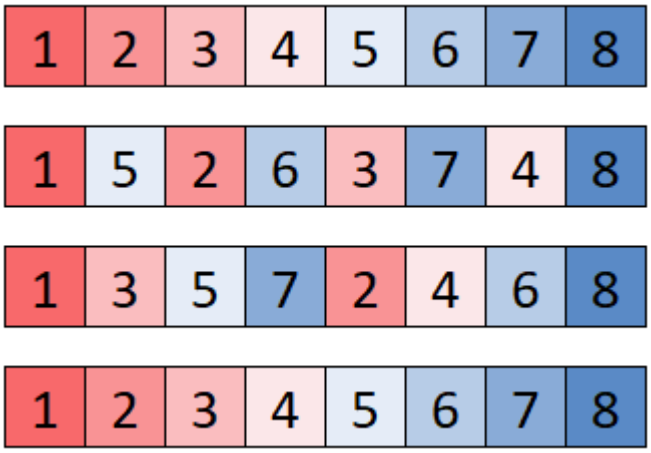
Shuffling Cards

Ensuring a deck of cards is in a sufficiently randomized order is important for lots of games. The Riffle Shuffle (AKA Faro, or Dovetail) is a popular method for shuffling involving interleaving alternate cards.

A riffle shuffle follows the same basic pattern, but is often performed more than once:

- ***Cut the deck (split it into two equal halves).***
- ***Merge them, alternating which half a card is put down from each time.***

If cards numbered 1 to 8 were shuffled repeatedly using this method, we would get:



What do you notice about the sequence after each successive shuffle?

If the process of shuffling can be described with enough precision, we can automate it, and make the computer do it for us. Try to write a sequence of instructions that would be specific enough for a programmer to convert into code.

Note: this process is sometimes referred to as writing 'Pseudo-English'. It is not specific to any one programming language, but has enough precision that someone fluent in any suitable coding language could readily convert it into a functional computer program.

Stuck? Imagine performing the task yourself. What would you do first? Then what? What would you have to show for it at the end? Break down the steps until they are small enough to form a precise instruction.

Turn over to see how to code it up …

Shuffling Cards *with Python*

Shuffling cards is a process that lends itself well to investigation with computers, because a deck of cards can be thought of as a list of elements in a certain order. The **list** object in Python is a data structure designed to hold elements in a specific order, with a host of methods for manipulating the list and the elements within.

Your Toolkit: This is a summary of a few useful tools you have at your disposal when using Python to rearrange lists. It is up to you to see how to combine these into functional code.

Lists	Python can store pretty much anything in a list, including a mixture of data types (such as `my_list = [2, "a", True]`), other lists (such as `my_lists = [my_list, list2, [1, 2, 3], "hi"]`) or even nothing at all (the empty list `results = []`).
Indexing	The elements in a list are indexed (numbered) from 0 to $n-1$, where n is the length of your list. For instance, if we define a list of factors of 6: `factors = [1, 2, 3, 6]` then `factors[0]` gives `1`, and `factors[3]` gives `6`. We can also *find* elements within a list using the **index** method: `factors.index(2)` gives `3`, because the element **3** appears in position **4** in the list.
Splicing	As well as extracting a single element, we can use a slightly different notation to extract a sub-list. Continuing with the example from above: `factors[1:3]` gives the sub-list `[2, 3]`. This notation gives a sub-list which goes from element 1 *up to but not including* element 3.
Add/Remove	The **append** method adds new elements to the end of a list: `factors.append(12)` gives `[1, 2, 3, 6, 12]`. The **insert** method adds new elements in a specific position: `factors.insert(4, 3)` gives `[1, 2, 3, 4, 6, 12]`. The **pop** method both *removes* an element and *provides it* as an output: `factors.pop(0)` returns the element `1` and leaves the list looking like this: `[2, 3, 4, 6, 12]`

Pseudo-English

This is a brief description of what I want the program to do, written in no specific programming language.

1. Make a list of values in order (say, 0 to 51).
2. Split the list into two separate halves (0 to 25, and 26 to 51). Call them **top** and **bottom**.
3. Create a new, empty, list into which I can add the cards. Call it **shuffled**.
3. 'Pop' the first card from **top** and 'append' it to **shuffled**. Do the same with **bottom**.
4. Repeat step 3 until all cards from **top** and **bottom** have been moved to **shuffled**.
5. Repeat steps 2 to 4, keeping track of the number of shuffles done, until **shuffled** is in order once more.
6. Print out the resulting number of shuffles. *I could also add more print statements along the way.*

Tip: if a piece of code will be called on repeatedly, we can store it in its own *function*.

Now try to turn each of these steps into Python code, then check back to see mine…

My Code

In general, it is hard even for the author of a program to easily make sense of an entire batch of code. First, read through the program from top to bottom, not worrying about the details, but concentrating instead on the general 'flow' of the program:
"Which bits happen when?" and "What loops are involved?" and "What gets printed and when?"

After that, you can read through again, focusing in detail on each individual block to work out exactly what it does and how: "What happens in this For loop?" or "What is meant by that If statement?"

