Card shuffling

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming, trimester 1, 2019

重新排列

```
In [2]: from random import randrange
        import matplotlib.pyplot as plt
        from sys import getsizeof
```

The program `card_shuffling.py` creates a widget to simulate the shuffling of a 56-card deck which to start with, is supposed to be brand new, with from top to bottom, all Hearts from Ace up to King, then all Clubs from Ace up to King, then all Diamonds from King down to Ace, and finally all Spades from King down to Ace. The Unicode character set has pictures of these cards, but in the order from Ace up to King for all colours, and with gaps between colours:

py程序card_shuffling.py创建一个小部件来模拟56张牌组的洗牌
一开始，应该是全新的，从上到下，所有的红桃从a到k，然后所有梅花从a到k，然后所有方块从k到a，最后所有黑桃从k到a。Unicode字符集有这些卡片的图片，但是从Ace到King的顺序是所有颜色的，并且颜色之间有间隔

- from code point 0x1F0A1 to code point 0x1F0AE, all Spades;
- from code point 0x1F0B1 to code point 0x1F0BE, all Hearts;
- from code point 0x1F0C1 to code point 0x1F0CE, all Diamonds;
- from code point 0x1F0D1 to code point 0x1F0DE, all Clubs.

Let us keep track of

颜色在Unicode字符集中出现的顺序，
第一个a的编码点，黑桃a，和
将偏移量添加到给定颜色的a的码点，以获得该颜色的所有卡片

- the colours in the order in which they appear in the Unicode character set,
- the code point of the first Ace, the Ace of Spades, and
- the offsets to add to the code point of an Ace of a given colour to get all cards of that colour, from Ace up to King:

```
In [3]: colours = 'spades', 'hearts', 'diamonds', 'clubs'
        first_ace_code_point = 0x1F0A1
        suit = range(14)
```

对于红心和梅花，抵消的顺序需要颠倒，以产生这些颜色的卡片在一个全新的牌组中的顺序;我们定义一个函数，对于给定的颜色，得到正确的顺序

For Hearts and Clubs, the order of offsets needs to be reversed to yield cards of those colours in the order they have in a brand new deck; we define a function to, for a given colour, get the right order:

```
In [4]: def default_suit_order(colour):
            return suit if colour in {'hearts', 'clubs'} else reversed(suit)
```

We can now define a deck as a list `deck` that stores the characters that represent all cards in the same order as in a brand new deck of cards. We use a list comprehension with two `for` statements. The syntax of (list and other forms of) comprehension allows for an arbitrary number of `for` statements, that can be associated with `if` statements, respecting the order they would have in the corresponding "classical" syntax. For instance, both code fragments build the same list:

现在我们可以将牌组定义为一个列表牌组，它存储表示所有牌的字符就像在一副全新的纸牌上下单一样。我们使用带有两个for语句的列表理解。的语法 of (list和其他形式)理解允许任意数量的for语句，它们可以与if语句相关联，并遵循它们在相应的“经典”语法中的顺序。例如，两个代码片段构建相同的列表：

```
In [5]: L = []
        for i in 'abcd':
            if i in 'ac':
                for j in '123':
                    for k in 'ABCD':
                        if k in 'BD':
                            L.append((i, j, k))
        L

Out[5]: [('a', '1', 'B'),
         ('a', '1', 'D'),
         ('a', '2', 'B'),
         ('a', '2', 'D'),
         ('a', '3', 'B'),
         ('a', '3', 'D'),
         ('c', '1', 'B'),
         ('c', '1', 'D'),
         ('c', '2', 'B'),
         ('c', '2', 'D'),
         ('c', '3', 'B'),
         ('c', '3', 'D')]

In [6]: L = [(i, j, k) for i in 'abcd' if i in 'ac'
                       for j in '123'
                           for k in 'ABCD' if k in 'BD'
            ]
        L

Out[6]: [('a', '1', 'B'),
         ('a', '1', 'D'),
         ('a', '2', 'B'),
         ('a', '2', 'D'),
         ('a', '3', 'B'),
         ('a', '3', 'D'),
         ('c', '1', 'B'),
         ('c', '1', 'D'),
         ('c', '2', 'B'),
         ('c', '2', 'D'),
         ('c', '3', 'B'),
         ('c', '3', 'D')]
```

