Card Games

There are no end of games based around playing cards. We are going to simulate a relatively straightforward one here, but hopefully lay the ground work for you to construct more sophisticated games of your own. I also intend to show you how to turn a player-based game into a bot simulation in order to test out different strategies.

Higher or Lower

I turn over cards from a shuffled deck (possible just a single suit), and after each card is revealed, you have to guess whether the next card will be 'higher' or 'lower'. If you're wrong, the game ends (and your score is the number of cards you saw). If you're right, another card is revealed.

Possible strategies:

There are a few obvious things you should do with a game like this, especially if you are playing with a reduced deck (note that the cards are drawn 'without replacement', so provided you're paying attention, there's no excuse for guessing wrong on the final card).

A few useful Python techniques:

Try the following code snippets and see if you can work out what they do. Write down the effect of each line beside it:

```
1   import random
2
3   deck = list(range(6))
4   original_deck = deck
5   print(deck)
6   random.shuffle(deck)
7   print(deck)
8
9   while len(deck) > 0:
10      next_card = deck.pop()
11      print(f"Card: {next_card} ({deck} remaining)")
12
13  print(original_deck)
```

The **random** module lets us shuffle lists (the original list has its elements reordered randomly). The built-in list method (function which can be applied to lists) **pop** lets us simultaneously read off the last item in a list and remove it (ideal for drawing a card from a deck, looking at it and then discarding it).

Did the last line confuse you? Surely **original_deck**, having been defined *before* shuffling took place, should preserve the original ordering. It turns out, when you assign a list to a new variable in Python, instead of creating a brand new separate object, Python simply creates a new 'pointer' to the same object. Change the underlying object (through reference to any of its pointers), and it will be changed.

If you play around, you'll find that it's not possible to **pop** elements from a **range** object in the same way as we can from a **list**. That's because **range** is a *generator*, which essentially means Python creates the 'next' value on the fly every time. Much more memory efficient to have, say **range(1000000)** than a list with a million values stored in it.

Coding a card game

The game loop

Pretty much any computer game, no matter how simple or complex, runs on the idea of a loop: do certain things, wait for user input, repeat until the game ends. Consider this pseudo-code version of Os and Xs:

```
While there are still spaces to fill, and no-one's won,
    If it's your turn,
        choose a space to mark X.
    Otherwise,
        wait for your opponent to fill in a space.
If there are three Xs in a row,
    you win.
Otherwise, if there are three Os in a row,
    you lose.
Otherwise,
    it's a draw.
```

We need a game loop for our own card game, Higher or Lower. Write a pseudo-code version of what needs to happen. Then try to generate for yourself some python code.

The Basic Structure

There are certain things that need to take place, although precisely how these happen is pretty much up to you. The user needs to be presented with the relevant information at each point, and you need to keep track of key things. See if you can complete the code:

```
import random
deck = # make a list called deck which contains the numbers from 0 to 13, then use the
random.shuffle method to mix them up.
card = # use the pop() method to get a card
score = 1 # use this to keep track of how many cards the player has seen (even if they
guess wrong at first, they've seen 1 card)
done = False
while not done:
  guess = # use the input method to get a guess from the user (eg ask for H or L)
  new_card = # pop out the next card, for comparison
  # do if / else type tests to determine whether the player guessed correctly.  You'll
need to compare card and new_card
  # make sure you tell the player if they were right or wrong using the print function.
Don't forget to tell them what the card was, too - they'll need that for their next
guess!
  # if the player guesses wrong at any point, you can set done to be True.
  # if the deck runs out, we can also be done.
  # if the player guesses correctly, increase their score by 1.
# at the end (after the loop has terminated), tell the player their final score

# optional: Before each player guess, show on screen which cards remain.  Something like:
# A 2 3 _ 5 6 _ _ | 10 J _ K
# if they've already seen the 4, 7, 8 and Queen, and have just drawn the 9.
```

Wanna bet?

A good next step is to incorporate betting into the program. Sometimes you're certain of the next result, sometimes 80% sure, and sometimes it really is an even chance. If you had the chance to bet money on each turn of a card, how much do you think you could make with the optimal strategy? First, we'll make a game that lets you bet money rather than simply guessing H or L. Then we'll make a bot to play it, using whatever strategy you favour, and see how it copes when we make it play a few thousand games.

