1- Yes, a Python list can hold a mixture of integers and strings. In Python, a list is an ordered collection of items, and the items can be of any data type, including integers and strings. For example, you can create a list that contains both integers and strings like this:

my\_list = [1, "hello", 2, "world"]

In this example, the list contains two integers (1 and 2) and two strings ("hello" and "world"). You can access individual items in the list using indexing, like this:

print(my\_list[0])  # output: 1

print(my\_list[1])  # output: hello

print(my\_list[2])  # output: 2

print(my\_list[3])  # output: world

You can also add or remove items from the list using various list methods.

2- In Python, you can access elements of a list using negative indexing, which means that the index starts from the end of the list. For example, my\_list[-1] would give you the last element of the list, my\_list[-2] would give you the second-to-last element, and so on.

If you attempt to access an element of a list using a negative index that is out of range, you will get an IndexError exception. For example, if you have a list with three elements and you try to access the fourth element using a negative index, you will get an IndexError:

my\_list = [1, 2, 3]

print(my\_list[-4])  # raises an IndexError: list index out of range

However, if the negative index is within the range of the list, you will get the element at the corresponding position from the end of the list. For example:

my\_list= [1, 2, 3]

print(my\_list[-1])  # output: 3

print(my\_list[-2])  # output: 2

print(my\_list[-3])  # output: 1

3- Here's how you can create a list containing the values 45, -3, 16 and 8 in that order using a Python statement:

my\_list = [45, -3, 16, 8]

This statement creates a list called my\_list and initializes it with the four values 45, -3, 16, and 8, in that order. You can access the individual elements of the list using indexing, like this:

print(my\_list[0])  # output: 45

print(my\_list[1])  # output: -3

print(my\_list[2])  # output: 16

print(my\_list[3])  # output: 8

4- (a) The expression lst[0] represents the very first element of lst, which is 10.

(b) The expression lst[-1] represents the very last element of lst, which is 29.

(c) lst[0] represents the first element of lst, which is 10.

(d) lst[3] represents the fourth element of lst, which is 29.

(e) lst[1] represents the second element of lst, which is -4.

(f) The expression lst[-1] represents the last element of lst, which is 29.

(g) The expression lst[-4] represents the first element of lst, which is 10.

(h) The expression lst[3.0] is illegal, because the index must be an integer. If you try to use a non-integer index, you will get a TypeError exception.

5- Using the statements lst = [3, 0, 1, 5, 2] and x = 2, the following expressions evaluate to:

(a) lst[0] evaluates to 3.

(b) lst[3] evaluates to 5.

(c) lst[x] evaluates to 1.

(d) lst[-x] is the same as lst[-2] and evaluates to 5.

(e) lst[x + 1] is the same as lst[3] and evaluates to 5.

(f) lst[x] + 1 is the same as 1 + 1 and evaluates to 2.

(g) lst[lst[x]] is the same as lst[1] and evaluates to 0.

(h) lst[lst[lst[x]]] is the same as lst[lst[1]] which is lst[0], so it evaluates to 3.

6- In Python, you can use the built-in function len() to get the number of elements in a list. The len() function takes a list as its argument and returns the number of elements in the list.

For example, if you have a list called my\_list with 5 elements, you can use the len() function to get the number of elements like this:

my\_list= [1, 2, 3, 4, 5]

num\_elements = len(my\_list)

print(num\_elements)  # output: 5

In this example, the len() function is used to get the number of elements in my\_list, which is 5. The result is then stored in a variable called num\_elements and printed to the console.

7- In Python, the empty list is represented by a set of empty square brackets, like this:

my\_list = []

This creates a new list called my\_list that contains no elements. You can also use the list() constructor to create an empty list, like this:

my\_list = list()

Both of these expressions create an empty list in Python. You can add elements to the list using the append() method or other list methods, or you can initialize the list with elements when you create it.

8- Given the list lst = [20, 1, -34, 40, -8, 60, 1, 3] in Python, the following expressions evaluate to:

(a) lst evaluates to [20, 1, -34, 40, -8, 60, 1, 3].

(b) lst[0:3] evaluates to [20, 1, -34].

(c) lst[4:8] evaluates to [-8, 60, 1, 3].

(d) lst[4:33] evaluates to [-8, 60, 1, 3]. Note that the slice is extended to the end of the list since there are not enough elements in the list to include all the elements up to index 33.

(e) lst[-5:-3] evaluates to [-8, 60].

(f) lst[-22:3] evaluates to [20, 1, -34]. Note that the slice is extended to the start of the list since there are not enough elements in the list to include all the elements up to index -22.

(g) lst[4:] evaluates to [-8, 60, 1, 3]. This slice includes all elements from index 4 to the end of the list.

(h) lst[:] evaluates to [20, 1, -34, 40, -8, 60, 1, 3]. This slice includes all elements in the list.

(i) lst[:4] evaluates to [20, 1, -34, 40]. This slice includes all elements from the start of the list up to (but not including) index 4.

(j) lst[1:5] evaluates to [1, -34, 40, -8]. This slice includes all elements from index 1 up to (but not including) index 5.