在一副全新的牌组中，颜色以[1]、[3]、[2]和的顺序出现
颜色[0]。另外，两个连续的ace代码点之间的差值为16。因此，我们可以如下定义deck

In a brand new deck of cards, colours appear in the order `colours[1]`, `colours[3]`, `colours[2]` and `colours[0]`. Also, the difference between the code points of two consecutive Aces is 16. Hence we can define `deck` as follows:

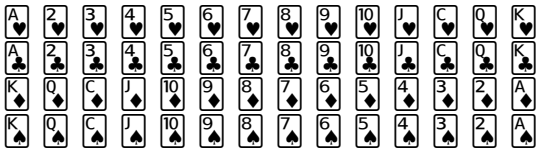
```
In [7]: deck = [chr(first_ace_code_point + i * 16 + j) for i in (1, 3, 2, 0)
                           for j in default_suit_order(colours[i])
                ]
```

让我们定义一个简单的函数，将牌组显示在4行以上，因此，当牌组表示尚未洗牌的全新牌组时，每种颜色对应一行

Let us define a simple function to display the deck over 4 lines, so with one line for each colour when `deck` represents a brand new deck of cards that has not been shuffled yet:

```
In [8]: def display_deck():
            print('\n'.join(' '.join(deck[i * 14 + j] for j in range(14))
                                                        for i in range(4)
                            )
                  )

        display_deck()
```



洗牌的第一步是把牌切下来，得到两叠牌，一叠和二叠。直观地，切牌将牌堆分成相等的两堆的几率大吗，切牌将牌堆分成相等的两堆的几率小吗。比另一个大;两者大小的差值越小，这种可能性就越大堆栈增加。两个堆栈都扮演对称的角色。我们假设牌面朝下索引0在牌堆底部，索引55的牌在牌堆顶部。由于二项分布的关系，切割甲板是很正式的。可以想象一枚均匀硬币抛56次，如果它以正面，正面，反面，正面，反面，反面，反面开始

The first step in shuffling the deck is to cut it and get two stacks, stack 1 and stack 2. Intuitively, there is a high chance that the cut divides the deck in two equal stacks, and a smaller chance that one stack is larger than the other; the chance is all the more smaller that the difference between the sizes of both stacks increases. Both stacks play a symmetric role. We assume that the cards face down, so the card of index 0 is at the bottom of the deck and the card of index 55 at the top of the deck. Cutting the deck is well formalised thanks to a binomial distribution. On can imagine 56 flips of a fair coin, that if it starts with Heads, Heads, Tail, Heads, Tail, Tail... can be interpreted as:

堆

- Card of index 0 bottommost card of stack 1
- Card of index 1 second bottommost card of stack 1
- Card of index 55 topmost card of stack 2
- Card of index 2 third bottommost card of stack 1
- Card of index 54 second topmost card of stack 2
- Card of index 53 third topmost card of stack 2
- ...

The number of Heads determines the size of stack 1. It can also be interpreted as the index of the card at the bottom of stack 2:

- equal to 0 in one of the two most unlikely cases, where stack 1 is empty and stack 2 is the whole deck,
- equal to 56 in the other most unlikely case, where stack 1 is the whole deck and stack 2 is empty,
- equal to 28 in the most likely case, where both stacks have the same size.

The code that follows implements the generation of a value for the variable `cut` in accordance with that random process:

```
In [9]: nb_of_cards = 56
        #  57 possible outcomes
        cut = sum(randrange(2) for _ in range(nb_of_cards))
        cut

Out[9]: 29
```

After the deck has been cut in two stacks, the cards of both stacks should be entwined to finish shuffling. The process is again probabilistic. If both stacks have the same number of cards, then there is an equal chance that either the card at the bottom of stack 1 or the card at the bottom of stack 2 becomes the card at the bottom of the shuffled deck. The larger one stack is compared to the other stack, the higher the chance that the bottom card in the shuffled deck comes from the former. If there are p cards in stack 1 and q cards in stack 2, a good formalisation gives a $\frac{p}{p+q}$ probability for the bottom card in the shuffled deck to come from the bottom of stack 1. To start with, p is equal to `cut` and q is equal to `nb_of_cards - cut`. The same reasoning applies to determine which card becomes the second bottommost card in the shuffled deck, which decreases either p or q by 1, and which card becomes the third bottommost card in the shuffled deck, which decreases either p or q by 1… until one of both stacks becomes empty – stack 1 if p becomes equal to 0 first, stack 2 if q becomes equal to 0 first. What remains of the other stack gets to the top of the shuffled deck.