Higher or Lower with betting

```
1   import random
2
3   names = ["Ace", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine",
    "Ten", "Jack", "Queen", "King"]
4
5   while True:
6     deck = [i for i in range(13)] # 13 cards (a single suit)
7     random.shuffle(deck) # deck is shuffled
8     discard = [] # a discard pile is created
9     card = deck.pop() # the first card is removed from the deck
10
11    print(f"Welcome to Higher or Lower! Aces are low.")
12    cash = 100
13    print(f"You start with £{cash}, and win or lose your stake each time you make
      a guess")
14    done = False # set up so we can break out of the loop for various reasons
15    while not done: # loop continues while game is being played
16      stake = int(input(f"Your card is: ___{names[card]}___.  Enter stake (+ve for
        Higher, -ve for Lower): £")) # get amount being risked
17      if stake > 0:
18        stake = min(cash, stake) # ensure stake doesn't exceed available cash
19      else:
20        stake = max(-cash, stake) # (and if the stake is negative, the same)
21      discard.append(card) # place previous card in the discard pile
22      card = deck.pop() # draw a new card
23      if card > discard[-1]: # compare latest card to most recent discarded card
24        cash += stake # gain stake if +ve, or lose if -ve
25        if stake > 0:
26          print(f"You made £{stake}: you have £{cash} altogether.")
27        else:
28          print(f"You lost £{stake}: you have £{cash} now.") # inform player
29      elif card < discard[-1]:
30        cash -= stake # gain stake if -ve, or lose if +ve
31        if stake < 0:
32          print(f"You made £{-stake}: you have £{cash} altogether.")
33        else:
34          print(f"You lost £{-stake}: you have £{cash} now.") # inform player
35      if len(deck) == 0 or cash <= 0: # if the deck is exhausted or the player
        bankrupted, quit the game
36        done = True
37    input(f"Game over!  You made {cash} altogether. Again? ")
```

Do your best to make sense of as much of this as possible before you run it. Try to trace the flow of the program from start to finish in your head. https://replit.com/@thechalkface2/higherorlower#main.py

Testing your strategy

It's hard to get a good sense of what works and what doesn't from just a handful of runs. A good way to properly compare strategies is to code a bot to play the game for you.

I've left minimised two functions from this program – the **stats** function you can choose to give you whatever particular statistics you want out of the data, so its precise details are not so important. The **bot_decide** function determines the strategy your bot should use – do you only bet when it's a sure-fire thing? Do you go all-in on anything about 80%? Do you always save half of your money? It's up to you. See if you can beat of 10,000 and median of 5,000 with your own strategy – that's about as good as mine can manage.

```
1   import random
2
3   names = ["Ace", "Two", "Three", "Four", "Five", "Six",
    "Seven", "Eight", "Nine", "Ten", "Jack", "Queen",
    "King"]
4
5   def bot_decide(latest_card, discard_pile, cash): ···
12
13  def stats(data_list): ···
24
25  while True:
26      num_sims = int(input("How many simulations do you want to run? "))
27      results = []
28      for i in range(num_sims):
29          deck = [i for i in range(13)]
30          random.shuffle(deck)
31          discard = []
32          card = deck.pop()
33          cash = 100
34          done = False
35          while not done:
36              stake = bot_decide(card, discard, cash)
37              discard.append(card)
38              card = deck.pop()
39              if card > discard[-1]:
40                  cash += stake
41              elif card < discard[-1]:
42                  cash -= stake
43              if len(deck) == 0 or cash <= 0:
44                  done = True
45          results.append(cash)
46
47      mean, median, minimum, maximum = stats(results)
48      print(f"Bot achieved a mean of {mean} and median {median} on the {num_sims}
        simulations.  Results ranged from {minimum} to {maximum}")
```

In some ways, this program is more elegant than the original – it doesn't have to parse imperfect user input, and constantly send statements to print to tell the user what's going on. The bot gets information about which cards have been seen, what the latest card was, and how much cash they have on hand, and made a decision based on that. It's also a lot quicker to run than if you played against a human, of course – it'll do 100,000 games in under 10 seconds, so you can get a fair degree of confidence from the spread of results.