SEQUENCES.
Collection of items which maintain their order.
Eg: Lists, Strings, Tuples, Dictionaries.

LIST.
An ordered sequence of items.
The order matters. The order is maintained with indices.
Created with [].

OPERATIONS ON AND PROPERTIES OF LISTS:

1. **Accessing:**

Typically, list variables hold addresses to the data/collection in memory, not the data (elements) themselves. Hence they are represented in the heap space.
Elements in a list are generally accessed using indices. The indices start at 0.

Examples. *(my_list = [1, 2], (list_len = len(my_list))*

```
my_list[0]                  >>> 1
my_list[-1]                 >>> 2
my_list[list_len - 1]       >>> 2
```

Lists are also mutable. We can change the elements at particular indices. To do this, we access the current element at our desired index and then set it to a new value: will be very useful when talking about for-loops.

```
my_list[0] = 4          >>> my_list[4, 2]
```

2. Insertion:

We can insert an item into a list. The item can be inserted at any position in the list. When we insert at the end of the list, we typically call it appending.

Methods.

```
my_list.insert(0, 3)      >>> [3, 1, 2]
```

```
my_list.append(5)       >>> [3, 1, 2, 5]
```

2. Slicing:

We can divide a list into sublists (which can even be the original list. Think every set is a subset of itself)

Examples. *(list_len = len(my_list)*

```
half_list = my_list[:list_len])  => my_list[:2]  >>> [3, 1]
```

3. Other operations: Addition, Reversals, Copying :

We can add 2 or more lists. The order of addition matters.

Eg:

```
[1, 2] + [3, 4] >>> [1, 2, 3, 4]
[3, 4] + [1, 2] >>> [3, 4, 1, 2]
```

We can reverse a list.

Eg:

```
rev_list1 = my_list.reverse()      >>> [5, 2, 1, 3]
rev_list2 = my_list[::-1]          >>> [5, 2, 1, 3]
```

Note:

Just assigning the list variable to another variable does not make a new copy of the list. To make a new copy of an existing list use [::]. We call this deep copy.

Eg: cpy_list = my_list : this will store in cpy_list an address to the same list that my_list points to. Any change using cpy_list will also be registered by my_list.

cpy_list = my_list[::] : this will create a new list for cpy_list. The new list will contain the same elements in my_list but a change done to cpy_list will not affect my_list and vice versa.

LOOPS.

A way to access the elements in a sequence. It is usually used for traversals (passing through the sequence).

Why a loop?

Imagine we have a list of 500 elements and we want to keep the elements at odd indices in a new list. A loop will provide an easy way to accomplish this in less lines of code! We needn't do :

```
new_list = [old_list[1], old_list[3], old_list[5].....]
```

LOOP VARIABLES.

In using loops, we need to track our progress- how far into the list we have gone. We do this by using a loop variable. This loop variable determines whether we should continue the process or not.

RANGE. *range(n)*

Range is an operator that returns a sequence(list) of numbers. It starts at 0 by default and goes to n-1. We can specify the start by throwing in another argument: range(start, end). Not that it will return [start, start+1….., end-1].

We generally use the elements in the list returned by range as the indices for our loop.

Eg:

```
my_letters = ["A", "B", "C"]
range(3) => [0, 1, 2]
```

We can create a mapping from range(3) to my_list since they have the same length.

FOR-LOOPS.

For-loops is one type of loop. It takes a number of forms depending on the kind of iteration you want to do.

We focus on 2 forms:

1. for each_item in my_list

2. for index in range(n)

for each_item in my_list.
Just as the name sounds! This format of the for-loop accesses each item of your list.
Eg:

```
for letter in my_letters:
    print(letter)

// Print only even numbers
for num in my_nums:
    if (num % 2) == 0:
        print(num)
```

Try and identify the loop variables.

for index in range(n).
It creates the appropriate list range(n). And the loops through that list. Since we can establish a relation between an existing list and the list from range(n), we can use the elements in the range(n) list as indices to access the items in our list.
NB: n should be equal to the length of the list when we want to go through the entire list.
Eg: *my_nums = [10, 20, 30, 40], n_len = 4*

```
for i in range(n_len):
    print(my_nums[i])
```

What happens if we do : print(i) instead? What is the loop variable?