To select the card at the bottommost of stack 1 with a probability of $\frac{p}{p+q}$, it suffices to randomly generate a natural number smaller than $p + q$, and test whether it is smaller than p:

```
In [10]: # 10,000 times, there are 2/5 chances for the test to be true,
         # so we expect the second number to be a reasonable approximation
         # of the first one.
         2/5, sum(randrange(5) < 2 for _ in range(10_000)) / 10_000
         # 10,000 times, there are 3/8 chances for the test to be true,
         # so we expect the second number to be a reasonable approximation
         # of the first one.
         3/8, sum(randrange(8) < 3 for _ in range(10_000)) / 10_000
```

```
Out[10]: (0.4, 0.4059)
```

```
Out[10]: (0.375, 0.3691)
```

Let us define a number of variables:

combined_size_of_stacks，初始化为nb_of_cards，用于跟踪两个堆栈中剩余的卡量

- `combined_size_of_stacks`, initialised to `nb_of_cards`, to keep track of how many cards are left in both stacks;
- `i_1`, initialised to `0`, to keep track of the position in stack 1 of the first card that is not part of the shuffled deck yet. i_1，初始化为0，以跟踪第一张牌的筹码1中的位置，该筹码不属于洗牌牌组的一部分
- `i_2`, initialised to `cut`, to keep track of the position in stack 2 of the first card that is not part of the shuffled deck yet. i_2，初始化切换，以跟踪第一张牌的筹码2中的位置，该筹码不属于洗牌牌组的一部分。

在任何阶段，要么i_1增加1，因为堆栈1剩余部分底部的卡片成为改组牌组的一部分，或者i_2增加1，因为它取而代之的是堆栈剩余部分的卡片成为洗牌的一部分。 由于cut -i_1评估堆栈1的当前大小，并且nb_of_card -i_2计算到堆栈2的当前大小，我们可以不用combined_size_f_stacks，而不是cut - i_1 nb_of_cards i_2

At any stage, either `i_1` gets increased by 1 because the card at the bottom of what remains of stack 1 becomes part of the shuffled deck, or `i_2` gets increased by 1 because it is instead the card at the bottom of what remains of stack 2 that becomes part of the shuffled deck. Since `cut - i_1` evaluates to the current size of stack 1, and `nb_of_cards - i_2` to the current size of stack 2, we could do without `combined_size_of_stacks` and work instead with `cut - i_1 + nb_of_cards - i_2`.

Let us first implement this method to the point where one of the stacks becomes empty and trace execution of the code, but with a small deck of only 7 cards, abstracted as the natural numbers 0, 1, 2, 3, 4, 5, 6 rather than being characters that depict cards, and fixing the value of `cut` to 4; so stack 1 consists of cards 0, 1, 2 and 3, while stack 2 consists of cards 4, 5 and 6. When execution terminates, either stack 1 is empty, in which case card 3 (facing down) is at the top of the shuffled deck, or stack 2 is empty, in which case card 6 (facing down) is at the top of the shuffled deck.