The riffle function:

```
1   def riffle(cards):
2     top = cards[0:26]
3     bottom = cards[26:52]
4     shuffled = []
5     for i in range(26):
6       shuffled.append(top.pop(0))
7       shuffled.append(bottom.pop(0))
8     return shuffled
9
10  original = list(range(52))
11  deck = original
12
13  deck = riffle(deck)
14  count = 1
15  print(f"After 1 shuffle: {deck}")
16
17  while deck != original:
18    deck = riffle(deck)
19    count += 1
20    print(f"After {count} shuffles: {deck}")
21
22  print(f"Done.  Took {count} shuffles altogether")
```

The **riffle** function takes a list as its input, and splits it into a **top** and a **bottom** using the **splice** method.

The **shuffled** list is initially empty.
It loops 26 times, each time removing the first element of **top** and of **bottom**, and appending them to the end of **shuffled**.

The output is a riffle shuffled deck.

Note: this function is defined at the top of the code, but won't be run until and unless it is 'called' during the main body of code (in this case, on lines 13 and 18).

Making a deck of cards:

```
1   def riffle(cards):
2     top = cards[0:26]
3     bottom = cards[26:52]
4     shuffled = []
5     for i in range(26):
6       shuffled.append(top.pop(0))
7       shuffled.append(bottom.pop(0))
8     return shuffled
9
10  original = list(range(52))
11  deck = original
12
13  deck = riffle(deck)
14  count = 1
15  print(f"After 1 shuffle: {deck}")
16
17  while deck != original:
18    deck = riffle(deck)
19    count += 1
20    print(f"After {count} shuffles: {deck}")
21
22  print(f"Done.  Took {count} shuffles altogether")
```

The variable **original** is created using the **range** object, which generates values from 0 to 51, and by passing this into a **list** function it is automatically converted to a list. If you try printing it, you'll see it contains the numbers 0 to 51 in order.

Defining **deck** to be equal to **original** may seem odd, but this allows us to modify **deck** each time we shuffle, but still preserve the state of the original list of cards so we can check after each shuffle whether the deck is back to normal yet.

The initial shuffle

```
1   def riffle(cards):
2     top = cards[0:26]
3     bottom = cards[26:52]
4     shuffled = []
5     for i in range(26):
6       shuffled.append(top.pop(0))
7       shuffled.append(bottom.pop(0))
8     return shuffled
9
10  original = list(range(52))
11  deck = original
12
13  deck = riffle(deck)
14  count = 1
15  print(f"After 1 shuffle: {deck}")
16
17  while deck != original:
18    deck = riffle(deck)
19    count += 1
20    print(f"After {count} shuffles: {deck}")
21
22  print(f"Done.  Took {count} shuffles altogether")
```

The first line here replaces the list **deck** with the result of shuffling **deck**. A **count** is also initialised, to enable us to keep track of the total number of shuffles required to return to the original order.

The **print** statement uses an **f-string** so that we can view the shuffled deck.

It may seem odd to shuffle the deck once before starting the loop that follows on line 17, but if you try omitting this first shuffle the program will finish too early. This is because as soon as the loop begins, Python checks to see if the deck is in the same order as the original. Since, after 0 shuffles, it clearly is, we'll get the boring answer of 0 rather than the more interesting result we are looking for!

Shuffling the deck until it's back to normal

```
1   def riffle(cards):
2     top = cards[0:26]
3     bottom = cards[26:52]
4     shuffled = []
5     for i in range(26):
6       shuffled.append(top.pop(0))
7       shuffled.append(bottom.pop(0))
8     return shuffled
9
10  original = list(range(52))
11  deck = original
12
13  deck = riffle(deck)
14  count = 1
15  print(f"After 1 shuffle: {deck}")
16
17  while deck != original:
18    deck = riffle(deck)
19    count += 1
20    print(f"After {count} shuffles: {deck}")
21
22  print(f"Done.  Took {count} shuffles altogether")
```

The **while** loop tests to see if the deck is the same as the original, and as long as it is *not* equal, the code within is executed.

The code does the same as above, replacing **deck** with a shuffled deck, and increments the count by 1 to keep track of the total number of shuffles carried out.

The first **print** statement is indented so that it is called every time the loop runs. It tells us how many shuffles have been done, and show the current deck.

The *second* **print** statement is *not* in line with the body of the loop, so will only happen once, after the loop has finished.

*A **while** loop like this may cause problems: if repeated shuffling never gets us back to the original order, it would loop forever.*

Extending the code

The program used here will solve the specific problem posed:
"How many shuffles will it take to get a normal 52-card deck back in the original order?"
Can we adapt our code to answer related questions, like:
"How many shuffles will it take to get a 54-card deck (ie, including jokers) back in the original order?"
"How does the number of shuffles required relate to the number of cards in the deck?"

To answer a broader range of questions, we could adapt our code so that the number of cards is no longer 'hard-coded' as 52. Instead, we could make it a variable and make some adaptations to the **riffle** function to account for different lengths of list (take extra care with decks with an odd number of cards!)