(k) -34 in lst evaluates to True. The element -34 is present in the list.

(l) -34 not in lst evaluates to False. The element -34 is present in the list.

(m) len(lst) evaluates to 8. The list contains 8 elements.

10-

(a) [8, 8, 8, 8]

This creates a list with a single element, 8, and then multiplies it by 4, resulting in a list with four 8s.

(b) [2, 7, 2, 7, 2, 7]

This creates a list with two elements, 2 and 7, and then multiplies it by 6, resulting in a list with six elements, alternating between 2 and 7.

(c) [1, 2, 3, 'a', 'b', 'c', 'd']

This concatenates two lists: [1, 2, 3] and ['a', 'b', 'c', 'd'], resulting in a new list with all the elements of both lists in order.

(d) [1, 2, 1, 2, 1, 2, 4, 2]

This creates a list with two elements, 1 and 2, and then multiplies it by 3, resulting in a list with six elements, alternating between 1 and 2. Then, it concatenates [4, 2] to the end of that list.

(e) [1, 2, 4, 2, 1, 2, 4, 2, 1, 2, 4, 2]

This first concatenates [1, 2] and [4, 2] to create a new list [1, 2, 4, 2]. Then, it multiplies that list by 3, resulting in a list with twelve elements, repeating the pattern of [1, 2, 4, 2] three times.

11-

(a) [3, 5, 7, 9]

This list comprehension adds 1 to each element of the list [2, 4, 6, 8], resulting in a new list with elements [3, 5, 7, 9].

(b) [50, 60, 70, 80, 90]

This list comprehension multiplies each number in the range range(5, 10) by 10, resulting in a new list with elements [50, 60, 70, 80, 90].

(c) [12, 15, 18]

This list comprehension creates a list of numbers in the range range(10, 21) that are divisible by 3, resulting in a new list with elements [12, 15, 18].

(d) [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3)]

This list comprehension creates all possible tuples (x, y) where x is in the range range(3) and y is in the range range(4). This results in a new list with 12 tuples.

(e) [(0, 0), (0, 2), (1, 1), (1, 3), (2, 0), (2, 2)]

This list comprehension creates all possible tuples (x, y) where x is in the range range(3) and y is in the range range(4), but only includes those tuples where the sum of x and y is even. This results in a new list with 6 tuples.

12-

(a) [x\*\*2 for x in range(1, 6)]

This list comprehension generates the squares of the numbers in the range range(1, 6), resulting in a new list with elements [1, 4, 9, 16, 25].

(b) [x/4 for x in range(1, 7)]

This list comprehension generates the numbers in the range range(1, 7) divided by 4, resulting in a new list with elements [0.25, 0.5, 0.75, 1.0, 1.25, 1.5].

(c) [(char, num) for char in ['a', 'b'] for num in range(3)]

This list comprehension generates all possible combinations of characters 'a' and 'b' with numbers in the range range(3), resulting in a new list with elements [('a', 0), ('a', 1), ('a', 2), ('b', 0), ('b', 1), ('b', 2)].

13-

In Python, the expression to check if x is a member of lst is x in lst.

This expression returns a boolean value True if x is present in lst, and False otherwise. Here's an example:

lst = [1, 2, 3, 4, 5]

x = 3

if x in lst:

    print("x is present in lst")

else:

    print("x is not present in lst")

In this example, since x is present in lst, the output will be "x is present in lst".

14-

In Python, reversed() is a built-in function that returns a reverse iterator. This iterator can be used to traverse a sequence (e.g. a list, tuple, or string) in reverse order.

Here's an example of how to use reversed() to iterate over a list in reverse order:

lst = [1, 2, 3, 4, 5]

for item in reversed(lst):

    print(item)

In this example, reversed(lst) returns a reverse iterator for the list lst. The for loop then iterates over this iterator, printing each element of the list in reverse order. The output would be:

5

4

3

2

1

Note that reversed() does not modify the original sequence. If you want to create a new list with the elements of a sequence in reverse order, you can use the [::-1] slice notation instead, like this:

lst = [1, 2, 3, 4, 5]

reversed\_lst = lst[::-1]

print(reversed\_lst)

In this example, lst[::-1] returns a new list with the elements of lst in reverse order, which is then assigned to the variable reversed\_lst. The output would be:

[5, 4, 3, 2, 1]

15-

Here's a Python function that adds up all the positive values in a list of integers:

def sum\_positive(a):

    # Initialize the sum to zero

    total = 0

    # Iterate over the list

    for num in a:

        # Check if the number is positive

        if num > 0:

            # Add the positive number to the total

            total += num

    # Return the sum of positive numbers

    return total

This function first initializes the sum to zero, then iterates over the input list a using a for loop. For each number in the list, it checks if the number is positive using an if statement. If the number is positive, it adds it to the total using the += operator. Finally, the function returns the sum of positive numbers.

If the input list is empty, the function will return zero since the total was initialized to zero and there are no positive numbers to add to it.