让我们首先实现这个方法，其中一个堆栈变为空并跟踪代码的执行，但是只有7个卡片的小牌，抽象为自然数0,1,2,3,4,5， 6而不是描绘卡片的人物，并将切割值固定为4；堆栈1由卡片0,1,2和3组成，而堆栈2由卡片4,5和6组成。当执行终止时，堆栈1为空，在这种情况下，卡片3（面朝下）位于顶部 洗牌的甲板或堆2是空的，在这种情况下，卡6（朝下）位于洗牌的甲板顶部

```
In [11]: nb_of_cards = 7
         deck = range(nb_of_cards)
         cut = 4
         shuffled_deck = []
         i_1 = 0
         i_2 = cut
         combined_size_of_stacks = nb_of_cards

         while i_1 < cut and i_2 < nb_of_cards:
             if randrange(combined_size_of_stacks) < cut - i_1:
                 shuffled_deck.append(deck[i_1])
                 i_1 += 1
                 print('Card coming from stack 1, shuffled deck becomes',
                       shuffled_deck
                      )
             else:
                 shuffled_deck.append(deck[i_2])
                 i_2 += 1
                 print('Card coming from stack 2, shuffled deck becomes',
                       shuffled_deck
                      )
             combined_size_of_stacks -= 1
```

```
Card coming from stack 1, shuffled deck becomes [0]
Card coming from stack 1, shuffled deck becomes [0, 1]
Card coming from stack 2, shuffled deck becomes [0, 1, 4]
Card coming from stack 2, shuffled deck becomes [0, 1, 4, 5]
Card coming from stack 1, shuffled deck becomes [0, 1, 4, 5, 2]
Card coming from stack 2, shuffled deck becomes [0, 1, 4, 5, 2, 6]
```

To bring to the top of the shuffled deck the rest of the nonempty stack, we can use the `extend()` method of the `list` class and use `slice` notation. Whereas `append()` adds only element to a list, `extend()` concatenates a sequence to the end of a list:

为了将非空堆栈的其余部分置于混洗层的顶部，我们可以使用list类的extend（）方法并使用切片表示法。 而append（）只将一个元素添加到列表中，而extend（）将序列连接到列表的末尾：

```
In [12]: L = [10, 11, 12]
         L.extend([]); L
         L.extend([0]); L
         L.extend([1, 2]); L
         L.extend((2, 3, 4)); L
         L.extend(range(5, 10)); L
```

```
Out[12]: [10, 11, 12]
```

```
Out[12]: [10, 11, 12, 0]
```

```
Out[12]: [10, 11, 12, 0, 1, 2]
```

```
Out[12]: [10, 11, 12, 0, 1, 2, 2, 3, 4]
```

```
Out[12]: [10, 11, 12, 0, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9]
```

Let us examine slice notation in detail. To get a copy of a selection of consecutive elements in a list, or a tuple, L, one optionally provides 让我们详细检查切片表示法。 获取列表中选定的连续元素的副本，或一个元组L，一个可选地提供

- the index of the first element to include, and
- the index beyond that of the last element e to include, which is therefore "invalid" in case e is L's last element:

```
In [13]: L = [10, 11, 12, 13, 14, 15, 16, 17, 18]
```

另外

```
         # Alternatively: L[0: x] x with x >= len(L)
         L[: ]
         # Alternatively: L[2: x] x with x >= len(L)
         L[2: ]
         # Alternatively: L[-2: x] x with x >= len(L)
         L[-2: ]
         # Alternatively: L[x: 4] with x == 0 or x < -len(L)
         L[: 4]
         # Alternatively: L[x: -4] with x == 0 or x < -len(L)
         L[: -4]
         L[60: -30]
         L[3: 6]
         L[-7: -1]
```

```
Out[13]: [10, 11, 12, 13, 14, 15, 16, 17, 18]

Out[13]: [12, 13, 14, 15, 16, 17, 18]

Out[13]: [17, 18]

Out[13]: [10, 11, 12, 13]

Out[13]: [10, 11, 12, 13, 14]

Out[13]: []

Out[13]: [13, 14, 15]

Out[13]: [12, 13, 14, 15, 16, 17]
```

With a second colon, we can optionally provide a step to jump over a given number of elements:

```
In [14]: # The step is 1 by default
         L[: : ]
         # Equivalent to [-1: -1 - len(L): -1]
         L[: : -1]
         L[: : 2]
         L[: : 3]
         L[: 7: 2]
         L[1: : 3]
         L[: 2: -2]
         L[-2: : -3]
```

```
Out[14]: [10, 11, 12, 13, 14, 15, 16, 17, 18]

Out[14]: [18, 17, 16, 15, 14, 13, 12, 11, 10]

Out[14]: [10, 12, 14, 16, 18]

Out[14]: [10, 13, 16]

Out[14]: [10, 12, 14, 16]

Out[14]: [11, 14, 17]

Out[14]: [17, 14]
```

