously passed tests.

EVALUATION

This assignment will use Gradescope's Autograder tool. Therefore, you will receive immediate feedback when submitting this assignment. You should strive to have your submitted code pass all of the tests run by the Gradescope Autograder by the time you have completed Step 10. When you have a program implementation that passes all of the Autograder tests, ensure that your program can be executed on its own (without producing any errors) when pressing the green run button in PyCharm. This is a requirement for getting full credit on the assignment.

Some docstrings for this assignment include doctests. These doctests are not enabled by default. If you would like to enable the doctests for testing the behavior of these functions on your local computer, you can do so by (carefully) editing the `if` statement at the bottom of the **blackjack.py** program file. Just uncomment the lines of code where instructed. Running the programs will then also run the doctests.

It is just as important to write clean and readable code as it is to write correct and functional code. A portion of your grade on this assignment will be based on how well you follow certain coding conventions. Make sure to follow the standards discussed in class, and before you submit your assignment, take a minute to review your code to check for stylistic issues like those mentioned below.

Comments

To make your program easier to read, add comments (docstrings) at the beginning of your function definitions to make your intention clearer. Good comments give the reader a clue about what a function does and, in some cases, how it works.

Be sure to acknowledge the honor pledge by replacing `<Your name>` with your name above the honor pledge at the top of your submission file.

Function design

Function names should include verbs indicating what is accomplished by the function. Most of the functions in this assignment have been named for you. However, if you add any additional functions to further decompose your solution, be sure to name these functions appropriately:

- function names should include verbs
- all words in a function name should be written in lower-case letters
- multiple words in a function name should be separated by underscores

Functions should be decomposed such that the length of each is small. This is an important part of proper decomposition. There is a bit of an art to this but you should ensure that your functions solve well-defined sub-problems. The function definitions should not, in general, have 10s of lines of codes. This is especially true for the programs written in this class.

Variable names

Variable naming conventions are as follows:

- names should be descriptive
- all characters in the name should be written in lowercase (with the exception of constant variables)
- multiple words should be separated by underscores ('_').

Spacing

Indentation should be consistent in the files that are submitted. The convention is that each block of indented code is indented using four spaces. Your program should follow this convention.

Add spaces between operators and operands in the expressions written in your code.

ASSIGNMENT SUBMISSION

Submit the following file on Gradescope (do not include any files not included in the list below):

- **blackjack.py**

The Autograder will assign points to passed tests. These results should be used as a guide for keeping you on track towards an implementation of this assignment that meets all of the requirements laid out above.

Good luck!