16-

def count\_evens(lst):

    count = 0

    for num in lst:

        if num % 2 == 0:

            count += 1

    return count

This function takes a list lst as an argument and initializes a variable count to zero. It then loops through each element in the list and checks if it is even by using the modulus operator % to check if the element is divisible by 2 with no remainder. If the element is even, the count variable is incremented by 1. Finally, the function returns the count variable, which represents the number of even elements in the list. If the list is empty, the function will return 0 because count was initialized to 0.

17-

def print\_big\_enough(lst, num):

    for element in lst:

        if element >= num:

            print(element)

This function takes two parameters: a list of numbers lst and a number num. It loops through each element in the list and checks if it is greater than or equal to the second parameter num. If it is, it prints the element to the console. If the element is less than num, it is skipped. This repeats for all elements in the list that meet the condition.

Note that the function only prints the elements that meet the condition, it does not return them as a list or modify the original list.

18-

def next\_number(lst):

    lst\_set = set(lst)  # Convert list to set for O(1) membership testing

    next\_num = 1

    while next\_num in lst\_set:

        next\_num += 1

    return next\_num

This function takes a list of integers lst and first converts it to a set lst\_set. This is done to enable O(1) membership testing, which will be used in the loop.

The function initializes a variable next\_num to 1, which is the smallest positive integer. It then enters a while loop that checks if next\_num is in the set lst\_set. If it is, it increments next\_num by 1 and checks again. This continues until next\_num is not in lst\_set, at which point it is the smallest positive integer not in the list. The function then returns next\_num.

Note that if the input list is empty, the function will return 1, as per the problem statement.

19-

def reverse(a):

    length = len(a)

    for i in range(length // 2):

        a[i], a[length - 1 - i] = a[length - 1 - i], a[i]

This function takes a list a as an argument and initializes a variable length to the length of the list. It then loops through the first half of the list using the range() function and swaps the elements at opposite ends of the list using a temporary variable. This effectively reverses the order of the elements in the list.

Note that this function modifies the original list a in place, rather than creating a new list.

20-

m = [[1]\*9 for i in range(6)]  # Creates a 6x9 matrix filled with 1's

import pprint

pprint.pprint(m)  # Pretty prints the matrix

m[2][4] = 0  # Reassigns the element at position [2][4] to 0

pprint.pprint(m)  # Pretty prints the modified matrix

The first line of code creates a 6x9 matrix filled with 1's using a list comprehension.

The second line of code imports the pprint module, which provides a way to pretty print Python data structures.

The third line of code pretty prints the matrix m using pprint.pprint(). This prints each row of the matrix on a separate line with appropriate spacing for readability.

The fourth line of code reassigns the element at position [2][4] to 0.

The fifth line of code pretty prints the modified matrix m using pprint.pprint(). This prints the matrix in the same format as before, but with the modified element at position [2][4] now equal to 0.

21-

1. Using a list comprehension:

lst = [i for i in range(1, 11)]

This creates a list lst using a list comprehension that iterates over the range 1 to 11 (exclusive) and adds each element to the list.

2. Using the list() function with the range() function:

lst = list(range(1, 11))

This creates a list lst by converting the range 1 to 11 (exclusive) to a list using the list() function.

3. Using the append() method in a for loop:

lst = []

for i in range(1, 11):

    lst.append(i)

This creates an empty list lst and then uses a for loop to iterate over the range 1 to 11 (exclusive) and append each element to the list.

4. Using the extend() method with a list:

lst = []

lst.extend([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

This creates an empty list lst and then extends it with another list containing the elements 1 through 10.

5. Using the \* operator with a list:

lst = [1, 2, 3, 4, 5] \* 2 + [6, 7, 8, 9, 10]

This creates a list lst by concatenating two copies of the list [1, 2, 3, 4, 5] using the \* operator and then appending the list [6, 7, 8, 9, 10].

22-

def check\_match(matrix):

    n = len(matrix)

    for i in range(n):

        if matrix[i] in matrix[:i] + matrix[i+1:]:

            return True

        col = [row[i] for row in matrix]

        if col in matrix[:i] + matrix[i+1:]:

            return True

    return False

This function takes a square 2D list matrix as input and returns True if the left to right contents of any row equals the top to bottom contents of any column. If no row matches any column, the function returns False.

The function loops through each row and column of the matrix, and checks whether the contents of the row match the contents of any other row, or whether the contents of the column match the contents of any other column. If a match is found, the function returns True. If no match is found, the function returns False.

23-

def check\_winner(board):

    for i in range(3):

        # Check rows

        if board[i][0] == board[i][1] == board[i][2] and board[i][0] != " ":

            return board[i][0]

        # Check columns

        if board[0][i] == board[1][i] == board[2][i] and board[0][i] != " ":

            return board[0][i]

    # Check diagonals

    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ":

        return board[0][0]

    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ":

        return board[0][2]

    # No winner found

    return " "

This function takes a 3x3 list board as input and checks for a winning Tic-Tac-Toe pattern. It loops through each row and column of the board and checks whether all three positions in a row or column contain the same non-empty string. If a winning pattern is found, the function returns the corresponding string ("X" or "O").

The function also checks for winning patterns along the diagonals of the board. If a winning pattern is found, the function returns the corresponding string ("X" or "O").

If no winning pattern exists, the function returns an empty string " ".