切片实际上可以自己定义，然后“ 应用 ”到列表或元组

Slices can actually be defined on their own and then be “applied” to a list or a tuple:

```
In [15]: s = slice(8); s.start, s.stop, s.step
         L[s]
         s = slice(2, 8); s.start, s.stop, s.step
         L[s]
         s = slice(2, 8, 2); s.start, s.stop, s.step
         L[s]

Out[15]: (None, 8, None)

Out[15]: [10, 11, 12, 13, 14, 15, 16, 17]

Out[15]: (2, 8, None)

Out[15]: [12, 13, 14, 15, 16, 17]

Out[15]: (2, 8, 2)

Out[15]: [12, 14, 16]
```

Getting back to our card simulation, let us complete the shuffling of the small deck of numbers between 0 and 7 initially cut at position 4:

```
In [16]: print('The shuffled deck is now', shuffled_deck)
         if i_1 < cut:
             shuffled_deck.extend(deck[i_1: cut])
         else:
             shuffled_deck.extend(deck[i_2: ])
         print('Finally, the shuffled deck is', shuffled_deck)

The shuffled deck is now [0, 1, 4, 5, 2, 6]
Finally, the shuffled deck is [0, 1, 4, 5, 2, 6, 3]
```

Eventually assigning `shuffled_deck` to `deck` in effect shuffles the deck, and allows one to shuffle it once again, and once again (every time with a possibly different cut). Theory establishes that after 7 shuffles, the ordering of the cards in the deck has become random, in a precise sense. Back to our list of 56 card representations:

最终将shuffled_deck分配给甲板实际上是对甲板进行洗牌，并且允许再次将其再次洗牌，并且再次（每次都有可能不同的切割）。 理论确定，在7次洗牌之后，牌组中牌的排序在精确意义上变得随机。 返回我们的56张卡表示列表：

```
In [17]: nb_of_cards = 56
         deck = [chr(first_ace_code_point + i * 16 + j) for i in (1, 3, 2, 0)
                                for j in default_suit_order(colours[i])
                 ]
         nb_of_shuffles = 7
         print('Deck before shuffling:')
         display_deck()
         for k in range(nb_of_shuffles):
             shuffled_deck = []
             cut = sum(randrange(2) for _ in range(nb_of_cards))
             i_1 = 0
             i_2 = cut
             combined_size_of_stacks = nb_of_cards
             while i_1 < cut and i_2 < nb_of_cards:
                 if randrange(combined_size_of_stacks) < cut - i_1:
                     shuffled_deck.append(deck[i_1])
                     i_1 += 1
                 else:
                     shuffled_deck.append(deck[i_2])
                     i_2 += 1
                 combined_size_of_stacks -= 1
             if i_1 < cut:
                 shuffled_deck.extend(deck[i_1: cut])
             else:
                 shuffled_deck.extend(deck[i_2: ])
             deck = shuffled_deck
             print(f'Shuffle {k + 1}, deck is now:')
             display_deck()
```

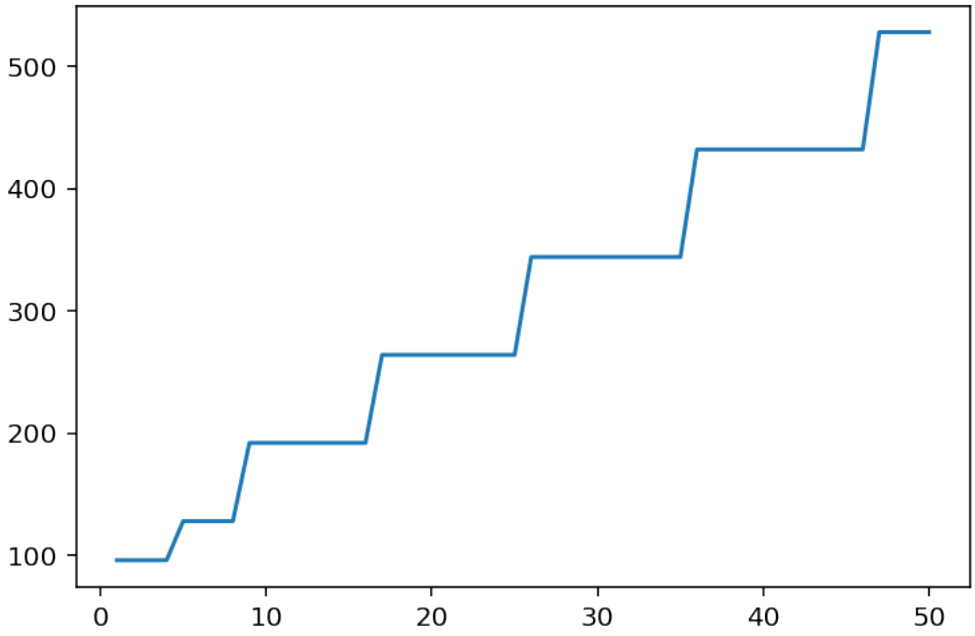
```
Deck before shuffling:
🂱 🂲 🂳 🂴 🂵 🂶 🂷 🂸 🂹 🂺 🂻 🂼 🂽 🂾
🃑 🃒 🃓 🃔 🃕 🃖 🃗 🃘 🃙 🃚 🃛 🃜 🃝 🃞
🃎 🃍 🃌 🃋 🃊 🃉 🃈 🃇 🃆 🃅 🃄 🃃 🃂 🃁
🂮 🂭 🂬 🂫 🂪 🂩 🂨 🂧 🂦 🂥 🂤 🂣 🂢 🂡
Shuffle 1, deck is now:
🂱 🃘 🃙 🂲 🂳 🂴 🃚 🃛 🃜 🃝 🂵 🂶 🃞 🂷
🃎 🃍 🃌 🂸 🂹 🂺 🃋 🃊 🃉 🃈 🂻 🃇 🃆 🃅
🂼 🂽 🂾 🃄 🃑 🃃 🃒 🃂 🃓 🃁 🃔 🂮 🂭 🂬
🃕 🂫 🂪 🂩 🂨 🂧 🂦 🂥 🂤 🂣 🂢 🃖 🂡 🃗
Shuffle 2, deck is now:
🂱 🃘 🃙 🃅 🂲 🂳 🂼 🂴 🂽 🂾 🃚 🃄 🃛 🃑
🃃 🃜 🃒 🃂 🃓 🃝 🂵 🂶 🃁 🃞 🃔 🂷 🂮 🂭
🂬 🃎 🃍 🃌 🂸 🃕 🂫 🂪 🂩 🂹 🂨 🂧 🂺 🂦
🂥 🃋 🃊 🂤 🂣 🃉 🂢 🃈 🃖 🂻 🃇 🂡 🃆 🃗
Shuffle 3, deck is now:
🂱 🃕 🃘 🃙 🂫 🂪 🂩 🃅 🂹 🂨 🂲 🂳 🂧 🂺
🂼 🂴 🂽 🂦 🂥 🂾 🃋 🃚 🃊 🃄 🂤 🃛 🃑 🂣
🃃 🃜 🃒 🃉 🃂 🃓 🂢 🃝 🃈 🂵 🃖 🂶 🂻 🃁
```

```
🃗 🃞 🃔 🃗 🃞 🂡 🃖 🂠 🃃 🃞 🃗 🂠 🃃 🃘
Shuffle 4, deck is now:
🂡 🃓 🃓 🃃 🂡 🃕 🂲 🃈 🃙 🃊 🃙 🂲 🃓 🃚
🃙 🂲 🃕 🂠 🃈 🃙 🃕 🃈 🃖 🂲 🃓 🃖 🃊 🂡
🃗 🃚 🃗 🃞 🃃 🃔 🃔 🂠 🃗 🃞 🃖 🃕 🂡 🃖
🃞 🂠 🃃 🃊 🃞 🃗 🂠 🃃 🃚 🃚 🃔 🃔 🃈 🃊
Shuffle 5, deck is now:
🂲 🃓 🃖 🂡 🃓 🃊 🃓 🂡 🃃 🃗 🃚 🃗 🃞 🂡
🃃 🃕 🃔 🃔 🂠 🃗 🃞 🂲 🃖 🃈 🃕 🂡 🃙 🃊
🃖 🃞 🃙 🂲 🂠 🃃 🃊 🃞 🃗 🃓 🃚 🂠 🃙 🂲
🃃 🃕 🂠 🃚 🃚 🃔 🃔 🃈 🃈 🃙 🃕 🃈 🃖 🃊
Shuffle 6, deck is now:
🃊 🂲 🃓 🃖 🃖 🂡 🃞 🃙 🃓 🂲 🃊 🃓 🂡 🃃
🂠 🃃 🃊 🃗 🃚 🃞 🃗 🃞 🃗 🂡 🃓 🃚 🂲 🃕
🃔 🂠 🃔 🂠 🃙 🂲 🃗 🃞 🃃 🃕 🂠 🃚 🃚 🂲
🃔 🃔 🃈 🃈 🃖 🃙 🃈 🃕 🃕 🂡 🃙 🃈 🃖 🃊
Shuffle 7, deck is now:
🃙 🂲 🃊 🂲 🃓 🃗 🃞 🃖 🃃 🃖 🃕 🂡 🃞 🃙
🃓 🂲 🃊 🂠 🃓 🃚 🂡 🃃 🃚 🂠 🃃 🃊 🃗 🂲
🃚 🃞 🃗 🃔 🃞 🃔 🃈 🃈 🃗 🃖 🃙 🂡 🃈 🃕
🃓 🃕 🃚 🂡 🃙 🃃 🃈 🃖 🃊 🃕 🃔 🂠 🃔 🂠
```

`shuffled_deck` starts as an empty list, which a new element is appended to 56 times. An alternative would have been to initialise `shuffled_deck` to `[None] * nb_of_cards`, introduce a new variable `j` initialised to `0`, and change the statements `shuffled_deck.append(deck[i])` with `i_1` or `i_2` for `i` to the two statements `shuffled_deck[j] = deck[i]; j += 1`. Creating an empty list and appending an element to this list n times is not obviously more efficient than creating a list of size n with a default value and changing each of its n elements to the desired value. Indeed, the size of a list L is not the same as its length: a list can possibly accomodate more elements than it currently "officially" has. When L is full, that is, when the size and the length of L are the same, `append()` does not necessarily just provide space for one extra element, but possibly space for many elements, and all the more space that the list is longer. So when appending elements again and again, times when the list is full and needs to be expanded to accomodate more elements come less and less frequently, in a way that the **amortised cost** of `append()` is constant: when the list is full and needs to be expanded and possibly copied over, the cost can be high, but the average cost over a large number of `append()` operations is constant. We can know the size of a list thanks to the function `getsizeof()` from the `sys` module. The following plot illustrates how the size of a list L remains constant and suddenly jumps as elements are appended to L one by one:

```
In [18]: data = []
         L = []
         for i in range(1, 51):
             L.append(None)
             data.append((i, getsizeof(L)))
         plt.plot(*zip(*data));
```

shuffled_deck以空列表开头，将新元素追加到56次。另一种方法是将shuffled_deck初始化为[None] * nb_of_cards，引入一个新的变量j初始化为0，并使用i_1或i_2更改语句shuffled_deck.append（deck [i]）i to
两个语句shuffled_deck [j] = deck [i]; j + = 1.创建一个空列表并将-元素附加到该列表n次显然不比使用默认值创建大小为n的列表并将其n个元素中的每一个更改为所需值更有效。实际上，列表L的大小与其长度不同：列表可能容纳比当前“ 正式 ”更多的元素。当L已满时，也就是说，当L的大小和长度相同时，append （） 不一定只为一个额外元素提供空间，但可能为多个元素提供空间，并且列表所需的空间更多更长的时间。因此，当一次又一次地附加元素时，列表已满并需要扩展到的时间
容纳更多元素的次数越来越少，因为append （） 的摊销成本是不变的：当列表已满且需要扩展并可能被复制时，成本可能很高，但平均成本高于大量的append （） 操作是不变的。由于sys模块中的函数getsizeof （），我们可以知道列表的大小。下图说明了列表L的大小如何保持不变并突然跳跃，因为元素逐个附加到L



Contrast the previous plot with the following, when we create a list of a given length; then we get exactly the size we are asking for, possibly plus some fixed extra space:

```
In [18]: data = []
         for i in range(1, 51):
             data.append((i, getsizeof([None] * i)))
         plt.plot(*zip(*data));
```